AGH UNIVERSITY OF SCIENCE AND TECHNOLOGY
Faculty of Physics and Applied Computer Science

Master thesis

Mikołaj Gralczyk

major: **medical physics**

specialisation: **dosimetry and electronics in medicine**

Programming and testing of the high-speed data acquisition system based on USB standard

Supervisor: **dr hab. inż. Bartosz Mindur**

Cracow, July 2018

Aware of criminal liability for making untrue statements I declare that the following thesis was written personally by myself and that I did not use any sources but the ones mentioned in the dissertation itself.

Cracow, July 2018

The subject of the master thesis and the internship by Mikołaj Gralczyk, student of 5th year major in medical physics, specialisation in dosimetry and electronics in medicine

The subject of the master thesis: **Programming and testing of the high-speed data acquisition system based on USB standard**

Supervisor:	dr hab. inż. Bartosz Mindur
Reviewer:	dr. inż. Paweł Hottowy
A place of the internship:	WFiIS AGH, Kraków

Programme of the master thesis and the internship

1. First discussion with the supervisor on a realization of the thesis.
2. Collecting and studying the references relevant to the thesis topic(s) (especially the OpalKelly's documentation).
3. The internship:
   - Familiarization with the electrical and functional diagram of the Opal Kelly XEM6310 module and its parameters.
   - Getting to know the API and trying to integrate the module with the computer.
   - Creating the first, simple programs, that communicate with the module.
   - Preparation of the internship report.
4. Developing the test environment both for PC and the module.
5. Performing tests on the module.
6. Final analysis of the results obtained, conclusions – discussion with and final approval by the thesis supervisor.
7. Typesetting the thesis.

Dean's office delivery:

. . . .
(signature of the supervisor)

*Pragnę podziękować mojemu promotorowi **dr hab. inż. Bartoszowi Mindurowi** za okazaną pomoc merytoryczną i cenne uwagi w trakcie powstawania niniejszej pracy.*

*Dziękuję również mojej **Rodzinie**, bez której ukończenie studiów nie byłoby możliwe.*

*Chciałbym również podziękować **Katarzynie Chmielewskiej** za emocjonalne wsparcie i nieustającą motywację w trakcie pisania pracy.*

Contents

List of Figures

List of Tables

1. Introduction

Thanks to advances in computer development which started during the second half of the twentieth century, modern neuroscience has a significant knowledge database about the nervous system nowadays, especially in anatomical and physiological contexts. The technology allowed researchers to learn more about the activity of a single neuron, the anatomy of the central nervous system (CNS) and how it responds to some stimulus. However, a brain is such a complex structure of multiple neuron connections that is still not investigated very well. One of the hardest issue to study is the principle of cooperation between its different parts. Hence, a research team from Department of Physics and Applied Computer Science AGH UST in Krakow started to develop a microelectronic system for large-scale electrical stimulation and recording of brain activity based on multielectrode arrays (MEAs). This solution allows to record the activity of neurons on the behaving animals (like mouse or rat) in different parts of their brain as well as to stimulate cells to discover algorithms of signal processing in the brain circuits [1].

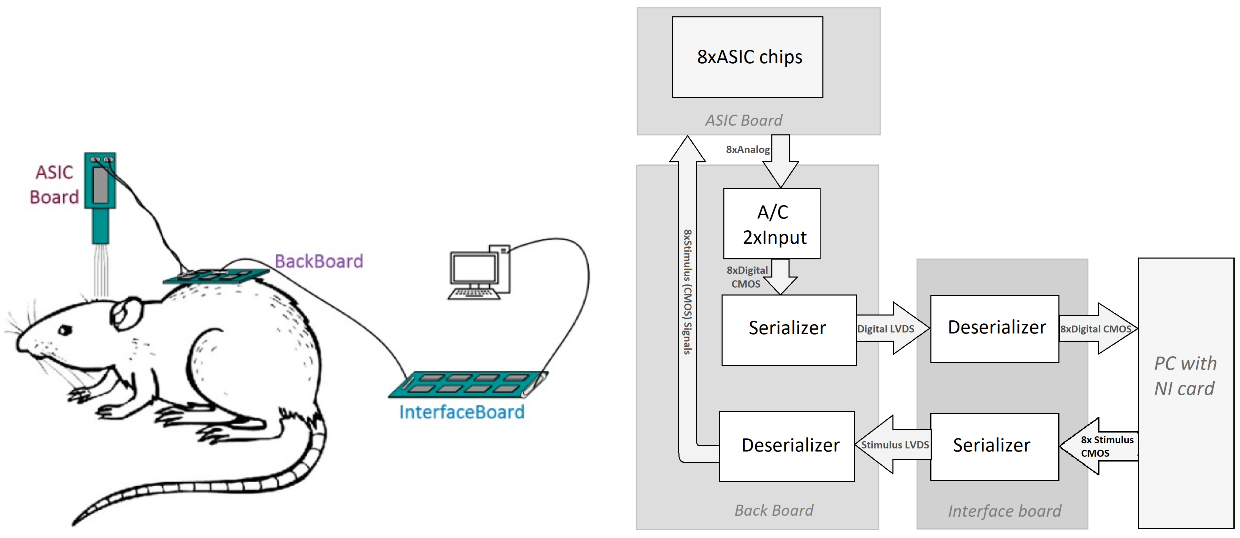
The purpose of this thesis is to use a dedicated OpalKelly's module that will be a part of the solution mentioned above. Its responsibility will be active data management. The module contains a primary interface that enables transferring data from and to PC using the USB 3.0 standard. The principal goal is to run this module and test its functionality and performance by building a dedicated environment on PC and module side.

The solution should meet the following requirements to achieve thesis goals:

- Data transfer should be *fast.* It means that transmission should have as highest data rate as possible. Moreover, the system requires asynchronous transfer, so the system should work in duplex mode. In practice, OpalKelly's modules that have USB 3.0 port allows synchronous data transfer up to 340 MB/s [2].
- Data transfer should be *efficient.* While transferring, no loss should be performed. OpalKelly's interface is intended to deliver 100% of the information. The cost of such solution is lack of isochronous transfer available for use as it is used for continuous error checking [6].
- The whole environment should be *universal and portable.* The solution should work with any modern computer (that is why USB port is used), independently from platform installed on it (Windows, Linux or macOS). An additional advantage will be the ability to reconfigure it in any way.

2. Description of the system for neuroscience studies

The system dedicated for studying brain activity (mentioned in section 1) is based on CMOS technology and has many elements that allow communication with PC. It has four fundamental components: ASIC[1] Board, Back Board, Interface Board and card connected to PC via PXI[2] interface (NI 6537 DIO). The first one is a sophisticated device that controls a matrix of electrodes implanted into an animal's brain. It can both stimulate and record neuronal activity using 64 independent channels. Analog signals from it go to the second component where ADC converts them to digital signals and then, after serialization, are transmitted to Interface Board using LVDS[3] lines. Interface Board deserialize signals and sends them via CMOS lines to NI card. Dedicated LabView application (that runs on PC) modifies signals into data understandable for neuroscientists. From the other perspective of the transmission, PC generates signals described in the previously mentioned application, which are sent then to Interface Board connected with NI card. After serialization, signals go to Back Board. There, they are deserialized and transmitted to the ASIC Board. Figure 1 present the concept of this system.



(a) The system concept drawing **(b)** Functional block diagram

Figure 1. Presentation of the system used in neuroscientific experiments [1].

The description above proves that the system is very complicated and needs very specialized devices which are expensive and difficult to handle. The most prominent advantage of this solution is high performance. However, some part of it can be simplified by replacing the NI card with a component that will work with any computer. This kind of solution may reduce costs and make the system more useful.

Hence, the NI card will be replaced by the OpalKelly's module (Figure 2 visualizes the system after replacement). Consequently, it will simplify the whole design and reduce costs of maintenance. Besides, the whole system will be more portable (as NI card requires a dedicated chassis). With plug&play interface, the system will be handier for neuroscientists, as their only duty will be inserting USB cable to port in their computers. The module will be a proxy between PC and Interface Board card. Thanks to expansion board BRK6110 provided with the module all CMOS lines will be directly connected with it. The module provides real-time data

[1] *Application-Specific Integrated Circuit* – a semi-custom for a particular application integrated circuit [7].
[2] *PCI eXtensions for Instrumentation* – modular PC-based instrumentation platform for measurement and automation systems [8].
[3] *Low-voltage differential signaling* – a signaling method used for high-speed, low-power transmission of binary data over copper [9].

manipulation so no latency should be noticed. The only apparent difference will be lower data transfer speed.

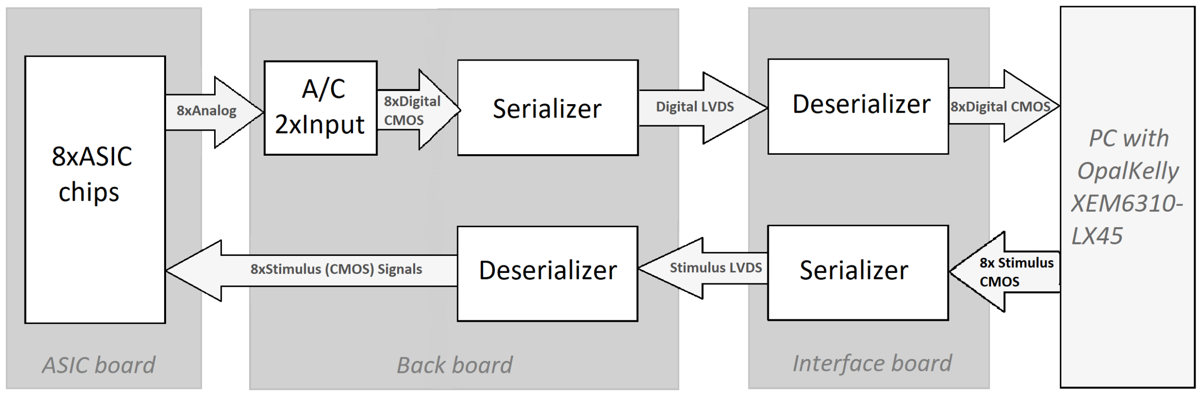


Figure 2. Functional block diagram of the system for neuroscientific experiments after replacing the NI card with the OpalKelly module.

3. Theory

3.1. Module for data transmission

The OpalKelly XEM6310-LX45 device was used as the module for data transmission to achieve the goal of this thesis. The reason for this choice is the fact that it is relatively small to other devices used in neuroscientific experiments (which makes it portable) and has two essential components embedded in:

- Spartan-6 XC6SLX45-2FGG484 – a special integrated circuit, made by Xilinx, known as Field-Programmable Gate Array (FPGA) that is fast, low-energy consuming, reprogrammable and cheaper than dedicated ASICs. Also, it can do some additional logic operations on data. It can convert signals from many wires to data transmitted by one USB cable in real-time.

- SuperSpeed USB 3.0 – Universal Serial Port that is a general-purpose interface for all modern PCs. Additionally, the USB 3.0 can achieve simultaneous read and write speed up to 5 Gb/s [10]. It is also simple to use because it is a popular plug&play component for data transmission.

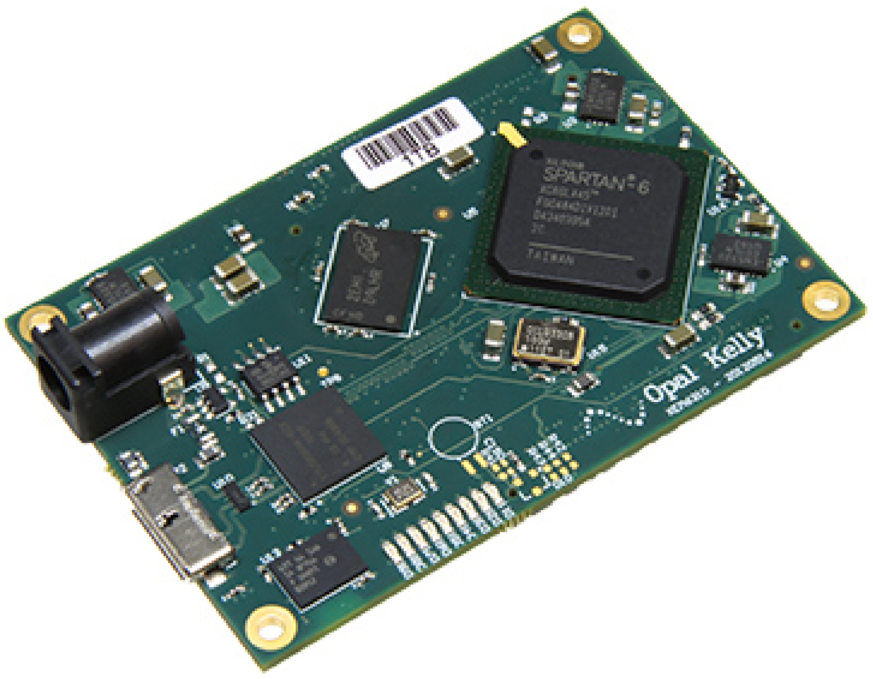


Figure 3. Picture of OpalKelly XEM6310-LX45 [2].

The module has an outside-world connection via USB 3.0. This port is used to connect the board with PC and is managed by Cypress FX3 microcontroller which is the host interface for the whole module and is responsible for data transfer, upload configuration file (which actually can be stored in 16 MiB system flash connected to the microcontroller) and debug. The connection between USB and FPGA provides the Host Interface Bus. The FPGA performs its operations with 100 MHz clock and coops with additional modules – a flash (where there can be stored some data even offline) and DDR2 SDRAM with the capacity of 128MiB. 5V DC voltage powers the whole module. A functional block diagram presented in Figure 4 shows all interconnections between components described above.

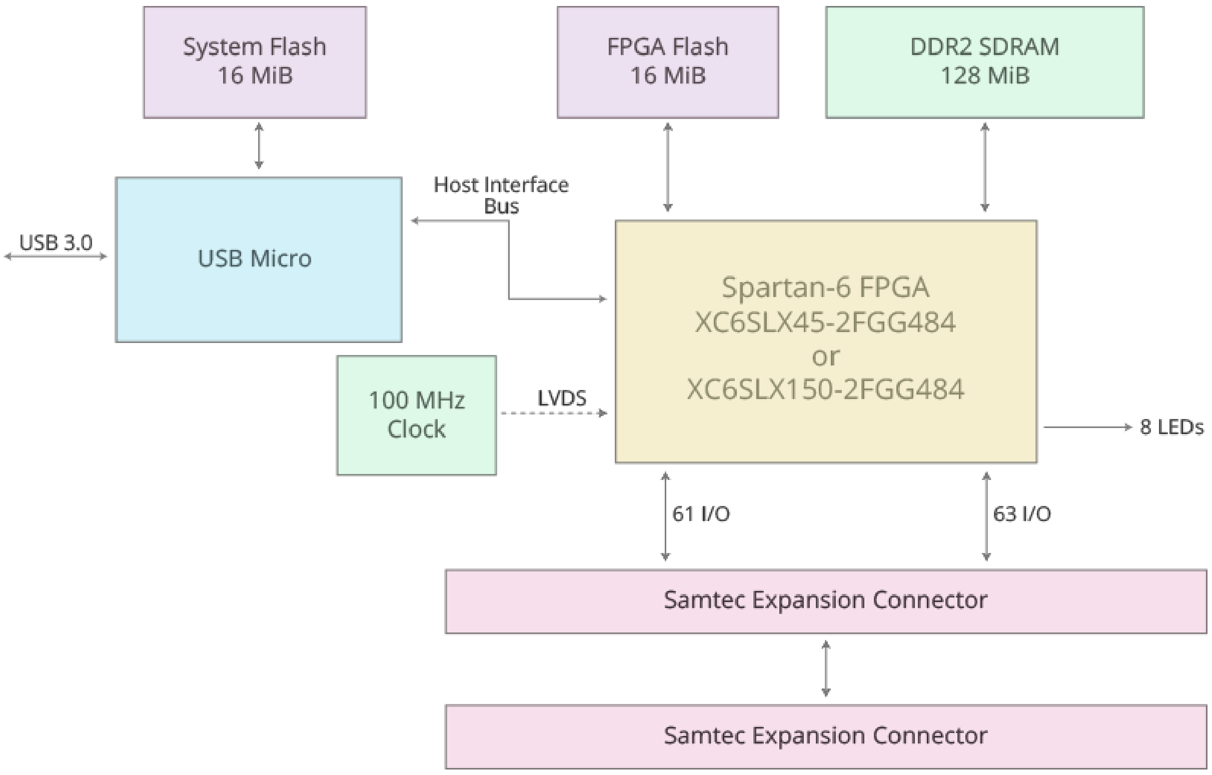


Figure 4. Functional block diagram of OpalKelly XEM6310-LX45 [2].

Samtec Expansion Connectors provides a direct connection with FPGA via 124 input/output ports. Expansion board BRK6110 can use these connectors and play a role as an extension for any electric device that can connect with FPGA. Also, the board has an additional XILINX JTAG port (for debugging) and power supply port. Although the board was not used in this thesis, after few modifications, it will be able to perform transfer between ASICs and FPGA.

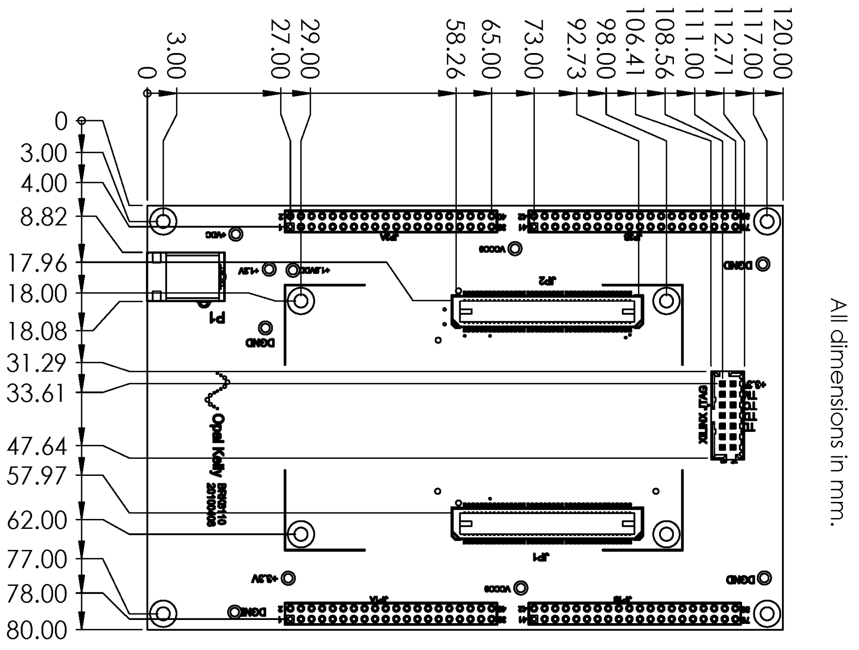


Figure 5. Drawing of BRK6110 – the expansion board of OpalKelly XEM6310-LX45 [3].

3.2. FrontPanel description

3.2.1. Preface

FrontPanel is a platform formed by OpalKelly to manage any physical device of this company. It provides necessary tools for configuring the FPGA as well as all other integrated components, such as RAM. The idea of this platform is to simplify the process of project building, minimize its development time and efficiently handle data transfer.

The FrontPanel environment contains the following components:

- FrontPanel SDK (Software Development Kit) – refers to all programming tools which are part of FrontPanel library.
- FrontPanel HDL (Hardware Description Language) – a library for Verilog and VHDL languages. It is a complete toolkit for low-level implementation of FPGA behavior.
- FrontPanel API (Application Programming Interface) – an interface for creating applications on PC which provides communication with the FPGA. The significant advantage of it is compatibility with many popular programming languages, such as C/C ++, C#, Python, Java, and Ruby.

FrontPanel is compatible with USB 2.0, USB 3.0 and PCI Express ports [11]. By using them (via PC), the user can freely configure the module and control data flow. The various data transfer speeds manifest the fundamental difference between these ports. The exact values of the clock controlling the transfer cause this distinction. In the case of the module used in this work, the clock frequency is 100.8 MHz (9.92 ns per period) due to the presence of USB 3.0 port [6].

3.2.2. FrontPanel's Application Programming Interface

The FrontPanel API is a cross-platform dynamically-linked library (written in C++) for PC applications that control OpalKelly module systems. It can be used with popular programming languages such as C/C++, Python, C#, Java, and Ruby. It provides compatibility with popular operating systems such as Windows, Linux or macOS (OS X) with both 32- and 64-bit architecture [12].

The FrontPanel API consists of several basic classes that allow managing the modules remotely. The most significant in the case of this work is the `okCFrontPanel` class. It is used to find and configure the device and to communicate directly with FPGA. One of the types of the `okCFrontPanel` class is `ErrorCode`. In case of an error event during usage, it allows identifying issue by returning its value in the form of a negative integer number.

Regarding functionality, the methods of the `okCFrontPanel` class are divided into three groups:

- Device interaction – methods for opening communication ports between the module and the PC, as well as systematizing modules information data (if the number of devices connected is higher than one). The most notable method is `OpenBySerial()` with its optional argument – serial number of the module which has to be connected. If it is not passed, the connection will be opened for the first detected device.
- Device configuration – when the connection with the module is already established, the methods of this group allow configuring the FPGA and all additional elements connected to it (e.g., flash memory). The most important is `ConfigureFPGA()`, which sets the FPGA using bitfile passed as the function argument.
- FPGA configuration – the methods of this group allow sending signals between FPGA and PC. The most important methods are described in 3.2.4 subsection.

3.2.3. FrontPanel's Hardware Description Language

One of the principal element of the FrontPanel's structure is the host interface. It is a software block that facilitates establishing communication between PC units and FPGA via USB bus. To be more specific, it controls the physical connection of the FPGA with the microcontroller managing the USB port.

Often, the FPGA implementations may require contact with PC application. As it is a very low-level operation, OpalKelly provides some high-level abstraction layer for developers. Therefore, FrontPanel uses the concept of endpoints. Somehow, they behave like external pins, but in fact, they play a role as ports for signals passing through the FPGA. Endpoints are divided into three group: wires, triggers, and pipes. Concerning the direction of data transfer, they are also classified into inputs (enter data into the device) and outputs (transfer data from the device) [13].

Endpoints and the host bus are permanently connected with each other. The designer can declare many of endpoints but must respect the naming rules which says that each endpoint type has a specific address range (see Table 1). The addresses can be sent in the form of an 8-bit signal via the `ep_addr` input port. Then, data can be transferred synchronously or asynchronously relative to the clock, depending on the type of endpoint.

Table 1. *List of FrontPanel's endpoint types with their parameters.*

Endpoint type	Address range	Synchronicity	Data type
Wire In	0x00 - 0x1F	No (asynchronous)	Signal state
Wire Out	0x20 - 0x3F	No (asynchronous)	Signal state
Trigger In	0x40 - 0x5F	Yes	One-shot signal
Trigger Out	0x60 - 0x7F	Yes	One-shot signal
Pipe In	0x80 - 0x9F	Yes	Multi-byte transfer
Pipe Out	0xA0 - 0xBF	Yes	Multi-byte transfer

In FrontPanel HDL, the host interface is known as the `okHost` module. This component is responsible for many aspects:

- On PC side – handling input signal `okUH` from PC to USB microcontroller, output signal `okHU` from USB to PC and input/output signals `okUHU` and `okAA`.
- On FPGA side – handling output signal `okHE` from USB microcontroller to the endpoints (via so-called target interface bus), input signal `okEH` from endpoints collected by `okWireOR`, and output signal `okClk` which is the clock for whole design (100.8 MHz).

Table 2. *List of signals handled by okHost with their direction and description.*

Signal	Direction	Description
`okUH[4:0]`	Input	Host interface input signals
`okHU[2:0]`	Output	Host interface output signals
`okUHU[31:0]`	Input/Output	Host interface bidirectional signals
`okAA`	Input/Output	Host interface bidirectional signal
`okHE[112:0]`	Output	Controls signals to the target endpoints
`okEH[64:0]`	Input	Controls signals from the target endpoints
`okClk`	Output	Buffered copy of the host interface clock (100.8 MHz)

Listing 1. *Instance of the* `okHost` *module.*

```
okHost okHI (.okUH(okUH), okHU(okHU), .okUHU(okUHU), .okAA(okAA),
             .okClk(okClk), .okHE(okHE), .okEH(okEH));
```

The `okWireOR` module is responsible for collecting all signals from all endpoints and transmit them to the `okHost` using the `okEH` output signal. By signal state, each endpoint is informed about permission to transfer its data. If the state is high, the permission is granted. In another case, the state is always low, and data transfer is interrupted. The `okWireOR` performs logical disjunctions (OR) on each bit of the interface bus and returns a result. Thus, many endpoints can share the bus at the same time.

A diagram presented in Figure 6 summarizes the FrontPanel HDL interface and shows all relations between modules described in this subsection.

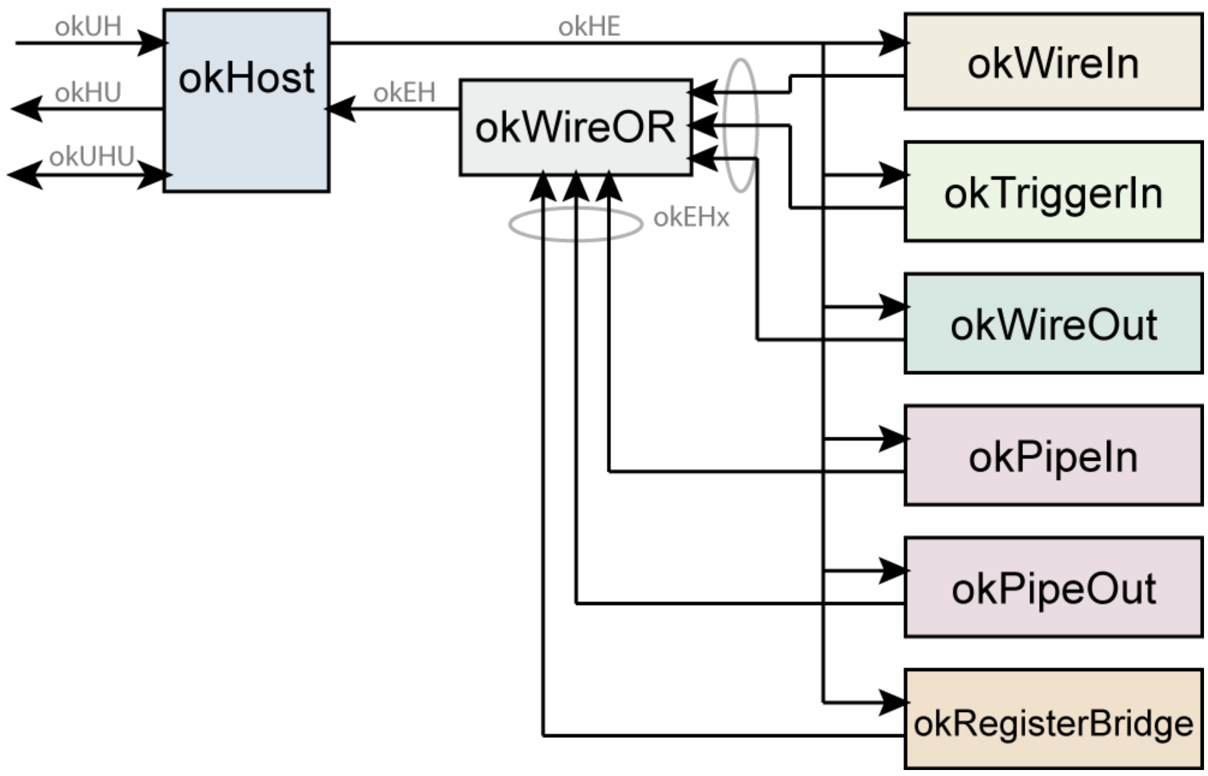


Figure 6. Diagram that illustrates the structural relationships in FrontPanel HDL between the various endpoints, the okWireOR, and okHost modules [4].

3.2.4. Endpoints

Wire

The wires' purpose is to receive and send asynchronous signals that are a state of any element in the design (e.g., LED). Often, the components can transmit many data with high rate via wires, which can lead to bandwidth overload. So, FrontPanel interface applies the poll mechanism periodically and register all wires' states at the same time. The user can specify the flow of this component but must keep in mind that transfer also depends on PC capabilities. When the maximum rate (25 milliseconds) is applied, the bandwidth is still low consumed, so the user should not notice any performance penalty.

`okWireIn` transfers signal state to the project via single 32-bit bus interface output `ep_datain`. On the other side, `okWireOut` returns 32-bit signal via input `ep_datain`. Both of them work asynchronously. The proper instance of `okWireIn` and `okWireOut` is shown in Listing 2.

Listing 2. *Instances of the* `okWire` *endpoint.*

```
okWireIn wire00 (.okHE(okHE), .ep_addr(8'h00), .ep_dataout(ep00data));

okWireOut wire20 (.okHE(okHE), .okEH(okEH), .ep_addr(8'h20),
                  .ep_datain(ep20data));
```

The user should use the method presented in Listing 3 to provide value to `okWireIn`.

Listing 3. `SetWireInValue()` *function.*

```
okCFrontPanel:ErrorCode okCFrontPanel::SetWireInValue (int ep,
                UINT32 val, UINT32 mask = 0xffffffff);
```

Arguments:

- `ep` – address of `okWireIn` target.
- `val` – value for `okWireIn` (32-bit digit number).
- `mask` – target mask for the new value.

Values returned:

- `NoError (0)` – process succeeded.
- `DeviceNotOpen (-8)` – no access to the device (even if plugged in).
- `InvalidEndpoint (-9)` – the address is out of the range specified for `okWireIn`.

All wire inputs have to be updated with the method presented in Listing 4 to deliver the given value.

Listing 4. `UpdateWireIns()` *function.*

```
okCFrontPanel::ErrorCode okCFrontPanel::UpdateWireIns();
```

All wire outputs need to be updated with the method presented in Listing 5 to receive value from `okWireOut`. After, the user should use the method shown in Listing 6 which returns a value of the target `okWireOut` endpoint of `epAddr` address.

Listing 5. `UpdateWireOuts()` *function.*

```
okCFrontPanel::ErrorCode okCFrontPanel::UpdateWireOuts();
```

Listing 6. `GetWireOutValue()` *function.*

```
UINT32 okCFrontPanel::GetWireOutValue(int epAddr);
```

Trigger

Triggers are useful endpoints that synchronize connection between PC and FPGA. They can be used in transfer operations (e.g., launching transmission) or they can inform about the module's state. Their instances are presented in Listing 7.

Listing 7. *Instances of the* `okTrigger` *endpoint.*

```
okTriggerIn trigIn40 (.okHE(okHE), .ep_addr(8'h40), .ep_clk(okClk),
                      .ep_trigger(ep40trig));

okTriggerOut trigOut60 (.okHE(okHE), .okEH(okEH), .ep_addr(8'h60),
   .ep_clk(okClk), .ep_trigger(ep60trig));
```

Both `okTriggerIn` and `okTriggerOut` are synchronized with the clock by the `ep_clk` port. The input trigger is asserted for a single clock cycle and affects the FPGA's process. In contrast, output trigger triggers the PC when rising edge of a signal is detected. As the consequence of polling mechanism, the trigger's state remains till next poll gets it and resets. For both of modules, the trigger value can be passed using the 32-bit `ep_trigger` port. The user can trigger action from PC using the method presented in Listing 8.

Listing 8. `ActivateTriggerIn()` *function.*

```
okCFrontPanel::ErrorCode okCFrontPanel::ActivateTriggerIn(int epAddr, int bit);
```

Arguments:

- `epAddr` – address of `okTriggerIn` target endpoint.
- `bit` – determine which bit of the 32-bit frame should reach a high state.

Values returned:

- `NoError (0)` – process succeeded.
- `InvalidEndpoint (-9)` – the address is out of the range specified for `okTriggeIn`.

To check states of output triggers, the poll method presented in Listing 9 needs to be used.

Listing 9. `UpdateTriggerOuts()` *function.*

```
okCFrontPanel::ErrorCode okCFrontPanel::UpdateTriggerOuts();
```

Next, the method shown in Listing 10 allows to check if a specific trigger changed its state. It returns true if the trigger reaches high state. The mask parameter determines which bit need to be taken into account.

Listing 10. `IsTriggered()` *function.*

```
bool okCFrontPanel::IsTriggered(int epAddr, UINT32 mask);
```

Pipe

Pipes are endpoints for synchronous multi-byte data transfer between PC and FPGA. They must obey the host (which is always master for whole design) and receive data flow with rate 100.8 MHz. Data is accepted by `okPipeIn` (via 32-bit `ep_dataout` output port) when it receives on output port `ep_write` 1-bite high-level signal. When the signal is still high during next clock intervals, host accepts new data till low. Analogically, the same situation happens for `okPipeOut`. If the host wants to read data from FPGA, it asserts a high-level signal on an `ep_read` input port. However, only from next clock interval data is transmitted.

The fundamental condition of proper data flow is the high interface sensitivity for incoming and outgoing data. If the target needs to handle data in block fashion, then it is recommended to link pipes module with FIFO (generated using Xilinx CORE).

Instances of `okPipe` are presented in Listing 11 with visualization of work synchronously to clock in Figure 7 and 8.

Listing 11. *Instances of the* `okPipe` *endpoint.*

```
okPipeIn pipeIn80 (.okHE(okHE), .okEH(okEH), .ep_addr(8'h80),
                   .ep_dataout(ep80pipe, .ep_write(ep80write)));

okPipeOut pipeOutA0 (.okHE(okHE), .okEH(okEH), .ep_addr(8'ha0),
                   .ep_datain(epA0pipe), .ep_read(epA0read));
```

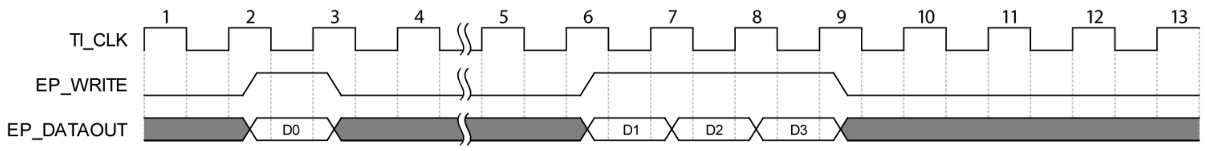


Figure 7. Timing diagram of okPipeIn [4].

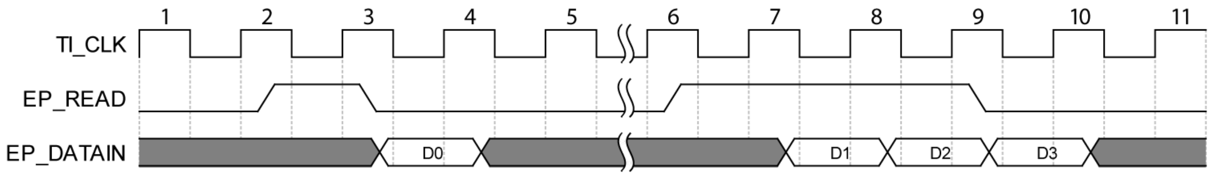


Figure 8. Timing diagram of okPipeOut [4].

There is a variant of the `okPipe` module that can handle the signal in block fashion – Blok-Throttled Pipe. In its instance, some additional ports can be found – `ep_blockstrobe` and `ep_ready`. FPGA controls the `ep_ready` and should assign on it high signal whenever is ready to transfer the whole block of data without any interruption. If the signal on `ep_ready` is low, then no transfer will be performed. The main difference between block throttle and `okPipe` is that transfer can be interrupted while using the regular pipe if the transferred data size is higher than 8192 bytes. It is because firmware put limitations on the maximum length per transfer. However, API will continue transferring extensive data by multiplying transfers if needed. The length of the data array must be a multiplied integer of a minimum transfer size which is 16 in the case of USB 3.0. In case of block variation, the rule says that the block size must be a power of two and fit in the range depending on the subtype of USB:

- FullSpeed: <16, 64>.
- HighSpeed: <16, 1024>.
- SuperSpeed: <16, 16384>.

Instances of `okBlokThrottledPipe` are presented in Listing 12 with visualization of work synchronously to clock in Figure 9 and 10.

Listing 12. *Instances of the* `okBTPipe` *endpoint.*

```
okBTPipeIn pipeIn90 (.okHE(okHE), .okEH(okEH), .ep_addr(8'h90),
                     .ep_dataout(ep90pipe), .ep_write(ep90write),
                     .ep_blockstrobe(ep90strobe), .ep_ready(ep90ready));

okBTPipeOut pipeOutB0 (.okHE(okHE), .okEH(okEH), .ep_addr(8'hb0),
                      .ep_datain(epB0pipe), .ep_read(epB0read),
                      .ep_blockstrobe(epB0strobe), .ep_ready(epB0ready));
```

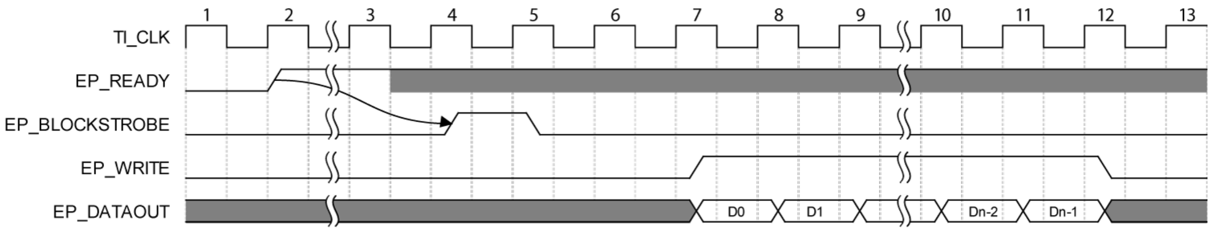


Figure 9. Timing diagram of okBTPipeIn [4].

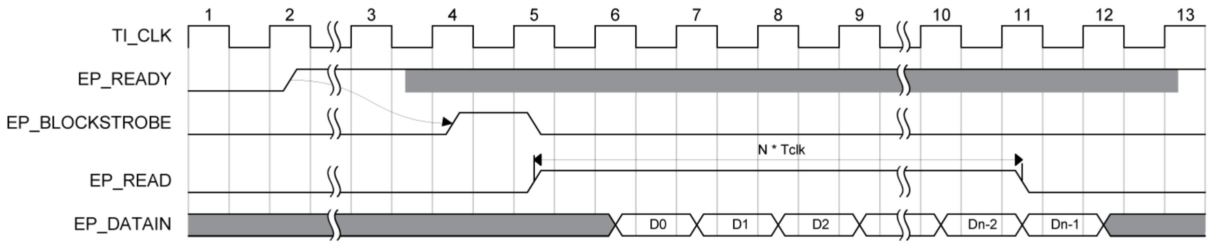


Figure 10. Timing diagram of okBTPipeOut [4].

Methods presented in Listing 13, 14, 15, and 16 allow manipulating data using pipes.

Listing 13. `WriteToPipeIn()` *function.*

```
long okCFrontPanel::WriteToPipeIn(int epAddr, long length,
                const unsigned char * data);
```

Listing 14. `WriteToBlockPipeIn()` *function.*

```
long okCFrontPanel::WriteToBlockPipeIn(int epAddr, int blockSize,
                long length, const unsigned char * data);
```

Listing 15. `ReadFromPipeOut()` *function.*

```
long okCFrontPanel::ReadFromPipeOut(int epAddr, long length,
                unsigned char * data);
```

Listing 16. `ReadFromBlockPipeOut()` *function.*

```
long okCFrontPanel::ReadFromBlockPipeOut(int epAddr, int blockSize,
                long length, unsigned char * data);
```

Arguments:

- `epAddr` – address of pipe target.
- `length` – length of a transferred tab (multiple of 16 in bytes).
- `blockSize` – size of a block (power of 2 in bytes).
- `data` – pointer to array with data.

Commonly returned values:

- A value above 0 indicates the size of the transmitted array.
- `Failed (-1)` – unsuccessful operation.
- `Timeout (-2)` – operation exceeded allowed time.
- `InvalidEndpoint (-9)` – the address is out of the range specified for pipes.
- `InvalidBlockSize (-10)` – the defined size of block of data is out of allowed range (BTPipe only)
- `UnsupportedFeature (-15)` – the size of the passed array is not correct.

The reason why `ep_dataout` and `ep_datain` ports are 32-bit is the rule of byte order for USB 3.0 OpalKelly's systems. FPGA (HDL) receives 4 bytes words per clock cycle. Though, these words are transferred from PC (API) perspective as 8-bit data type (*unsigned char*). According to this rule, the first byte sent via `okPipeIn` is transferred over the lower order bits of the data bus (7:0). The second byte is transferred over next higher order bits of the data bus (15:8) and so on. Analogically, when reading from `okPipeOut`, the lower order bits are the first byte read, and the next higher order bits are the second byte read.

3.3. First-in, first-out data structure

FIFO is a data structure that represents a queue. The meaning of this acronym is first-in, first-out. In other words, the element on the tail is always the one that has been in the set for the longest time [14].

FIFO is a valuable component in systems where data need to be arranged in a buffer, especially in a case where data incomes faster than it is processed. That will be a typical situation for proxy system described in the second chapter. Then, the implementation of FPGA system should consider such data structure. Fortunately, Xilinx system helps to achieve that by sharing a module called LogiCORE^{TM} IP FIFO Generator. The core provides an optimized solution for all FIFO configurations and delivers maximum performance (up to 500 MHz) while utilizing minimum resources [5]. It is entirely customizable so that a user can choose specific width, depth, memory type and some useful status flags. It is also very stable and compatible with Spartan-6 used in OpalKelly module.

FIFO needs memory space to store data in a queue. That is why Xilinx allows choosing three of memory types: Block RAM (BRAM), Distributed RAM and a shift register. Each of them has own low-level implementation but in general:

- *Block RAM* is used in situations when there is a need for a scalable system that will store a lot of data in a deep queue. It is the only memory type that allows non-symmetric aspect ratio implementations.
- *Distributed RAM* is suitable for the smaller amount of data but still in a deep queue.
- The *shift register* is proper for really small FIFOs.

Table 3. *FIFO's memory type characteristics.*

Memory type	**Clock supported**	**Buffering**	**Minimal resources required**	**Non-symmetric aspect ratio support**
Block RAM	Common/Independent	Medium-Large	Yes	Yes
Distributed RAM	Common/Independent	Small	No	No
Shift register	Common only	Small	No	No

The register size (for reading and writing data) of a FIFO is called width. The depth is a value that says how many words (of a size of width) can be stored in the queue. In general, both read and write ports are symmetric (has the same width). However, in case of non-symmetric aspect ratios, they can have different sizes but only in supported proportions write-to-read: 1:8, 1:4, 1:2, 1:1, 2:1, 4:1, 8:1. Also, the only memory type that supports such case is Block RAM.

There is also a rule for data ordering – when the write width is smaller than the read width (1:8, 1:4, 1:2), the most significant bits are read first (MSB to LSB). The analogical situation happens for reverse proportions. Two examples can be found in Figure 11 for 1:4 and 4:1 cases.

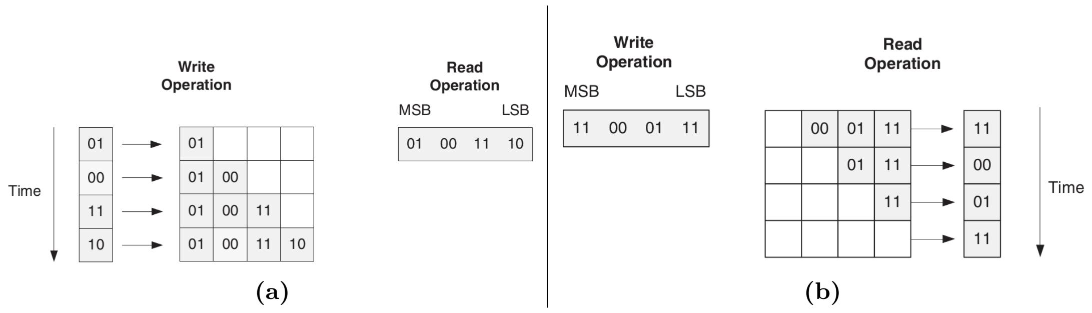


Figure 11. Data ordering in FIFO with write-to-read 1:4 (a) and 4:1 (b) proportions [5].

As previously mentioned, FIFO is fully customizable but in this project typically were used ports described in Table 4. Also, mostly Common Clock was implemented. Figure 12 shows how such system works.

Table 4. *FIFO ports used in the project with description.*

Port	Direction	Description
`clk`	input	a clock that can be synchronous to the whole design or asynchronous. Moreover, in case of non-symmetric aspect ratios, there are separated clock domains for read (`rd_clk`) and write (`wr_clk`) ports.
`rst`	input	an asynchronous reset for FIFO registers.
`din`	input	port for incoming data. From FrontPanel's perspective, the best wire/register width is 32 bit, so `din` should be [31:0].
`dout`	output	analogically to the `din`, but for outcoming data.
`wr_en`	input	port that receives signal whenever FIFO is obligated to receive data throughout the `din` port. It is synchronous to the `clk` and enables writing in the same clock cycle (only when FIFO is not full).
`rd_en`	input	port that receives signal whenever FIFO is obligated to send data throughout the `dout`. It is also synchronous to the `clk` but read operation is performed only from the next clock cycle (till the `rd_en` signal is de-asserted, or FIFO is empty).
`almost_full`	output	output signal that informs design whenever the only one more write can be performed before the FIFO is full.
`wr_ack`	output	output signal asserted when a write request (`wr_en`) succeeded during the prior clock cycle.
`valid`	output	output signal indicating that valid data is available on the `dout` bus.

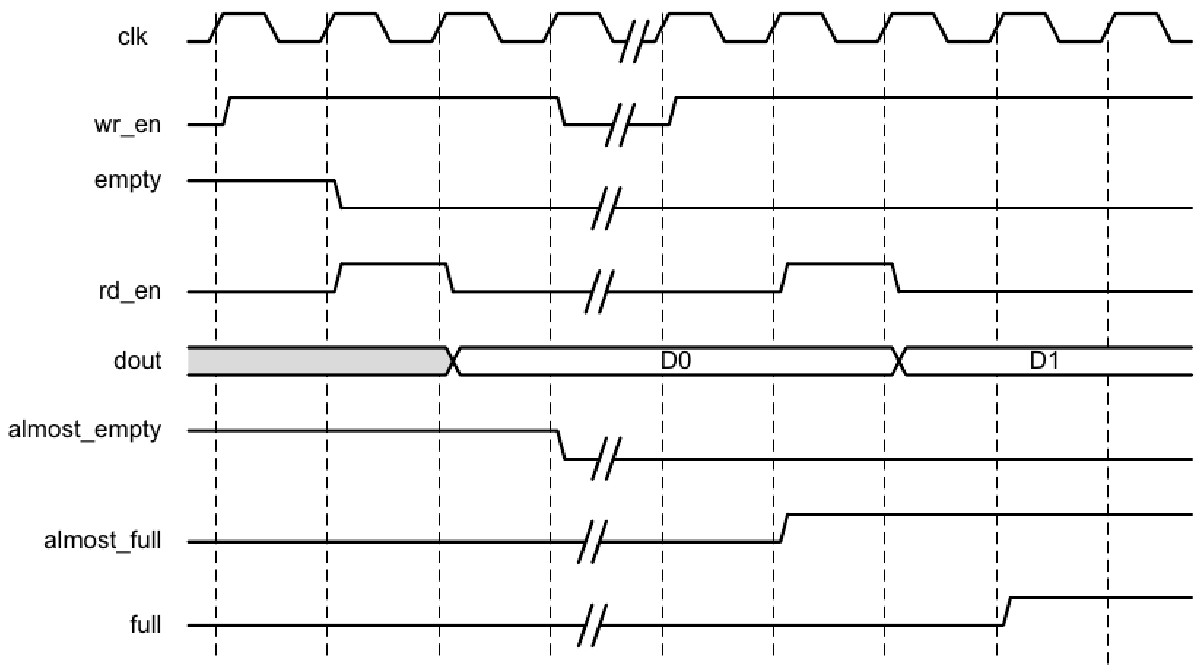


Figure 12. Timing diagram for FIFO with Common Clock [5].

4. Project description

4.1. Overview

This project is an environment for testing out the transfer capabilities of the OpalKelly's FPGA device which will serve in the future as a proxy between specialized ASICs and PC. It consists of programs both for PC and FPGA side and a script for results processing. In detail, its primary responsibility is measuring pipe write and read transfer speed (with applied FIFO in FPGA) by serving a bulk of data with specific pattern and size. The whole project is available on GitHub webpage: `https://github.com/mikgral/Master_thesis`.

The project is suitable for all popular operating systems (Windows, Linux, and macOS) and all hardware that supports USB 3.0. It is developed for PC in C++ (an object-oriented, general-purpose programming language [15]) and FPGA in Verilog (a hardware description language used during creation of electronic systems [16]). The script is written in Python (an interpreted, high-level, general-purpose programming language [17]). It enables to generate charts that visualize transfer speed and helps to understand which FIFO memory type (Block RAM, Distributed RAM or shift register) is the most efficient buffer to queue data. It also checks if specific data pattern or depth value affects bandwidth and proves that long arrays are the most efficient way to transfer information.

The test environment is prepared to act based on specified options. The most important of them with their possible parameters (and additional description) are presented below:

- **mode**:
  - *32bit* – FIFO with a symmetric aspect ratio (both its read and write ports are 32-bits width),
  - *nonsym* – FIFO with a non-symmetric aspect ratio (one of its ports is 32-bits width and the second one is 64-bits width. The *direction* option determines which port has which width),
  - *duplex* – FIFO prepared for pseudo-duplex transfer with a symmetric aspect ratio. The reason why full duplex is not available is the fact that FrontPanel does not support isochronous transfer, providing in return reliable transfer error correction [6];
- **direction**:
  - *write* – transfer from PC to FPGA,
  - *read* – transfer to PC from FPGA;
- **memory** – FIFO memory type:
  - *blockram* – refers to Block RAM,
  - *distributedram* – refers to Distributed RAM,
  - *shiftregister* – refers to a shift register;
- **depth** – depth of FIFO. Valid values are: 16 (except *nonsym write* mode), 32 (*nonsym write* mode only), 64, 256, 1024, 2048;
- **pattern**:
  - *counter_8bit* – a counter based on an 8-bit register. It starts from 0 to 255 and is increased by 8-bit 1 per word,
  - *counter_32bit* – a counter based on a 32-bit register. It starts from 0 to 8589934591 and is increased by 32-bit 1 per 4 words,

– *walking_1* – per each iteration, there is a left logical shift in 32-bit (for *32bit* mode) or 64-bit (for *nonsym* mode) register. The initial value equals 1,
– *asic* – ASIC pattern simulator based on a 64-bit register (*nonsym* mode only).

The *asic* pattern was developed for the need of the project to simulate the activity of STIM64 IC chip that is used in neuroscientific projects [18]. The width of the register was arbitrarily set to 64-bit so that it can be used only in *nonsym* mode. First four bits are reserved for an ASIC ID number, next eight bits are reserved for a channel number, next 16 are for amplitude, and the rest is for a timestamp. The amplitude takes pseudo-random values generated by LFSR[4] per each iteration according to the equation (1).

$$f(x) = x^{12} + x^6 + x^4 \tag{1}$$

where $f(1) = 123_{16}$ and x is a value of the previous state. The values of the powers have been chosen arbitrarily in such a way as to give the highest possible dispersion between successive results.

These parameters can be combined with each other to run a specific test. However, the combinations have some limitations. Table 5 presents valid options of a higher level (looking from the left column) that can be linked with lower level options.

Table 5. *Valid combinations of parameters with additional information about FIFO read and write ports width.*

Mode	Direction	Memory	Depth	Pattern	FIFO write port width [bits]	FIFO read port width [bits]
32bit	write	blockram, distributedram, shiftregister	16, 64, 256, 1024, 2048	counter_8bit, counter_32bit, walking_1	32	32
	read					
nonsym	write	blockram	32, 64, 256, 1024, 2048	counter_8bit, counter_32bit, walking_1, asic	32	64
	read	blockram	16, 64, 256, 1024, 2048		64	32
duplex	-	blockram	2048	counter_8bit, counter_32bit, walking_1	32	32

Some limitations result from software restrictions. E.g., in the case of non-symmetric aspect ratios (in *nonsym* mode), the only memory applicable for FIFO is *blockram* [5]. The *asic* pattern is available only in *nonsym* mode as initially is based on a 64-bit register. The 32 *depth* (instead of 16) in *nonsym write* mode results from mathematic limitations — the read port is two times wider than write, so more data is stored.

[4] *Linear-feedback shift register* – a shift register that uses a specified linear function of the previous state as an input bit.

4.2. FPGA side implementation

4.2.1. The core of the project's implementation on FPGA side

The project on FPGA side is divided into 5 top modules: *32bit write, 32bit read, nonsym write, nonsym read, duplex bidir.* In general, the test sequence is the same for *32bit* and *nonsym.* The difference is in a low-level implementation of FIFO (*32bit* uses symmetric, while *nonsym* uses non-symmetric aspect ratios). The most significant variation is visible between modules that are responsible for testing *write* and *read* performance.

Figure 13 presents a timing diagram for *write* module. After launching the timer with `start_timer` wire, the `pipe_in_write` is triggered and orders FIFO to collect data. Just after receiving the first word, FIFO informs that is not empty. That triggers port `fifo_read_enable`, which in turn activates an additional `checkData` module (via `enable_pattern` port). It is responsible for comparing data received from a PC with the data generated by `dataGenerator` module. If the transfer is stopped, FIFO tries to empty its content.

In the *read* implementation (its timing diagram is presented in Figure 14), the timer triggers data generation (in `dataGenerator` module) that fills FIFO. However, the `fifo_almost_full` wire controls the creation and prevents from overflowing the queue. In next step, the PC requests read operation and stops the timer at the end.

The implementation of *duplex* (its timing diagram is presented in Figure 15), is the combination of *write* and *read* solutions. The whole control of data holds the PC.

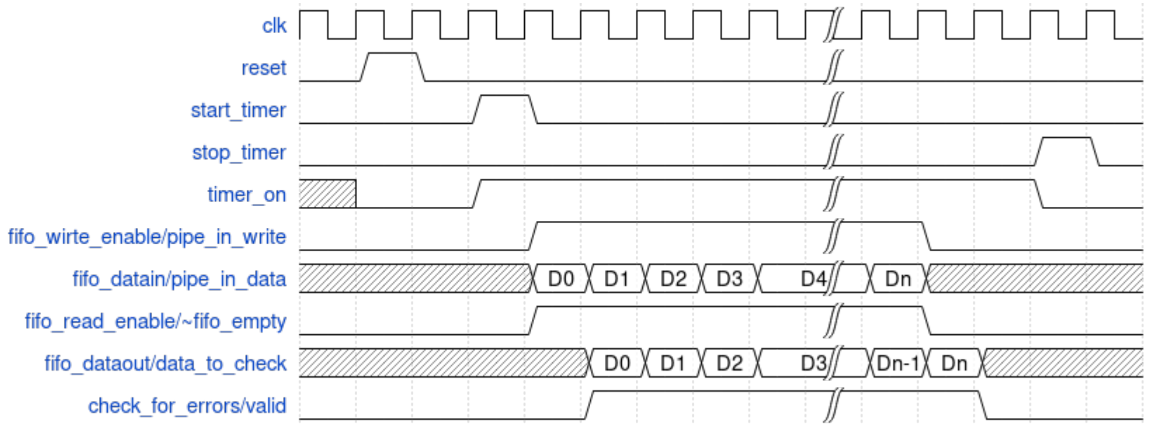


Figure 13. Timing diagram for *write* module.

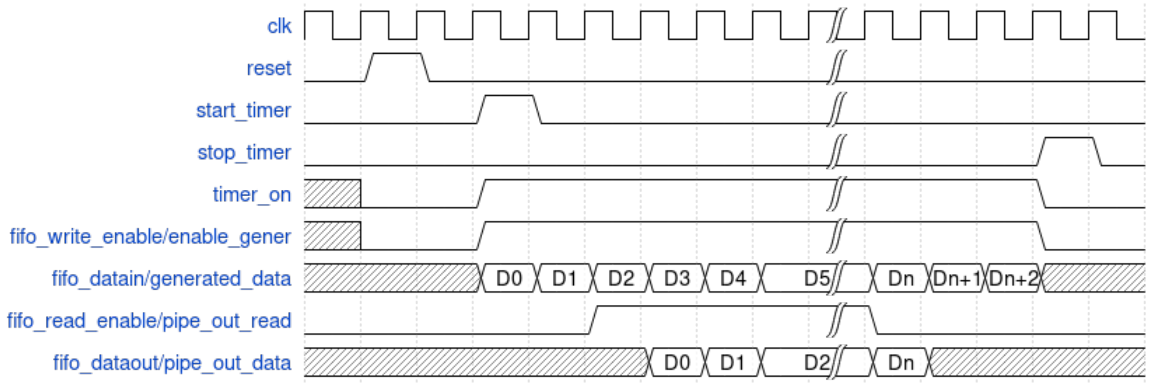

clk
reset
start_timer
stop_timer
timer_on
fifo_write_enable/enable_gener
fifo_datain/generated_data
D0
D1
D2
D3
D4
D5
Dn
Dn+1
Dn+2
fifo_read_enable/pipe_out_read
fifo_dataout/pipe_out_data
D0
D1
D2
Dn

Figure 14. Timing diagram for *read* module.

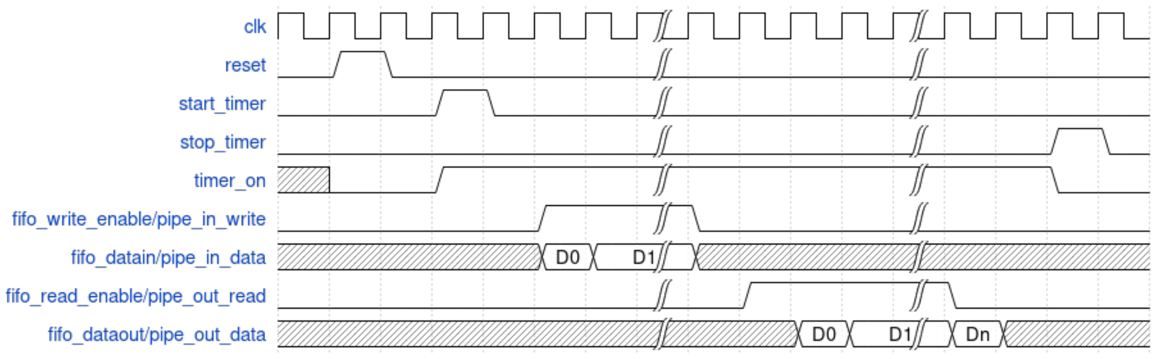

clk
reset
start_timer
stop_timer
timer_on
fifo_write_enable/pipe_in_write
fifo_datain/pipe_in_data
fifo_read_enable/pipe_out_read
fifo_dataout/pipe_out_data
D0
D1
D0
D1
Dn

Figure 15. Timing diagram for *bidir* module.

All implementations respect the convention of wires and registers naming, created for needs of the project (see Table 6).

Table 6. *The project's convention for endpoints.*

Wire/register name	Endpoint address	Description
`pattern_to_generate`	0x00	Informs `dataGenerator` module which pattern need to be generated. Valid values are from 0 to 3 respectively for `counter_8bit`, `counter_32bit`, `walking_1`, `asic`
`trigger`	0x40	Triggers specified wire. Valid values are from 0 to 3 responsible for (respectively) `reset`, `start_timer`, `stop_timer`, `reset_pattern`
`clk_counts`	0x20	Register that holds clock counts in range <0, 31> bits
`clk_counts`	0x21	Register that holds clock counts in range <32, 63> bits
`error_count`	0x22	Register that gets errors counted by `checkData` module (used only in *write* direction mode)
`pipe_in_data`	0x80	Pipe write endpoint (not used in *read*)
`pipe_out_data`	0xA0	Pipe read endpoint (not used in *write*)

The FIFO is always coupled with pipes or timer, so it will never overflow, as an emptying mechanism is automated (even with small depth). The user can collect counts of clock cycles via 0x20 and 0x21 wires (as `clk_counts` register is 64 bit). By dividing it by clock frequency (100.8 MHz), transfer time can be counted. An additional wire in *write* mode is implemented – `error_counts`, which delivers counted errors from `checkData` module.

4.2.2. Timer implementation

Timer implementation is the same for all top modules. The `start_timer` wire should be triggered if a user wants to launch the timer. Then, a 64-bit `clk_counts` register stores number of errors that are counted during all clock intervals till `stop_timer` is triggered. In *write* mode, before the beginning of sending data from PC to FPGA, some clock cycles can elapse (the delay depends on PC processor speed. See Figure 13). The `stop_timer` trigger may also delay depending on PC capabilities (see Figure 14 and 15).

The timer is implemented in `always` block, which is controlled by clock `okClk`. When `start_timer` is at a high value, a non-zero value is set for the `timer_on` register. In next clock cycles, once this register is high, the `clk_counts` register is increased by one per each cycle. The `stop_timer` wire overrides the `timer_on` register to 0. A user should use the `reset` trigger to zero `clk_counts` register. Listing 17 presents this implementation in Verilog language.

Listing 17. *The timer implementation in FPGA project.*

```
always @(posedge okClk) begin
  if (reset) begin
    clk_counts <= 64'd0;
    timer_on   <= 0;
  end

  if (start_timer) begin
    timer_on   <= 1;
    clk_counts <= clk_counts + 1;
  end

  if (timer_on) begin
    clk_counts <= clk_counts + 1;
  end

  if (stop_timer) begin
    timer_on   <= 0;
  end
end
```

4.2.3. Description of module responsible for generating data

The `dataGenerator` module is a standalone part of the project. It is responsible for filling register (that handles data) with a specified pattern. Its implementation is the same for both *read* and *write* direction mode. Though, it differs in case of *32bit* and *nonsym* mode as the register size is respectively 32- and 64-bit width. The module's instance is presented in Listing 18.

Listing 18. *Instance of the* `dataGenerator` *module.*

```
dataGenerator dataGenCheck(
  .clk(clk),
  .enable_gener(enable_gener),
  .pattern(pattern),
  .reset(reset_pattern),
  .dataout(dataout),
  .dataout_available(dataout_available)
);
```

It is important to know each endpoint's role to implement the module properly in a project:

- `clk` – input for a clock that synchronizes the project.
- `enable_gener` – input 32-bit wire synchronized with a clock that triggers generation of pattern.
- `pattern` – input wire that handles the request of specified data pattern generation. Valid values are described below (with their Verilog's binary format equivalent in brackets):
  - 0 (3'b000) – *counter_8ibit.*
  - 1 (3'b001) – *counter_32bit.*
  - 2 (3'b010) – *walking_1.*
  - 3 (3'b011) – *asic* (valid only for *nonsym* mode).
- `reset` – input wire that triggers reset of all registers used for generating data.
- `dataout` – output register that returns data filled with specified pattern. Its width depends on which mode (32-bit for *32bit* and 64-bit for *nonsym*) the project works.
- `dataout_available` – output register that gets high state whenever the pattern generation is done in current clock cycle (mostly used in *read* direction mode to inform FIFO that data is ready to receive).

There are some additional helper components in the module showed in Listing 19, e.g., registers that are used in generating *asic* data pattern. To the extra four wires are assigned maximal values that the registers can reach.

Listing 19. *Helper registers in* `dataGenerator` *module that are used in generating asic data pattern.*

```
reg [3:0] id;
reg [7:0] channel;
reg [15:0] amplitude;
reg [35:0] timestamp;

wire [3:0] max_id;
wire [7:0] max_channel;
wire [35:0] max_timestamp;

assign max_id = 4'hF;
assign max_channel = 8'hFF;
assign max_timestamp = 36'hFFFFFFFFF;
```

The generation is controlled by `reset` and `enable_gener` triggers synchronized with the clock in `always` block. When the `reset` is triggered, the `dataout` register gets a default value, depending on the requested pattern type in `pattern` endpoint. Listing 20 presents an example of this action from *32bit* mode implementation.

Listing 20. *The reset operation in* `dataGenerator` *module in 32bit mode implementation.*

```
if (reset) begin
  dataout_available = 0;
  case (pattern)
    3’b000: begin
    dataout[7:0] = -8’h4;;
    dataout[15:8] = -8’h3;
    dataout[23:16] = -8’h2;
    dataout[31:24] = -8’h1;
    end
    3’b001: dataout = -32’b1;
    3’b010: dataout = 32’b10000000000000000000000000000000;
  endcase
end
```

When `enable_gener` is at a high state, the generation is launched, and `dataout_available` gets a value of 1. At the same time, when `pattern` wire is set to 0 value, each byte (8 bits) of 32- or 64-bit `dataout` register is increased by 1 per clock cycle. That is how *counter_8bit* pattern is formed. When the wire is set to 1, the register is increased by a constant wire of value of 1 (of the same width), creating the *counter_32bit* pattern. When the wire is set to 2, a concatenation happens of the whole register (without the last bit) with its last bit. Thus, the generation of the *walking_1* pattern is performed. Listing 21 shows an implementation in the *32bit* mode of generation of data patterns described above. When the `pattern` wire is set to 3 value, the *asic* pattern generation is performed. The `dataout` register is the result of a concatenation of helper registers showed on Listing 19. Per each clock cycle, `timestamp` and `channel` registers are increased by 1. If the `channel` register reaches its max value, it is zeroed, and the value of `id` register is increased by one. Meanwhile, the concatenation happens in `amplitude` register that complies with the LFSR operation described in equation (1). Listing 22 shows the specific implementation in *nonsym* mode.

Listing 21. *The part of* `dataGenerator` *module responsible for generating counter_8bit, counter_32bit, walking_1 patterns in 32bit mode implementation.*

```
if (enable_gener) begin
  dataout_available = 1;
  case (pattern)
    3’b000: begin
      dataout[7:0] = dataout[7:0] + 4’b1000;
      dataout[15:8] = dataout[15:8] + 4’b1000;
      dataout[23:16] = dataout[23:16] + 4’b1000;
      dataout[31:24] = dataout[31:24] + 4’b1000;
      dataout[39:32] = dataout[39:32] + 4’b1000;
      dataout[47:40] = dataout[47:40] + 4’b1000;
      dataout[55:48] = dataout[55:48] + 4’b1000;
      dataout[63:56] = dataout[63:56] + 4’b1000;
    end
    3’b001: dataout = dataout + 64’b1;
3’b010: dataout = {dataout[62:0], dataout[63]};
...
```

Listing 22. *The part of dataGenerator module responsible for generating the asic pattern in nonsym mode implementation.*

```
    ...
    3'b011: begin
      dataout = {timestamp[35:0], amplitude[15:0], channel[7:0], id[3:0]};
      amplitude = {amplitude[14:0],
                   amplitude[11] ^ amplitude[5] ^ amplitude[3]};
      if (timestamp == max_timestamp) begin
        timestamp = timestamp + 36'b1;
      end
      if (channel == max_channel) begin
        channel = 8'b1;
        id = id + 4'b1;
      end else begin
        channel = channel + 8'b1;
      end
      if (id == max_id) begin
        id = 4'b1;
      end
    end
  endcase
end else begin
  dataout_available = 0;
end
```

4.2.4. Description of module responsible for checking data

Data check happens only in *write* direction mode. The module responsible for this is `checkData`. Its primary goal is to verify whether incoming data from PC is compatible with the pattern. It uses `dataGenerator` module to produce a correct sample and compares it with the one from source. The module's instance is presented in Listing 23.

Listing 23. *Instance of the* `checkData` *module.*

```
checkData checkDataFromPipeIn (
  .data_to_check(data_to_check),
  .pattern(pattern),
  .clk(clk),
  .reset_err_counter(reset_err_counter),
  .reset_pattern(reset_pattern),
  .check_for_errors(check_for_errors),
  .enable_pattern(enable_pattern),
  .error_count(error_count)
)
```

It is essential to know each endpoint's role to implement the module accurately in a project:

- `data_to_check` – input wire that holds data to check. Its width depends on which mode ([31:0] in *32bit* and [63:0] in *nonsym*) the project works.
- `pattern` – input wire that handles the request of specified data pattern generation. Valid values are the same that the ones described in Listing 18 in subsection 4.2.3.
- `clk` – input for a clock that synchronizes the project.
- `reset_err_counter` – input wire that zeroes `error_count` register.
- `reset_pattern` – input wire that is passed to the `reset` endpoint of a `dataGenerator` module.
- `check_for_errors` – input wire that triggers error checking.
- `enable_pattern` – input wire that is passed to the `enable_gener` endpoint of the `dataGenerator` module.
- `error_count` – output 32-bit register that returns from the module counted errors.

The module's `always` block controls two possible states. The first one is triggered by `reset_err_counter` where `error_count` is zeroed. The second one is triggered by `check_for_errors`. If the incoming data (`data_to_check`) is not equal with the data (stored in additional wire `correct_data`) generated by the `dataGenerator` module, the `error_count` register is increased by 1.

Listing 24. *Implementation of* `always` *block in the* `checkData` *module in 32bit mode. A width of an additional wire* `correct_data` *is 64-bit in nonsym mode.*

```
wire [31:0] correct_data;
always @(posedge clk) begin
  if (reset_err_counter) begin
    error_count <= 32'b0;
  end
  if (check_for_errors) begin
    if(data_to_check != correct_data) begin
      error_count <= error_count + 1;
    end
  end
end
```

4.2.5. LEDs application in the project

The OpalKelly module has 8 LEDs that can be used in any project. In this case, they play a role as states indicators of some wires, which can be useful during debugging or troubleshooting. Listing 25 presents assignments of relevant wires to appropriate LEDs. Assignment for led[6] was intentionally omitted as it varies depending on mode:

- `~valid` is assigned in *32bit write* mode.
- `~fifo_empty` is assigned in *nonsym write* mode.
- `1'b1` (always off) is assigned in *32bit read*, *nonsym read*, and *duplex bidir* modes.

Listing 25. *Universal LED array assignment to specific wires.*

```
assign led[0] = ~reset;
assign led[1] = ~start_timer;
assign led[2] = ~timer_on;
assign led[3] = ~stop_timer;
assign led[4] = ~fifo_write_enable;
assign led[5] = ~fifo_read_enable;
assign led[7] = ~1'b1;
```

4.2.6. FIFO instance in the project

An instance of FIFO (which was described in detail in section 3.3) is placed in each project's top module. Listing 26 shows it as a universal pattern which does not take into account several ports that additionally appear depending on the mode:

- `.clk(okClk)` – *32bit write*, *32bit read*, *duplex bidir.*
- `.wr_clk(okClk)` and `.rd_clk(okClk)` – *nonsym write*, *nonsym read.*
- `.almost_full(fifo_almost_full)` – *32bit read*, *nonsym read.*
- `.valid(valid)` – *32bit write*, *nonsym write.*
- `.wr_ack(wr_ack)` – *32bit write.*
- `.empty(fifo_empty)` – *nonsym write.*

Listing 26. *FIFO instance that is common for all top modules. Square brackets should be replaced with relevant words.*

```
FIFO_[mode] fifoFor[direction]Test (
  .rst(reset),
  .din(fifo_datain),
  .dout(fifo_dataout),
  .wr_en(fifo_write_enable),
  .rd_en(fifo_read_enable),
  [other ports that appear depending on used mode]
);
```

4.3. PC side implementation

4.3.1. Overview

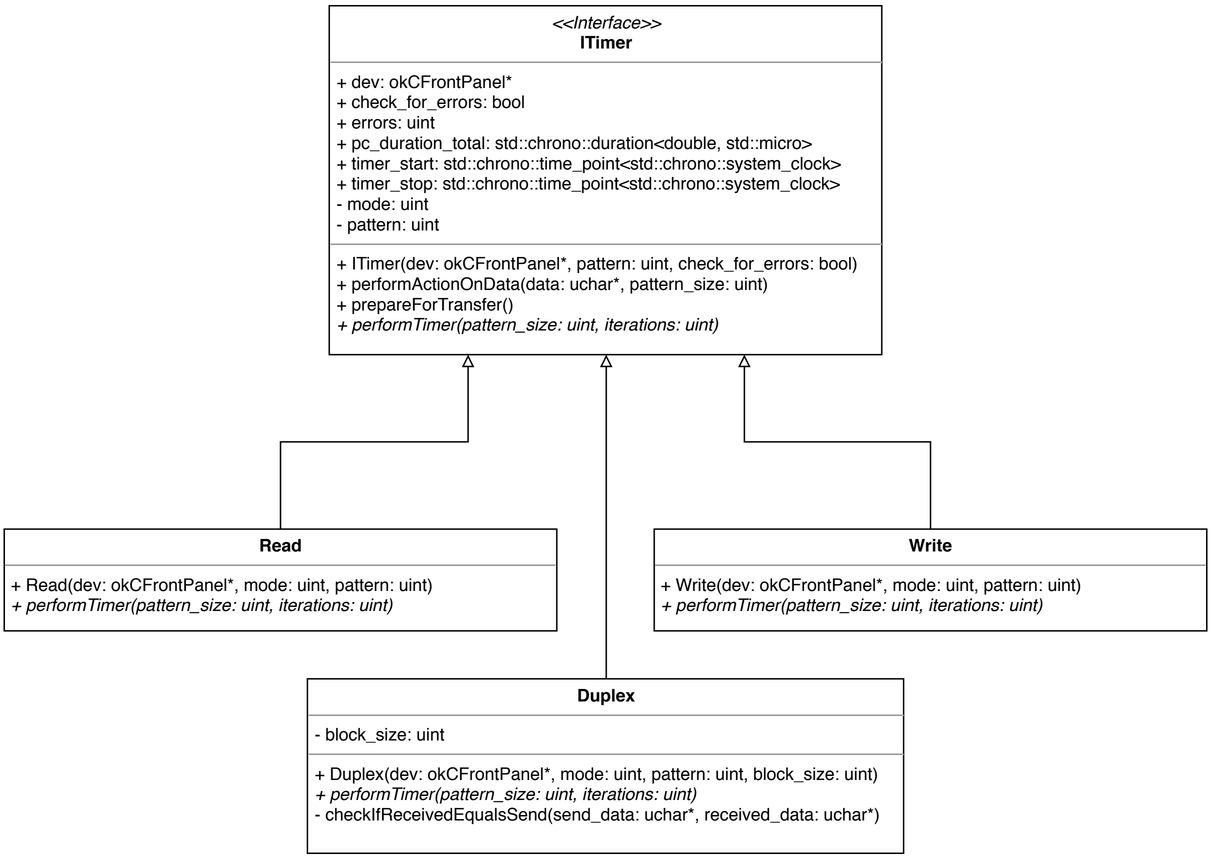
The test environment on the PC side is fully customizable and can execute on any platform. It is ready to be used in future projects that will include the OpalKelly module. The application strictly for the test is an object-oriented program written in C++11 [19]. It uses the standard library (std) and some other cross-platform libraries:

- *GLOG* – an open source logging library that supports tracing program execution [20].
- *libconfig* – an open source project for reading config files by a program written in C++ [21].
- *FrontPanelDLL* – library developed by OpalKelly, available only for customers of their products. This library contains a header file for API purposes (*okFrontPanelDLL.h*) and dynamically linked libraries for Windows, macOS, and Linux.

The application can perform many tests on the OpalKelly device, considering specified options. Results are saved in a separated file. Moreover, the application has own logging systems, so the program execution can be investigated in order to understand dependencies between implemented classes. The program can be compiled in a standard release mode or debug (where additional logs are added to find bugs during a test). The application comes with ready CMake file to facilitate compilation process.

The application contains own implementation of 7 classes, one interface, and one namespace. All of their definitions are stored in a header file called *performance.h*, which can be used as the project's API in future projects. Names and descriptions of these classes (along with the name of the file in which they are developed) are described below:

- `Configurations` (*config.cpp*) – a class responsible for parsing test options from a particular config file. It uses *libconfig* methods as the configuration is written in unique JSON-like format. A path to the file is passed as an argument to the class's constructor. Then, parsing is performed to public variables which will be used by other classes. Some default variables are hardcoded in that class, so a user should check them in a header file to adequately prepare the config file.
- `Results` (*results.cpp*) – a class used to store results from the test and save them to a result file.
- `TransferController` (*transfer.cpp*) – a class that controls the test based on specified configuration. It uses many loops to reach the test's goals.
- `DataGenerator` (*datagen.cpp*) – a class that fills arrays with specified pattern or checks if received data is following the pattern.
- `ITimer` (*timer.cpp*) – an abstract class that holds data counted by timers implemented in its child classes. It has one pure virtual method - `performTimer`. Figure 16 presents its class diagram. The child classes are:
  - `Read` – a class used in *read* mode.
  - `Write` – a class used in *write* mode.
  - `Duplex` – a class used in *duplex* mode. It has one additional method (`checkIfReceivedEqualsSend`) that compares sent and received data.



<<Interface>>
ITimer
+ dev: okCFrontPanel*
+ check_for_errors: bool
+ errors: uint
+ pc_duration_total: std::chrono::duration<double, std::micro>
+ timer_start: std::chrono::time_point<std::chrono::system_clock>
+ timer_stop: std::chrono::time_point<std::chrono::system_clock>
- mode: uint
- pattern: uint
+ ITimer(dev: okCFrontPanel*, pattern: uint, check_for_errors: bool)
+ performActionOnData(data: uchar*, pattern_size: uint)
+ prepareForTransfer()
+ performTimer(pattern_size: uint, iterations: uint)
Read
+ Read(dev: okCFrontPanel*, mode: uint, pattern: uint)
+ performTimer(pattern_size: uint, iterations: uint)
Write
+ Write(dev: okCFrontPanel*, mode: uint, pattern: uint)
+ performTimer(pattern_size: uint, iterations: uint)
Duplex
- block_size: uint
+ Duplex(dev: okCFrontPanel*, mode: uint, pattern: uint, block_size: uint)
+ performTimer(pattern_size: uint, iterations: uint)
- checkIfReceivedEqualsSend(send_data: uchar*, received_data: uchar*)

Figure 16. Class diagram of *Timer* interface.

The last element of the project's API is `okdev` namespace (developed in *okdev.cpp* file) that has three helper functions for `okCFrontPanel` objects:

- `void openDevice(okCFrontPanel *dev)` – if an OpalKelly's device is connected with PC, then this method opens a communication port.
- `void checkIfOpen(okCFrontPanel *dev)` – checks if the device is still connected. If not, it closes the program with a fatal error. It may be useful before performing some critical operation on the device.
- `void setupFPGA(okCFrontPanel *dev, const string &path_to_bitfile)` – loads a bitfile to the FPGA.

4.3.2. The program's flow description

When starting the program, a path to a config file should be passed as an argument (otherwise, the program will use by default: "../performance.cfg"). `Configurations` class parses determined options and stores them. Then, the object of this class is passed (with `okCFrontPanel` object responsible for device management) to `TransferController` class that handles the test. There, many loops iterate through mode and direction of the transfer, depth, and memory of FIFO, pattern and its size. At one runtime, many mixed tests can be performed. Based on these parameters, `TransferController` class uses a class from *Timer* interface: `Read`, `Write` or `Duplex`. Each of them launches timer and sends (or receives) data generated by `DataGenerator` class, based on specified pattern type. After measurements, all results are saved to file in CVS format by `Results` class. This flow is presented graphically in Figure 17.

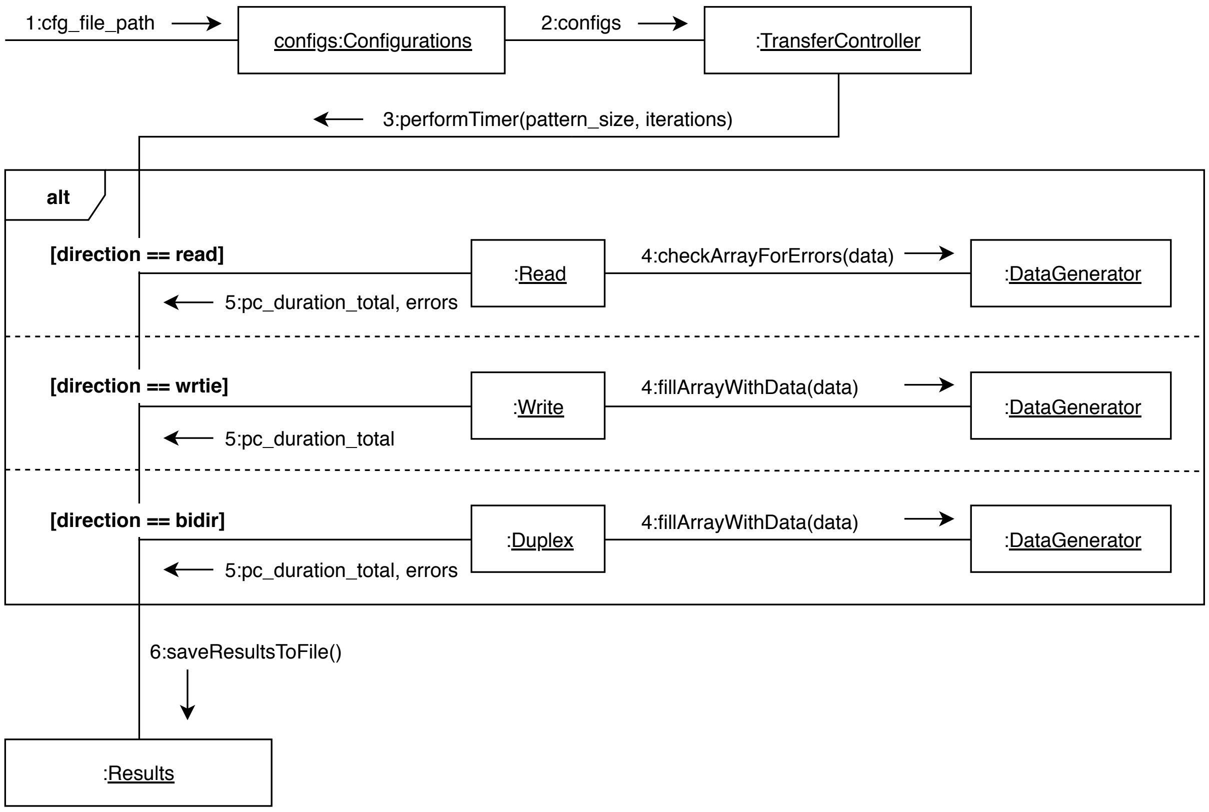


Figure 17. Collaboration diagram of the PC program execution.

4.3.3. Detailed description of the timer

There are three timer versions in the application as there are three possible direction modes: *read, write and bidir (duplex)*. The principal idea is to start a chronometer, perform transfer operation, stop the measurement and count elapsed time. Technically, the application uses a function from the standard library called `chrono::system_clock::now()` to save time value in both start and stop timer variables. The result is saved in `pc_duration_total` variable of a `chrono::duration<double, std::micro>` type, so *count()* operation on it returns a time result in microseconds.

According to *ITimer* interface, the *performTimer* method takes two arguments: pattern size and a number of iterations of the transfer operation. First, a method called `prepareForTransfer()` is executed where time result and error variables are zeroed. Also, information about the current pattern type is sent to the FPGA with reset trigger. Second, memory is allocated for an array that will hold data. Then, depending on the direction mode, there are three possible scenarios. If the direction is set to *read*, a loop is executed where the read operation on a pipe is performed per each iteration, an elapsed time is counted, and data is checked for errors (Listing 27). Else, if the direction is set to *write*, the data array is filled with the pattern. The data is sent to the FPGA and the transfer time is measured (Listing 28). Else, if the project works in *duplex* mode, the data array is filled with the pattern, and memory is allocated for an array that will hold received data. During time measurement, write and read operations are done along with comparing if the sent and received arrays are the same (Listing 29).

Listing 27. *The implementation of performTimer in* `Read` *class.*

```
prepareForTransfer();
unsigned char *data = new unsigned char[pattern_size];
for (unsigned int i=0; i<iterations; i++)
{
  DLOG(INFO) << "Current iteration: " << i;
  dev->ActivateTriggerIn(TRIGGER, RESET_PATTERN);
  timer_start = std::chrono::system_clock::now();
  dev->ActivateTriggerIn(TRIGGER, START_TIMER);

  dev->ReadFromPipeOut(PIPE_OUT, pattern_size, data);

  dev->ActivateTriggerIn(TRIGGER, STOP_TIMER);
  timer_stop = std::chrono::system_clock::now();

  pc_duration_total += (timer_stop - timer_start);
  performActionOnData(data, pattern_size);
}
delete[] data;
```

Listing 28. *The implementation of performTimer in* `Write` *class.*

```
prepareForTransfer();
unsigned char *data = new unsigned char[pattern_size];
performActionOnData(data, pattern_size);
timer_start = std::chrono::system_clock::now();
dev->ActivateTriggerIn(TRIGGER, START_TIMER);
for (unsigned int i=0; i<iterations; i++)
{
  dev->ActivateTriggerIn(TRIGGER, RESET_PATTERN);
  dev->WriteToPipeIn(PIPE_IN, pattern_size, data);
}
dev->ActivateTriggerIn(TRIGGER, STOP_TIMER);
timer_stop = std::chrono::system_clock::now();
pc_duration_total = timer_stop - timer_start;
delete[] data;
```

Listing 29. *The implementation of performTimer in* `Duplex` *class.*

```
prepareForTransfer();
unsigned char *data = new unsigned char[pattern_size];
performActionOnData(data, pattern_size);
unsigned char *received_data = new unsigned char[block_size];
unsigned char *send_data;
for (unsigned int i=0; i<iterations; i++)

  for (unsigned int j = 0; j < pattern_size; j+=block_size)
  {
    send_data = data + j;

    timer_start = std::chrono::system_clock::now();
    dev->ActivateTriggerIn(TRIGGER, START_TIMER);

    dev->WriteToPipeIn(PIPE_IN, block_size, send_data);
    dev->ReadFromPipeOut(PIPE_OUT, block_size, received_data);

    dev->ActivateTriggerIn(TRIGGER, STOP_TIMER);
    timer_stop = std::chrono::system_clock::now();
    pc_duration_total += (timer_stop - timer_start);

    // Error checking
    checkIfReceivedEqualsSend(send_data, received_data);
  }
}

delete[] received_data;
delete[] data;
```

4.3.4. Description of pattern generators

Generators for the four patterns are implemented in `DataGenerator` class. Its constructor takes such parameters like mode, pattern type, and size. It has two public methods:

- `void fillArrayWithData(unsigned char *data)` – fills the data array (passed as a pointer) with the pattern.
- `unsigned int checkArrayForErrors(unsigned char *data)` – checks whether data array equals expected pattern.

After detection of an expected pattern to generate and determining some helper parameters (like a maximal value of register size), the generation starts. A helper method is used called `performActionOnGeneratedData` which fills the data or check it with the pattern, depending on transfer direction. The generation algorithms are the same as described in section 4.2.3. Listings 30, 31 and 32 show implementation of (respectively) *counter_8bit, counter_32bit, walking_1* patterns. Listing 33 shows declaration of helper variables that are used in *asic* pattern generation. Listing 34 shows shift operations on an `asic_data` array (that will be finally used to fill data pattern) to place id, channel, amplitude and timestamp variables on proper bit positions. Listing 35 shows the way of generating data for id, channel, and amplitude variables.

Listing 30. *The implementation of counter_8bit pattern in* `DataGenerator::counter8Bit()` *method.*

```
uint64_t iter = 0;
for (unsigned int i = 0; i < pattern_size; i++)
{
  unsigned char data_char = static_cast<unsigned char>(iter);
  performActionOnGeneratedData(data_char, i);
  if (iter > max_register_size) iter = 0;
  else ++iter;
}
```

Listing 31. *The implementation of counter_32bit pattern in* `DataGenerator::counter32Bit()` *method.*

```
uint64_t iter = 0;
for (unsigned int i = 0; i < pattern_size; i+=register_size)
{
  for (unsigned int j = 0; j < register_size; j++)
  {
    unsigned char data_char =
                  static_cast<unsigned char>((iter >> j*8) & 0xFF);
    performActionOnGeneratedData(data_char, i+j);
  }
  if (iter > max_register_size) iter = 0;
  else ++iter;
}
```

Listing 32. *The implementation of walking_1 pattern in* `DataGenerator::walking1()` *method.*

```
uint64_t iter = 1;
uint64_t last_possible_value = max_register_size / 2 + 1;
for (unsigned int i=0; i < pattern_size; i+=register_size)
{
  for (unsigned int j=0; j < register_size; j++)
  {
    unsigned char data_char =
                  static_cast<unsigned char>((iter >> j*8) & 0xFF);
    performActionOnGeneratedData(data_char, i+j);
  }

  if (iter == last_possible_value) iter = 1;
  else iter *= 2;
}
```

Listing 33. *Declaration of variables used in the generation of asic pattern in* `DataGenerator::asic()` *method. Comments inform about their bit size.*

```
uint16_t amplitude = 0x123; // 16b
uint64_t timestamp = 0; // 36b

const uint8_t max_id = 15; // 4b
const uint8_t max_channel = 255; // 8b
const uint16_t max_amplitude = 65535; // 16b
const uint64_t max_timestamp = 68719476735; // 36b

unsigned char asic_data[8];

uint8_t id = 1;
uint8_t channel = 1;
unsigned int i_data = 0;
```

Listing 34. *Filling asic pattern with id, channel, amplitude and timestamp data in* `while (i_data < pattern_size)` *loop in* `DataGenerator::asic()` *method. Comments inform about steps taken to fill* `asic_data` *array correctly.*

```
timestamp = i_data + 1;

// The first byte is filled with the 4-bit id and
// a half of the 8-bit channel
asic_data[0] = static_cast<unsigned char>(id);
asic_data[0] += static_cast<unsigned char>(channel << 4);

// The second byte is filled with the second half of channel and
// a quarter of the 16-bit amplitude
asic_data[1] = static_cast<unsigned char>(channel >> 4);
asic_data[1] += static_cast<unsigned char>(amplitude << 4);

// The third byte is filled with the next series of bits from amplitude
asic_data[2] = static_cast<unsigned char>(amplitude >> 4);

// The fourth byte is filled with the rest bits of amplitude and
// a 1/9 part of the 36-bit timestamp
asic_data[3] = static_cast<unsigned char>(amplitude >> 12);
asic_data[3] += static_cast<unsigned char>(timestamp << 4);

// Rest of bytes are filled with the rest of timestamp
asic_data[4] = static_cast<unsigned char>(timestamp >> 4);
asic_data[5] = static_cast<unsigned char>(timestamp >> 12);
asic_data[6] = static_cast<unsigned char>(timestamp >> 20);
asic_data[7] = static_cast<unsigned char>(timestamp >> 28);
```

Listing 35. *Id, channel and amplitude generation in* `while (i_data < pattern_size)` *loop in* `DataGenerator::asic()` *method.*

```
amplitude = (amplitude << 1) |
            ((((amplitude >> 11)^(amplitude >> 5)^(amplitude >> 3)) & 1));

for (std::size_t i = 0; i < sizeof(asic_data); i++)
{
  if (check_for_errors && i >= 3) break;
  performActionOnGeneratedData(asic_data[i], i_data + i);
}
i_data += 8;

if (channel == max_channel)
{
  channel = 1;
  ++id;
}
else
  ++channel;

if (id == max_id)
  id = 1;
```

4.4. Description of script for results processing

One performance test can produce large result file with plenty of data. Results are in CSV format so they can be passed to a program that will visualize and analyze them. Nevertheless, for fast data processing, a script was developed which is an integral part of the project. Not only it allows to visualize results by generating a series of charts, but also it produces tables that show a comparison of data in different cases so that it facilitates the analysis. The script is written in Python3 and uses matplotlib library (a Python 2D plotting library [22]). It consists of two files: *run_analysis.py* and *cfg.py*. The first one has the main implementation. The second one has global attributes that affect script execution. Comments above these variables inform what they do and how to use them in order to change script options, but a description of the most important of them can be found below:

- CSV_FILE – defines a path to file (in CSV format) that contains transfer results from the test.
- SEPARATOR – defines separator used in results file (as the CSV file does not necessarily need to be separated by a comma).
- STATISTICAL_ITERATIONS – defines the number of statistical iterations (should be the same as the one defined in *performance.cfg* file).
- CHECK_FOR_ERRORS – determines if the script should check error occurrence during transfer (default value is *False*). The result is returned on stdout.
- TARGET_SPEED – specifies the target speed that will be collocated with pattern sizes on charts. The default value is 'AV' (average speed of average speeds from PC and FPGA) but also available are 'PC' (average speed from PC) and 'FPGA' (average speed from FPGA).
- PARAMETERS_SEPARATED – determines if each combination of parameters should be presented on a separated chart. The default answer is False, so aggregated parameters are indicated in a legend on a figure.

The main implementation is composed of 4 classes:

- `ResultsParser` – transforms data from CSV file to a list of dictionaries that will be easier in use by rest of classes.
- `CounterParams` – uses methods from *statistics* and *math* libraries to count such values like average transfer speed.
- `ResultsHandler` – handles result data whereby another list of dictionaries is created that can be managed in any way, e.g., to create charts or tables.
- `Figure` – responsible for generating charts with *matplotlib* library.

4.5. Project repository structure and description of the config file

The project is divided into PC, FPGA and script section. In folder '*src*' is a source code for PC application. In '*HDL*' folder are top modules (with additional components) for the board. In folder '*script*' is a source code for the script that analyses results with the result file from the test described in section 5. Optionally, *CMakeList.txt* file can be used with *cmake* program that controls the build process [23].

Compilation of the program will be successful only if additional libraries like *GLOG*, *libconfig* and *FrontPanelDLL* are installed on a computer or, at least, their sources are placed in the project's folder. The program can be compiled in a *release* or *debug* mode (in the second case, additional traces are implemented).

The config file (*performance.cfg*) specifies how exactly the device should be tested. It has attributes that are divided into two groups:

- *output* – specifies how result file should look like:
  - *headers* – names output arguments in the first row of results file. They can be modified, but it is not generally recommended.
  - *resultfile_name* – the name of the result file. It is recommended to use *.csv* extension.
  - *results_path* – a path where results need to be stored. Warning: the folder specified in this path must exist! Otherwise, the program will terminate with a fatal error.
  - *result_sep* – a separator that separates results in rows (comma or other).
- *params* – test options:
  - *mode* – three modes are available to choose: *32bit*, *nonsym*, and *duplex*.
  - *direction* – two directions are available: *read* (from FPGA to PC) or *write* (from PC to FPGA). Valid only for *32bit* or *nonsym* mode.
  - *memory* – determines which FIFO memory need to be used: Block RAM (*blockram*), Distributed RAM (*distributedram*) or shift register (*shiftregister*).
  - *depth* – FIFO memory depth. Valid options are: 16 (except *nonsym* write mode), 32 (*nonsym* write mode only), 64, 256, 1024, 2048.
  - *pattern_size* – list of data sizes. Generally, size must be in a closed range from 16 to 1073741824 (1 GB). If no argument passed, then default values will be generated according to the pattern: $a_n = 16 \cdot 2^{n-1}$, where $n \in \mathbb{N} \bigwedge n \in < 0, 27 >$
  - *block_size_duplex* – specifies a size of data blocks. The block size must be divisible by data size and be in the range: <16, 64, 256, 1024>. Valid only for *duplex* mode.
  - *pattern_size_duplex* – sizes of data patterns for *duplex* mode. Requirements are the same as in *pattern_size* except that range begins from 1024.
  - *pattern* – data pattern to generate.
  - *statistic_iter* – iterations that improve statistical error of speed transfer.
  - *iterations* – numbers of transfers for a single mode. It generally decreases statistical error for short patterns.

To run the script for data analysis, the user should execute *script/run_analysis.py* in Python3 environment. Also, a config file *script/cfg.py* can be modified to custom analysis (see section 4.4).

5. Testing the transfer

5.1. Introduction to test

The experiment was launched on PC that had Intel® Core^{TM} i7-8700 CPU @ 3.20GHz with integrated USB 3.0 Intel Corporation Device controller. It had 16 GB RAM DDR4 and ran on Linux Debian 9.4. Redundant processes and daemons were off to reduce actions performed in OS that may interfere with the transfer. All test options were enabled in the test's configuration while respecting rules of combination described in Table 5. Both *statistic_iter* and *iterations* values were set to 10 to meet a representative statistical sample. The measurement errors of the speed results were counted using standard deviation formula according to the equation (2).

$$s = \sqrt{\frac{1}{10-1}\sum_{i=1}^{10}(speed_i - \overline{speed})^2} \tag{2}$$

The transfer results shown in figures (in next subsections) are sums of average speeds counted on PC and FPGA, divided by two (mean value). Their statistical errors were computed using error propagation formula presented in the equation (3) and were included in the charts in the form of vertical lines.

$$\begin{aligned} u_c(av\_speed) &= \sqrt{\left[\frac{\partial av\_speed}{\partial av\_fpga\_speed} u\left(av\_fpga\_speed\right)\right]^2 + \left[\frac{\partial av\_speed}{\partial av\_pc\_speed} u\left(av\_pc\_speed\right)\right]^2} \\ &= \sqrt{\left[\frac{u(av\_fpga\_speed)}{2}\right]^2 + \left[\frac{u(av\_pc\_speed)}{2}\right]^2} \end{aligned} \tag{3}$$

The primary purpose of the test is to examine whether, in *nonsym* and *32bit* mode, FIFO memory type, depth value of FIFO or data pattern type (sent via pipes) has an impact on transfer speed. Therefore, three levels of parameters were introduced in each part of the test. Description of these levels can be found at the beginning of subsections. Also, each part of the experiment was performed in four configurations considering transfer direction and mode: *nonsym read, nonsym write, 32bit read, 32bit write.*

In case of *duplex* mode, instead of FIFO memory type (as there is only one valid for this test – *blockram*) there is another parameter – block size. The primary purpose of this test is to find out how efficiently divide bulk of data to imitate full duplex transfer. In the future implementation of OpalKelly module, it will be essential to perform many read and write operations as much as possible at the same time, so the best transfer manner, in this case, should be developed.

Graphical presentation of results and their analysis were possible thanks to the script described in section 4.4. Charts and tables that were created using it are included in the Appendix. As many results were similar to each other, only one example per subtest was moved to proper subsection (unless some anomalies appeared. Their examples were also moved from the Appendix).

5.2. Investigation of data pattern type impact on transfer speed

The first level parameter in this subtest is FIFO memory type. The second one is FIFO depth value. The third parameter is data pattern type. This subsection examines the impact on bandwidth considering a type of data sent in bulks.

In case of *nonsym* mode, no anomalies were detected. All *read* results (from 16 to 2048 depth value) are very similar to those presented in Figure 18. In case of *write* direction – all results (from 32 to 2048 depth value) are similar to those in Figure 19.

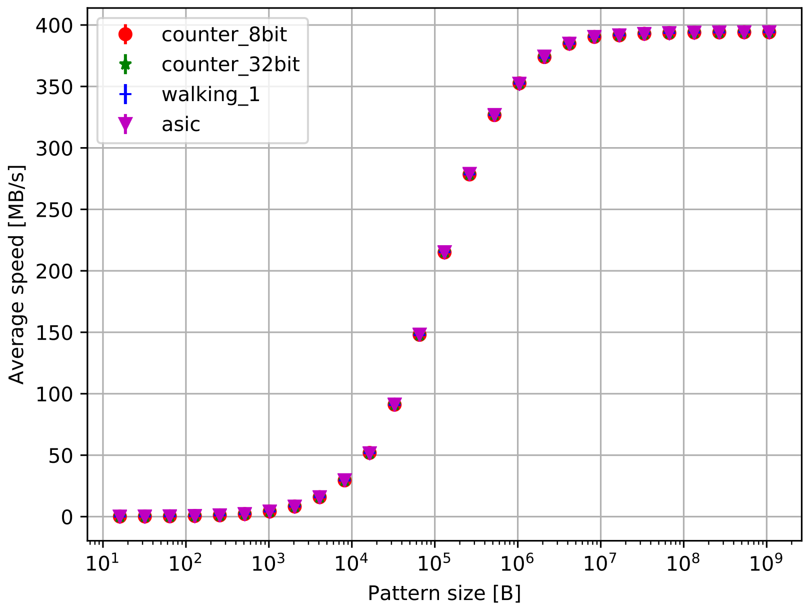


Figure 18. Example chart that presents speed results from the test of data pattern impact investigation in *nonsym read* mode without anomalies (*blockram* FIFO memory type with *16* depth value).

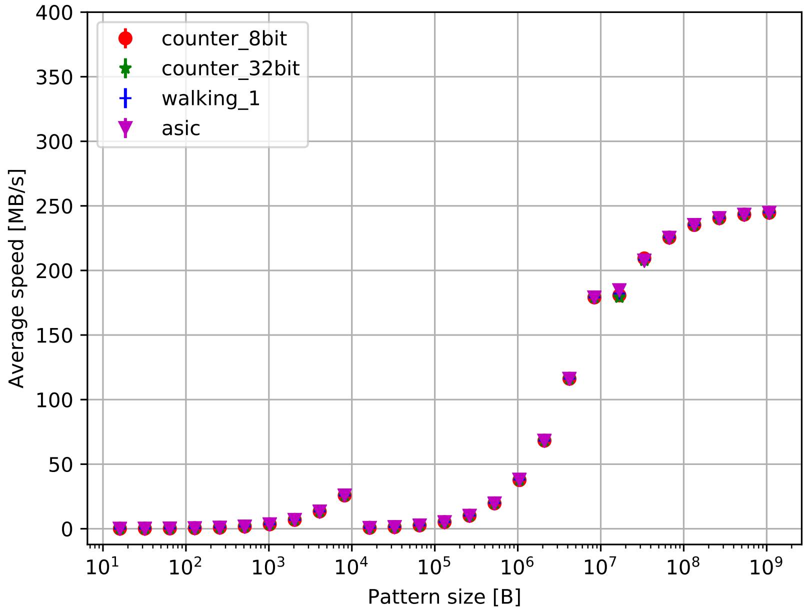


Figure 19. Example chart that presents speed results from the test of data pattern impact investigation in *nonsym write* mode without anomalies (*blockram* FIFO memory type with *32* depth value).

In case of *32bit* mode and *read* direction, for *blockram* and *distributedram* (16, 64, 256, 1024, and 2048 depth values), and *shiftregister* (256, 1024, and 2048 depth values), no anomalies were detected, and results were similar to those presented in Figure 20. However, in the case of *shiftregister* (16 and 64 depth values), there were some disorders during transfer and looked like that those presented in Figures 22 and 23. For *write* direction, results were similar to those presented in Figure 21.

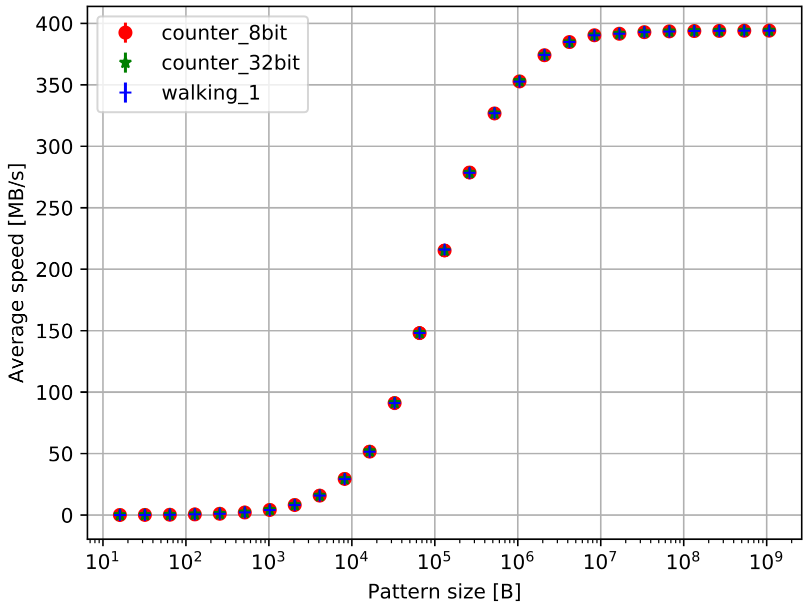


Figure 20. Example chart that presents speed results from the test of data pattern impact investigation in *32bit read* mode without anomalies (*blockram* FIFO memory type with *16* depth value).

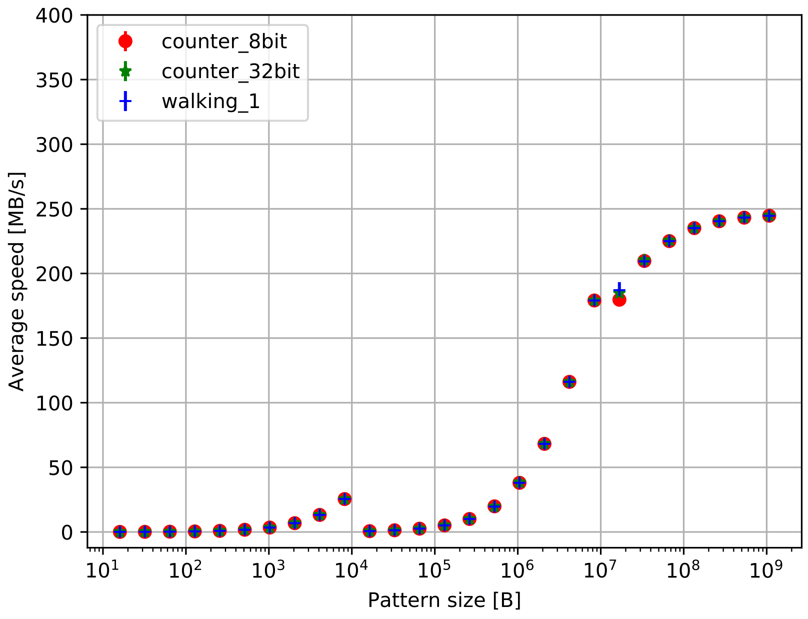


Figure 21. Example chart that presents speed results from the test of data pattern impact investigation in *32bit write* mode without anomalies (*blockram* FIFO memory type with *16* depth value).

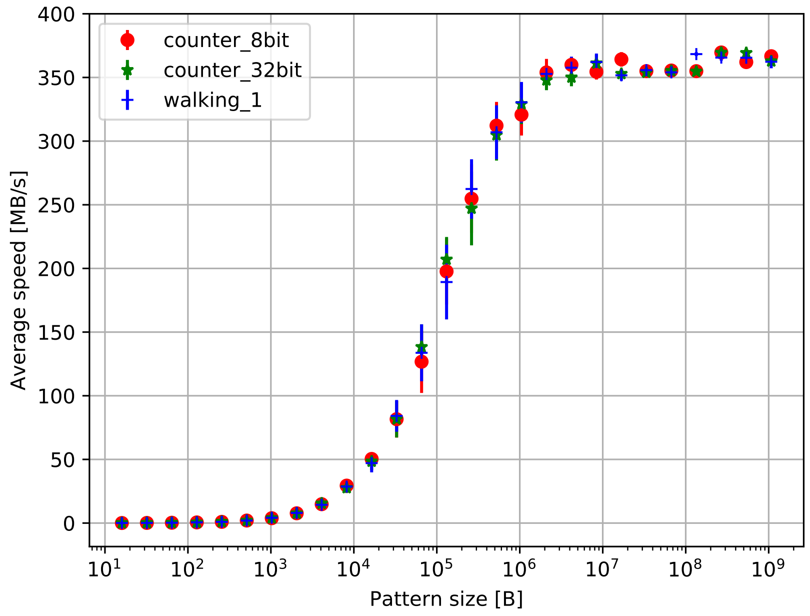


Figure 22. Example chart that presents speed results from the test of data pattern impact investigation in *32bit read* mode with anomalies (*shiftregister* FIFO memory type with *16* depth value).

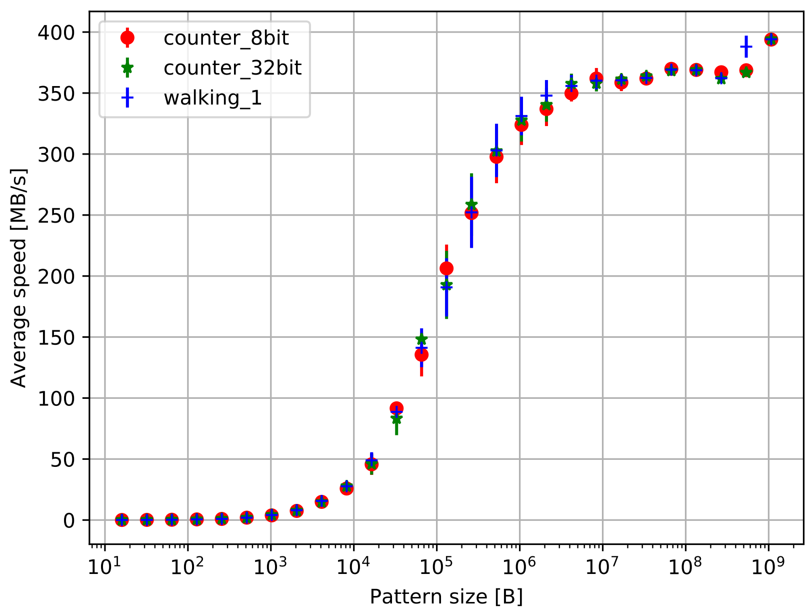


Figure 23. Example chart that presents speed results from the test of data pattern impact investigation in *32bit read* mode with anomalies (*shiftregister* FIFO memory type with *64* depth value).

This subtest showed two major problems during the transfer. The first concerns the anomalies observed during testing shift register FIFO memory type. The second showed atypical data writing speeds (from the PC to the FPGA).

The problem with the *shiftregister* is that the maximal transfer speed values are slightly lower than in the other cases (there are even some small discrepancies between pattern types). Besides, in the range of the pattern size from 10kB to 1 MB, the quality of the results is noticeably worse, as the measurement uncertainties are significant. One could conclude that the shift register is not suitable for long patterns, which would be in agreement with the theory that in case of big queues this type of memory is not optimal (see Table 3). However, the problem appears only in the case of 16 and 64 depth values. The rest of the results for that FIFO memory type do not differ from those without anomalies. The reason for these abnormalities should probably be looked for in the operating system. Possibly, some processes or daemons affected the transfer.

Reading data from the FPGA runs smoothly with speed up to 400 MB/s. On the other hand, sending data to the FPGA shows two basic anomalies. First, maximal transfer speed is about 250 MB/s, so it is over half of the reading average top values. Second, there are visible bumps for around 10 kB and 10 MB data pattern size. Reasons for their appearance are not clear. As the measurement uncertainties are low (the results were reproducible) and considering the fact that reading works correctly, the module's microcontroller responsible for transfer handling should not be treated as defected. Probably, reasons should be sought in the PC's USB controller or operating system. Perhaps, changing hardware on the PC side or OS would solve the problem.

In overall, there was no significant difference in transfer speed results between *nonsym* and *32bit* modes. Besides, no impact on transfer was detected that might be potentially affected by pattern types. The whole analysis was based on a graphical representation of results of this subtest placed in the Appendix section A.1.

5.3. Investigation of FIFO memory type impact on transfer speed

The first level parameter in this subtest is FIFO depth value. The second one is data pattern type. The third parameter is FIFO memory type. This subsection examines the impact on bandwidth considering a type of memory implemented in FIFO's core.

In case of *nonsym* mode, both *read* and *write* direction did not reveal any anomalies (except that speed results between these directions differ. This case has been described more in subsection 5.2). In fact, nothing particular this subtest shows as the only valid memory type in that mode is *blockram*. All results are similar to those presented in Figure 24 for *read* and Figure 25 for *write*.

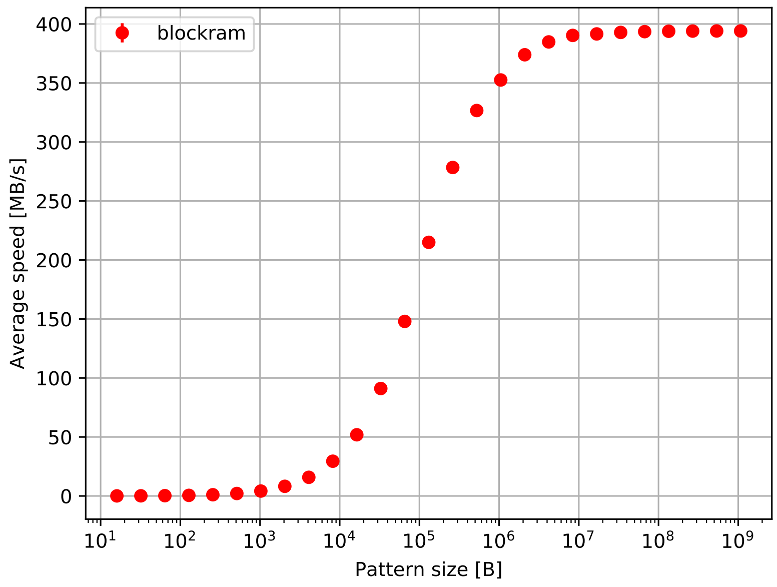


Figure 24. Example chart that presents speed results from the test of FIFO memory type impact investigation in *nonsym read* mode (*16* FIFO depth value and *counter_8bit* pattern type).

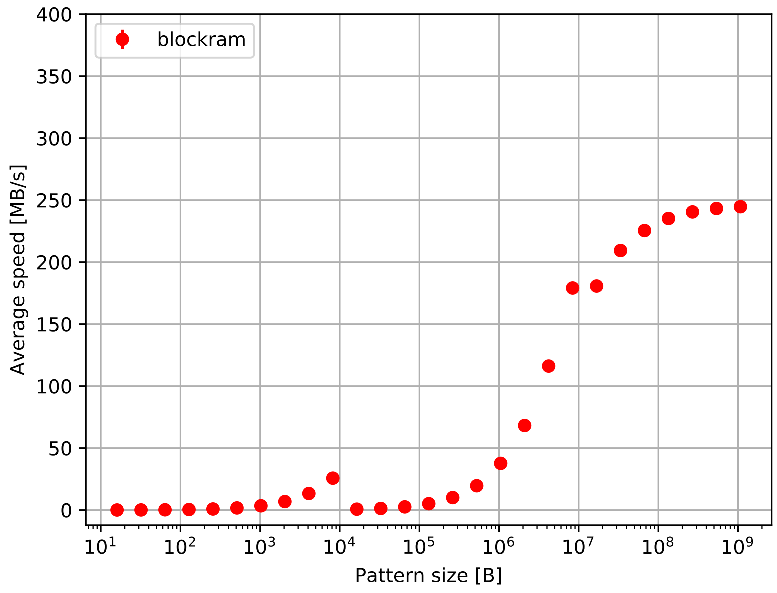


Figure 25. Example chart that presents speed results from the test of FIFO memory type impact investigation in *nonsym write* mode (*32* FIFO depth value and *counter_8bit* pattern type).

In case of *32bit* mode and *read* direction, for all depth values (and all pattern types), no anomalies were detected, and the results were similar to those presented in Figure 26. Though, for 16 and 64 depth values are noticeable anomalies only in *shiftregister* (see Figure 28 and 29). For *write* direction, the transfer ran as shown in Figure 27.

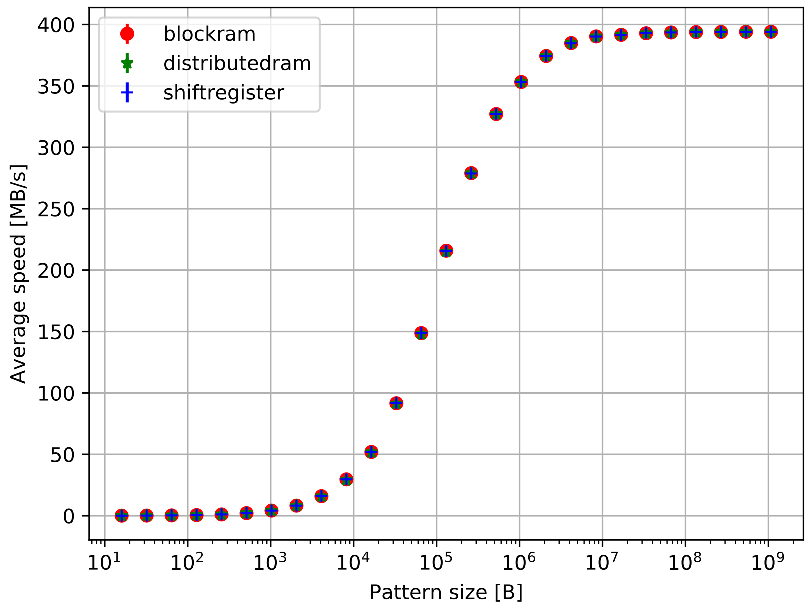


Figure 26. Example chart that presents speed results from the test of FIFO memory type impact investigation in *32bit read* mode without anomalies (*256* FIFO depth value and *counter_8bit* pattern type).

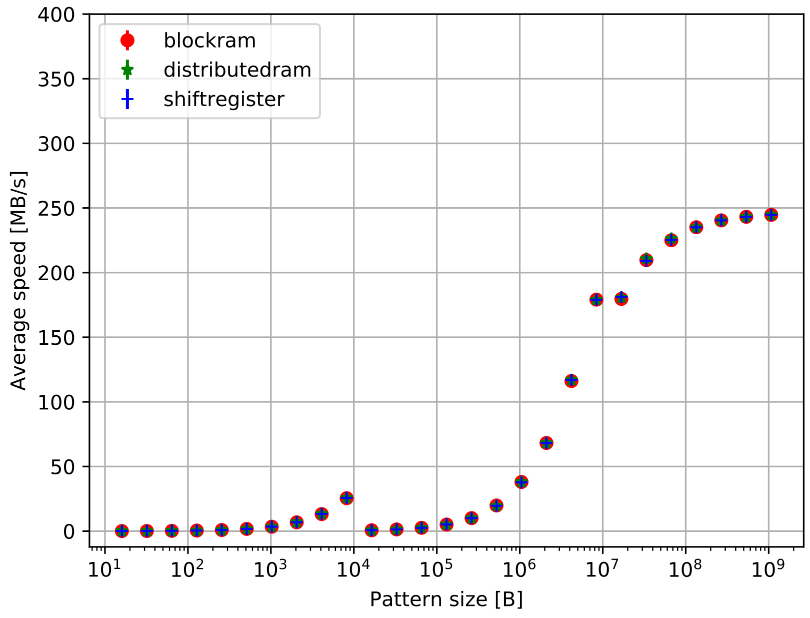


Figure 27. Example chart that presents speed results from the test of FIFO memory type impact investigation in *32bit write* mode without anomalies (*16* FIFO depth value and *counter_8bit* pattern type).

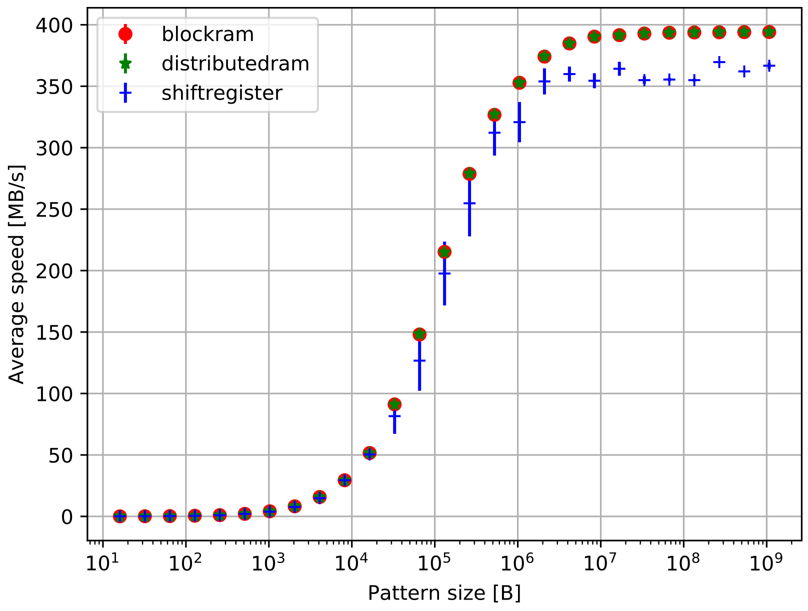


Figure 28. Example chart that presents speed results from the test of FIFO memory type impact investigation in *32bit read* mode with anomalies (*16* FIFO depth value and *counter_8bit* pattern type).

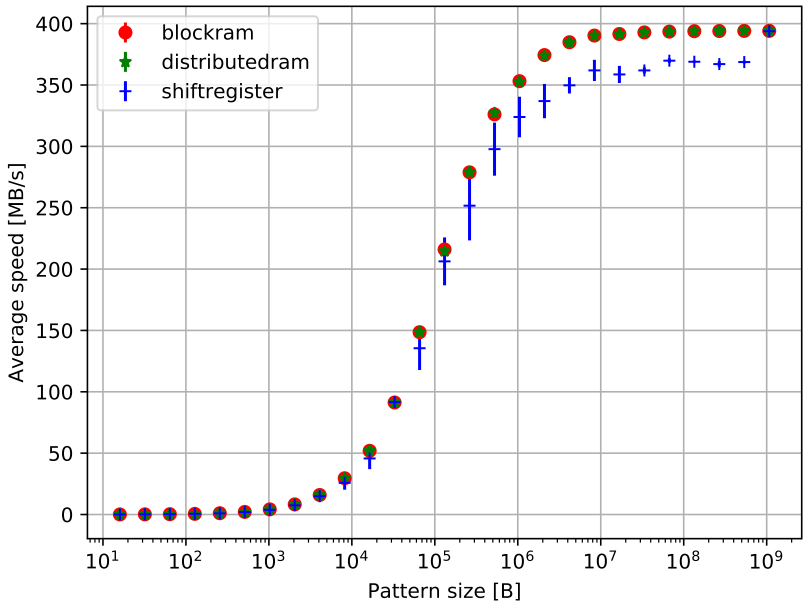


Figure 29. Example chart that presents speed results from the test of FIFO memory type impact investigation in *32bit read* mode with anomalies (*64* FIFO depth value and *counter_8bit* pattern type).

In overall, there was no significant difference in transfer speed results between *nonsym* and *32bit* modes. Except for the mentioned anomalies in *32bit* mode (*shiftregister* with 16 and 64 depth value, which were more described in subsection 5.2), no impact on transfer was detected that might be potentially affected by FIFO memory types. The whole analysis was based on a graphical representation of results of this subtest placed in the Appendix section A.2.

5.4. Investigation of FIFO depth value impact on transfer speed

The first level parameter in this subtest is data pattern type. The second one is FIFO memory type. The third parameter is FIFO depth value. This subsection examines the impact on bandwidth considering the depth of a queue implemented in FIFO.

In case of *nonsym* mode, no anomalies were detected. All *read* results are very similar to those presented in Figure 30. In case of *write* direction – all results are similar to those in Figure 31.

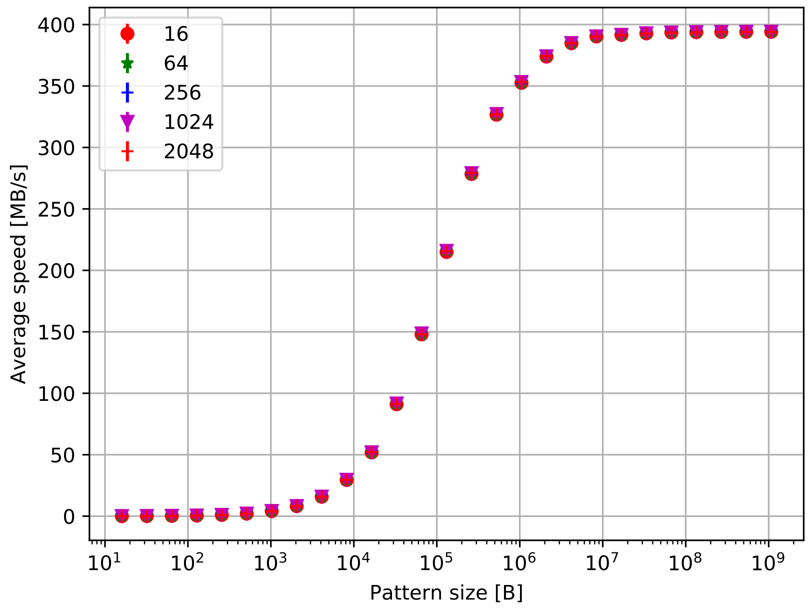


Figure 30. Example chart that presents speed results from the test of FIFO depth value impact investigation in *nonsym read* mode (*counter_8bit* pattern type and *blockram* FIFO memory type).

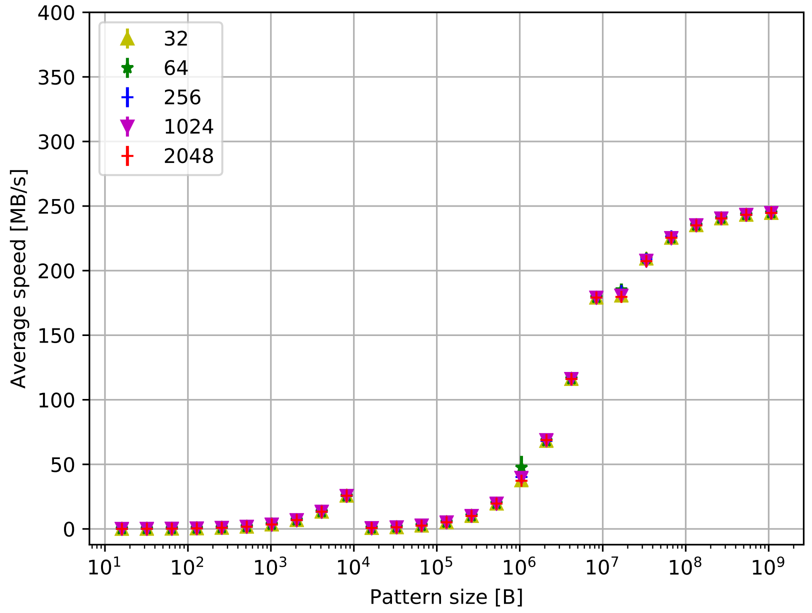


Figure 31. Example chart that presents speed results from the test of FIFO depth value impact investigation in *nonsym write* mode (*counter_8bit* pattern type and *blockram* FIFO memory type).

In case of *32bit* mode and *read* direction, for all pattern types, no anomalies were detected, and the results were similar to those presented in Figure 32. However, some anomalies were found for *shiftregister* memory type – for all pattern types, significant discrepancies were found among 16 and 64 depth values. For comparison, examples for *counter_8bit* and *counter_32bit* were presented in Figures 34 and 35. For *write* direction, the transfer ran as shown in Figure 33.

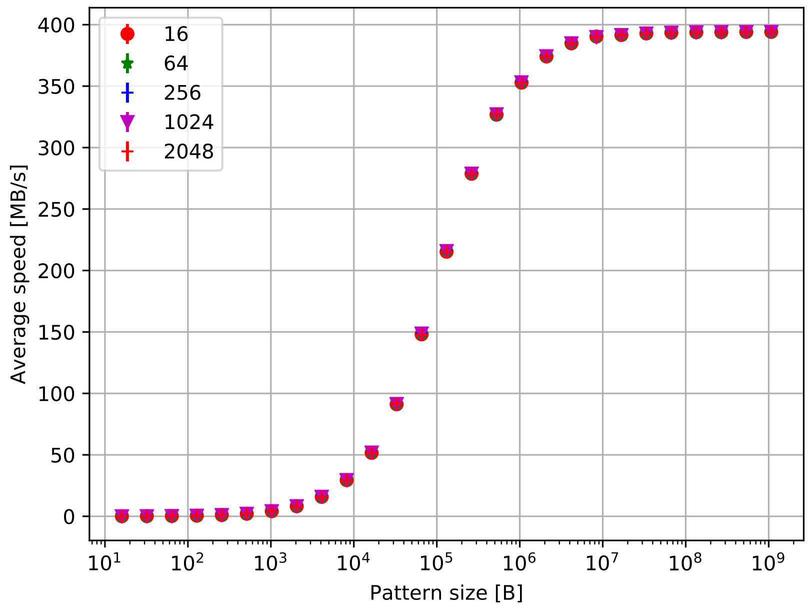


Figure 32. Example chart that presents speed results from the test of FIFO depth value impact investigation in *32bit read* mode without anomalies (*counter_8bit* pattern type and *blockram* FIFO memory type).

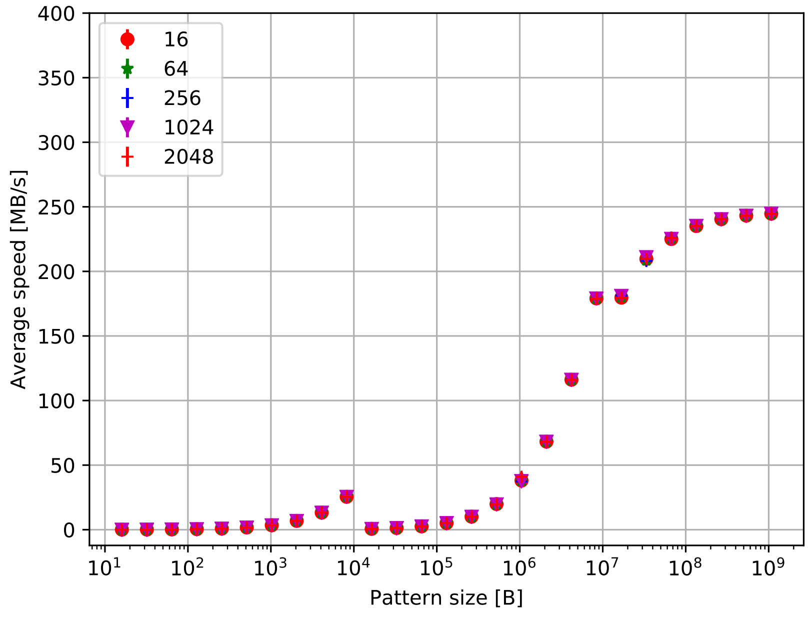


Figure 33. Example chart that presents speed results from the test of FIFO depth value impact investigation in *write* mode without anomalies (*counter_8bit* pattern type and *blockram* FIFO memory type).

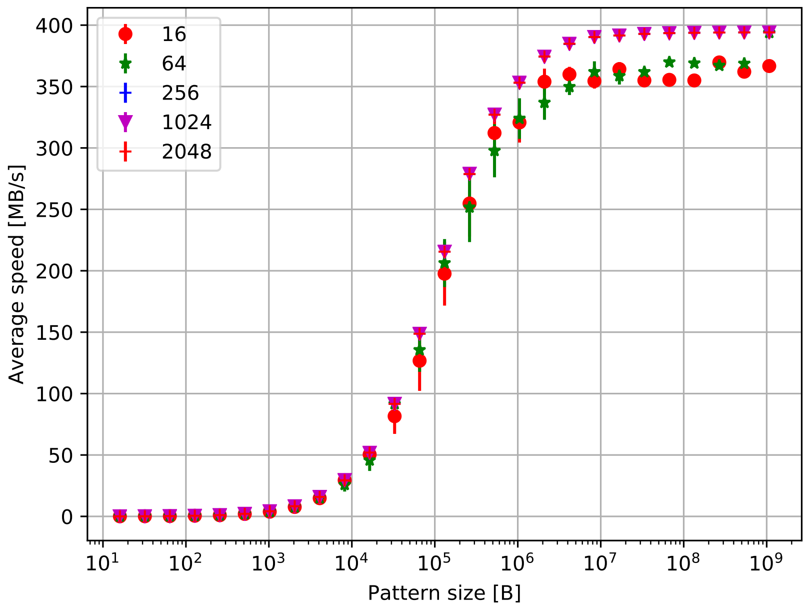


Figure 34. Example chart that presents speed results from the test of FIFO depth value impact investigation in *32bit read* mode with anomalies (*counter_8bit* pattern type and *shiftregister* FIFO memory type).

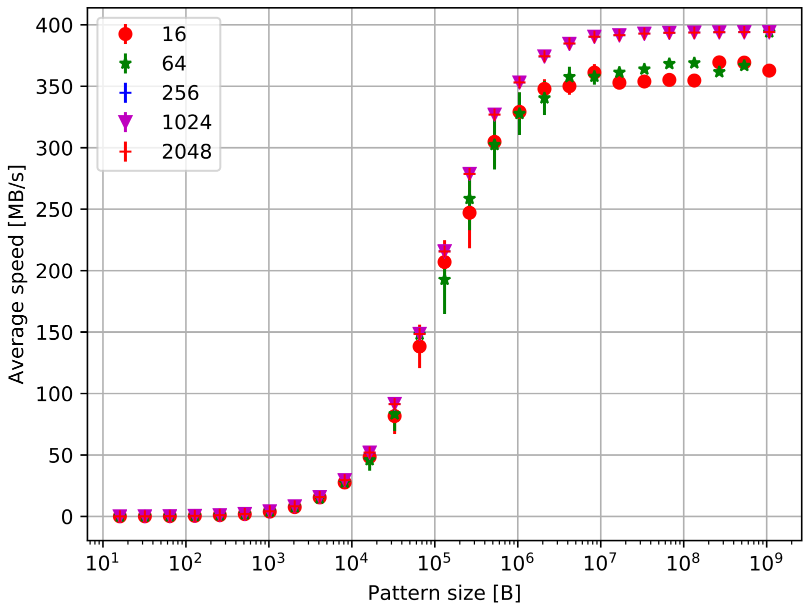


Figure 35. Example chart that presents speed results from the test of FIFO depth value impact investigation in *32bit read* mode with anomalies (*counter_32bit* pattern type and *shiftregister* FIFO memory type).

In overall, there was no significant difference in transfer speed results between *nonsym* and *32bit* modes. Except for the mentioned anomalies in *32bit* mode (*shiftregister* with 16 and 64 depth value), no impact on transfer was detected that might be potentially affected by the specific depth of FIFO. The whole analysis was based on a graphical representation of results of this subtest placed in the Appendix section A.3.

5.5. Investigation of data pattern type impact on transfer speed in pseudo-duplex mode

The first level parameter in this subtest is FIFO memory type (with only one valid value – blockram). The second one is data block size. The third parameter is data pattern type. This subsection examines the impact on bandwidth considering a data pattern type sent in small blocks in pseudo-duplex mode. Graphical presentation of results can be found in Figures 36, 37, 38, and 39.

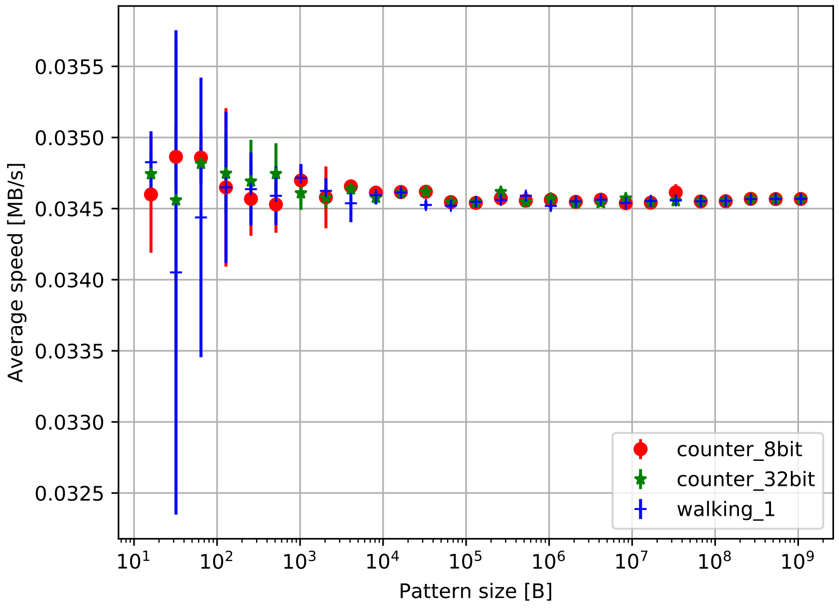


Figure 36. Transfer results for the test that investigates data pattern impact in *duplex* mode (*blockram* FIFO memory type, 16 *block_size*).

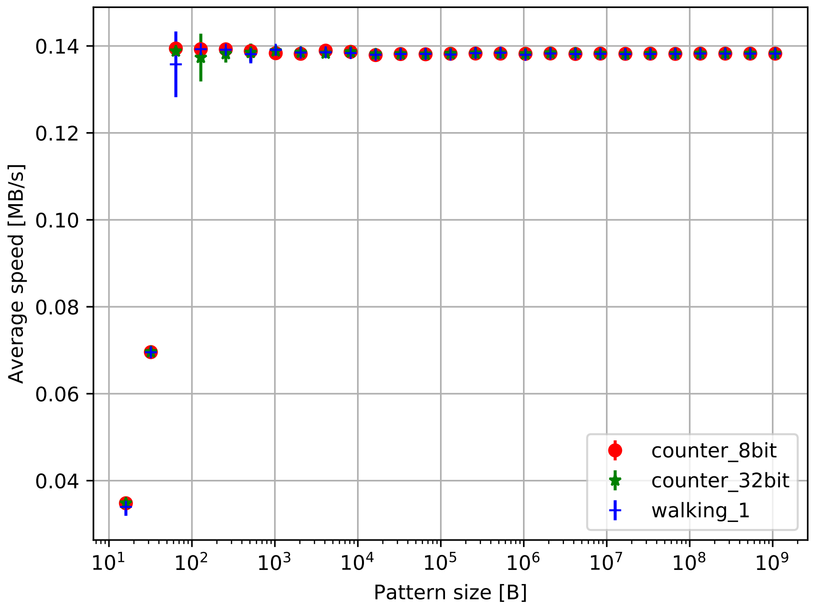


Figure 37. Transfer results for the test that investigates data pattern impact in *duplex* mode (*blockram* FIFO memory type, 64 *block_size*).

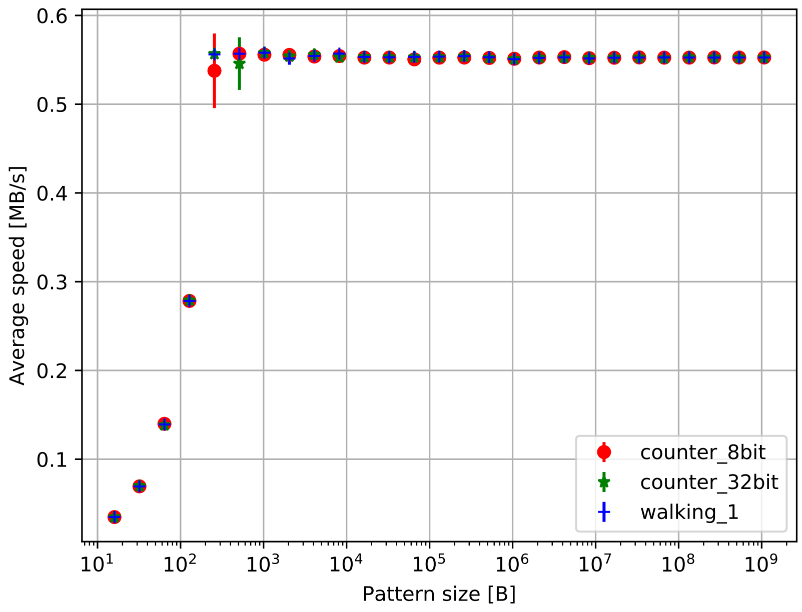


Figure 38. Transfer results for the test that investigates data pattern impact in *duplex* mode (*blockram* FIFO memory type, 256 *block_size*).

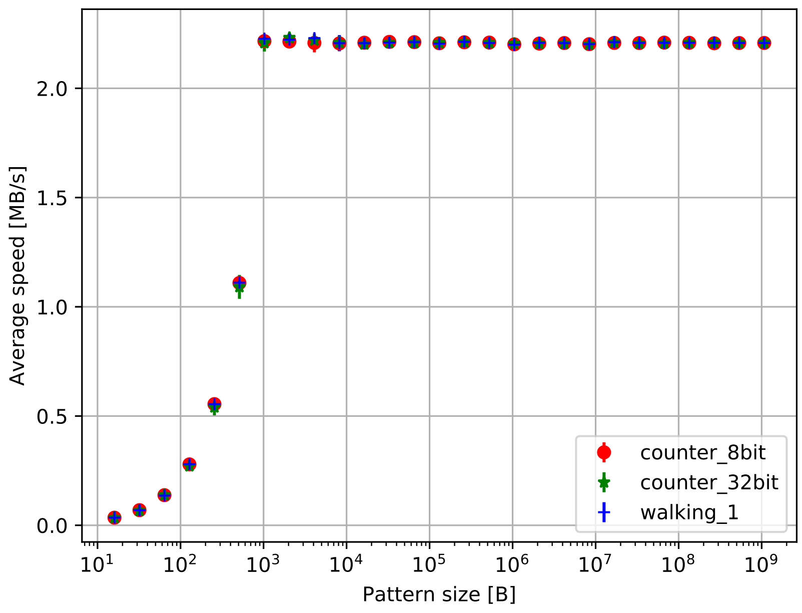


Figure 39. Transfer results for the test that investigates data pattern impact in *duplex* mode (*blockram* FIFO memory type, 1024 *block_size*).

The graphical presentation shows that despite the large uncertainties for the small size of patterns (for smaller block sizes), the results between pattern types coincide, so the speed transfer speed does not depend on the pattern used (considering pseudo-duplex mode). More precise result values can be found in tables in the Appendix section A.4.

5.6. Research of the most optimal block size value for pseudo-duplex mode

The first level parameter in this subtest is data pattern type. The second one is FIFO memory type (with only one valid value – *blockram*). The third parameter is block size. This subsection examines the best block size value that allows a fast pseudo-duplex transfer. Graphical presentation of results can be found in Figure 40, 41 and 42.

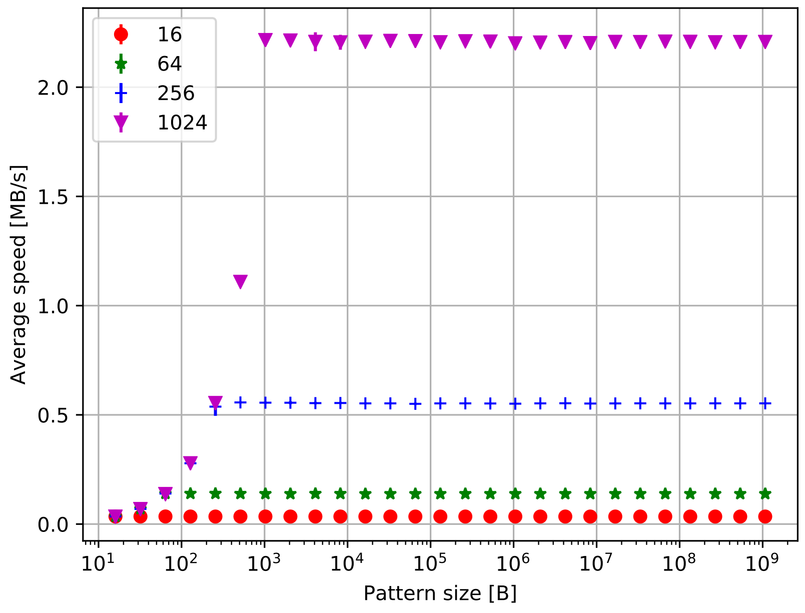


Figure 40. Transfer results for the research of the most optimal block size value in *duplex* mode (*counter_8bit* pattern type, *blockram* FIFO memory type).

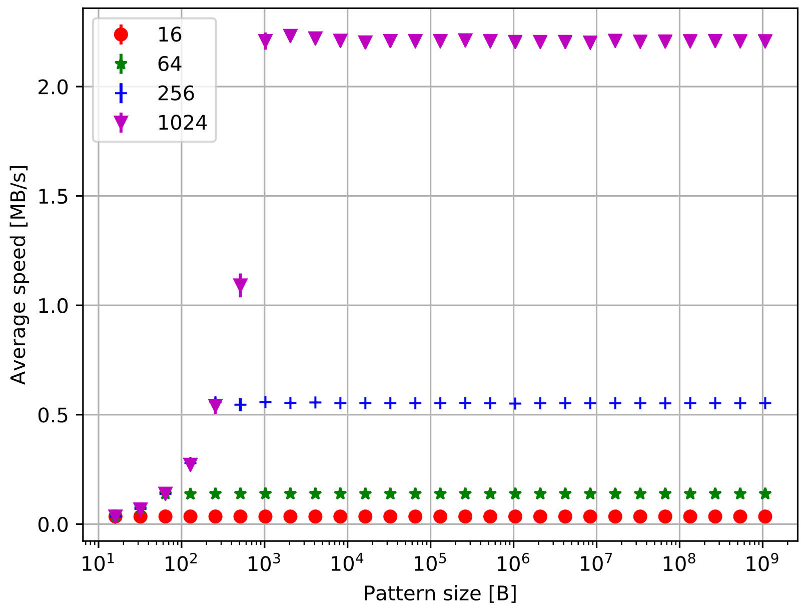


Figure 41. Transfer results for the research of the most optimal block size value in *duplex* mode (*counter_32bit* pattern type, *blockram* FIFO memory type).

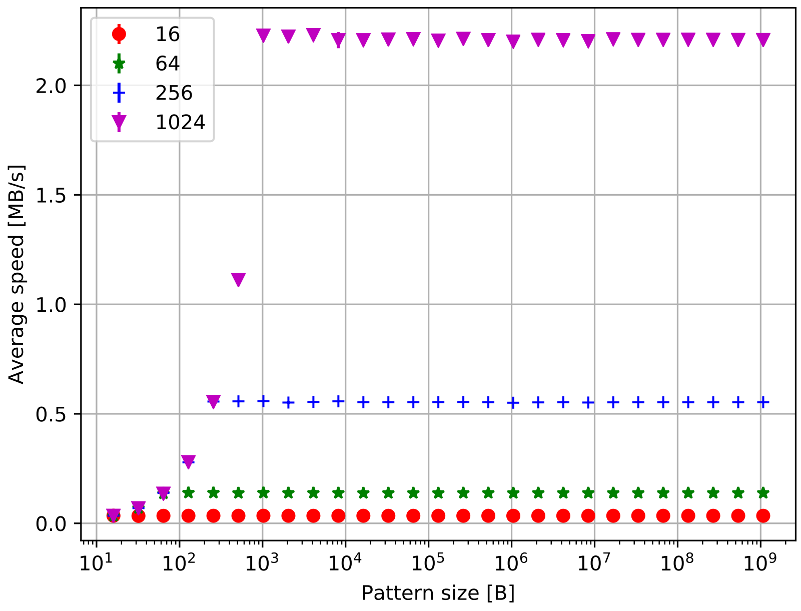


Figure 42. Transfer results for the research of the most optimal block size value in *duplex* mode (*walking_1* pattern type, *blockram* FIFO memory type).

It can be clearly seen that the size of the block of data affects the transfer speed. When the block is larger, the speed increases. Moreover, every pattern size has own speed plateau which depends on the block size. The larger piece of data, the shorter and higher the limit is. More precise result values can be found in tables in the Appendix section A.5.

6. Summary and conclusions

The bandwidth may be affected by multiple aspects. The OpalKelly documentation says that even small operations performed on not real-time operating systems (such as Windows, Linux or macOS) may significantly reduce transfer speed [6]. It is noticeable that in the testing section, in the case of non-duplex modes, there is a deviation between *read* and *write* direction. Typically, the difference should be of a few percents. Besides, the *write* results have two peaks around 10 kB and 10 MB of pattern size, the origin of which is not entirely clear. Perhaps, when changing the PC hardware or OS, the results may be equal considering direction. However, the key of this thesis was to check if implementation on FPGA side may impact transfer, e.g. when complementary cores like FIFO are embedded. Moreover, the test of pseudo-duplex mode should say which block size implementation is the best to simulate fully duplex mode.

Subection 5.2 examined if data pattern sent between FPGA and PC matters considering speed transfer. The analysis showed that there is no apparent dependence between transfer rate and sent content.

Subection 5.3 examined if FIFO memory type has an impact on transfer speed. In *nonsym* mode, only Block RAM implementation is valid, so no further conclusions can be drawn. However, in *32bit* mode in some cases, there is a visible difference between the shift register and other implementations, especially in more extended patterns. It follows the theory that assumes that shift register is not the best memory type for long patterns. Though, this difference might be caused by other aspects that came from OS side (e.g., other processes have disrupted the execution of the program), as the results had a large dispersion compared to the other cases (with high measurement uncertainty), and appeared only in situations when two specific depth values were used.

Subection 5.4 examined if depth of FIFO has an impact on transfer speed. Here, the situation is clear – no effect should be visible.

Subection 5.5 examined if data pattern type may be crucial when sending small blocks of data in pseudo-duplex mode. Also here, the conclusion is – no impact. The last subsection (5.6), however, showed that block size matters. Up to some value of pattern size, the total speed increases exponentially and eventually reaches max value. But, these speeds are still too low to perform pseudo-duplex transfers efficiently. Thus, bigger block sizes should be used. After all, although the isochronous transfer is not supported by FrontPanel (for the sake of efficient transfer error checking), the module can imitate full duplex mode.

To sum up, there is no significant impact on transfer speed considering FIFO implementation and sent data. Only the size of pattern matters (and its block size in case of pseudo-duplex applications). From 10 MB of transmitted data, the rate reaches max value up to 400 MB/s (in *read* direction), so the module met the expected speed. Plus, no errors during transfer were performed, so the transfer was efficient at 100%. Furthermore, the test environment is built on cross-platform libraries, so it is universal in use. The OpalKelly module is entirely suitable for neuroscientific experiments.

References

[1] Szypulska M., Dąbrowski W., Hottowy P. et al., “Modular asic-based system for large-scale electrical stimulation and recording of brain activity in behaving animals,” tech. rep., Faculty of Physics and Applied Computer Science AGH University of Science and Technology Krakow, Poland, 2016.

[2] OpalKelly Inc., “XEM6310 product overview.” `https://www.opalkelly.com/products/xem6310/`. Access: 17.05.2018.

[3] OpalKelly Inc., “BRK6110 breakout board.” `https://docs.opalkelly.com/display/XEM6310/BRK6110+Breakout+Board`. Access: 17.05.2018.

[4] OpalKelly Inc., “FrontPanel HDL – USB 3.0.” `https://docs.opalkelly.com/display/FPSDK/FrontPanel+HDL+-+USB+3.0`. Access: 17.05.2018.

[5] Xilinx Inc., *FIFO Generator v13.1. LogiCORE IP Product Guide*, April 2017. Document number: PG057.

[6] OpalKelly Inc., “FrontPanel’s performance.” `https://docs.opalkelly.com/display/FPSDK/Performance`. Access: 17.05.2018.

[7] Vahid F., *Digital Design with RTL Design, VHDL, and Verilog.* John Wiley & Sons, Inc., second ed., 2011.

[8] PXI Systems Alliance, “About PXI.” `http://www.pxisa.org/About/AboutPXI.aspx`. Access: 17.05.2018.

[9] Texas Instruments Inc., *Interface Circuits for TIA/EIA-644 (LVDS)*, September 2002. Document number: SLLA038B.

[10] Hewlett-Packard Company, Intel Corporation, Microsoft Corporation, NEC Corporation, ST-NXP Wireless, and Texas Instruments, *Universal Serial Bus 3.0 Specification*, November 2008. Revision 1.0.

[11] OpalKelly Inc., “FrontPanel presentation.” `https://www.opalkelly.com/products/frontpanel/`. Access: 17.05.2018.

[12] OpalKelly Inc., “FrontPanel API.” `https://docs.opalkelly.com/display/FPSDK/FrontPanel+API`. Access: 17.05.2018.

[13] OpalKelly Inc., “FrontPanel HDL.” `https://docs.opalkelly.com/display/FPSDK/FrontPanel+HDL`. Access: 17.05.2018.

[14] Cormen T. and Leiserson C. and Rivest R. and Stein C., *Introduction to Algorithms.* Massachusetts Institute of Technology, third ed., 2009.

[15] Stroustrup B., *The C++ Programming Language.* Addison Wesley, fourth ed., 2013.

[16] The Institute of Electrical and Electronics Engineers, Inc., *IEEE Standard for Verilog® Hardware Description Language*, April 2006. IEEE Std 1364^{TM}-2005.

[17] Python Software Foundation, “Python official website.” `https://www.python.org/`. Access: 17.05.2018.

[18] AGH University of Science and Technology, Faculty of Physics and Applied Computer Science, Department of Nuclear Electronics, *Specification of STIM64 IC*, July 2015. version 1.2.

[19] International Organization for Standardization, *ISO/IEC 14882:2011*, September 2011.

[20] Google Inc., "C++ implementation of the Google logging module." `https://github.com/google/glog`. Access: 17.05.2018.

[21] hyperrealm, "C/C++ library for processing configuration files." `https://github.com/hyperrealm/libconfig`. Access: 17.05.2018.

[22] The Matplotlib development team, "Matplotlib: Python plotting." `https://matplotlib.org/`. Access: 17.05.2018.

[23] Kitware, Inc., "CMake official website." `https://cmake.org/`. Access: 17.05.2018.

A. Appendix

A.1. Graphical presentation of results from the data pattern type impact investigation subtest

A.1.1. nonsym mode part of the subtest

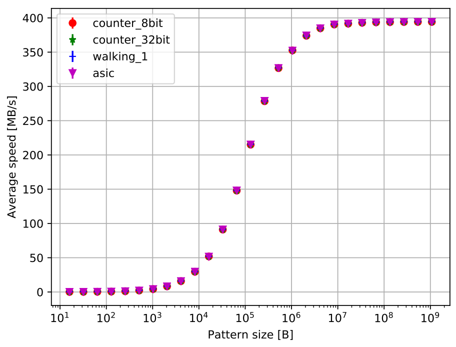


Figure 43. Speed results for data pattern impact subtest in *nonsym read* mode (*blockram* FIFO memory type with *16* depth value).

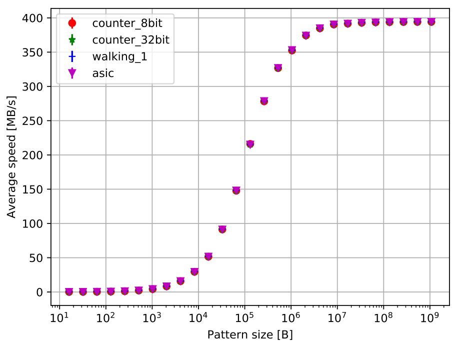


Figure 44. Speed results for data pattern impact subtest in *nonsym read* mode (*blockram* FIFO memory type with *64* depth value).

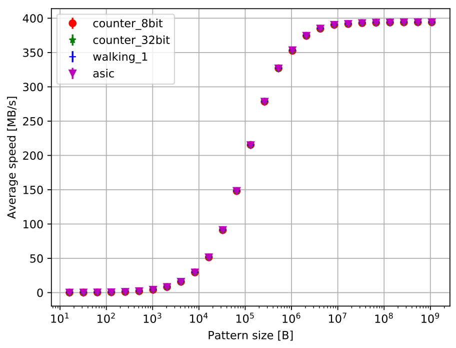


Figure 45. Speed results for data pattern impact subtest in *nonsym read* mode (*blockram* FIFO memory type with *256* depth value).

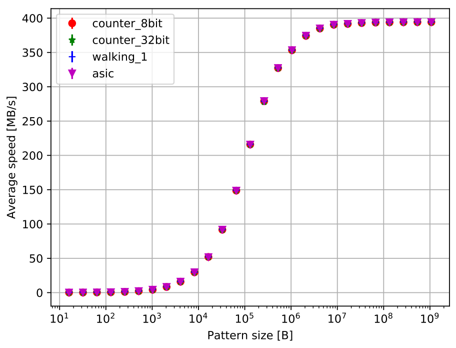


Figure 46. Speed results for data pattern impact subtest in *nonsym read* mode (*blockram* FIFO memory type with *1024* depth value).

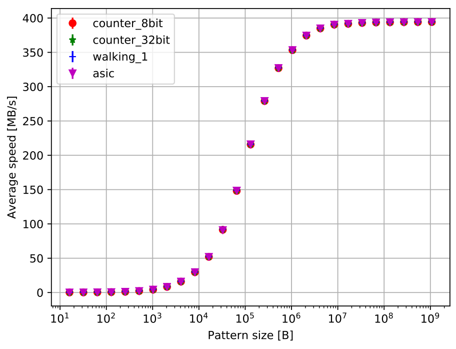


Figure 47. Speed results for data pattern impact subtest in *nonsym read* mode (*blockram* FIFO memory type with *2048* depth value).

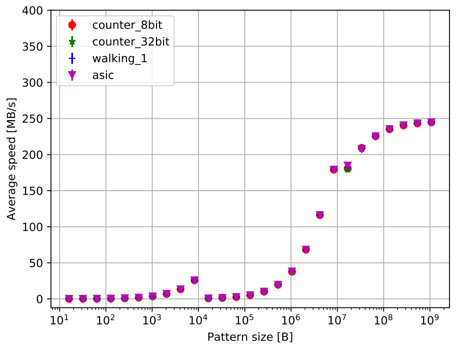


Figure 48. Speed results for data pattern impact subtest in *nonsym write* mode (*blockram* FIFO memory type with *32* depth value).

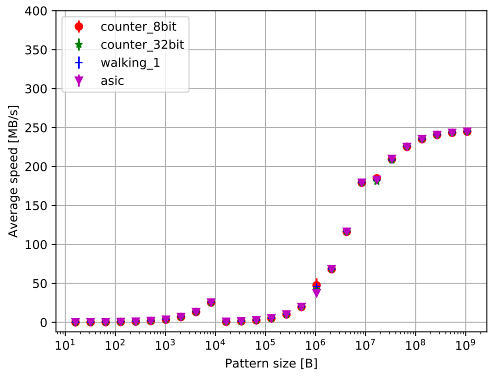


Figure 49. Speed results for data pattern impact subtest in *nonsym write* mode (*blockram* FIFO memory type with *64* depth value).

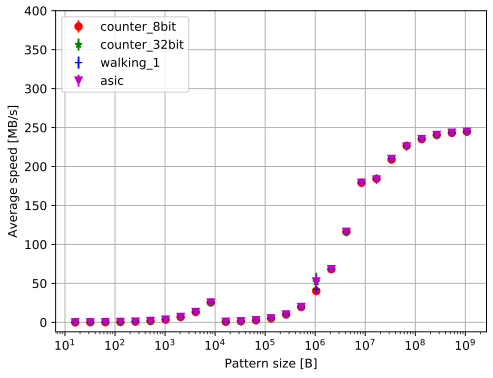


Figure 50. Speed results for data pattern impact subtest in *nonsym write* mode (*blockram* FIFO memory type with *256* depth value).

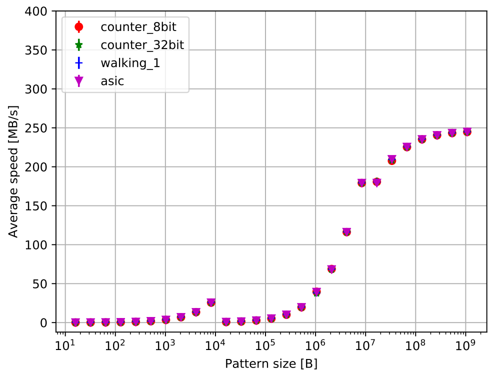


Figure 51. Speed results for data pattern impact subtest in *nonsym write* mode (*blockram* FIFO memory type with *1024* depth value).

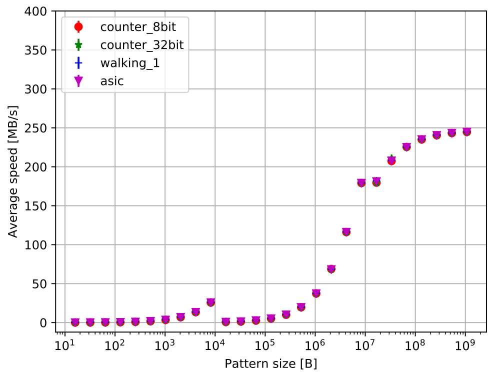


Figure 52. Speed results for data pattern impact subtest in *nonsym write* mode (*blockram* FIFO memory type with *2048* depth value).

Table 7. *The fastest pattern types (values in square brackets are in MB/s) compared between depth values (nonsym mode, read direction, blockram FIFO memory type).*

Pattern size [B]	**16**	**64**	**256**	**1024**	**2048**
16	asic [*0.065*]	walking_1 [*0.065*]	counter_8bit [*0.065*]	counter_8bit [*0.066*]	counter_8bit [*0.066*]
32	counter_32bit [*0.132*]	walking_1 [*0.131*]	walking_1 [*0.131*]	counter_8bit [*0.132*]	walking_1 [*0.132*]
64	counter_32bit [*0.262*]	counter_8bit [*0.263*]	counter_32bit [*0.262*]	counter_32bit [*0.264*]	asic [*0.263*]
128	counter_32bit [*0.523*]	walking_1 [*0.523*]	walking_1 [*0.522*]	asic [*0.527*]	counter_8bit [*0.528*]
256	counter_8bit [*1.046*]	walking_1 [*1.045*]	walking_1 [*1.047*]	counter_8bit [*1.054*]	asic [*1.054*]
512	counter_8bit [*2.096*]	counter_8bit [*2.092*]	counter_8bit [*2.091*]	walking_1 [*2.107*]	asic [*2.107*]
1024	counter_32bit [*4.174*]	counter_8bit [*4.177*]	counter_32bit [*4.179*]	walking_1 [*4.209*]	asic [*4.217*]
2048	counter_8bit [*8.191*]	counter_32bit [*8.173*]	walking_1 [*8.190*]	counter_32bit [*8.246*]	counter_32bit [*8.242*]
4096	counter_8bit [*15.816*]	counter_32bit [*15.812*]	asic [*15.813*]	counter_8bit [*15.911*]	walking_1 [*15.933*]
8192	asic [*29.613*]	walking_1 [*29.466*]	walking_1 [*29.438*]	counter_32bit [*29.619*]	asic [*29.611*]
16384	counter_8bit [*51.862*]	asic [*51.569*]	counter_32bit [*51.499*]	counter_8bit [*51.961*]	counter_8bit [*51.974*]
32768	asic [*91.198*]	counter_8bit [*91.236*]	counter_32bit [*91.425*]	counter_32bit [*91.721*]	counter_32bit [*91.823*]
65536	asic [*148.105*]	asic [*148.043*]	asic [*148.193*]	counter_32bit [*148.868*]	walking_1 [*148.646*]
131072	walking_1 [*215.186*]	counter_8bit [*216.054*]	counter_8bit [*215.370*]	counter_8bit [*215.849*]	counter_32bit [*215.974*]
262144	asic [*278.859*]	counter_32bit [*278.772*]	counter_32bit [*278.689*]	asic [*279.065*]	asic [*279.087*]
524288	counter_32bit [*327.077*]	asic [*326.875*]	counter_8bit [*327.005*]	asic [*327.257*]	counter_32bit [*327.184*]
1048576	counter_32bit [*352.594*]	asic [*352.982*]	asic [*353.293*]	asic [*353.288*]	walking_1 [*353.290*]
2097152	asic [*374.325*]	counter_8bit [*374.297*]	walking_1 [*374.400*]	walking_1 [*374.413*]	counter_32bit [*374.425*]
4194304	counter_8bit [*384.797*]	asic [*384.823*]	asic [*384.871*]	asic [*384.887*]	asic [*384.875*]
8388608	walking_1 [*390.318*]	walking_1 [*390.345*]	counter_32bit [*390.345*]	walking_1 [*390.349*]	counter_32bit [*390.347*]
16777216	walking_1 [*391.577*]	walking_1 [*391.585*]	counter_8bit [*391.587*]	walking_1 [*391.596*]	counter_32bit [*391.591*]
33554432	counter_8bit [*392.854*]	counter_8bit [*392.872*]	walking_1 [*392.871*]	counter_32bit [*392.865*]	asic [*392.875*]
67108864	walking_1 [*393.514*]	counter_8bit [*393.517*]	asic [*393.520*]	asic [*393.514*]	counter_8bit [*393.493*]
134217728	counter_8bit [*393.841*]	counter_32bit [*393.820*]	counter_8bit [*393.840*]	asic [*393.825*]	asic [*393.827*]
268435456	asic [*393.995*]	asic [*393.983*]	asic [*394.005*]	walking_1 [*393.999*]	counter_8bit [*393.996*]
536870912	counter_8bit [*394.081*]	asic [*394.077*]	counter_8bit [*394.082*]	counter_32bit [*394.071*]	counter_32bit [*394.074*]
1073741824	counter_32bit [*394.112*]	counter_8bit [*394.113*]	counter_8bit [*394.093*]	walking_1 [*394.113*]	walking_1 [*394.099*]
Most frequent parameter	counter_8bit	counter_8bit	counter_8bit	counter_32bit, asic, walking_1	asic

Table 8. *The fastest pattern types (values in square brackets are in MB/s) compared between depth values (nonsym mode, write direction, blockram FIFO memory type).*

Pattern size [B]	**32**	**64**	**256**	**1024**	**2048**
16	asic *[0.054]*	counter_32bit *[0.053]*	asic *[0.054]*	asic *[0.054]*	walking_1 *[0.054]*
32	asic *[0.108]*	walking_1 *[0.107]*	counter_8bit *[0.107]*	asic *[0.107]*	counter_8bit *[0.107]*
64	counter_32bit *[0.215]*	asic *[0.214]*	counter_8bit *[0.214]*	walking_1 *[0.215]*	asic *[0.215]*
128	walking_1 *[0.430]*	walking_1 *[0.428]*	counter_8bit *[0.429]*	counter_8bit *[0.430]*	asic *[0.430]*
256	walking_1 *[0.861]*	asic *[0.855]*	counter_32bit *[0.856]*	walking_1 *[0.858]*	counter_8bit *[0.860]*
512	counter_32bit *[1.717]*	asic *[1.703]*	asic *[1.709]*	asic *[1.716]*	asic *[1.719]*
1024	counter_32bit *[3.438]*	walking_1 *[3.414]*	counter_32bit *[3.421]*	walking_1 *[3.439]*	asic *[3.441]*
2048	counter_32bit *[6.852]*	walking_1 *[6.808]*	walking_1 *[6.815]*	counter_32bit *[6.846]*	asic *[6.849]*
4096	walking_1 *[13.363]*	counter_8bit *[13.208]*	counter_8bit *[13.229]*	counter_8bit *[13.350]*	counter_8bit *[13.357]*
8192	asic *[25.778]*	walking_1 *[25.408]*	counter_8bit *[25.509]*	walking_1 *[25.589]*	counter_8bit *[25.737]*
16384	counter_8bit *[0.717]*	walking_1 *[0.696]*	walking_1 *[0.700]*	asic *[0.727]*	counter_8bit *[0.738]*
32768	asic *[1.312]*	walking_1 *[1.343]*	counter_8bit *[1.344]*	counter_32bit *[1.326]*	counter_32bit *[1.313]*
65536	walking_1 *[2.561]*	walking_1 *[2.651]*	counter_8bit *[2.561]*	counter_32bit *[2.758]*	walking_1 *[2.824]*
131072	counter_8bit *[5.147]*	counter_32bit *[5.093]*	counter_32bit *[5.147]*	asic *[5.146]*	counter_8bit *[5.146]*
262144	asic *[10.060]*	counter_8bit *[10.060]*	counter_32bit *[10.166]*	counter_8bit *[10.061]*	asic *[10.061]*
524288	walking_1 *[19.828]*	walking_1 *[19.628]*	counter_8bit *[19.629]*	asic *[19.627]*	counter_32bit *[19.629]*
1048576	walking_1 *[38.026]*	counter_8bit *[47.582]*	asic *[52.397]*	counter_8bit *[39.562]*	counter_8bit *[37.295]*
2097152	asic *[68.184]*	walking_1 *[68.190]*	counter_8bit *[68.163]*	counter_8bit *[68.772]*	counter_8bit *[68.771]*
4194304	walking_1 *[116.138]*	counter_32bit *[116.144]*	counter_8bit *[116.110]*	counter_32bit *[116.127]*	asic *[116.146]*
8388608	counter_8bit *[179.106]*	counter_8bit *[179.142]*	counter_32bit *[180.102]*	asic *[179.102]*	counter_8bit *[179.129]*
16777216	asic *[184.707]*	counter_8bit *[184.886]*	counter_32bit *[185.187]*	counter_32bit *[181.192]*	walking_1 *[182.194]*
33554432	counter_8bit *[209.324]*	asic *[209.682]*	walking_1 *[210.356]*	asic *[209.637]*	counter_32bit *[211.023]*
67108864	counter_32bit *[225.823]*	walking_1 *[225.813]*	counter_8bit *[226.582]*	walking_1 *[225.430]*	counter_8bit *[225.439]*
134217728	asic *[235.216]*	counter_32bit *[235.279]*	counter_32bit *[235.195]*	asic *[235.189]*	counter_32bit *[235.190]*
268435456	asic *[240.531]*	counter_8bit *[240.485]*	counter_32bit *[240.524]*	counter_32bit *[240.444]*	counter_32bit *[240.473]*
536870912	counter_32bit *[243.254]*	counter_32bit *[243.228]*	asic *[243.328]*	asic *[243.224]*	asic *[243.268]*
1073741824	walking_1 *[244.743]*	counter_32bit *[244.770]*	counter_8bit *[244.705]*	counter_32bit *[244.747]*	counter_32bit *[244.743]*
Most frequent parameter	asic	walking_1	counter_8bit	asic	counter_8bit

A.1.2. 32bit mode part of the subtest

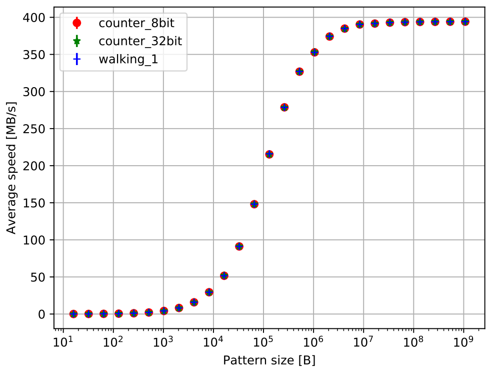


Figure 53. Speed results for data pattern impact subtest in *32bit read* mode (*blockram* FIFO memory type with *16* depth value).

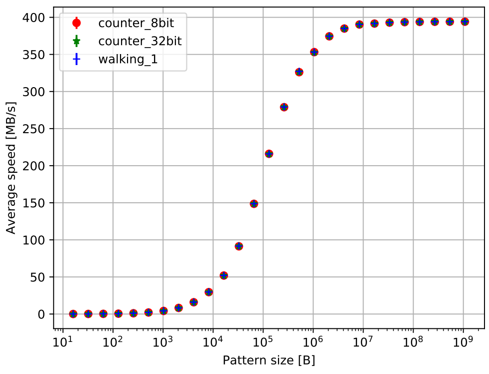


Figure 54. Speed results for data pattern impact subtest in *32bit read* mode (*blockram* FIFO memory type with *64* depth value).

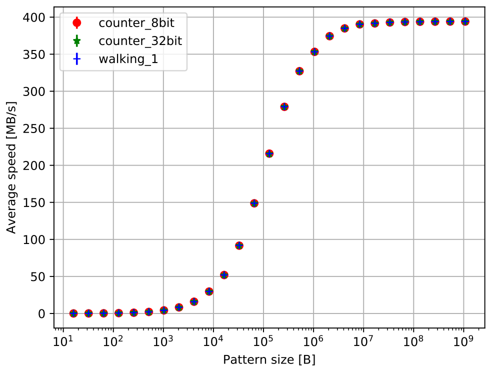


Figure 55. Speed results for data pattern impact subtest in *32bit read* mode (*blockram* FIFO memory type with *256* depth value).

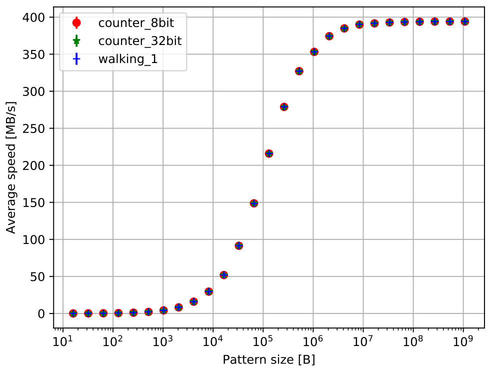


Figure 56. Speed results for data pattern impact subtest in *32bit read* mode (*blockram* FIFO memory type with *1024* depth value).

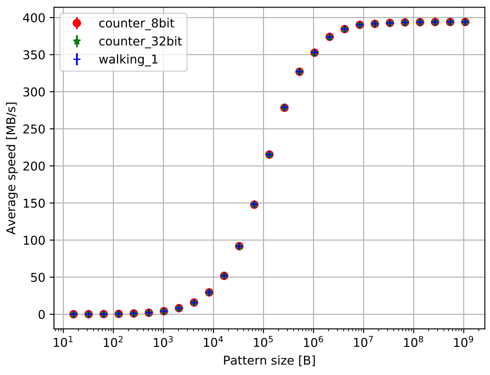


Figure 57. Speed results for data pattern impact subtest in *32bit read* mode (*blockram* FIFO memory type with *2048* depth value).

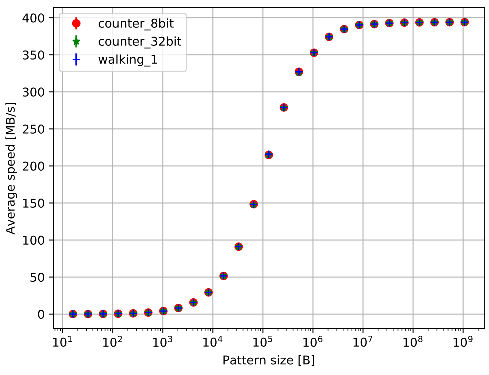


Figure 58. Speed results for data pattern impact subtest in *32bit read* mode (*distributedram* FIFO memory type with *16* depth value).

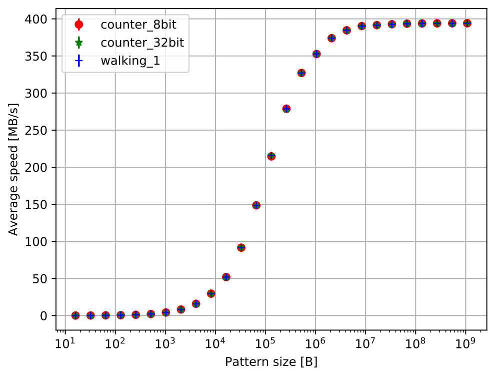


Figure 59. Speed results for data pattern impact subtest in *32bit read* mode (*distributedram* FIFO memory type with *64* depth value).

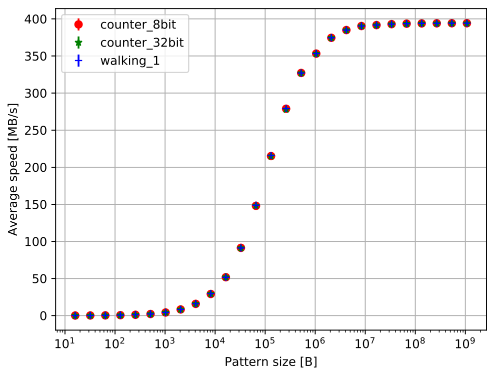


Figure 60. Speed results for data pattern impact subtest in *32bit read* mode (*distributedram* FIFO memory type with *256* depth value).

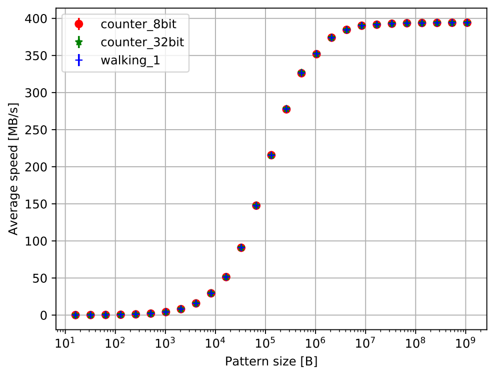


Figure 61. Speed results for data pattern impact subtest in *32bit read* mode (*distributedram* FIFO memory type with *1024* depth value).

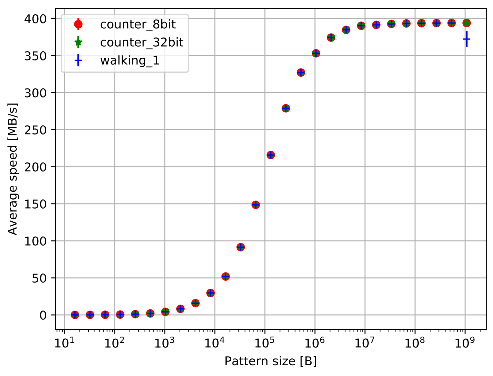


Figure 62. Speed results for data pattern impact subtest in *32bit read* mode (*distributedram* FIFO memory type with *2048* depth value).

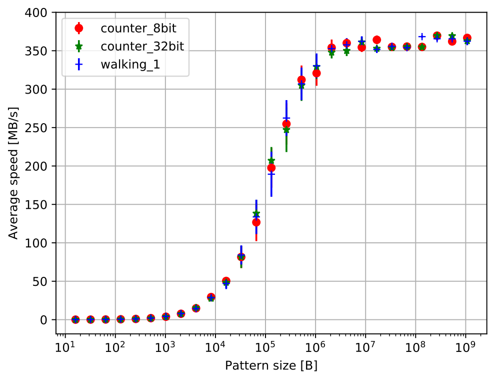


Figure 63. Speed results for data pattern impact subtest in *32bit read* mode (*shiftregister* FIFO memory type with *16* depth value).

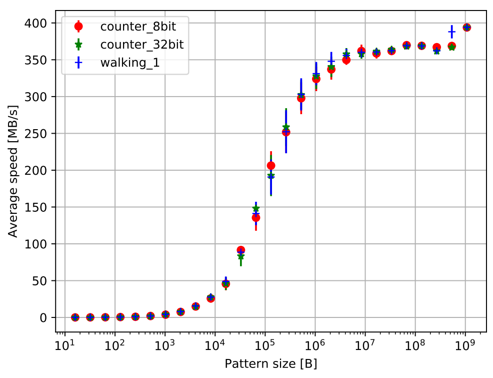


Figure 64. Speed results for data pattern impact subtest in *32bit read* mode (*shiftregister* FIFO memory type with *64* depth value).

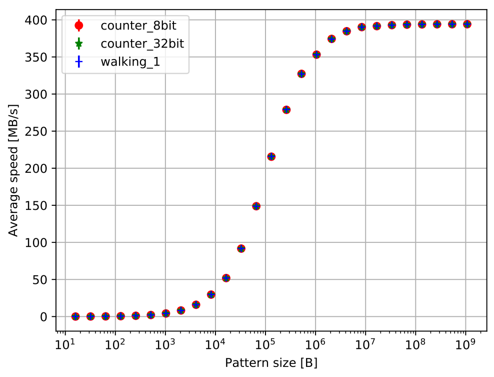


Figure 65. Speed results for data pattern impact subtest in *32bit read* mode (*shiftregister* FIFO memory type with *256* depth value).

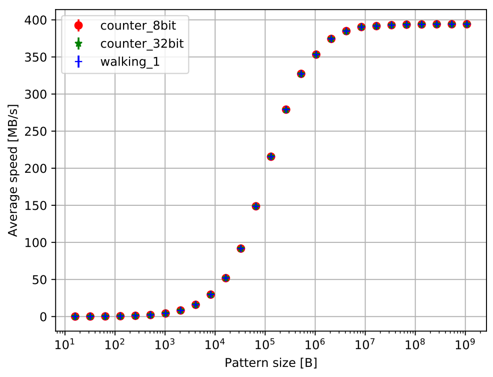


Figure 66. Speed results for data pattern impact subtest in *32bit read* mode (*shiftregister* FIFO memory type with *1024* depth value).

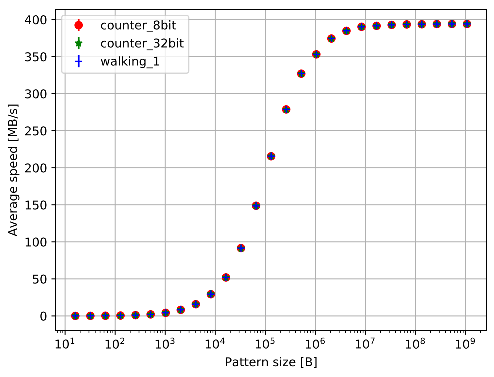


Figure 67. Speed results for data pattern impact subtest in *32bit read* mode (*shiftregister* FIFO memory type with *2048* depth value).

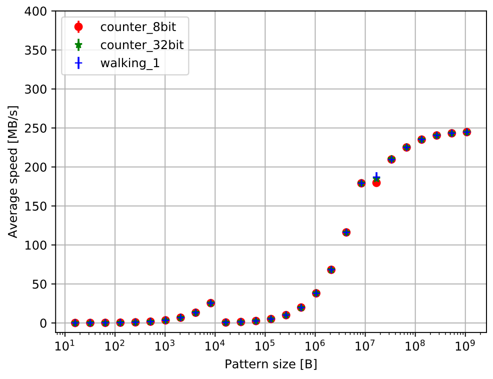


Figure 68. Speed results for data pattern impact subtest in *32bit write* mode (*blockram* FIFO memory type with *16* depth value).

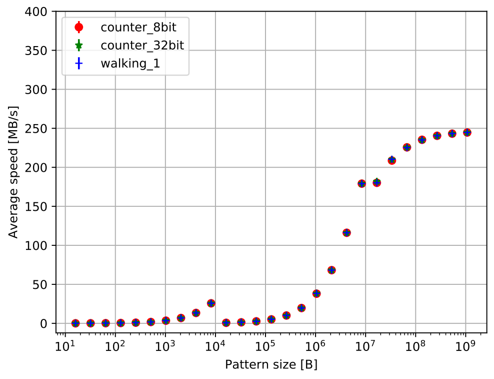


Figure 69. Speed results for data pattern impact subtest in *32bit write* mode (*blockram* FIFO memory type with *64* depth value).

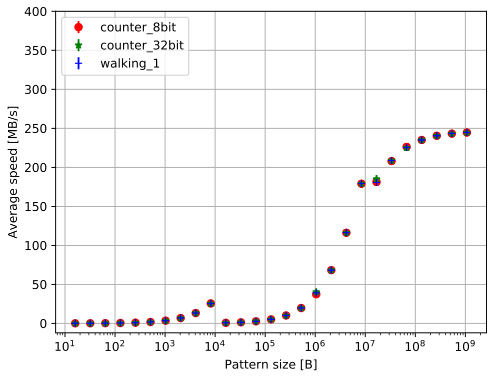


Figure 70. Speed results for data pattern impact subtest in *32bit write* mode (*blockram* FIFO memory type with *256* depth value).

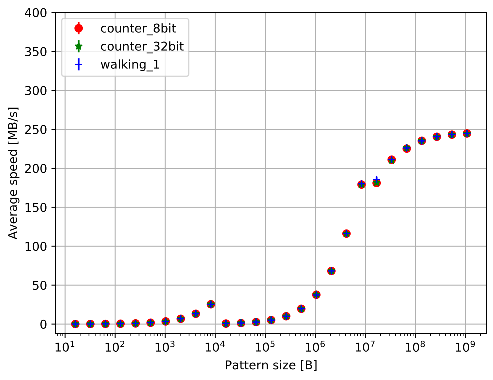


Figure 71. Speed results for data pattern impact subtest in *32bit write* mode (*blockram* FIFO memory type with *1024* depth value).

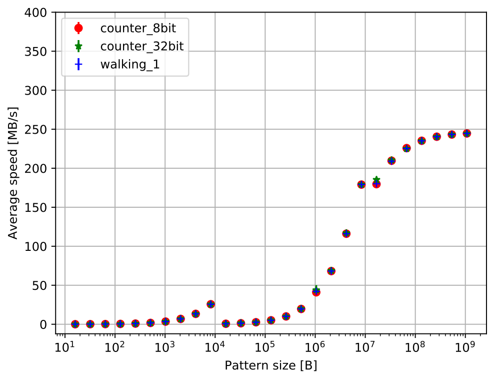


Figure 72. Speed results for data pattern impact subtest in *32bit write* mode (*blockram* FIFO memory type with *2048* depth value).

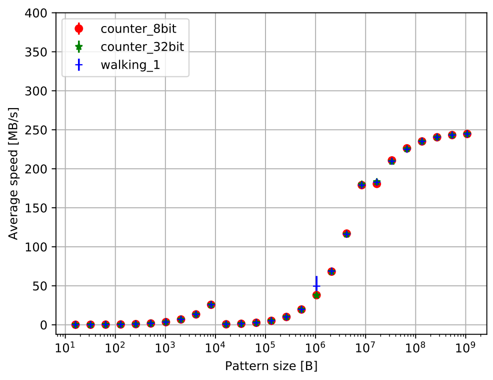


Figure 73. Speed results for data pattern impact subtest in *32bit write* mode (*distributedram* FIFO memory type with *16* depth value).

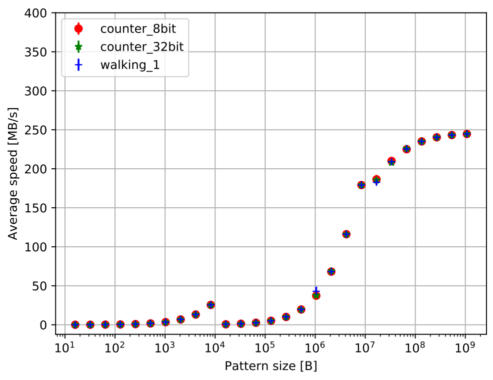


Figure 74. Speed results for data pattern impact subtest in *32bit write* mode (*distributedram* FIFO memory type with *64* depth value).

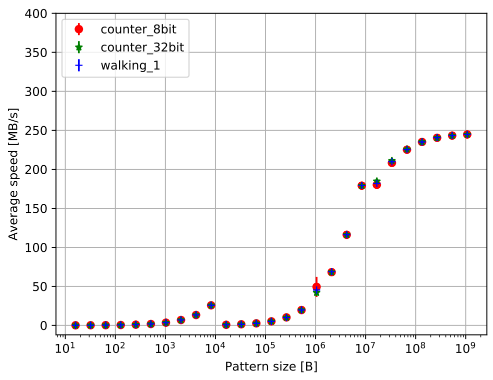


Figure 75. Speed results for data pattern impact subtest in *32bit write* mode (*distributedram* FIFO memory type with *256* depth value).

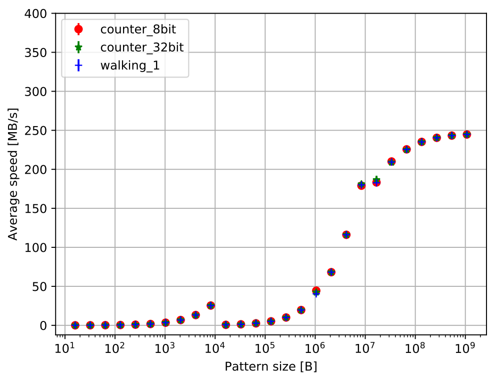


Figure 76. Speed results for data pattern impact subtest in *32bit write* mode (*distributedram* FIFO memory type with *1024* depth value).

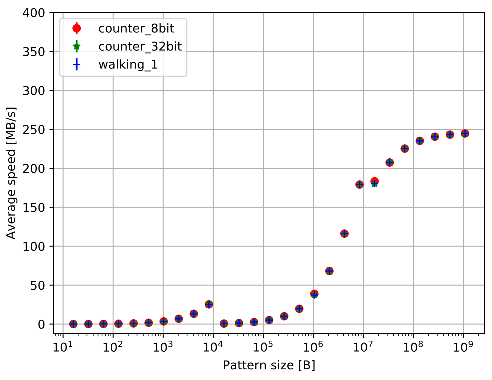


Figure 77. Speed results for data pattern impact subtest in *32bit write* mode (*distributedram* FIFO memory type with *2048* depth value).

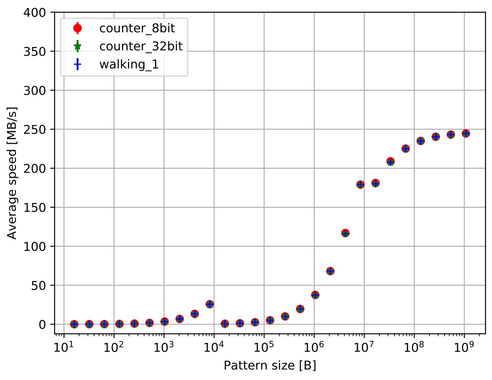


Figure 78. Speed results for data pattern impact subtest in *32bit write* mode (*shiftregister* FIFO memory type with *16* depth value).

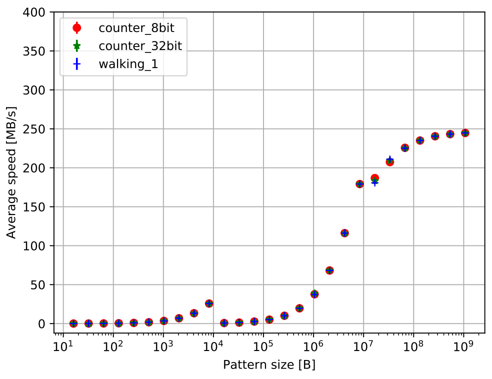


Figure 79. Speed results for data pattern impact subtest in *32bit write* mode (*shiftregister* FIFO memory type with *64* depth value).

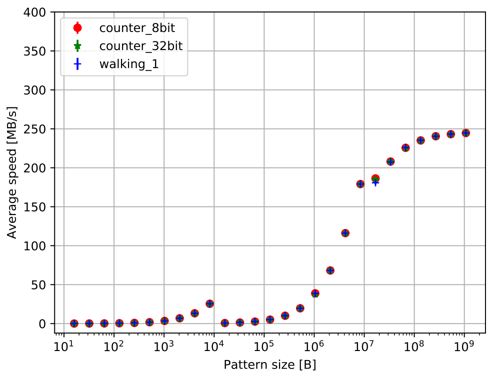


Figure 80. Speed results for data pattern impact subtest in *32bit write* mode (*shiftregister* FIFO memory type with *256* depth value).

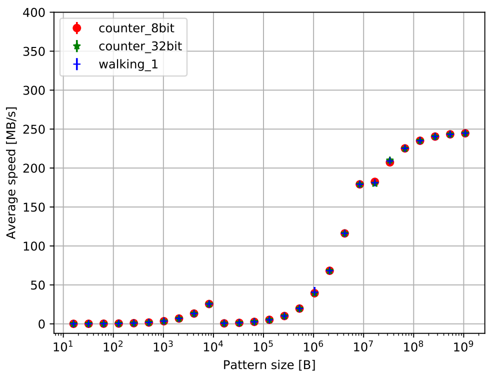


Figure 81. Speed results for data pattern impact subtest in *32bit write* mode (*shiftregister* FIFO memory type with *1024* depth value).

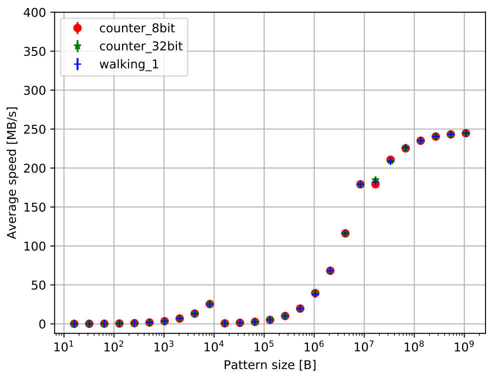


Figure 82. Speed results for data pattern impact subtest in *32bit write* mode (*shiftregister* FIFO memory type with *2048* depth value).

Table 9. *The fastest pattern types (values in square brackets are in MB/s) compared between depth values (32bit mode, read direction, blockram FIFO memory type).*

Pattern size [B]	**16**	**64**	**256**	**1024**	**2048**
16	counter_32bit [*0.065*]	counter_32bit [*0.066*]	counter_32bit [*0.066*]	counter_32bit [*0.066*]	walking_1 [*0.066*]
32	counter_8bit [*0.131*]	walking_1 [*0.132*]	counter_32bit [*0.132*]	counter_8bit [*0.132*]	walking_1 [*0.131*]
64	walking_1 [*0.262*]	walking_1 [*0.264*]	walking_1 [*0.263*]	walking_1 [*0.263*]	counter_8bit [*0.262*]
128	counter_8bit [*0.523*]	counter_8bit [*0.528*]	counter_32bit [*0.527*]	walking_1 [*0.528*]	counter_32bit [*0.526*]
256	counter_8bit [*1.044*]	walking_1 [*1.054*]	counter_8bit [*1.053*]	counter_8bit [*1.054*]	counter_8bit [*1.049*]
512	counter_8bit [*2.087*]	counter_8bit [*2.111*]	counter_8bit [*2.105*]	counter_32bit [*2.106*]	counter_8bit [*2.104*]
1024	counter_32bit [*4.167*]	counter_32bit [*4.207*]	walking_1 [*4.203*]	walking_1 [*4.194*]	walking_1 [*4.195*]
2048	walking_1 [*8.189*]	counter_8bit [*8.249*]	counter_8bit [*8.239*]	walking_1 [*8.259*]	counter_32bit [*8.244*]
4096	walking_1 [*15.820*]	walking_1 [*15.918*]	counter_8bit [*15.909*]	walking_1 [*15.928*]	counter_8bit [*15.885*]
8192	counter_8bit [*29.411*]	counter_32bit [*29.643*]	counter_32bit [*29.630*]	counter_32bit [*29.602*]	walking_1 [*29.579*]
16384	counter_32bit [*51.933*]	walking_1 [*51.995*]	walking_1 [*51.965*]	counter_32bit [*51.930*]	walking_1 [*51.941*]
32768	counter_32bit [*91.284*]	walking_1 [*91.600*]	walking_1 [*91.875*]	counter_32bit [*91.613*]	counter_8bit [*91.806*]
65536	counter_32bit [*148.171*]	walking_1 [*148.682*]	walking_1 [*148.762*]	counter_32bit [*148.776*]	walking_1 [*148.415*]
131072	walking_1 [*216.013*]	counter_8bit [*216.011*]	counter_8bit [*215.799*]	counter_32bit [*215.868*]	counter_32bit [*215.507*]
262144	counter_8bit [*278.685*]	walking_1 [*279.071*]	walking_1 [*279.096*]	walking_1 [*278.931*]	walking_1 [*279.044*]
524288	walking_1 [*326.896*]	walking_1 [*327.092*]	counter_32bit [*327.203*]	counter_8bit [*327.099*]	counter_32bit [*327.188*]
1048576	counter_32bit [*353.151*]	counter_8bit [*353.133*]	counter_8bit [*353.209*]	walking_1 [*353.194*]	walking_1 [*352.906*]
2097152	walking_1 [*374.329*]	counter_32bit [*374.437*]	walking_1 [*374.390*]	walking_1 [*374.452*]	counter_8bit [*374.021*]
4194304	walking_1 [*384.840*]	counter_8bit [*384.869*]	counter_32bit [*384.872*]	walking_1 [*384.853*]	counter_32bit [*384.708*]
8388608	counter_8bit [*390.351*]	walking_1 [*390.332*]	walking_1 [*390.320*]	walking_1 [*390.256*]	counter_32bit [*390.272*]
16777216	counter_32bit [*391.590*]	counter_8bit [*391.578*]	walking_1 [*391.580*]	counter_32bit [*391.590*]	counter_8bit [*391.556*]
33554432	walking_1 [*392.870*]	counter_32bit [*392.871*]	walking_1 [*392.860*]	counter_8bit [*392.854*]	counter_8bit [*392.811*]
67108864	counter_32bit [*393.518*]	walking_1 [*393.515*]	counter_8bit [*393.514*]	counter_32bit [*393.506*]	counter_32bit [*393.494*]
134217728	walking_1 [*393.826*]	counter_8bit [*393.833*]	counter_32bit [*393.817*]	counter_8bit [*393.835*]	walking_1 [*393.827*]
268435456	walking_1 [*393.971*]	walking_1 [*393.998*]	counter_32bit [*393.967*]	counter_32bit [*394.002*]	counter_8bit [*393.974*]
536870912	walking_1 [*394.067*]	counter_8bit [*394.080*]	walking_1 [*394.076*]	walking_1 [*394.069*]	counter_32bit [*394.078*]
1073741824	counter_8bit [*394.109*]	counter_8bit [*394.103*]	walking_1 [*394.104*]	walking_1 [*394.099*]	walking_1 [*394.096*]
Most frequent parameter	walking_1	walking_1	walking_1	walking_1	walking_1

Table 10. *The fastest pattern types (values in square brackets are in MB/s) compared between depth values (32bit mode, read direction, distributedram FIFO memory type).*

Pattern size [B]	**16**	**64**	**256**	**1024**	**2048**
16	counter_32bit [*0.066*]	walking_1 [*0.066*]	counter_8bit [*0.065*]	counter_32bit [*0.065*]	counter_8bit [*0.066*]
32	counter_8bit [*0.131*]	counter_32bit [*0.132*]	counter_8bit [*0.131*]	counter_8bit [*0.131*]	counter_8bit [*0.132*]
64	walking_1 [*0.262*]	counter_8bit [*0.263*]	counter_8bit [*0.260*]	counter_8bit [*0.261*]	counter_32bit [*0.264*]
128	counter_32bit [*0.523*]	walking_1 [*0.526*]	counter_32bit [*0.527*]	counter_32bit [*0.522*]	counter_8bit [*0.526*]
256	counter_8bit [*1.046*]	walking_1 [*1.053*]	counter_32bit [*1.057*]	walking_1 [*1.046*]	counter_8bit [*1.053*]
512	walking_1 [*2.089*]	counter_32bit [*2.108*]	counter_8bit [*2.103*]	counter_32bit [*2.086*]	counter_8bit [*2.107*]
1024	counter_32bit [*4.179*]	counter_8bit [*4.198*]	walking_1 [*4.208*]	counter_8bit [*4.135*]	counter_32bit [*4.204*]
2048	counter_8bit [*8.245*]	walking_1 [*8.239*]	counter_8bit [*8.249*]	walking_1 [*8.174*]	walking_1 [*8.246*]
4096	walking_1 [*15.814*]	counter_32bit [*15.884*]	counter_32bit [*15.903*]	counter_8bit [*15.806*]	counter_32bit [*15.919*]
8192	counter_32bit [*29.400*]	walking_1 [*29.587*]	walking_1 [*29.602*]	counter_32bit [*29.502*]	walking_1 [*29.614*]
16384	counter_32bit [*51.544*]	counter_8bit [*51.938*]	counter_32bit [*51.693*]	counter_32bit [*51.387*]	walking_1 [*51.898*]
32768	counter_8bit [*91.047*]	counter_32bit [*91.664*]	counter_8bit [*91.322*]	walking_1 [*90.921*]	counter_32bit [*91.617*]
65536	counter_8bit [*148.380*]	counter_32bit [*148.720*]	walking_1 [*148.760*]	counter_32bit [*147.826*]	counter_8bit [*148.649*]
131072	walking_1 [*215.960*]	walking_1 [*215.778*]	walking_1 [*215.921*]	counter_8bit [*215.599*]	counter_32bit [*215.991*]
262144	walking_1 [*278.996*]	counter_8bit [*278.908*]	walking_1 [*279.031*]	counter_32bit [*278.172*]	walking_1 [*279.068*]
524288	counter_8bit [*327.099*]	counter_8bit [*327.086*]	counter_8bit [*327.057*]	counter_32bit [*326.884*]	counter_8bit [*327.229*]
1048576	counter_32bit [*353.030*]	counter_32bit [*352.868*]	walking_1 [*353.158*]	walking_1 [*352.287*]	walking_1 [*353.215*]
2097152	walking_1 [*374.334*]	walking_1 [*374.108*]	walking_1 [*374.457*]	counter_8bit [*374.013*]	counter_8bit [*374.409*]
4194304	counter_32bit [*384.781*]	counter_32bit [*384.698*]	counter_8bit [*384.898*]	counter_8bit [*384.619*]	walking_1 [*384.868*]
8388608	counter_32bit [*390.295*]	counter_8bit [*390.269*]	counter_8bit [*390.332*]	counter_32bit [*390.199*]	counter_8bit [*390.351*]
16777216	counter_32bit [*391.569*]	counter_8bit [*391.540*]	counter_8bit [*391.590*]	counter_32bit [*391.567*]	walking_1 [*391.584*]
33554432	counter_8bit [*392.849*]	counter_32bit [*392.829*]	walking_1 [*392.867*]	counter_8bit [*392.853*]	counter_8bit [*392.873*]
67108864	walking_1 [*393.508*]	counter_32bit [*393.513*]	walking_1 [*393.507*]	counter_8bit [*393.507*]	walking_1 [*393.518*]
134217728	walking_1 [*393.838*]	walking_1 [*393.833*]	counter_32bit [*393.841*]	counter_8bit [*393.810*]	walking_1 [*393.818*]
268435456	counter_32bit [*393.999*]	counter_8bit [*393.995*]	counter_8bit [*393.984*]	walking_1 [*393.996*]	walking_1 [*393.999*]
536870912	walking_1 [*394.064*]	counter_8bit [*394.066*]	counter_8bit [*394.070*]	counter_8bit [*394.074*]	walking_1 [*394.072*]
1073741824	counter_8bit [*394.106*]	counter_32bit [*394.102*]	counter_8bit [*394.097*]	counter_8bit [*394.103*]	counter_8bit [*394.101*]
Most frequent parameter	counter_32bit	counter_32bit	counter_8bit	counter_8bit	counter_8bit, walking_1

Table 11. *The fastest pattern types (values in square brackets are in MB/s) compared between depth values (32bit mode, read direction, shiftregister FIFO memory type).*

Pattern size [B]	**16**	**64**	**256**	**1024**	**2048**
16	counter_32bit [*0.065*]	counter_32bit [*0.065*]	walking_1 [*0.066*]	counter_32bit [*0.066*]	counter_32bit [*0.066*]
32	counter_32bit [*0.123*]	counter_8bit [*0.131*]	counter_32bit [*0.132*]	counter_8bit [*0.132*]	counter_8bit [*0.132*]
64	counter_8bit [*0.256*]	counter_32bit [*0.255*]	counter_32bit [*0.263*]	walking_1 [*0.263*]	counter_8bit [*0.264*]
128	counter_8bit [*0.521*]	counter_8bit [*0.526*]	counter_8bit [*0.527*]	counter_8bit [*0.527*]	counter_32bit [*0.528*]
256	counter_32bit [*1.040*]	counter_32bit [*0.982*]	counter_32bit [*1.054*]	counter_32bit [*1.054*]	walking_1 [*1.054*]
512	counter_8bit [*1.986*]	counter_8bit [*1.957*]	counter_32bit [*2.107*]	walking_1 [*2.107*]	counter_32bit [*2.107*]
1024	walking_1 [*4.175*]	counter_32bit [*4.182*]	counter_8bit [*4.198*]	counter_8bit [*4.203*]	walking_1 [*4.210*]
2048	walking_1 [*7.981*]	walking_1 [*8.062*]	counter_8bit [*8.237*]	counter_32bit [*8.253*]	walking_1 [*8.256*]
4096	counter_32bit [*15.438*]	walking_1 [*15.705*]	counter_8bit [*15.919*]	walking_1 [*15.907*]	walking_1 [*15.853*]
8192	counter_8bit [*29.417*]	counter_32bit [*27.679*]	walking_1 [*29.657*]	walking_1 [*29.628*]	counter_32bit [*29.557*]
16384	counter_8bit [*50.380*]	walking_1 [*48.964*]	walking_1 [*51.920*]	counter_32bit [*51.991*]	counter_32bit [*51.976*]
32768	walking_1 [*84.201*]	counter_8bit [*91.571*]	walking_1 [*91.676*]	counter_8bit [*91.683*]	counter_8bit [*91.588*]
65536	counter_32bit [*138.324*]	counter_32bit [*148.110*]	counter_8bit [*148.781*]	counter_8bit [*148.700*]	counter_8bit [*148.745*]
131072	counter_32bit [*207.014*]	counter_8bit [*206.285*]	walking_1 [*215.823*]	counter_32bit [*215.740*]	walking_1 [*215.717*]
262144	walking_1 [*262.387*]	counter_32bit [*258.534*]	walking_1 [*279.032*]	counter_8bit [*278.995*]	walking_1 [*278.803*]
524288	counter_8bit [*312.172*]	walking_1 [*302.974*]	counter_8bit [*327.212*]	counter_8bit [*327.205*]	walking_1 [*327.069*]
1048576	walking_1 [*330.465*]	walking_1 [*331.175*]	counter_8bit [*353.100*]	counter_8bit [*353.185*]	walking_1 [*353.198*]
2097152	counter_8bit [*353.948*]	walking_1 [*347.914*]	counter_8bit [*374.337*]	counter_8bit [*374.327*]	counter_8bit [*374.468*]
4194304	counter_8bit [*359.897*]	counter_32bit [*357.630*]	counter_32bit [*384.704*]	walking_1 [*384.839*]	walking_1 [*384.881*]
8388608	walking_1 [*362.094*]	counter_8bit [*361.903*]	counter_32bit [*390.228*]	counter_32bit [*390.343*]	counter_32bit [*390.336*]
16777216	counter_8bit [*364.196*]	counter_32bit [*361.250*]	walking_1 [*391.583*]	counter_8bit [*391.585*]	counter_8bit [*391.584*]
33554432	walking_1 [*355.745*]	counter_32bit [*363.850*]	counter_32bit [*392.873*]	walking_1 [*392.869*]	counter_8bit [*392.864*]
67108864	counter_8bit [*355.445*]	counter_8bit [*369.825*]	counter_8bit [*393.516*]	counter_8bit [*393.514*]	counter_8bit [*393.504*]
134217728	walking_1 [*368.340*]	counter_8bit [*368.964*]	counter_8bit [*393.844*]	counter_32bit [*393.815*]	walking_1 [*393.829*]
268435456	counter_8bit [*369.640*]	counter_8bit [*367.055*]	walking_1 [*393.986*]	counter_8bit [*393.990*]	counter_32bit [*394.003*]
536870912	counter_32bit [*369.263*]	walking_1 [*388.025*]	counter_32bit [*394.071*]	walking_1 [*394.080*]	counter_8bit [*394.078*]
1073741824	counter_8bit [*366.695*]	counter_32bit [*394.096*]	walking_1 [*394.104*]	counter_32bit [*394.083*]	walking_1 [*394.106*]
Most frequent parameter	counter_8bit	counter_32bit	counter_8bit	counter_8bit	walking_1

Table 12. *The fastest pattern types (values in square brackets are in MB/s) compared between depth values (32bit mode, write direction, blockram FIFO memory type).*

Pattern size [B]	**16**	**64**	**256**	**1024**	**2048**
16	walking_1 [*0.054*]	counter_32bit [*0.054*]	walking_1 [*0.054*]	counter_32bit [*0.053*]	walking_1 [*0.054*]
32	walking_1 [*0.107*]	walking_1 [*0.108*]	counter_8bit [*0.107*]	counter_32bit [*0.107*]	walking_1 [*0.107*]
64	counter_8bit [*0.214*]	counter_8bit [*0.215*]	counter_32bit [*0.215*]	counter_32bit [*0.214*]	walking_1 [*0.215*]
128	walking_1 [*0.429*]	walking_1 [*0.430*]	counter_8bit [*0.428*]	counter_8bit [*0.428*]	counter_8bit [*0.430*]
256	counter_32bit [*0.856*]	walking_1 [*0.860*]	walking_1 [*0.856*]	counter_32bit [*0.855*]	counter_32bit [*0.860*]
512	counter_8bit [*1.711*]	counter_8bit [*1.718*]	walking_1 [*1.706*]	walking_1 [*1.707*]	counter_32bit [*1.710*]
1024	counter_32bit [*3.420*]	counter_8bit [*3.439*]	counter_8bit [*3.432*]	walking_1 [*3.418*]	walking_1 [*3.437*]
2048	walking_1 [*6.815*]	counter_32bit [*6.842*]	walking_1 [*6.823*]	counter_32bit [*6.809*]	counter_8bit [*6.847*]
4096	counter_32bit [*13.233*]	counter_32bit [*13.355*]	counter_32bit [*13.226*]	counter_8bit [*13.223*]	walking_1 [*13.343*]
8192	walking_1 [*25.494*]	walking_1 [*25.717*]	walking_1 [*25.501*]	counter_32bit [*25.500*]	counter_8bit [*25.663*]
16384	counter_32bit [*0.664*]	counter_8bit [*0.689*]	counter_32bit [*0.673*]	counter_32bit [*0.650*]	counter_8bit [*0.650*]
32768	counter_8bit [*1.312*]	counter_32bit [*1.330*]	counter_32bit [*1.298*]	counter_32bit [*1.344*]	counter_32bit [*1.598*]
65536	counter_32bit [*2.616*]	counter_8bit [*2.561*]	counter_8bit [*2.589*]	counter_32bit [*2.589*]	counter_32bit [*2.616*]
131072	walking_1 [*5.093*]	counter_32bit [*5.203*]	counter_32bit [*5.093*]	counter_8bit [*5.148*]	counter_8bit [*5.091*]
262144	walking_1 [*10.062*]	walking_1 [*10.061*]	walking_1 [*10.061*]	walking_1 [*10.059*]	counter_32bit [*10.059*]
524288	counter_8bit [*19.833*]	counter_8bit [*19.627*]	counter_32bit [*19.628*]	walking_1 [*19.629*]	counter_32bit [*19.829*]
1048576	counter_32bit [*38.390*]	counter_32bit [*38.390*]	counter_32bit [*40.090*]	counter_32bit [*38.027*]	counter_32bit [*43.821*]
2097152	walking_1 [*68.187*]	walking_1 [*68.177*]	counter_8bit [*68.175*]	counter_8bit [*68.181*]	counter_32bit [*68.766*]
4194304	counter_8bit [*116.133*]	walking_1 [*116.129*]	walking_1 [*116.124*]	counter_32bit [*116.137*]	counter_32bit [*116.953*]
8388608	counter_32bit [*179.115*]	counter_8bit [*179.112*]	counter_32bit [*179.101*]	walking_1 [*180.109*]	walking_1 [*179.122*]
16777216	walking_1 [*187.065*]	counter_32bit [*181.690*]	counter_32bit [*184.868*]	walking_1 [*185.761*]	counter_32bit [*185.262*]
33554432	counter_32bit [*210.333*]	walking_1 [*210.329*]	walking_1 [*208.665*]	walking_1 [*211.339*]	counter_32bit [*210.343*]
67108864	counter_32bit [*225.429*]	counter_8bit [*225.634*]	counter_8bit [*226.216*]	counter_32bit [*226.194*]	counter_8bit [*225.794*]
134217728	counter_32bit [*235.169*]	counter_8bit [*235.332*]	walking_1 [*235.078*]	counter_8bit [*235.274*]	counter_8bit [*235.226*]
268435456	counter_32bit [*240.482*]	walking_1 [*240.438*]	counter_32bit [*240.484*]	walking_1 [*240.437*]	counter_8bit [*240.433*]
536870912	walking_1 [*243.259*]	counter_32bit [*243.182*]	counter_32bit [*243.278*]	counter_32bit [*243.244*]	walking_1 [*243.283*]
1073741824	counter_32bit [*244.689*]	counter_32bit [*244.686*]	walking_1 [*244.698*]	counter_8bit [*244.690*]	counter_8bit [*244.643*]
Most frequent parameter	counter_32bit	counter_32bit, walking_1, counter_8bit	counter_32bit	counter_32bit	counter_32bit

Table 13. *The fastest pattern types (values in square brackets are in MB/s) compared between depth values (32bit mode, write direction, distributedram FIFO memory type).*

Pattern size [B]	**16**	**64**	**256**	**1024**	**2048**
16	walking_1 [*0.054*]	counter_8bit [*0.053*]	counter_32bit [*0.054*]	walking_1 [*0.053*]	counter_32bit [*0.053*]
32	counter_8bit [*0.107*]	counter_8bit [*0.107*]	counter_8bit [*0.108*]	counter_32bit [*0.107*]	counter_8bit [*0.107*]
64	walking_1 [*0.215*]	walking_1 [*0.214*]	counter_32bit [*0.215*]	walking_1 [*0.214*]	counter_8bit [*0.214*]
128	counter_32bit [*0.430*]	walking_1 [*0.428*]	counter_32bit [*0.430*]	walking_1 [*0.429*]	counter_8bit [*0.426*]
256	counter_8bit [*0.860*]	counter_8bit [*0.855*]	counter_8bit [*0.859*]	counter_8bit [*0.860*]	counter_8bit [*0.851*]
512	walking_1 [*1.717*]	counter_8bit [*1.709*]	counter_8bit [*1.716*]	counter_8bit [*1.705*]	walking_1 [*1.702*]
1024	counter_8bit [*3.439*]	counter_8bit [*3.410*]	walking_1 [*3.437*]	walking_1 [*3.420*]	counter_8bit [*3.411*]
2048	counter_32bit [*6.853*]	counter_8bit [*6.819*]	counter_8bit [*6.847*]	counter_8bit [*6.817*]	counter_32bit [*6.819*]
4096	counter_32bit [*13.352*]	walking_1 [*13.201*]	counter_32bit [*13.351*]	walking_1 [*13.245*]	counter_8bit [*13.224*]
8192	counter_8bit [*25.680*]	walking_1 [*25.502*]	walking_1 [*25.706*]	counter_32bit [*25.478*]	counter_32bit [*25.490*]
16384	walking_1 [*0.710*]	walking_1 [*0.678*]	counter_32bit [*0.709*]	walking_1 [*0.659*]	counter_32bit [*0.657*]
32768	counter_32bit [*1.312*]	walking_1 [*1.313*]	counter_8bit [*1.358*]	counter_32bit [*1.330*]	counter_8bit [*1.299*]
65536	counter_8bit [*2.616*]	counter_8bit [*2.617*]	walking_1 [*2.588*]	counter_32bit [*2.800*]	walking_1 [*2.623*]
131072	walking_1 [*5.092*]	counter_8bit [*5.147*]	counter_32bit [*5.093*]	counter_8bit [*5.145*]	counter_8bit [*5.147*]
262144	counter_8bit [*10.060*]	walking_1 [*10.168*]	counter_8bit [*10.166*]	counter_32bit [*10.061*]	counter_8bit [*10.060*]
524288	walking_1 [*19.627*]	walking_1 [*19.629*]	counter_8bit [*19.627*]	counter_32bit [*19.627*]	counter_32bit [*19.627*]
1048576	walking_1 [*49.550*]	walking_1 [*42.967*]	counter_8bit [*49.369*]	counter_8bit [*44.433*]	counter_8bit [*38.913*]
2097152	walking_1 [*68.748*]	counter_32bit [*68.772*]	walking_1 [*68.769*]	counter_32bit [*68.161*]	counter_32bit [*68.187*]
4194304	counter_8bit [*116.909*]	counter_8bit [*116.139*]	counter_32bit [*116.109*]	walking_1 [*116.088*]	walking_1 [*116.151*]
8388608	counter_32bit [*180.048*]	counter_8bit [*179.136*]	walking_1 [*179.137*]	counter_32bit [*180.997*]	counter_32bit [*179.112*]
16777216	counter_32bit [*183.197*]	counter_8bit [*186.456*]	counter_32bit [*184.733*]	counter_32bit [*186.452*]	counter_8bit [*183.216*]
33554432	counter_8bit [*210.713*]	counter_8bit [*210.001*]	counter_32bit [*211.014*]	counter_8bit [*210.047*]	counter_32bit [*208.675*]
67108864	counter_8bit [*226.209*]	counter_32bit [*225.831*]	counter_32bit [*225.780*]	counter_8bit [*225.624*]	counter_32bit [*225.625*]
134217728	walking_1 [*235.081*]	counter_32bit [*235.082*]	counter_8bit [*235.085*]	counter_8bit [*235.184*]	counter_32bit [*235.271*]
268435456	counter_32bit [*240.475*]	counter_32bit [*240.480*]	walking_1 [*240.535*]	counter_32bit [*240.495*]	counter_8bit [*240.492*]
536870912	counter_8bit [*243.219*]	counter_32bit [*243.271*]	counter_8bit [*243.252*]	counter_8bit [*243.238*]	walking_1 [*243.277*]
1073741824	counter_32bit [*244.689*]	counter_32bit [*244.765*]	walking_1 [*244.711*]	counter_32bit [*244.693*]	counter_32bit [*244.664*]
Most frequent parameter	counter_8bit	counter_8bit	counter_32bit, counter_8bit	counter_32bit	counter_8bit

Table 14. *The fastest pattern types (values in square brackets are in MB/s) compared between depth values (32bit mode, write direction, shiftregister FIFO memory type).*

Pattern size [B]	**16**	**64**	**256**	**1024**	**2048**
16	counter_32bit *[0.054]*	counter_32bit *[0.054]*	walking_1 *[0.054]*	walking_1 *[0.054]*	walking_1 *[0.054]*
32	counter_32bit *[0.107]*	counter_8bit *[0.108]*	counter_32bit *[0.107]*	counter_32bit *[0.107]*	counter_32bit *[0.107]*
64	walking_1 *[0.215]*	counter_32bit *[0.215]*	counter_32bit *[0.214]*	counter_32bit *[0.214]*	walking_1 *[0.214]*
128	walking_1 *[0.430]*	counter_32bit *[0.430]*	counter_8bit *[0.429]*	counter_8bit *[0.428]*	walking_1 *[0.429]*
256	counter_32bit *[0.860]*	walking_1 *[0.861]*	counter_32bit *[0.856]*	walking_1 *[0.855]*	counter_8bit *[0.859]*
512	counter_8bit *[1.718]*	counter_8bit *[1.718]*	counter_8bit *[1.710]*	walking_1 *[1.706]*	counter_8bit *[1.709]*
1024	walking_1 *[3.439]*	counter_8bit *[3.434]*	counter_8bit *[3.418]*	walking_1 *[3.425]*	counter_32bit *[3.418]*
2048	counter_32bit *[6.851]*	counter_8bit *[6.834]*	counter_32bit *[6.810]*	walking_1 *[6.854]*	counter_8bit *[6.814]*
4096	counter_8bit *[13.354]*	counter_8bit *[13.288]*	counter_8bit *[13.223]*	counter_8bit *[13.221]*	walking_1 *[13.212]*
8192	counter_8bit *[25.712]*	walking_1 *[25.740]*	counter_32bit *[25.460]*	counter_8bit *[25.466]*	counter_32bit *[25.479]*
16384	walking_1 *[0.714]*	counter_8bit *[0.696]*	counter_32bit *[0.666]*	walking_1 *[0.712]*	counter_32bit *[0.680]*
32768	walking_1 *[1.372]*	walking_1 *[1.298]*	walking_1 *[1.534]*	counter_32bit *[1.284]*	counter_8bit *[1.344]*
65536	walking_1 *[2.561]*	counter_8bit *[2.616]*	walking_1 *[2.651]*	counter_32bit *[2.644]*	walking_1 *[2.561]*
131072	counter_32bit *[5.147]*	counter_32bit *[5.147]*	counter_32bit *[5.092]*	counter_8bit *[5.201]*	counter_32bit *[5.146]*
262144	counter_32bit *[10.061]*	counter_32bit *[10.061]*	walking_1 *[10.168]*	walking_1 *[10.060]*	counter_8bit *[10.060]*
524288	walking_1 *[19.628]*	counter_8bit *[19.629]*	counter_8bit *[19.629]*	walking_1 *[19.629]*	counter_32bit *[19.629]*
1048576	counter_8bit *[37.657]*	counter_32bit *[38.835]*	counter_8bit *[38.835]*	walking_1 *[40.664]*	counter_32bit *[40.008]*
2097152	counter_8bit *[68.168]*	walking_1 *[68.186]*	counter_8bit *[68.182]*	counter_32bit *[68.177]*	counter_8bit *[68.181]*
4194304	counter_8bit *[116.976]*	walking_1 *[116.135]*	counter_32bit *[116.132]*	counter_8bit *[116.117]*	walking_1 *[116.125]*
8388608	counter_8bit *[179.072]*	walking_1 *[179.114]*	counter_32bit *[179.111]*	counter_32bit *[179.094]*	walking_1 *[179.116]*
16777216	counter_8bit *[181.116]*	counter_8bit *[186.764]*	counter_8bit *[186.337]*	counter_8bit *[182.277]*	counter_32bit *[184.629]*
33554432	counter_8bit *[208.940]*	walking_1 *[211.025]*	counter_32bit *[208.022]*	counter_32bit *[209.960]*	counter_8bit *[210.642]*
67108864	counter_32bit *[225.372]*	counter_8bit *[225.626]*	walking_1 *[226.014]*	walking_1 *[225.241]*	walking_1 *[226.376]*
134217728	counter_32bit *[235.114]*	counter_32bit *[235.218]*	walking_1 *[235.350]*	counter_32bit *[235.085]*	walking_1 *[235.214]*
268435456	walking_1 *[240.413]*	counter_8bit *[240.449]*	counter_8bit *[240.481]*	counter_32bit *[240.430]*	walking_1 *[240.523]*
536870912	counter_8bit *[243.190]*	counter_32bit *[243.219]*	walking_1 *[243.259]*	counter_8bit *[243.279]*	counter_32bit *[243.272]*
1073741824	counter_8bit *[244.624]*	walking_1 *[244.705]*	counter_32bit *[244.756]*	walking_1 *[244.709]*	counter_8bit *[244.697]*
Most frequent parameter	counter_8bit	counter_8bit	counter_32bit	walking_1	walking_1

A.2. Graphical presentation of results from the FIFO memory type impact investigation subtest

A.2.1. nonsym mode part of the subtest

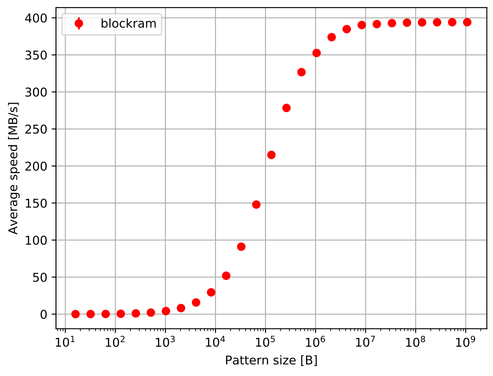


Figure 83. Speed results for FIFO memory type impact subtest in *nonsym read* mode (*16* FIFO depth value and *counter_8bit* pattern type).

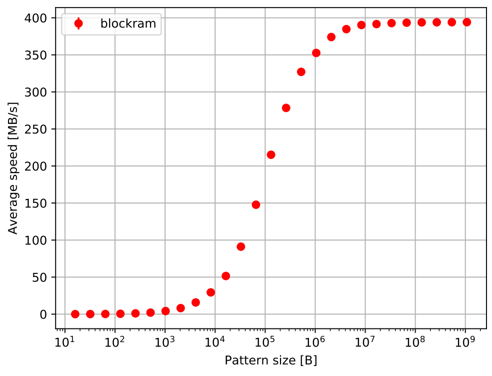


Figure 84. Speed results for FIFO memory type impact subtest in *nonsym read* mode (*16* FIFO depth value and *counter_32bit* pattern type).

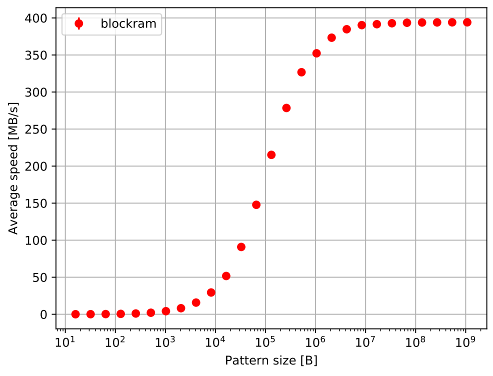


Figure 85. Speed results for FIFO memory type impact subtest in *nonsym read* mode (*16* FIFO depth value and *walking_1* pattern type).

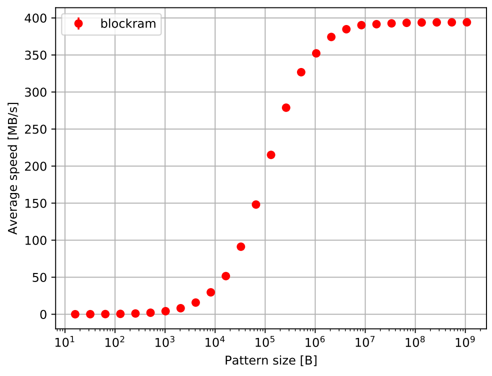


Figure 86. Speed results for FIFO memory type impact subtest in *nonsym read* mode (*16* FIFO depth value and *asic* pattern type).

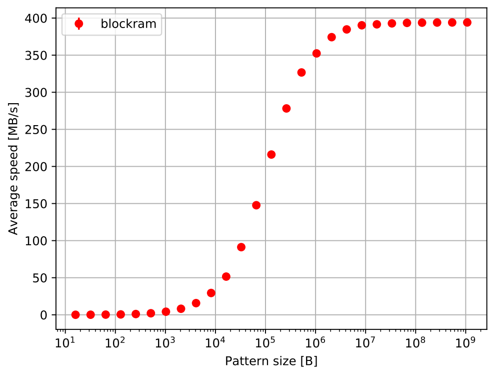


Figure 87. Speed results for FIFO memory type impact subtest in *nonsym read* mode (*64* FIFO depth value and *counter_8bit* pattern type).

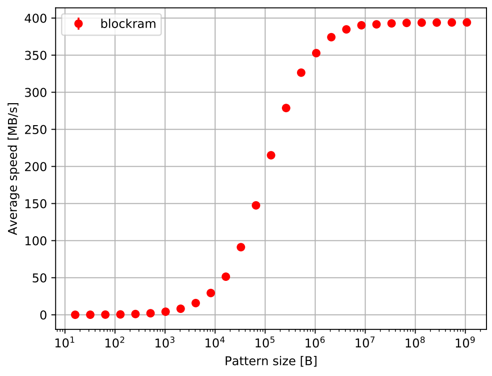


Figure 88. Speed results for FIFO memory type impact subtest in *nonsym read* mode (*64* FIFO depth value and *counter_32bit* pattern type).

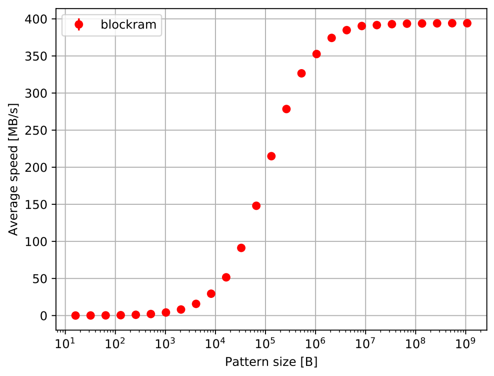


Figure 89. Speed results for FIFO memory type impact subtest in *nonsym read* mode (*64* FIFO depth value and *walking_1* pattern type).

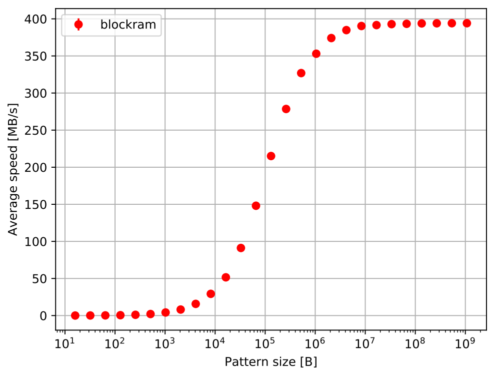


Figure 90. Speed results for FIFO memory type impact subtest in *nonsym read* mode (*64* FIFO depth value and *asic* pattern type).

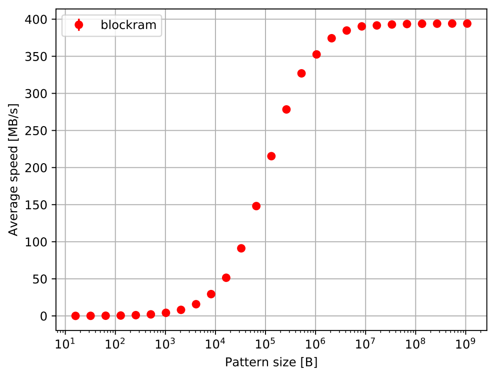


Figure 91. Speed results for FIFO memory type impact subtest in *nonsym read* mode (*256* FIFO depth value and *counter_8bit* pattern type).

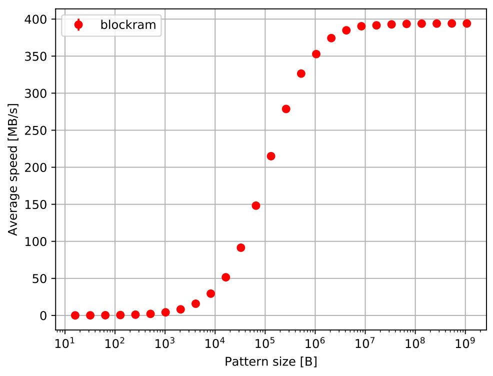


Figure 92. Speed results for FIFO memory type impact subtest in *nonsym read* mode (*256* FIFO depth value and *counter_32bit* pattern type).

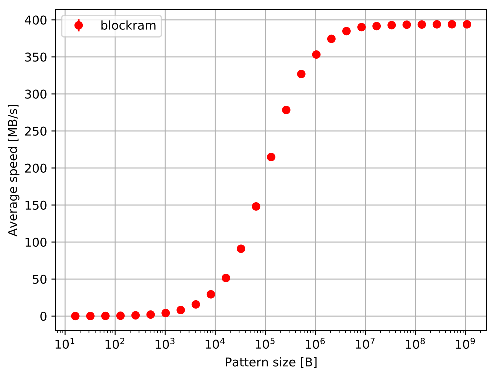


Figure 93. Speed results for FIFO memory type impact subtest in *nonsym read* mode (*256* FIFO depth value and *walking_1* pattern type).

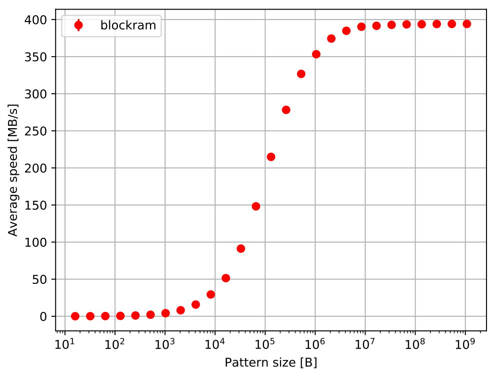


Figure 94. Speed results for FIFO memory type impact subtest in *nonsym read* mode (*256* FIFO depth value and *asic* pattern type).

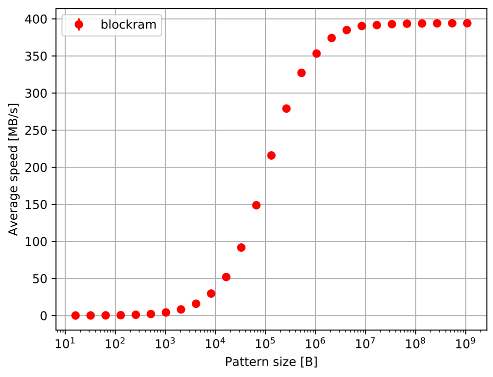


Figure 95. Speed results for FIFO memory type impact subtest in *nonsym read* mode (*1024* FIFO depth value and *counter_8bit* pattern type).

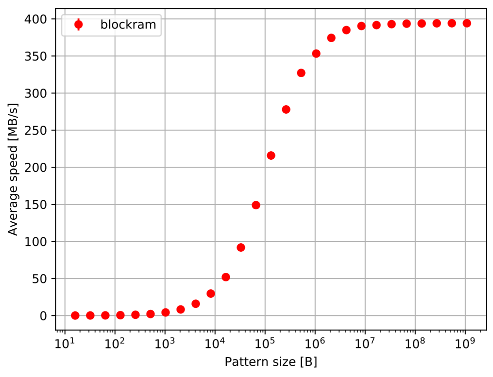


Figure 96. Speed results for FIFO memory type impact subtest in *nonsym read* mode (*1024* FIFO depth value and *counter_32bit* pattern type).

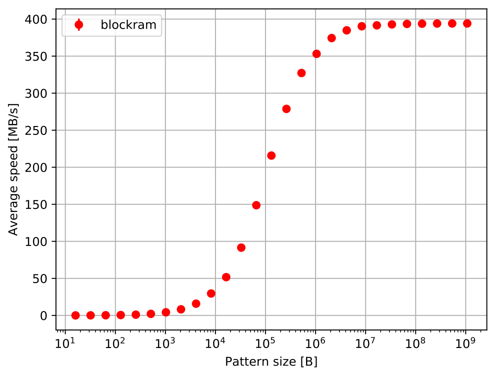


Figure 97. Speed results for FIFO memory type impact subtest in *nonsym read* mode (*1024* FIFO depth value and *walking_1* pattern type).

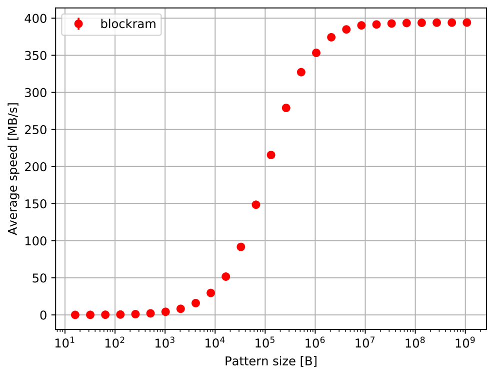


Figure 98. Speed results for FIFO memory type impact subtest in *nonsym read* mode (*1024* FIFO depth value and *asic* pattern type).

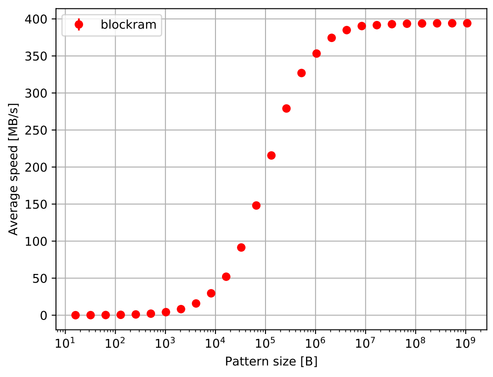


Figure 99. Speed results for FIFO memory type impact subtest in *nonsym read* mode (*2048* FIFO depth value and *counter_8bit* pattern type).

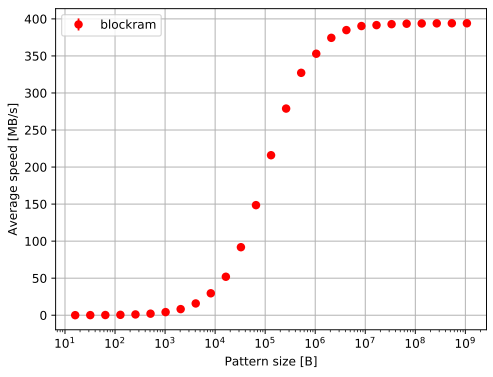


Figure 100. Speed results for FIFO memory type impact subtest in *nonsym read* mode (*2048* FIFO depth value and *counter_32bit* pattern type).

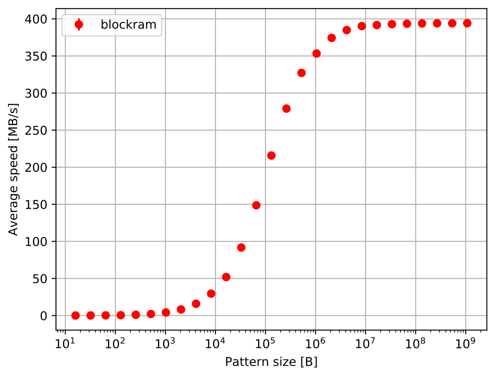


Figure 101. Speed results for FIFO memory type impact subtest in *nonsym read* mode (*2048* FIFO depth value and *walking_1* pattern type).

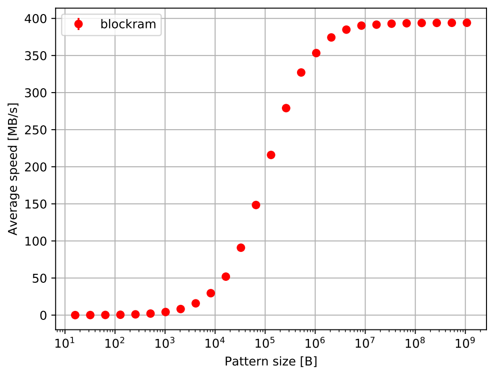


Figure 102. Speed results for FIFO memory type impact subtest in *nonsym read* mode (*2048* FIFO depth value and *asic* pattern type).

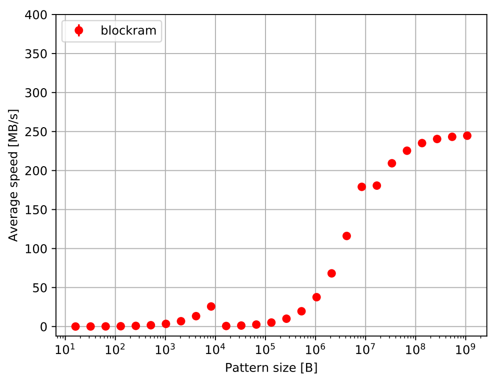


Figure 103. Speed results for FIFO memory type impact subtest in *nonsym write* mode (*32* FIFO depth value and *counter_8bit* pattern type).

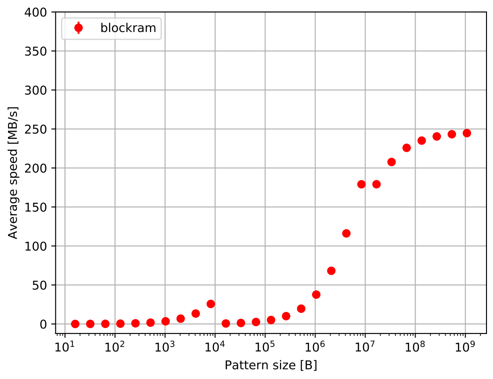


Figure 104. Speed results for FIFO memory type impact subtest in *nonsym write* mode (*32* FIFO depth value and *counter_32bit* pattern type).

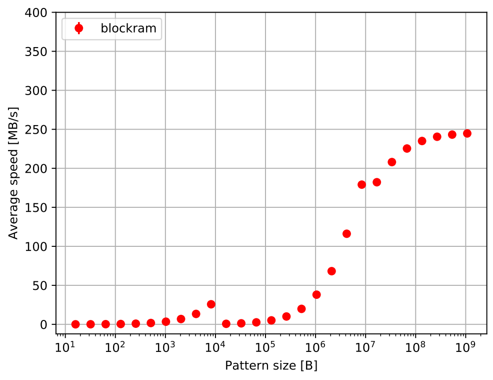


Figure 105. Speed results for FIFO memory type impact subtest in *nonsym write* mode (*32* FIFO depth value and *walking_1* pattern type).

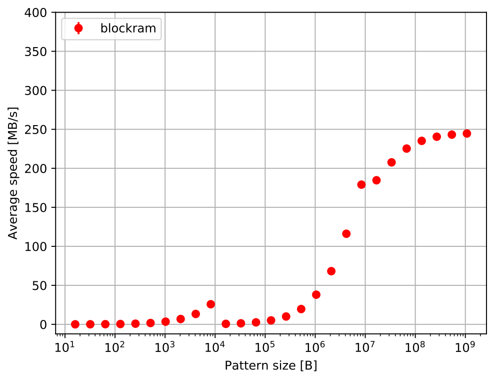


Figure 106. Speed results for FIFO memory type impact subtest in *nonsym write* mode (*32* FIFO depth value and *asic* pattern type).

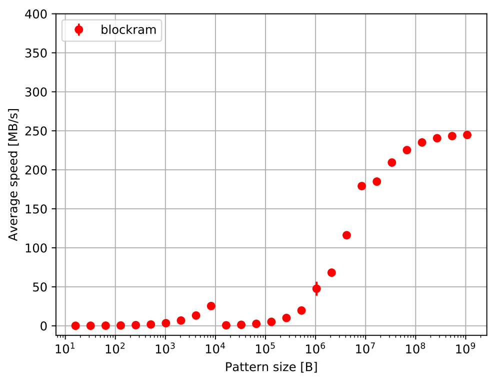


Figure 107. Speed results for FIFO memory type impact subtest in *nonsym write* mode (*64* FIFO depth value and *counter_8bit* pattern type).

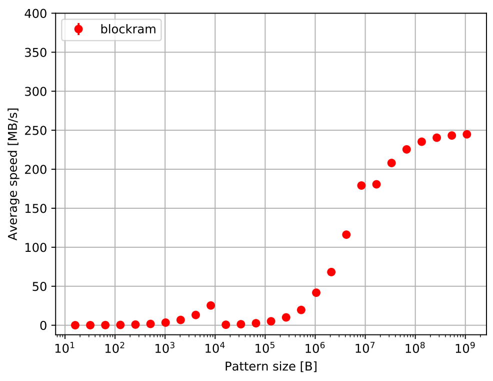


Figure 108. Speed results for FIFO memory type impact subtest in *nonsym write* mode (*64* FIFO depth value and *counter_32bit* pattern type).

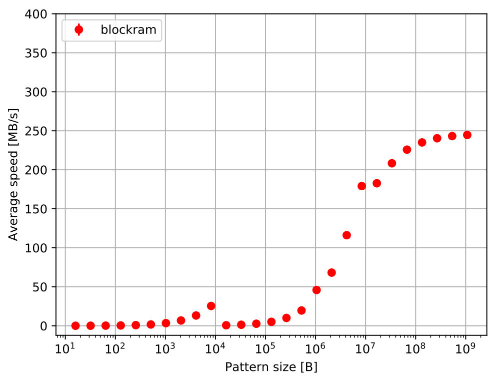


Figure 109. Speed results for FIFO memory type impact subtest in *nonsym write* mode (*64* FIFO depth value and *walking_1* pattern type).

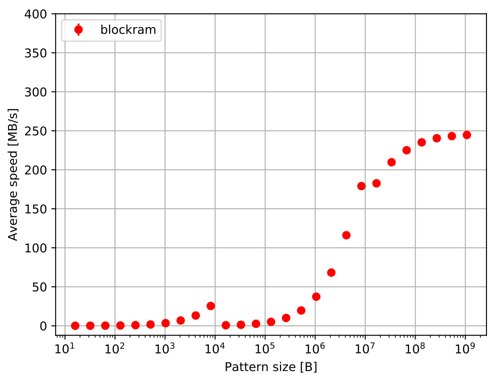


Figure 110. Speed results for FIFO memory type impact subtest in *nonsym write* mode (*64* FIFO depth value and *asic* pattern type).

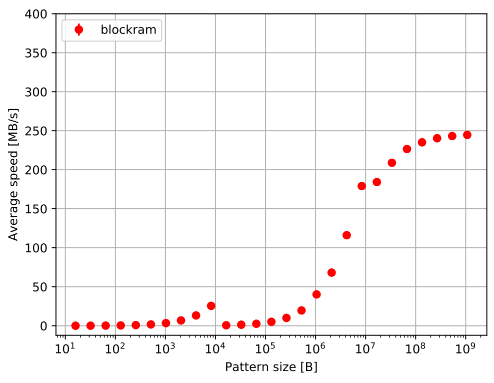


Figure 111. Speed results for FIFO memory type impact subtest in *nonsym write* mode (*256* FIFO depth value and *counter_8bit* pattern type).

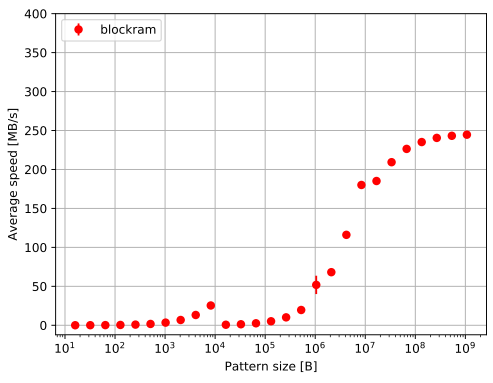


Figure 112. Speed results for FIFO memory type impact subtest in *nonsym write* mode (*256* FIFO depth value and *counter_32bit* pattern type).

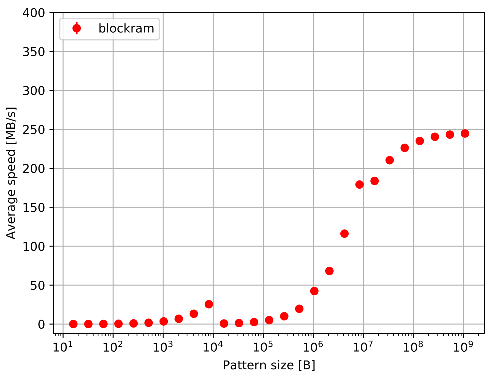


Figure 113. Speed results for FIFO memory type impact subtest in *nonsym write* mode (*256* FIFO depth value and *walking_1* pattern type).

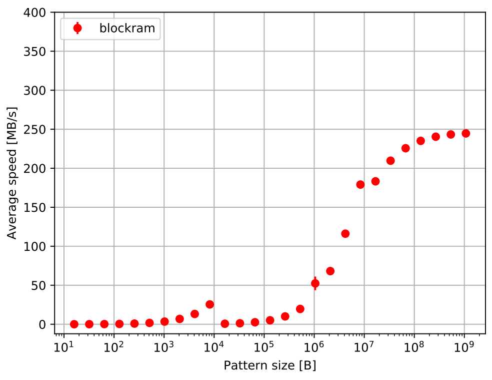


Figure 114. Speed results for FIFO memory type impact subtest in *nonsym write* mode (*256* FIFO depth value and *asic* pattern type).

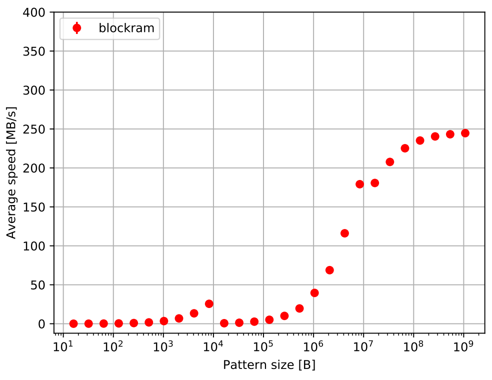


Figure 115. Speed results for FIFO memory type impact subtest in *nonsym write* mode (*1024* FIFO depth value and *counter_8bit* pattern type).

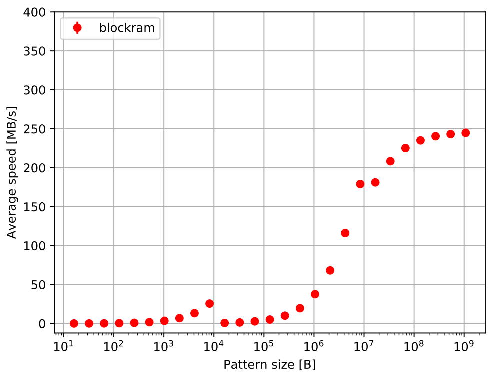


Figure 116. Speed results for FIFO memory type impact subtest in *nonsym write* mode (*1024* FIFO depth value and *counter_32bit* pattern type).

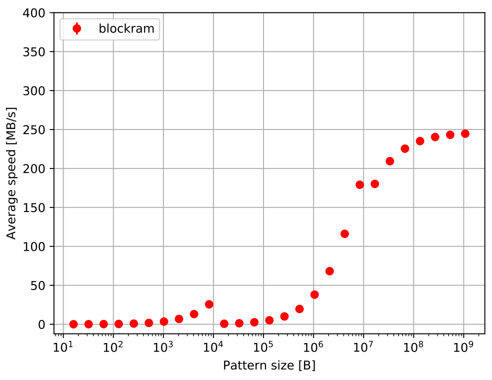


Figure 117. Speed results for FIFO memory type impact subtest in *nonsym write* mode (*1024* FIFO depth value and *walking_1* pattern type).

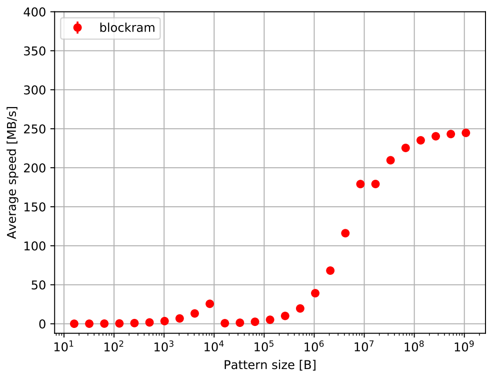


Figure 118. Speed results for FIFO memory type impact subtest in *nonsym write* mode (*1024* FIFO depth value and *asic* pattern type).

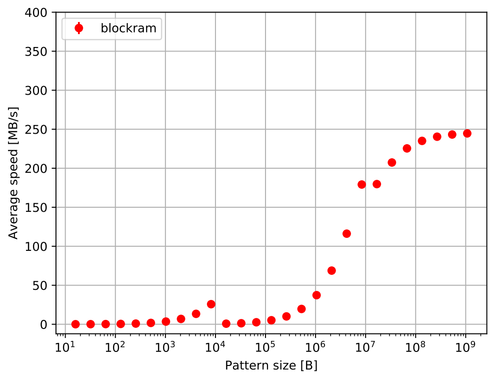


Figure 119. Speed results for FIFO memory type impact subtest in *nonsym write* mode (*2048* FIFO depth value and *counter_8bit* pattern type).

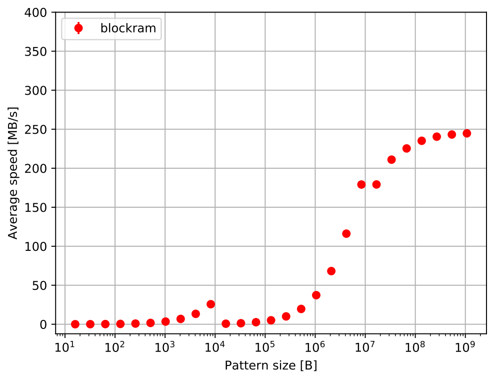


Figure 120. Speed results for FIFO memory type impact subtest in *nonsym write* mode (*2048* FIFO depth value and *counter_32bit* pattern type).

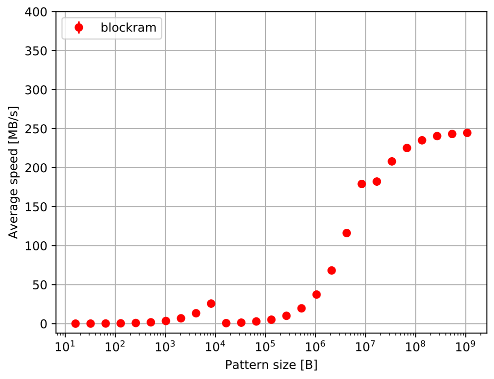


Figure 121. Speed results for FIFO memory type impact subtest in *nonsym write* mode (*2048* FIFO depth value and *walking_1* pattern type).

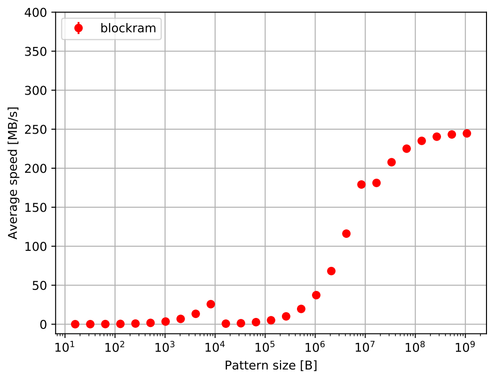


Figure 122. Speed results for FIFO memory type impact subtest in *nonsym write* mode (*2048* FIFO depth value and *asic* pattern type).

Table 15. *Speed results (in MB/s) for blockram memory type in nonsym mode. The first value in cells concerns 16 depth value for read direction and the second one in the square bracket concerns 32 depth value for write direction.*

Pattern size [B]	**counter_8bit**	**counter_32bit**	**walking_1**	**asic**
16	0.065 [*0.054*]	0.065 [*0.054*]	0.065 [*0.054*]	0.065 [*0.054*]
32	0.130 [*0.107*]	0.132 [*0.107*]	0.131 [*0.107*]	0.131 [*0.108*]
64	0.261 [*0.215*]	0.262 [*0.215*]	0.261 [*0.215*]	0.262 [*0.215*]
128	0.523 [*0.430*]	0.523 [*0.430*]	0.523 [*0.430*]	0.523 [*0.429*]
256	1.046 [*0.859*]	1.046 [*0.858*]	1.045 [*0.861*]	1.046 [*0.859*]
512	2.096 [*1.717*]	2.084 [*1.717*]	2.091 [*1.715*]	2.082 [*1.715*]
1024	4.170 [*3.437*]	4.174 [*3.438*]	4.174 [*3.395*]	4.165 [*3.438*]
2048	8.191 [*6.849*]	8.172 [*6.852*]	8.191 [*6.831*]	8.173 [*6.845*]
4096	15.816 [*13.357*]	15.813 [*13.360*]	15.779 [*13.363*]	15.793 [*13.343*]
8192	29.453 [*25.731*]	29.375 [*25.703*]	29.396 [*25.718*]	29.613 [*25.778*]
16384	51.862 [*0.717*]	51.507 [*0.650*]	51.670 [*0.671*]	51.535 [*0.650*]
32768	91.061 [*1.284*]	91.110 [*1.298*]	90.863 [*1.298*]	91.198 [*1.312*]
65536	147.929 [*2.561*]	147.719 [*2.561*]	147.752 [*2.561*]	148.105 [*2.561*]
131072	214.958 [*5.147*]	215.179 [*5.093*]	215.186 [*5.093*]	215.140 [*5.093*]
262144	278.392 [*10.059*]	278.409 [*10.060*]	278.471 [*10.059*]	278.859 [*10.060*]
524288	326.661 [*19.624*]	327.077 [*19.621*]	326.736 [*19.828*]	326.771 [*19.629*]
1048576	352.548 [*37.661*]	352.594 [*37.660*]	352.282 [*38.026*]	352.217 [*38.024*]
2097152	373.910 [*68.182*]	374.079 [*68.179*]	373.343 [*68.177*]	374.325 [*68.184*]
4194304	384.797 [*116.137*]	384.758 [*116.132*]	384.763 [*116.138*]	384.787 [*116.134*]
8388608	390.315 [*179.106*]	390.308 [*179.102*]	390.318 [*179.095*]	390.281 [*179.106*]
16777216	391.573 [*180.700*]	391.576 [*179.153*]	391.577 [*182.190*]	391.567 [*184.707*]
33554432	392.854 [*209.324*]	392.852 [*207.688*]	392.842 [*208.021*]	392.697 [*207.677*]
67108864	393.506 [*225.427*]	393.440 [*225.823*]	393.514 [*225.410*]	393.392 [*225.319*]
134217728	393.841 [*235.185*]	393.834 [*235.085*]	393.839 [*235.039*]	393.814 [*235.216*]
268435456	393.983 [*240.462*]	393.970 [*240.433*]	393.949 [*240.484*]	393.995 [*240.531*]
536870912	394.081 [*243.246*]	394.067 [*243.254*]	394.069 [*243.220*]	394.059 [*243.199*]
1073741824	394.096 [*244.673*]	394.112 [*244.684*]	394.093 [*244.743*]	394.103 [*244.642*]

Table 16. *Speed results (in MB/s) for blockram memory type in nonsym mode. The first value in cells concerns 64 depth value for read direction and the second one in the square bracket concerns 64 depth value for write direction.*

Pattern size [B]	**counter__8bit**	**counter__32bit**	**walking__1**	**asic**
16	0.065 [*0.053*]	0.065 [*0.053*]	0.065 [*0.053*]	0.065 [*0.053*]
32	0.131 [*0.106*]	0.130 [*0.107*]	0.131 [*0.107*]	0.130 [*0.107*]
64	0.263 [*0.213*]	0.261 [*0.213*]	0.261 [*0.214*]	0.261 [*0.214*]
128	0.523 [*0.428*]	0.522 [*0.427*]	0.523 [*0.428*]	0.522 [*0.426*]
256	1.045 [*0.849*]	1.044 [*0.854*]	1.045 [*0.852*]	1.042 [*0.855*]
512	2.092 [*1.700*]	2.086 [*1.698*]	2.089 [*1.693*]	2.086 [*1.703*]
1024	4.177 [*3.400*]	4.169 [*3.412*]	4.148 [*3.414*]	4.173 [*3.387*]
2048	8.153 [*6.770*]	8.173 [*6.805*]	8.166 [*6.808*]	8.123 [*6.797*]
4096	15.757 [*13.208*]	15.812 [*13.189*]	15.772 [*13.206*]	15.768 [*13.182*]
8192	29.415 [*25.351*]	29.390 [*25.322*]	29.466 [*25.408*]	29.284 [*25.403*]
16384	51.510 [*0.650*]	51.390 [*0.664*]	51.561 [*0.696*]	51.569 [*0.680*]
32768	91.236 [*1.298*]	91.177 [*1.298*]	91.215 [*1.343*]	91.127 [*1.312*]
65536	147.735 [*2.561*]	147.530 [*2.588*]	148.007 [*2.651*]	148.043 [*2.589*]
131072	216.054 [*5.092*]	215.082 [*5.093*]	214.880 [*5.092*]	215.061 [*5.092*]
262144	278.207 [*10.060*]	278.772 [*10.058*]	278.437 [*10.059*]	278.503 [*10.060*]
524288	326.701 [*19.625*]	326.438 [*19.624*]	326.565 [*19.628*]	326.875 [*19.623*]
1048576	352.404 [*47.582*]	352.770 [*41.738*]	352.586 [*45.874*]	352.982 [*37.287*]
2097152	374.297 [*68.169*]	374.273 [*68.189*]	374.289 [*68.190*]	374.106 [*68.182*]
4194304	384.738 [*116.143*]	384.779 [*116.144*]	384.777 [*116.141*]	384.823 [*116.132*]
8388608	390.311 [*179.142*]	390.324 [*179.123*]	390.345 [*179.121*]	390.341 [*179.135*]
16777216	391.579 [*184.886*]	391.563 [*180.687*]	391.585 [*182.716*]	391.579 [*182.679*]
33554432	392.872 [*209.321*]	392.868 [*208.018*]	392.842 [*208.347*]	392.863 [*209.682*]
67108864	393.517 [*225.253*]	393.491 [*225.441*]	393.509 [*225.813*]	393.305 [*225.054*]
134217728	393.816 [*235.078*]	393.820 [*235.279*]	393.731 [*235.077*]	393.817 [*235.191*]
268435456	393.973 [*240.485*]	393.919 [*240.424*]	393.942 [*240.434*]	393.983 [*240.431*]
536870912	394.066 [*243.219*]	394.061 [*243.228*]	394.075 [*243.175*]	394.077 [*243.171*]
1073741824	394.113 [*244.688*]	394.107 [*244.770*]	394.101 [*244.648*]	394.097 [*244.644*]

Table 17. *Speed results (in MB/s) for blockram memory type in nonsym mode. The first value in cells concerns 256 depth value for read direction and the second one in the square bracket concerns 256 depth value for write direction.*

Pattern size [B]	**counter_8bit**	**counter_32bit**	**walking_1**	**asic**
16	0.065 [*0.053*]	0.065 [*0.054*]	0.065 [*0.054*]	0.065 [*0.054*]
32	0.131 [*0.107*]	0.130 [*0.107*]	0.131 [*0.107*]	0.131 [*0.107*]
64	0.260 [*0.214*]	0.262 [*0.213*]	0.261 [*0.214*]	0.260 [*0.214*]
128	0.520 [*0.429*]	0.522 [*0.427*]	0.522 [*0.428*]	0.522 [*0.427*]
256	1.045 [*0.855*]	1.042 [*0.856*]	1.047 [*0.855*]	1.042 [*0.854*]
512	2.091 [*1.707*]	2.085 [*1.707*]	2.090 [*1.706*]	2.090 [*1.709*]
1024	4.173 [*3.411*]	4.179 [*3.421*]	4.169 [*3.417*]	4.165 [*3.417*]
2048	8.179 [*6.808*]	8.180 [*6.789*]	8.190 [*6.815*]	8.177 [*6.813*]
4096	15.795 [*13.229*]	15.812 [*13.218*]	15.787 [*13.219*]	15.813 [*13.184*]
8192	29.407 [*25.509*]	29.428 [*25.395*]	29.438 [*25.489*]	29.380 [*25.431*]
16384	51.454 [*0.643*]	51.499 [*0.687*]	51.476 [*0.700*]	51.443 [*0.678*]
32768	91.132 [*1.344*]	91.425 [*1.312*]	90.949 [*1.326*]	91.220 [*1.284*]
65536	148.080 [*2.561*]	148.191 [*2.561*]	148.067 [*2.561*]	148.193 [*2.561*]
131072	215.370 [*5.093*]	214.947 [*5.147*]	214.811 [*5.093*]	214.888 [*5.092*]
262144	278.370 [*10.061*]	278.689 [*10.166*]	278.318 [*10.059*]	278.250 [*10.061*]
524288	327.005 [*19.629*]	326.432 [*19.626*]	326.897 [*19.628*]	326.772 [*19.625*]
1048576	352.535 [*40.199*]	352.778 [*51.847*]	353.203 [*42.372*]	353.293 [*52.397*]
2097152	374.350 [*68.163*]	374.346 [*68.158*]	374.400 [*68.160*]	374.394 [*68.160*]
4194304	384.741 [*116.110*]	384.859 [*116.086*]	384.824 [*116.109*]	384.871 [*116.106*]
8388608	390.317 [*179.098*]	390.345 [*180.102*]	390.215 [*179.110*]	390.317 [*179.069*]
16777216	391.587 [*184.246*]	391.578 [*185.187*]	391.579 [*183.759*]	391.587 [*183.229*]
33554432	392.864 [*208.936*]	392.864 [*209.349*]	392.871 [*210.356*]	392.864 [*209.652*]
67108864	393.495 [*226.582*]	393.511 [*226.403*]	393.509 [*226.211*]	393.520 [*225.640*]
134217728	393.840 [*235.174*]	393.836 [*235.195*]	393.716 [*235.076*]	393.674 [*235.084*]
268435456	393.959 [*240.403*]	393.959 [*240.524*]	393.993 [*240.417*]	394.005 [*240.425*]
536870912	394.082 [*243.202*]	394.062 [*243.210*]	394.078 [*243.225*]	394.068 [*243.328*]
1073741824	394.093 [*244.705*]	394.080 [*244.639*]	393.990 [*244.659*]	394.083 [*244.655*]

Table 18. *Speed results (in MB/s) for blockram memory type in nonsym mode. The first value in cells concerns 1024 depth value for read direction and the second one in the square bracket concerns 1024 depth value for write direction.*

Pattern size [B]	**counter_8bit**	**counter_32bit**	**walking_1**	**asic**
16	0.066 [*0.053*]	0.066 [*0.053*]	0.066 [*0.053*]	0.066 [*0.054*]
32	0.132 [*0.107*]	0.132 [*0.106*]	0.132 [*0.107*]	0.132 [*0.107*]
64	0.263 [*0.214*]	0.264 [*0.215*]	0.263 [*0.215*]	0.263 [*0.215*]
128	0.525 [*0.430*]	0.527 [*0.429*]	0.525 [*0.430*]	0.527 [*0.429*]
256	1.054 [*0.857*]	1.053 [*0.858*]	1.054 [*0.858*]	1.047 [*0.857*]
512	2.096 [*1.714*]	2.105 [*1.713*]	2.107 [*1.715*]	2.107 [*1.716*]
1024	4.195 [*3.412*]	4.201 [*3.434*]	4.209 [*3.439*]	4.196 [*3.435*]
2048	8.242 [*6.826*]	8.246 [*6.846*]	8.230 [*6.839*]	8.231 [*6.822*]
4096	15.911 [*13.350*]	15.905 [*13.248*]	15.886 [*13.088*]	15.909 [*13.220*]
8192	29.604 [*25.576*]	29.619 [*25.571*]	29.609 [*25.589*]	29.617 [*25.585*]
16384	51.961 [*0.657*]	51.853 [*0.666*]	51.690 [*0.682*]	51.634 [*0.727*]
32768	91.679 [*1.284*]	91.721 [*1.326*]	91.536 [*1.312*]	91.695 [*1.326*]
65536	148.700 [*2.672*]	148.868 [*2.758*]	148.779 [*2.589*]	148.565 [*2.561*]
131072	215.849 [*5.093*]	215.783 [*5.093*]	215.802 [*5.093*]	215.596 [*5.146*]
262144	279.065 [*10.061*]	277.893 [*10.060*]	278.787 [*10.059*]	279.065 [*10.060*]
524288	327.189 [*19.626*]	327.090 [*19.626*]	327.228 [*19.627*]	327.257 [*19.627*]
1048576	353.228 [*39.562*]	353.200 [*37.655*]	353.125 [*38.018*]	353.288 [*39.192*]
2097152	374.189 [*68.772*]	374.340 [*68.163*]	374.413 [*68.162*]	374.334 [*68.165*]
4194304	384.885 [*116.111*]	384.874 [*116.127*]	384.848 [*116.109*]	384.887 [*116.114*]
8388608	390.348 [*179.100*]	390.342 [*179.094*]	390.349 [*179.085*]	390.332 [*179.102*]
16777216	391.592 [*180.671*]	391.582 [*181.192*]	391.596 [*180.167*]	391.567 [*179.184*]
33554432	392.857 [*207.682*]	392.865 [*208.341*]	392.856 [*209.318*]	392.863 [*209.637*]
67108864	393.508 [*225.236*]	393.513 [*225.220*]	393.503 [*225.430*]	393.514 [*225.426*]
134217728	393.822 [*235.170*]	393.721 [*235.076*]	393.789 [*235.179*]	393.825 [*235.189*]
268435456	393.978 [*240.421*]	393.997 [*240.444*]	393.999 [*240.382*]	393.980 [*240.381*]
536870912	394.071 [*243.220*]	394.071 [*243.149*]	394.062 [*243.193*]	394.049 [*243.224*]
1073741824	394.103 [*244.641*]	394.095 [*244.747*]	394.113 [*244.645*]	394.110 [*244.721*]

Table 19. *Speed results (in MB/s) for blockram memory type in nonsym mode. The first value in cells concerns 2048 depth value for read direction and the second one in the square bracket concerns 2048 depth value for write direction.*

Pattern size [B]	counter_8bit	counter_32bit	walking_1	asic
16	0.066 [*0.054*]	0.066 [*0.054*]	0.066 [*0.054*]	0.066 [*0.054*]
32	0.131 [*0.107*]	0.132 [*0.107*]	0.132 [*0.107*]	0.132 [*0.107*]
64	0.263 [*0.215*]	0.263 [*0.215*]	0.263 [*0.215*]	0.263 [*0.215*]
128	0.528 [*0.430*]	0.527 [*0.430*]	0.526 [*0.430*]	0.527 [*0.430*]
256	1.053 [*0.860*]	1.053 [*0.859*]	1.053 [*0.857*]	1.054 [*0.854*]
512	2.106 [*1.716*]	2.091 [*1.704*]	2.105 [*1.717*]	2.107 [*1.719*]
1024	4.167 [*3.434*]	4.204 [*3.433*]	4.197 [*3.432*]	4.217 [*3.441*]
2048	8.238 [*6.847*]	8.242 [*6.842*]	8.241 [*6.843*]	8.236 [*6.849*]
4096	15.866 [*13.357*]	15.888 [*13.342*]	15.933 [*13.343*]	15.901 [*13.352*]
8192	29.582 [*25.737*]	29.611 [*25.700*]	29.571 [*25.690*]	29.611 [*25.720*]
16384	51.974 [*0.738*]	51.919 [*0.703*]	51.930 [*0.671*]	51.904 [*0.737*]
32768	91.423 [*1.285*]	91.823 [*1.313*]	91.752 [*1.299*]	90.902 [*1.284*]
65536	148.115 [*2.561*]	148.597 [*2.623*]	148.646 [*2.824*]	148.478 [*2.678*]
131072	215.654 [*5.146*]	215.974 [*5.092*]	215.725 [*5.092*]	215.950 [*5.092*]
262144	279.060 [*10.060*]	278.947 [*10.057*]	279.013 [*10.061*]	279.087 [*10.061*]
524288	326.936 [*19.624*]	327.184 [*19.629*]	327.157 [*19.625*]	327.104 [*19.628*]
1048576	353.204 [*37.295*]	352.968 [*37.291*]	353.290 [*37.294*]	353.277 [*37.293*]
2097152	374.402 [*68.771*]	374.425 [*68.166*]	374.357 [*68.167*]	374.359 [*68.174*]
4194304	384.867 [*116.137*]	384.870 [*116.139*]	384.861 [*116.145*]	384.875 [*116.146*]
8388608	390.335 [*179.129*]	390.347 [*179.114*]	390.245 [*179.091*]	390.296 [*179.110*]
16777216	391.551 [*179.676*]	391.591 [*179.195*]	391.583 [*182.194*]	391.587 [*181.162*]
33554432	392.866 [*207.358*]	392.874 [*211.023*]	392.869 [*208.013*]	392.875 [*207.690*]
67108864	393.493 [*225.439*]	393.488 [*225.380*]	393.489 [*225.197*]	393.480 [*225.048*]
134217728	393.764 [*235.068*]	393.804 [*235.190*]	393.791 [*235.062*]	393.827 [*235.071*]
268435456	393.996 [*240.420*]	393.961 [*240.473*]	393.993 [*240.469*]	393.934 [*240.418*]
536870912	394.042 [*243.224*]	394.074 [*243.229*]	394.028 [*243.180*]	394.044 [*243.268*]
1073741824	394.089 [*244.662*]	394.081 [*244.743*]	394.099 [*244.608*]	394.095 [*244.636*]

A.2.2. 32bit mode part of the subtest

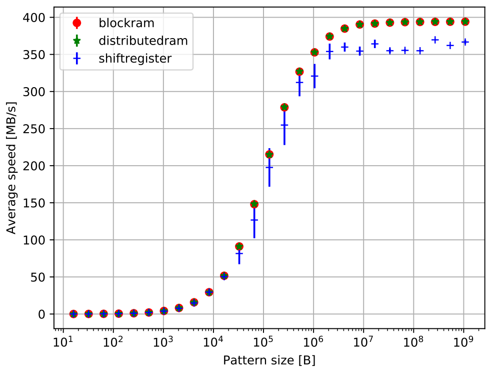


Figure 123. Speed results for FIFO memory type impact subtest in *32bit read* mode (*16* FIFO depth value and *counter_8bit* pattern type).

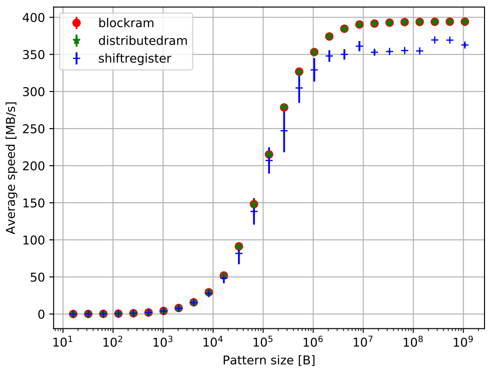


Figure 124. Speed results for FIFO memory type impact subtest in *32bit read* mode (*16* FIFO depth value and *counter_32bit* pattern type).

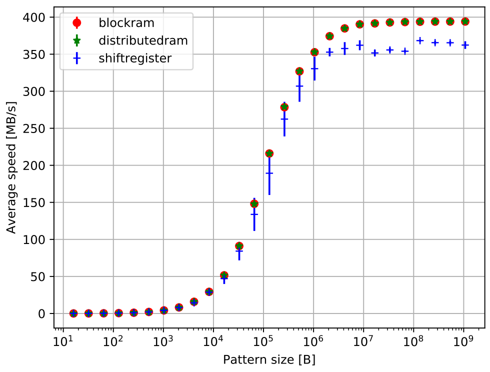


Figure 125. Speed results for FIFO memory type impact subtest in *32bit read* mode (*16* FIFO depth value and *walking_1* pattern type).

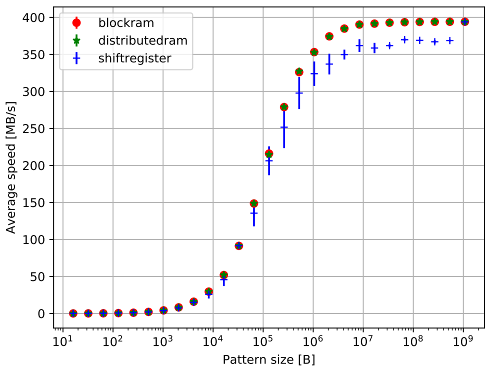


Figure 126. Speed results for FIFO memory type impact subtest in *32bit read* mode (*64* FIFO depth value and *counter_8bit* pattern type).

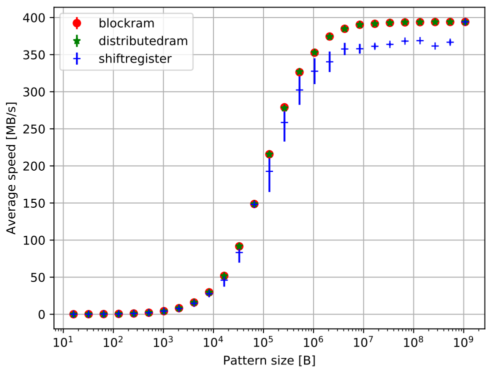


Figure 127. Speed results for FIFO memory type impact subtest in *32bit read* mode (*64* FIFO depth value and *counter_32bit* pattern type).

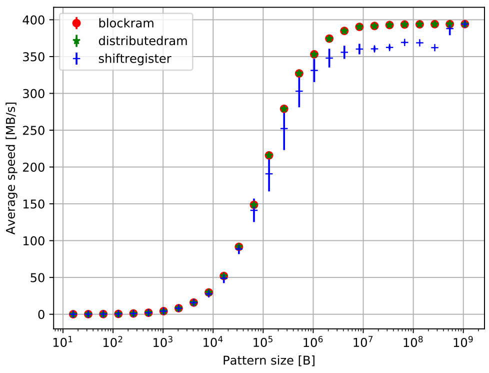


Figure 128. Speed results for FIFO memory type impact subtest in *32bit read* mode (*64* FIFO depth value and *walking_1* pattern type).

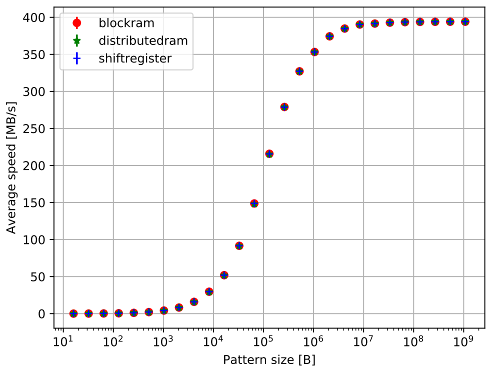


Figure 129. Speed results for FIFO memory type impact subtest in *32bit read* mode (*256* FIFO depth value and *counter_8bit* pattern type).

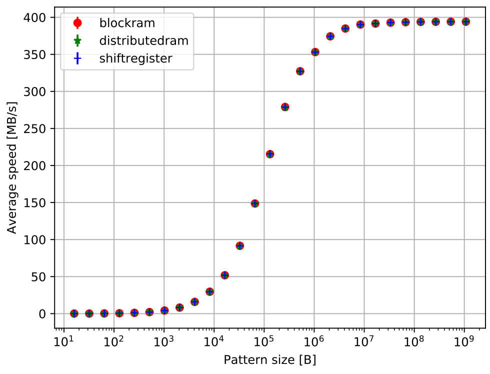


Figure 130. Speed results for FIFO memory type impact subtest in *32bit read* mode (*256* FIFO depth value and *counter_32bit* pattern type).

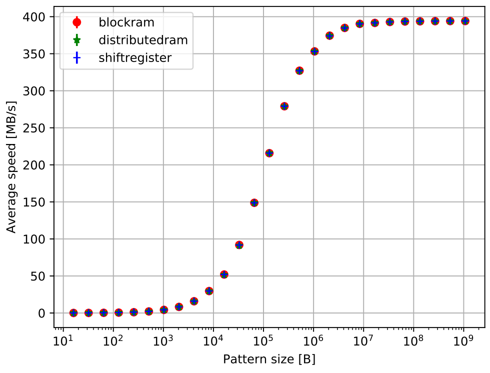


Figure 131. Speed results for FIFO memory type impact subtest in *32bit read* mode (*256* FIFO depth value and *walking_1* pattern type).

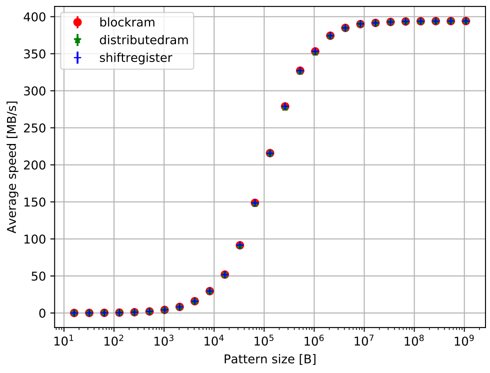


Figure 132. Speed results for FIFO memory type impact subtest in *32bit read* mode (*1024* FIFO depth value and *counter_8bit* pattern type).

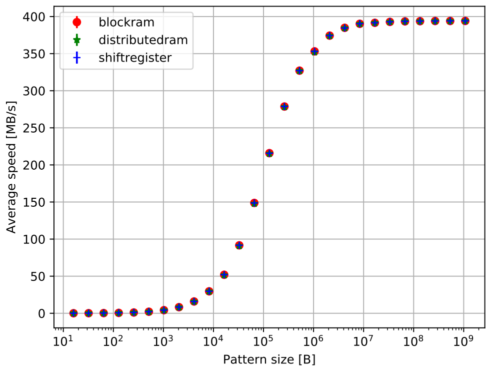


Figure 133. Speed results for FIFO memory type impact subtest in *32bit read* mode (*1024* FIFO depth value and *counter_32bit* pattern type).

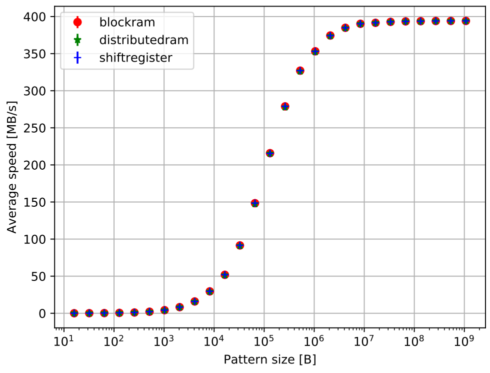


Figure 134. Speed results for FIFO memory type impact subtest in *32bit read* mode (*1024* FIFO depth value and *walking_1* pattern type).

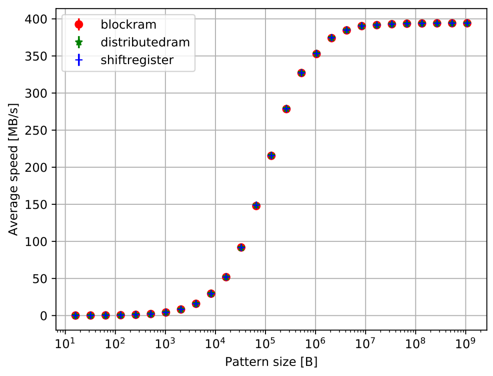


Figure 135. Speed results for FIFO memory type impact subtest in *32bit read* mode (*2048* FIFO depth value and *counter_8bit* pattern type).

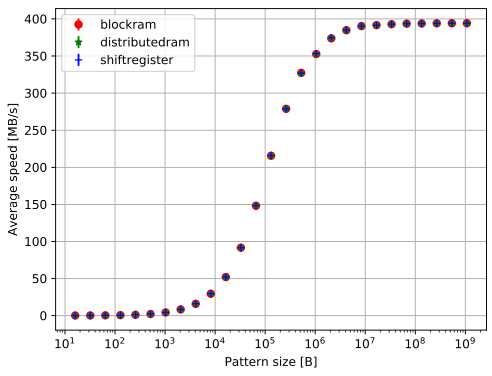


Figure 136. Speed results for FIFO memory type impact subtest in *32bit read* mode (*2048* FIFO depth value and *counter_32bit* pattern type).

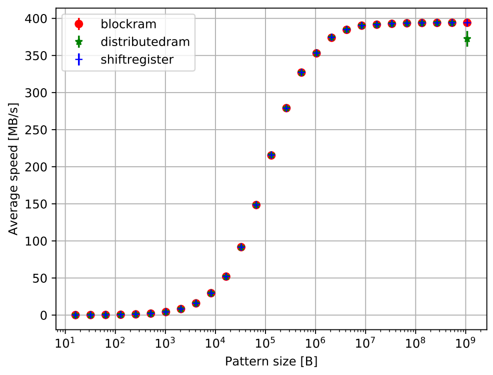


Figure 137. Speed results for FIFO memory type impact subtest in *32bit read* mode (*2048* FIFO depth value and *walking_1* pattern type).

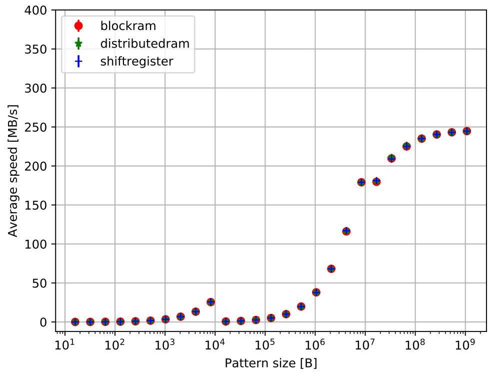


Figure 138. Speed results for FIFO memory type impact subtest in *32bit write* mode (*16* FIFO depth value and *counter_8bit* pattern type).

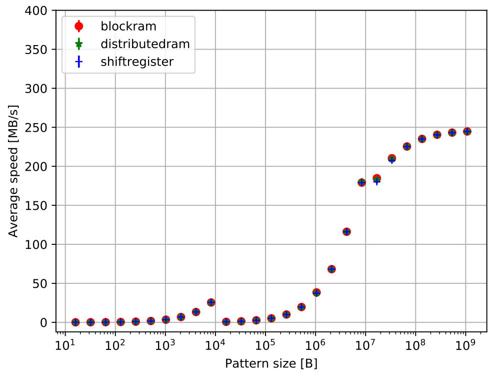


Figure 139. Speed results for FIFO memory type impact subtest in *32bit write* mode (*16* FIFO depth value and *counter_32bit* pattern type).

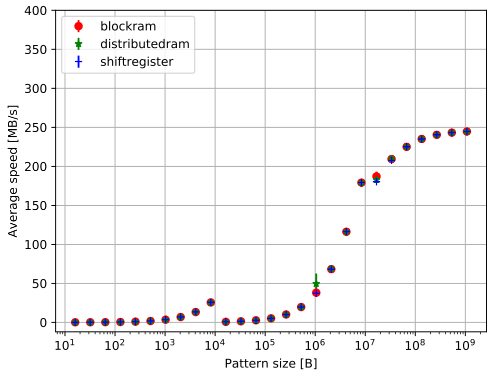


Figure 140. Speed results for FIFO memory type impact subtest in *32bit write* mode (*16* FIFO depth value and *walking_1* pattern type).

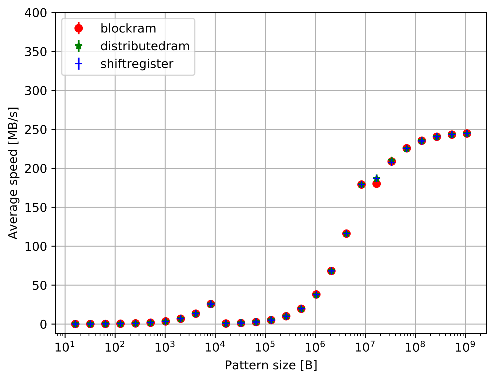


Figure 141. Speed results for FIFO memory type impact subtest in *32bit write* mode (*64* FIFO depth value and *counter_8bit* pattern type).

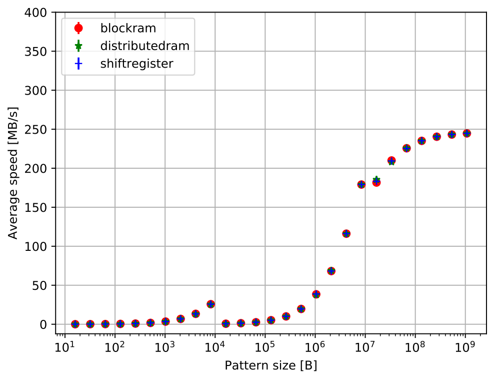


Figure 142. Speed results for FIFO memory type impact subtest in *32bit write* mode (*64* FIFO depth value and *counter_32bit* pattern type).

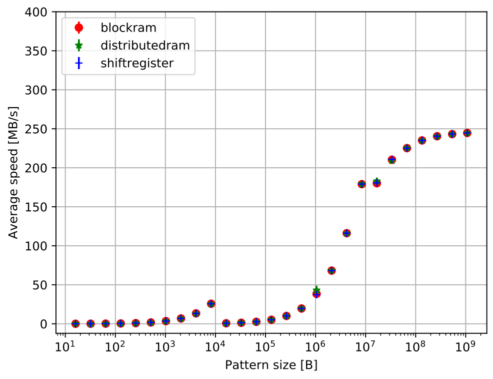


Figure 143. Speed results for FIFO memory type impact subtest in *32bit write* mode (*64* FIFO depth value and *walking_1* pattern type).

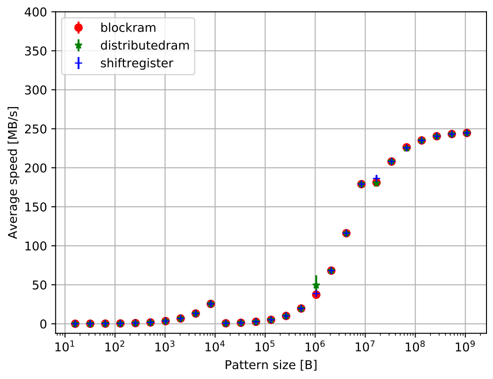


Figure 144. Speed results for FIFO memory type impact subtest in *32bit write* mode (*256* FIFO depth value and *counter_8bit* pattern type).

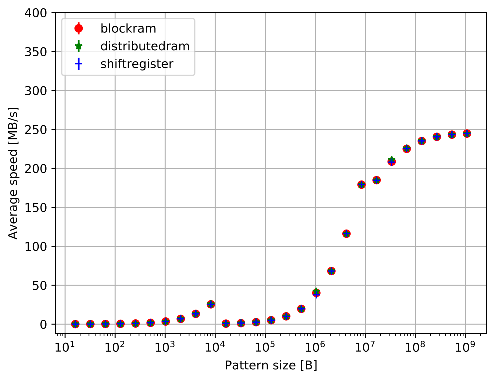


Figure 145. Speed results for FIFO memory type impact subtest in *32bit write* mode (*256* FIFO depth value and *counter_32bit* pattern type).

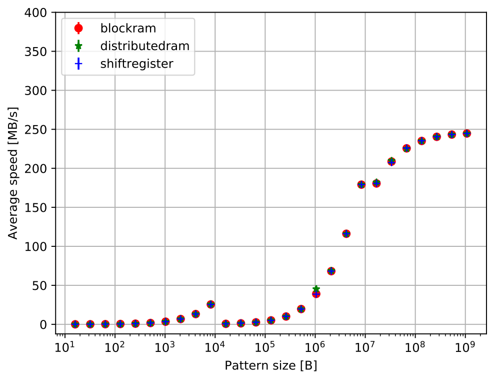


Figure 146. Speed results for FIFO memory type impact subtest in *32bit write* mode (*256* FIFO depth value and *walking_1* pattern type).

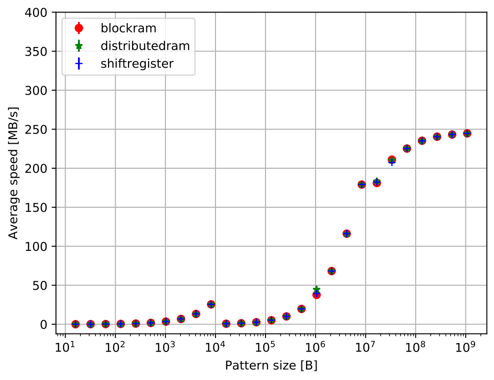


Figure 147. Speed results for FIFO memory type impact subtest in *32bit write* mode (*1024* FIFO depth value and *counter_8bit* pattern type).

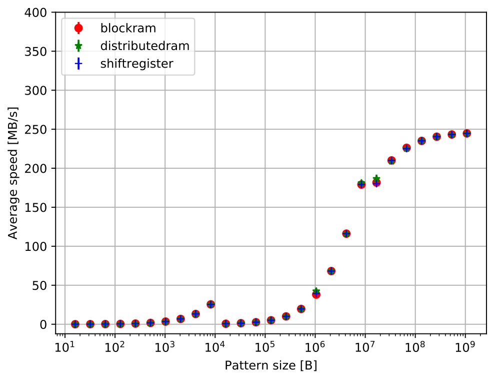


Figure 148. Speed results for FIFO memory type impact subtest in *32bit write* mode (*1024* FIFO depth value and *counter_32bit* pattern type).

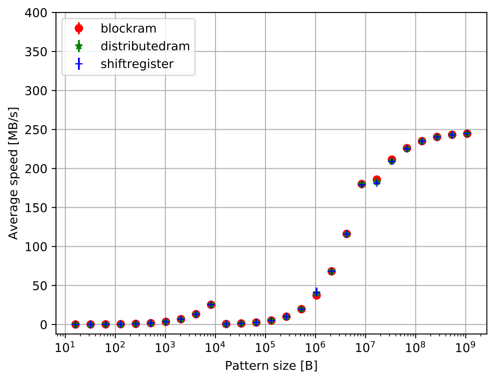


Figure 149. Speed results for FIFO memory type impact subtest in *32bit write* mode (*1024* FIFO depth value and *walking_1* pattern type).

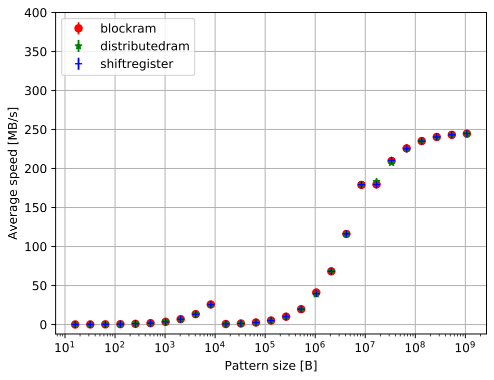


Figure 150. Speed results for FIFO memory type impact subtest in *32bit write* mode (*2048* FIFO depth value and *counter_8bit* pattern type).

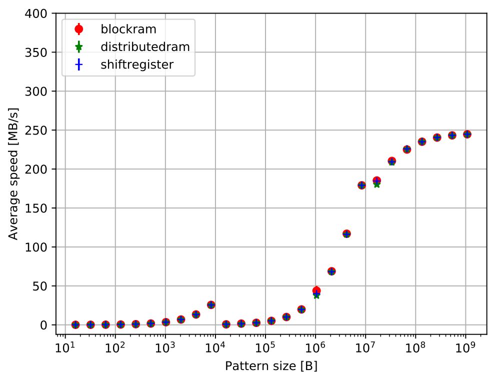


Figure 151. Speed results for FIFO memory type impact subtest in *32bit write* mode (*2048* FIFO depth value and *counter_32bit* pattern type).

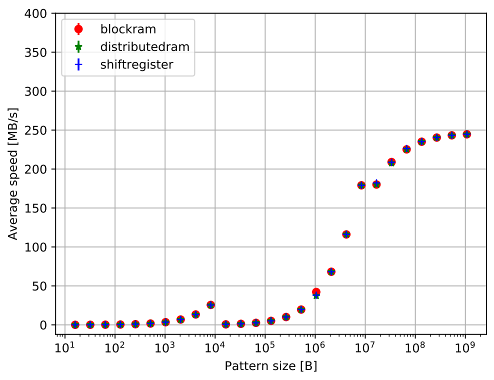


Figure 152. Speed results for FIFO memory type impact subtest in *32bit write* mode (*2048* FIFO depth value and *walking_1* pattern type).

Table 20. *The fastest memory types (values in square brackets are in MB/s) compared between pattern types (32bit mode, read direction, 16 depth value).*

Pattern size [B]	**counter__8bit**	**counter__32bit**	**walking__1**
16	distributedram [*0.065*]	distributedram [*0.066*]	distributedram [*0.065*]
32	distributedram [*0.131*]	distributedram [*0.131*]	distributedram [*0.130*]
64	distributedram [*0.261*]	blockram [*0.261*]	distributedram [*0.262*]
128	blockram [*0.523*]	distributedram [*0.523*]	distributedram [*0.522*]
256	distributedram [*1.046*]	distributedram [*1.043*]	distributedram [*1.046*]
512	blockram [*2.087*]	distributedram [*2.084*]	distributedram [*2.089*]
1024	distributedram [*4.170*]	distributedram [*4.179*]	shiftregister [*4.175*]
2048	distributedram [*8.245*]	distributedram [*8.199*]	blockram [*8.189*]
4096	blockram [*15.811*]	blockram [*15.806*]	blockram [*15.820*]
8192	shiftregister [*29.417*]	distributedram [*29.400*]	distributedram [*29.385*]
16384	blockram [*51.601*]	blockram [*51.933*]	distributedram [*51.534*]
32768	blockram [*91.125*]	blockram [*91.284*]	blockram [*91.067*]
65536	distributedram [*148.380*]	blockram [*148.171*]	blockram [*148.088*]
131072	blockram [*215.242*]	distributedram [*215.213*]	blockram [*216.013*]
262144	distributedram [*278.975*]	distributedram [*278.940*]	distributedram [*278.996*]
524288	distributedram [*327.099*]	blockram [*326.805*]	distributedram [*327.012*]
1048576	distributedram [*352.820*]	blockram [*353.151*]	distributedram [*352.855*]
2097152	distributedram [*374.261*]	distributedram [*374.224*]	distributedram [*374.334*]
4194304	blockram [*384.814*]	distributedram [*384.781*]	blockram [*384.840*]
8388608	blockram [*390.351*]	blockram [*390.341*]	blockram [*390.338*]
16777216	blockram [*391.581*]	blockram [*391.590*]	blockram [*391.583*]
33554432	blockram [*392.863*]	blockram [*392.862*]	blockram [*392.870*]
67108864	blockram [*393.517*]	blockram [*393.518*]	distributedram [*393.508*]
134217728	distributedram [*393.798*]	distributedram [*393.836*]	distributedram [*393.838*]
268435456	distributedram [*393.991*]	distributedram [*393.999*]	distributedram [*393.988*]
536870912	distributedram [*394.057*]	distributedram [*394.059*]	blockram [*394.067*]
1073741824	blockram [*394.109*]	blockram [*394.096*]	blockram [*394.096*]
Most frequent parameter	distributedram	distributedram	distributedram

Table 21. *The fastest memory types (values in square brackets are in MB/s) compared between pattern types (32bit mode, read direction, 64 depth value).*

Pattern size [B]	**counter_8bit**	**counter_32bit**	**walking_1**
16	blockram *[0.066]*	blockram *[0.066]*	distributedram *[0.066]*
32	blockram *[0.131]*	distributedram *[0.132]*	blockram *[0.132]*
64	blockram *[0.264]*	blockram *[0.263]*	blockram *[0.264]*
128	blockram *[0.528]*	distributedram *[0.524]*	distributedram *[0.526]*
256	distributedram *[1.051]*	distributedram *[1.051]*	blockram *[1.054]*
512	blockram *[2.111]*	distributedram *[2.108]*	blockram *[2.103]*
1024	distributedram *[4.198]*	blockram *[4.207]*	blockram *[4.206]*
2048	blockram *[8.249]*	blockram *[8.249]*	distributedram *[8.239]*
4096	blockram *[15.909]*	blockram *[15.900]*	blockram *[15.918]*
8192	distributedram *[29.584]*	blockram *[29.643]*	blockram *[29.635]*
16384	distributedram *[51.938]*	blockram *[51.778]*	blockram *[51.995]*
32768	shiftregister *[91.571]*	distributedram *[91.664]*	distributedram *[91.623]*
65536	distributedram *[148.613]*	distributedram *[148.720]*	blockram *[148.682]*
131072	blockram *[216.011]*	blockram *[215.752]*	blockram *[215.840]*
262144	distributedram *[278.908]*	blockram *[279.063]*	blockram *[279.071]*
524288	distributedram *[327.086]*	distributedram *[326.851]*	blockram *[327.092]*
1048576	blockram *[353.133]*	distributedram *[352.868]*	blockram *[353.087]*
2097152	blockram *[374.407]*	blockram *[374.437]*	blockram *[374.435]*
4194304	blockram *[384.869]*	blockram *[384.861]*	blockram *[384.863]*
8388608	blockram *[390.303]*	blockram *[390.327]*	blockram *[390.332]*
16777216	blockram *[391.578]*	blockram *[391.577]*	blockram *[391.573]*
33554432	blockram *[392.856]*	blockram *[392.871]*	blockram *[392.865]*
67108864	blockram *[393.514]*	distributedram *[393.513]*	blockram *[393.515]*
134217728	blockram *[393.833]*	distributedram *[393.831]*	distributedram *[393.833]*
268435456	distributedram *[393.995]*	blockram *[393.973]*	blockram *[393.998]*
536870912	blockram *[394.080]*	blockram *[394.064]*	distributedram *[394.062]*
1073741824	blockram *[394.103]*	distributedram *[394.102]*	distributedram *[394.099]*
Most frequent parameter	blockram	blockram	blockram

Table 22. *The fastest memory types (values in square brackets are in MB/s) compared between pattern types (32bit mode, read direction, 256 depth value).*

Pattern size [B]	counter__8bit	counter__32bit	walking__1
16	blockram *[0.066]*	blockram *[0.066]*	blockram *[0.066]*
32	shiftregister *[0.131]*	blockram *[0.132]*	shiftregister *[0.131]*
64	blockram *[0.263]*	shiftregister *[0.263]*	blockram *[0.263]*
128	shiftregister *[0.527]*	blockram *[0.527]*	blockram *[0.527]*
256	distributedram *[1.056]*	distributedram *[1.057]*	shiftregister *[1.052]*
512	blockram *[2.105]*	shiftregister *[2.107]*	blockram *[2.103]*
1024	shiftregister *[4.198]*	blockram *[4.195]*	distributedram *[4.208]*
2048	distributedram *[8.249]*	distributedram *[8.244]*	blockram *[8.234]*
4096	shiftregister *[15.919]*	distributedram *[15.903]*	shiftregister *[15.902]*
8192	blockram *[29.621]*	blockram *[29.630]*	shiftregister *[29.657]*
16384	blockram *[51.932]*	shiftregister *[51.843]*	blockram *[51.965]*
32768	shiftregister *[91.657]*	blockram *[91.614]*	blockram *[91.875]*
65536	shiftregister *[148.781]*	shiftregister *[148.732]*	blockram *[148.762]*
131072	blockram *[215.799]*	shiftregister *[215.700]*	distributedram *[215.921]*
262144	blockram *[278.965]*	blockram *[278.996]*	blockram *[279.096]*
524288	shiftregister *[327.212]*	blockram *[327.203]*	blockram *[327.124]*
1048576	blockram *[353.209]*	blockram *[353.114]*	blockram *[353.172]*
2097152	distributedram *[374.434]*	distributedram *[374.423]*	distributedram *[374.457]*
4194304	distributedram *[384.898]*	blockram *[384.872]*	distributedram *[384.856]*
8388608	distributedram *[390.332]*	distributedram *[390.293]*	distributedram *[390.326]*
16777216	distributedram *[391.590]*	blockram *[391.572]*	shiftregister *[391.583]*
33554432	shiftregister *[392.868]*	shiftregister *[392.873]*	distributedram *[392.867]*
67108864	shiftregister *[393.516]*	shiftregister *[393.511]*	shiftregister *[393.515]*
134217728	shiftregister *[393.844]*	distributedram *[393.841]*	distributedram *[393.830]*
268435456	shiftregister *[393.986]*	shiftregister *[393.973]*	shiftregister *[393.986]*
536870912	shiftregister *[394.071]*	shiftregister *[394.071]*	blockram *[394.076]*
1073741824	distributedram *[394.097]*	blockram *[394.103]*	blockram *[394.104]*
Most frequent parameter	shiftregister	blockram	blockram

Table 23. *The fastest memory types (values in square brackets are in MB/s) compared between pattern types (32bit mode, read direction, 1024 depth value).*

Pattern size [B]	**counter__8bit**	**counter__32bit**	**walking__1**
16	shiftregister [*0.066*]	shiftregister [*0.066*]	shiftregister [*0.066*]
32	shiftregister [*0.132*]	shiftregister [*0.132*]	shiftregister [*0.132*]
64	blockram [*0.263*]	shiftregister [*0.263*]	shiftregister [*0.263*]
128	shiftregister [*0.527*]	shiftregister [*0.527*]	blockram [*0.528*]
256	blockram [*1.054*]	shiftregister [*1.054*]	shiftregister [*1.053*]
512	blockram [*2.105*]	blockram [*2.106*]	shiftregister [*2.107*]
1024	shiftregister [*4.203*]	shiftregister [*4.197*]	shiftregister [*4.201*]
2048	shiftregister [*8.249*]	shiftregister [*8.253*]	blockram [*8.259*]
4096	blockram [*15.924*]	blockram [*15.908*]	blockram [*15.928*]
8192	shiftregister [*29.603*]	shiftregister [*29.607*]	shiftregister [*29.628*]
16384	blockram [*51.878*]	shiftregister [*51.991*]	shiftregister [*51.920*]
32768	shiftregister [*91.683*]	shiftregister [*91.670*]	shiftregister [*91.553*]
65536	blockram [*148.728*]	blockram [*148.776*]	shiftregister [*148.546*]
131072	blockram [*215.834*]	blockram [*215.868*]	blockram [*215.822*]
262144	shiftregister [*278.995*]	blockram [*278.726*]	shiftregister [*278.988*]
524288	shiftregister [*327.205*]	blockram [*327.098*]	shiftregister [*327.098*]
1048576	shiftregister [*353.185*]	shiftregister [*353.108*]	blockram [*353.194*]
2097152	blockram [*374.333*]	shiftregister [*374.326*]	blockram [*374.452*]
4194304	blockram [*384.844*]	blockram [*384.837*]	blockram [*384.853*]
8388608	shiftregister [*390.341*]	shiftregister [*390.343*]	shiftregister [*390.286*]
16777216	shiftregister [*391.585*]	blockram [*391.590*]	shiftregister [*391.572*]
33554432	shiftregister [*392.856*]	shiftregister [*392.868*]	shiftregister [*392.869*]
67108864	shiftregister [*393.514*]	shiftregister [*393.512*]	blockram [*393.491*]
134217728	blockram [*393.835*]	shiftregister [*393.815*]	blockram [*393.769*]
268435456	shiftregister [*393.990*]	blockram [*394.002*]	blockram [*393.996*]
536870912	distributedram [*394.074*]	shiftregister [*394.067*]	shiftregister [*394.080*]
1073741824	distributedram [*394.103*]	distributedram [*394.100*]	blockram [*394.099*]
Most frequent parameter	shiftregister	shiftregister	shiftregister

Table 24. *The fastest memory types (values in square brackets are in MB/s) compared between pattern types (32bit mode, read direction, 2048 depth value).*

Pattern size [B]	**counter__8bit**	**counter__32bit**	**walking__1**
16	distributedram *[0.066]*	shiftregister *[0.066]*	distributedram *[0.066]*
32	shiftregister *[0.132]*	shiftregister *[0.131]*	distributedram *[0.132]*
64	shiftregister *[0.264]*	shiftregister *[0.264]*	shiftregister *[0.263]*
128	shiftregister *[0.527]*	shiftregister *[0.528]*	shiftregister *[0.527]*
256	distributedram *[1.053]*	shiftregister *[1.053]*	shiftregister *[1.054]*
512	distributedram *[2.107]*	shiftregister *[2.107]*	shiftregister *[2.104]*
1024	shiftregister *[4.201]*	distributedram *[4.204]*	shiftregister *[4.210]*
2048	shiftregister *[8.254]*	blockram *[8.244]*	shiftregister *[8.256]*
4096	distributedram *[15.913]*	distributedram *[15.919]*	distributedram *[15.915]*
8192	distributedram *[29.598]*	distributedram *[29.613]*	distributedram *[29.614]*
16384	shiftregister *[51.947]*	shiftregister *[51.976]*	blockram *[51.941]*
32768	blockram *[91.806]*	distributedram *[91.617]*	blockram *[91.628]*
65536	shiftregister *[148.745]*	shiftregister *[148.496]*	distributedram *[148.574]*
131072	distributedram *[215.843]*	distributedram *[215.991]*	distributedram *[215.828]*
262144	distributedram *[278.975]*	distributedram *[279.038]*	distributedram *[279.068]*
524288	distributedram *[327.229]*	blockram *[327.188]*	shiftregister *[327.069]*
1048576	distributedram *[353.201]*	shiftregister *[353.153]*	distributedram *[353.215]*
2097152	shiftregister *[374.468]*	shiftregister *[374.409]*	shiftregister *[374.417]*
4194304	shiftregister *[384.840]*	distributedram *[384.863]*	shiftregister *[384.881]*
8388608	distributedram *[390.351]*	shiftregister *[390.336]*	distributedram *[390.341]*
16777216	shiftregister *[391.584]*	shiftregister *[391.568]*	distributedram *[391.584]*
33554432	distributedram *[392.873]*	distributedram *[392.860]*	shiftregister *[392.854]*
67108864	distributedram *[393.512]*	distributedram *[393.510]*	distributedram *[393.518]*
134217728	distributedram *[393.816]*	distributedram *[393.804]*	shiftregister *[393.829]*
268435456	shiftregister *[393.985]*	shiftregister *[394.003]*	distributedram *[393.999]*
536870912	shiftregister *[394.078]*	blockram *[394.078]*	distributedram *[394.072]*
1073741824	distributedram *[394.101]*	distributedram *[394.096]*	shiftregister *[394.106]*
Most frequent parameter	distributedram	shiftregister	distributedram

Table 25. *The fastest memory types (values in square brackets are in MB/s) compared between pattern types (32bit mode, write direction, 16 depth value).*

Pattern size [B]	**counter__8bit**	**counter__32bit**	**walking__1**
16	shiftregister [*0.054*]	shiftregister [*0.054*]	distributedram [*0.054*]
32	distributedram [*0.107*]	shiftregister [*0.107*]	shiftregister [*0.107*]
64	shiftregister [*0.215*]	distributedram [*0.215*]	distributedram [*0.215*]
128	distributedram [*0.430*]	shiftregister [*0.430*]	shiftregister [*0.430*]
256	distributedram [*0.860*]	shiftregister [*0.860*]	shiftregister [*0.860*]
512	shiftregister [*1.718*]	distributedram [*1.710*]	distributedram [*1.717*]
1024	distributedram [*3.439*]	distributedram [*3.437*]	shiftregister [*3.439*]
2048	distributedram [*6.844*]	distributedram [*6.853*]	distributedram [*6.844*]
4096	shiftregister [*13.354*]	distributedram [*13.352*]	distributedram [*13.351*]
8192	shiftregister [*25.712*]	shiftregister [*25.709*]	shiftregister [*25.666*]
16384	shiftregister [*0.701*]	shiftregister [*0.687*]	shiftregister [*0.714*]
32768	blockram [*1.312*]	distributedram [*1.312*]	shiftregister [*1.372*]
65536	distributedram [*2.616*]	blockram [*2.616*]	blockram [*2.589*]
131072	shiftregister [*5.147*]	shiftregister [*5.147*]	blockram [*5.093*]
262144	shiftregister [*10.060*]	shiftregister [*10.061*]	blockram [*10.062*]
524288	blockram [*19.833*]	blockram [*19.630*]	blockram [*19.630*]
1048576	blockram [*38.029*]	blockram [*38.390*]	distributedram [*49.550*]
2097152	blockram [*68.183*]	blockram [*68.176*]	distributedram [*68.748*]
4194304	shiftregister [*116.976*]	blockram [*116.124*]	blockram [*116.129*]
8388608	blockram [*179.102*]	distributedram [*180.048*]	blockram [*179.108*]
16777216	shiftregister [*181.116*]	blockram [*184.786*]	blockram [*187.065*]
33554432	distributedram [*210.713*]	blockram [*210.333*]	distributedram [*209.683*]
67108864	distributedram [*226.209*]	distributedram [*225.434*]	distributedram [*225.034*]
134217728	blockram [*235.094*]	blockram [*235.169*]	blockram [*235.094*]
268435456	distributedram [*240.414*]	blockram [*240.482*]	blockram [*240.427*]
536870912	distributedram [*243.219*]	blockram [*243.243*]	blockram [*243.259*]
1073741824	blockram [*244.678*]	distributedram [*244.689*]	distributedram [*244.645*]
Most frequent parameter	shiftregister, distributedram	blockram	distributedram, blockram

Table 26. *The fastest memory types (values in square brackets are in MB/s) compared between pattern types (32bit mode, write direction, 64 depth value).*

Pattern size [B]	counter__8bit	counter__32bit	walking__1
16	shiftregister [*0.054*]	shiftregister [*0.054*]	shiftregister [*0.054*]
32	shiftregister [*0.108*]	shiftregister [*0.108*]	shiftregister [*0.108*]
64	shiftregister [*0.215*]	shiftregister [*0.215*]	shiftregister [*0.215*]
128	shiftregister [*0.430*]	shiftregister [*0.430*]	blockram [*0.430*]
256	shiftregister [*0.860*]	blockram [*0.859*]	shiftregister [*0.861*]
512	blockram [*1.718*]	shiftregister [*1.717*]	blockram [*1.718*]
1024	blockram [*3.439*]	blockram [*3.436*]	blockram [*3.435*]
2048	blockram [*6.840*]	blockram [*6.842*]	blockram [*6.836*]
4096	blockram [*13.340*]	blockram [*13.355*]	blockram [*13.347*]
8192	shiftregister [*25.730*]	blockram [*25.713*]	shiftregister [*25.740*]
16384	shiftregister [*0.696*]	blockram [*0.687*]	distributedram [*0.678*]
32768	distributedram [*1.313*]	blockram [*1.330*]	distributedram [*1.313*]
65536	distributedram [*2.617*]	blockram [*2.561*]	distributedram [*2.589*]
131072	distributedram [*5.147*]	blockram [*5.203*]	blockram [*5.147*]
262144	blockram [*10.061*]	shiftregister [*10.061*]	distributedram [*10.168*]
524288	shiftregister [*19.629*]	shiftregister [*19.628*]	distributedram [*19.629*]
1048576	blockram [*38.024*]	shiftregister [*38.835*]	distributedram [*42.967*]
2097152	shiftregister [*68.182*]	distributedram [*68.772*]	shiftregister [*68.186*]
4194304	distributedram [*116.139*]	shiftregister [*116.129*]	shiftregister [*116.135*]
8388608	distributedram [*179.136*]	distributedram [*179.125*]	distributedram [*179.125*]
16777216	shiftregister [*186.764*]	distributedram [*185.361*]	distributedram [*182.744*]
33554432	distributedram [*210.001*]	blockram [*209.955*]	shiftregister [*211.025*]
67108864	blockram [*225.634*]	distributedram [*225.831*]	distributedram [*225.438*]
134217728	blockram [*235.332*]	shiftregister [*235.218*]	blockram [*235.090*]
268435456	shiftregister [*240.449*]	distributedram [*240.480*]	blockram [*240.438*]
536870912	distributedram [*243.221*]	distributedram [*243.271*]	distributedram [*243.244*]
1073741824	shiftregister [*244.673*]	distributedram [*244.765*]	shiftregister [*244.705*]
Most frequent parameter	shiftregister	shiftregister, blockram	distributedram

Table 27. *The fastest memory types (values in square brackets are in MB/s) compared between pattern types (32bit mode, write direction, 256 depth value).*

Pattern size [B]	counter__8bit	counter__32bit	walking__1
16	blockram *[0.054]*	distributedram *[0.054]*	blockram *[0.054]*
32	distributedram *[0.108]*	distributedram *[0.108]*	distributedram *[0.108]*
64	distributedram *[0.215]*	distributedram *[0.215]*	distributedram *[0.215]*
128	distributedram *[0.430]*	distributedram *[0.430]*	distributedram *[0.430]*
256	distributedram *[0.859]*	distributedram *[0.859]*	distributedram *[0.859]*
512	distributedram *[1.716]*	distributedram *[1.715]*	distributedram *[1.716]*
1024	blockram *[3.432]*	distributedram *[3.430]*	distributedram *[3.437]*
2048	distributedram *[6.847]*	distributedram *[6.839]*	distributedram *[6.840]*
4096	distributedram *[13.267]*	distributedram *[13.351]*	distributedram *[13.306]*
8192	distributedram *[25.652]*	distributedram *[25.706]*	distributedram *[25.706]*
16384	distributedram *[0.705]*	distributedram *[0.709]*	distributedram *[0.671]*
32768	distributedram *[1.358]*	shiftregister *[1.413]*	shiftregister *[1.534]*
65536	blockram *[2.589]*	blockram *[2.588]*	shiftregister *[2.651]*
131072	distributedram *[5.093]*	blockram *[5.093]*	distributedram *[5.092]*
262144	distributedram *[10.166]*	blockram *[10.060]*	shiftregister *[10.168]*
524288	shiftregister *[19.629]*	shiftregister *[19.628]*	shiftregister *[19.629]*
1048576	distributedram *[49.369]*	distributedram *[42.002]*	distributedram *[45.246]*
2097152	shiftregister *[68.182]*	shiftregister *[68.176]*	distributedram *[68.769]*
4194304	shiftregister *[116.120]*	shiftregister *[116.132]*	shiftregister *[116.124]*
8388608	shiftregister *[179.094]*	distributedram *[179.112]*	distributedram *[179.137]*
16777216	shiftregister *[186.337]*	blockram *[184.868]*	distributedram *[182.062]*
33554432	distributedram *[208.281]*	distributedram *[211.014]*	distributedram *[209.704]*
67108864	blockram *[226.216]*	distributedram *[225.780]*	shiftregister *[226.014]*
134217728	shiftregister *[235.096]*	shiftregister *[235.185]*	shiftregister *[235.350]*
268435456	shiftregister *[240.481]*	blockram *[240.484]*	distributedram *[240.535]*
536870912	distributedram *[243.252]*	blockram *[243.278]*	blockram *[243.278]*
1073741824	distributedram *[244.699]*	shiftregister *[244.756]*	distributedram *[244.711]*
Most frequent parameter	distributedram	distributedram	distributedram

Table 28. *The fastest memory types (values in square brackets are in MB/s) compared between pattern types (32bit mode, write direction, 1024 depth value).*

Pattern size [B]	**counter__8bit**	**counter__32bit**	**walking__1**
16	shiftregister [*0.053*]	blockram [*0.053*]	shiftregister [*0.054*]
32	blockram [*0.107*]	blockram [*0.107*]	distributedram [*0.107*]
64	distributedram [*0.214*]	blockram [*0.214*]	distributedram [*0.214*]
128	blockram [*0.428*]	distributedram [*0.429*]	distributedram [*0.429*]
256	distributedram [*0.860*]	blockram [*0.855*]	shiftregister [*0.855*]
512	shiftregister [*1.706*]	shiftregister [*1.706*]	blockram [*1.707*]
1024	distributedram [*3.416*]	distributedram [*3.416*]	shiftregister [*3.425*]
2048	shiftregister [*6.851*]	blockram [*6.809*]	shiftregister [*6.854*]
4096	blockram [*13.223*]	shiftregister [*13.213*]	distributedram [*13.245*]
8192	shiftregister [*25.466*]	blockram [*25.500*]	shiftregister [*25.462*]
16384	distributedram [*0.657*]	distributedram [*0.650*]	shiftregister [*0.712*]
32768	distributedram [*1.312*]	blockram [*1.344*]	blockram [*1.340*]
65536	blockram [*2.589*]	distributedram [*2.800*]	distributedram [*2.650*]
131072	shiftregister [*5.201*]	blockram [*5.092*]	shiftregister [*5.092*]
262144	distributedram [*10.060*]	distributedram [*10.061*]	shiftregister [*10.060*]
524288	blockram [*19.628*]	shiftregister [*19.628*]	shiftregister [*19.629*]
1048576	distributedram [*44.433*]	distributedram [*42.366*]	shiftregister [*40.664*]
2097152	blockram [*68.181*]	shiftregister [*68.177*]	blockram [*68.179*]
4194304	blockram [*116.136*]	blockram [*116.137*]	blockram [*116.128*]
8388608	shiftregister [*179.084*]	distributedram [*180.997*]	blockram [*180.109*]
16777216	distributedram [*183.243*]	distributedram [*186.452*]	blockram [*185.761*]
33554432	blockram [*211.038*]	blockram [*210.028*]	blockram [*211.339*]
67108864	distributedram [*225.624*]	blockram [*226.194*]	blockram [*226.019*]
134217728	blockram [*235.274*]	shiftregister [*235.085*]	distributedram [*235.076*]
268435456	blockram [*240.434*]	distributedram [*240.495*]	blockram [*240.437*]
536870912	shiftregister [*243.279*]	blockram [*243.244*]	distributedram [*243.230*]
1073741824	blockram [*244.690*]	distributedram [*244.693*]	shiftregister [*244.709*]
Most frequent parameter	blockram	blockram	shiftregister

Table 29. *The fastest memory types (values in square brackets are in MB/s) compared between pattern types (32bit mode, write direction, 2048 depth value).*

Pattern size [B]	**counter__8bit**	**counter__32bit**	**walking__1**
16	blockram *[0.054]*	blockram *[0.054]*	blockram *[0.054]*
32	distributedram *[0.107]*	blockram *[0.107]*	blockram *[0.107]*
64	blockram *[0.214]*	blockram *[0.215]*	blockram *[0.215]*
128	blockram *[0.430]*	blockram *[0.429]*	blockram *[0.429]*
256	blockram *[0.859]*	blockram *[0.860]*	blockram *[0.852]*
512	blockram *[1.710]*	blockram *[1.710]*	blockram *[1.708]*
1024	blockram *[3.433]*	blockram *[3.434]*	blockram *[3.437]*
2048	blockram *[6.847]*	blockram *[6.846]*	blockram *[6.839]*
4096	blockram *[13.339]*	blockram *[13.334]*	blockram *[13.343]*
8192	blockram *[25.663]*	blockram *[25.623]*	blockram *[25.574]*
16384	shiftregister *[0.657]*	shiftregister *[0.680]*	shiftregister *[0.650]*
32768	shiftregister *[1.344]*	blockram *[1.598]*	shiftregister *[1.326]*
65536	blockram *[2.589]*	blockram *[2.616]*	distributedram *[2.623]*
131072	distributedram *[5.147]*	distributedram *[5.147]*	distributedram *[5.092]*
262144	shiftregister *[10.060]*	distributedram *[10.060]*	shiftregister *[10.060]*
524288	shiftregister *[19.628]*	blockram *[19.829]*	shiftregister *[19.628]*
1048576	blockram *[41.010]*	blockram *[43.821]*	blockram *[42.179]*
2097152	distributedram *[68.182]*	blockram *[68.766]*	shiftregister *[68.179]*
4194304	distributedram *[116.131]*	blockram *[116.953]*	distributedram *[116.151]*
8388608	distributedram *[179.105]*	blockram *[179.115]*	blockram *[179.122]*
16777216	distributedram *[183.216]*	blockram *[185.262]*	shiftregister *[182.219]*
33554432	shiftregister *[210.642]*	blockram *[210.343]*	blockram *[209.005]*
67108864	blockram *[225.794]*	shiftregister *[225.824]*	shiftregister *[226.376]*
134217728	blockram *[235.226]*	distributedram *[235.271]*	shiftregister *[235.214]*
268435456	distributedram *[240.492]*	shiftregister *[240.465]*	shiftregister *[240.523]*
536870912	blockram *[243.221]*	distributedram *[243.272]*	blockram *[243.283]*
1073741824	shiftregister *[244.697]*	distributedram *[244.664]*	distributedram *[244.637]*
Most frequent parameter	blockram	blockram	blockram

A.3. Graphical presentation of results from the FIFO depth value impact investigation subtest

A.3.1. nonsym mode part of the subtest

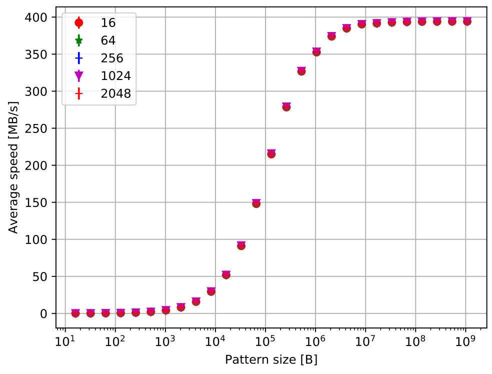


Figure 153. Speed results for FIFO depth value impact subtest in *nonsym read* mode (*counter_8bit* pattern type and *blockram* FIFO memory type).

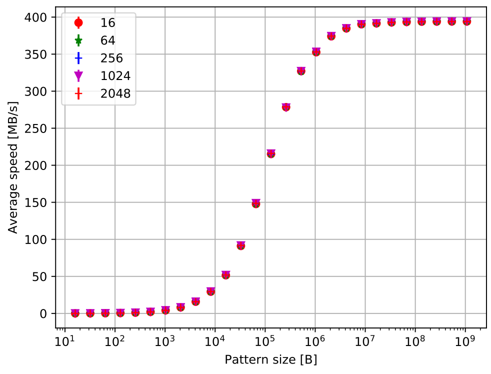


Figure 154. Speed results for FIFO depth value impact subtest in *nonsym read* mode (*counter_32bit* pattern type and *blockram* FIFO memory type).

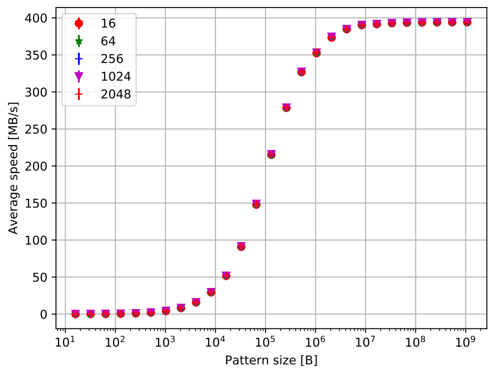


Figure 155. Speed results for FIFO depth value impact subtest in *nonsym read* mode (*walking_1* pattern type and *blockram* FIFO memory type).

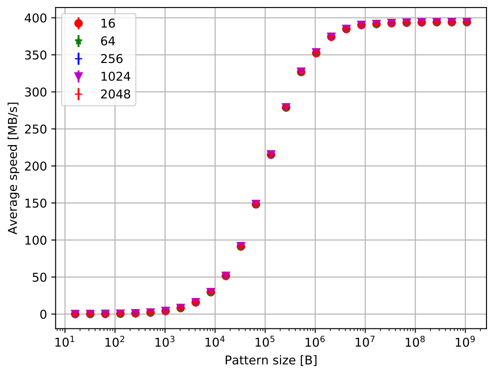


Figure 156. Speed results for FIFO depth value impact subtest in *nonsym read* mode (*asic* pattern type and *blockram* FIFO memory type).

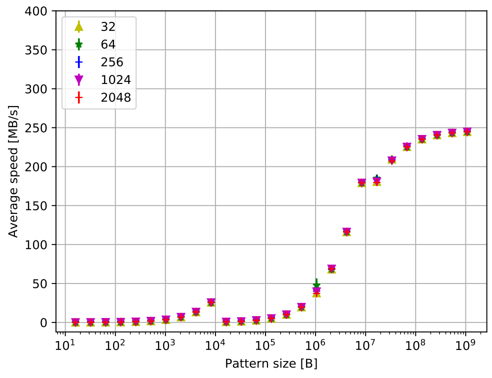


Figure 157. Speed results for FIFO depth value impact subtest in *nonsym write* mode (*counter_8bit* pattern type and *blockram* FIFO memory type).

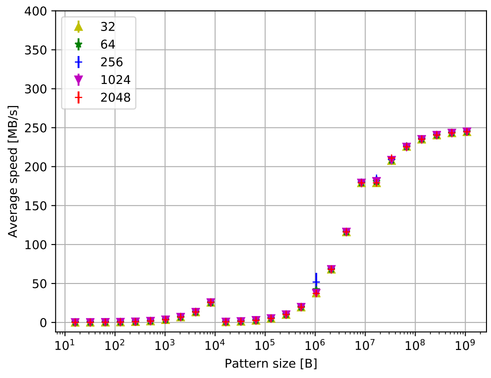


Figure 158. Speed results for FIFO depth value impact subtest in *nonsym write* mode (*counter_32bit* pattern type and *blockram* FIFO memory type).

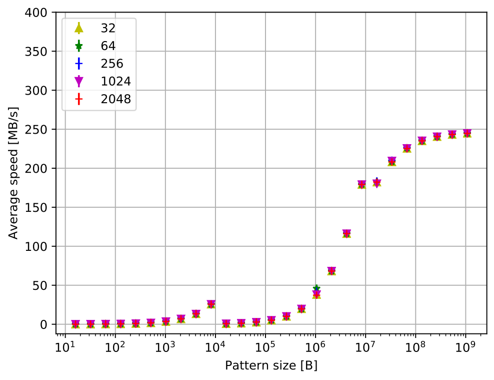


Figure 159. Speed results for FIFO depth value impact subtest in *nonsym write* mode (*walking_1* pattern type and *blockram* FIFO memory type).

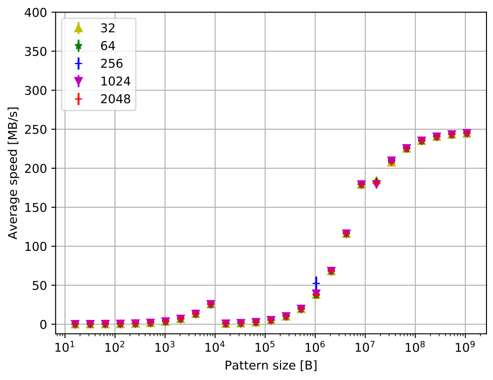


Figure 160. Speed results for FIFO depth value impact subtest in *nonsym write* mode (*asic* pattern type and *blockram* FIFO memory type).

Table 30. *The fastest depth values (speeds in square brackets are in MB/s) compared between pattern types (nonsym mode, read direction, blockram memory type).*

Pattern size [B]	**counter__8bit**	**counter__32bit**	**walking__1**
16	2048 [*0.066*]	1024 [*0.066*]	2048 [*0.066*]
32	1024 [*0.132*]	1024 [*0.132*]	1024 [*0.132*]
64	1024 [*0.263*]	1024 [*0.264*]	1024 [*0.263*]
128	2048 [*0.528*]	1024 [*0.527*]	2048 [*0.526*]
256	1024 [*1.054*]	2048 [*1.053*]	1024 [*1.054*]
512	2048 [*2.106*]	1024 [*2.105*]	1024 [*2.107*]
1024	1024 [*4.195*]	2048 [*4.204*]	1024 [*4.209*]
2048	1024 [*8.242*]	1024 [*8.246*]	2048 [*8.241*]
4096	1024 [*15.911*]	1024 [*15.905*]	2048 [*15.933*]
8192	1024 [*29.604*]	1024 [*29.619*]	1024 [*29.609*]
16384	2048 [*51.974*]	2048 [*51.919*]	2048 [*51.930*]
32768	1024 [*91.679*]	2048 [*91.823*]	2048 [*91.752*]
65536	1024 [*148.700*]	1024 [*148.868*]	1024 [*148.779*]
131072	64 [*216.054*]	2048 [*215.974*]	1024 [*215.802*]
262144	1024 [*279.065*]	2048 [*278.947*]	2048 [*279.013*]
524288	1024 [*327.189*]	2048 [*327.184*]	1024 [*327.228*]
1048576	1024 [*353.228*]	1024 [*353.200*]	2048 [*353.290*]
2097152	2048 [*374.402*]	2048 [*374.425*]	1024 [*374.413*]
4194304	1024 [*384.885*]	1024 [*384.874*]	2048 [*384.861*]
8388608	1024 [*390.348*]	2048 [*390.347*]	1024 [*390.349*]
16777216	1024 [*391.592*]	2048 [*391.591*]	1024 [*391.596*]
33554432	64 [*392.872*]	2048 [*392.874*]	256 [*392.871*]
67108864	64 [*393.517*]	1024 [*393.513*]	16 [*393.514*]
134217728	16 [*393.841*]	256 [*393.836*]	16 [*393.839*]
268435456	2048 [*393.996*]	1024 [*393.997*]	1024 [*393.999*]
536870912	256 [*394.082*]	2048 [*394.074*]	256 [*394.078*]
1073741824	64 [*394.113*]	16 [*394.112*]	1024 [*394.113*]
Most frequent parameter	1024	1024	1024

Table 31. *The fastest depth values (speeds in square brackets are in MB/s) compared between pattern types (nonsym mode, write direction, blockram memory type).*

Pattern size [B]	counter__8bit	counter__32bit	walking__1
16	2048 [*0.054*]	32 [*0.054*]	32 [*0.054*]
32	32 [*0.107*]	32 [*0.107*]	32 [*0.107*]
64	2048 [*0.215*]	1024 [*0.215*]	1024 [*0.215*]
128	1024 [*0.430*]	32 [*0.430*]	32 [*0.430*]
256	2048 [*0.860*]	2048 [*0.859*]	32 [*0.861*]
512	32 [*1.717*]	32 [*1.717*]	2048 [*1.717*]
1024	32 [*3.437*]	32 [*3.438*]	1024 [*3.439*]
2048	32 [*6.849*]	32 [*6.852*]	2048 [*6.843*]
4096	2048 [*13.357*]	32 [*13.360*]	32 [*13.363*]
8192	2048 [*25.737*]	32 [*25.703*]	32 [*25.718*]
16384	2048 [*0.738*]	2048 [*0.703*]	256 [*0.700*]
32768	256 [*1.344*]	1024 [*1.326*]	64 [*1.343*]
65536	1024 [*2.672*]	1024 [*2.758*]	2048 [*2.824*]
131072	32 [*5.147*]	256 [*5.147*]	32 [*5.093*]
262144	256 [*10.061*]	256 [*10.166*]	2048 [*10.061*]
524288	256 [*19.629*]	2048 [*19.629*]	32 [*19.828*]
1048576	64 [*47.582*]	256 [*51.847*]	64 [*45.874*]
2097152	1024 [*68.772*]	64 [*68.189*]	64 [*68.190*]
4194304	64 [*116.143*]	64 [*116.144*]	2048 [*116.145*]
8388608	64 [*179.142*]	256 [*180.102*]	64 [*179.121*]
16777216	64 [*184.886*]	256 [*185.187*]	256 [*183.759*]
33554432	32 [*209.324*]	2048 [*211.023*]	256 [*210.356*]
67108864	256 [*226.582*]	256 [*226.403*]	256 [*226.211*]
134217728	32 [*235.185*]	64 [*235.279*]	1024 [*235.179*]
268435456	64 [*240.485*]	256 [*240.524*]	32 [*240.484*]
536870912	32 [*243.246*]	32 [*243.254*]	256 [*243.225*]
1073741824	256 [*244.705*]	64 [*244.770*]	32 [*244.743*]
Most frequent parameter	32	32	32

A.3.2. 32bit mode part of the subtest

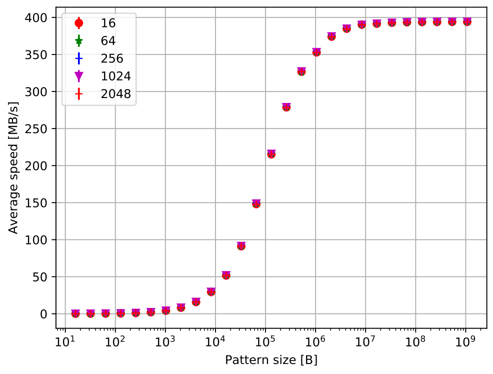


Figure 161. Speed results for FIFO depth value impact subtest in *32bit read* mode (*counter_8bit* pattern type and *blockram* FIFO memory type).

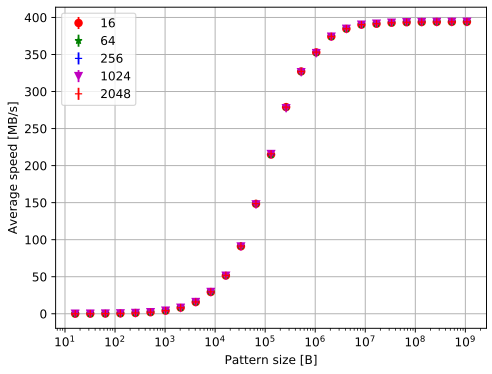


Figure 162. Speed results for FIFO depth value impact subtest in *32bit read* mode (*counter_8bit* pattern type and *distributedram* FIFO memory type).

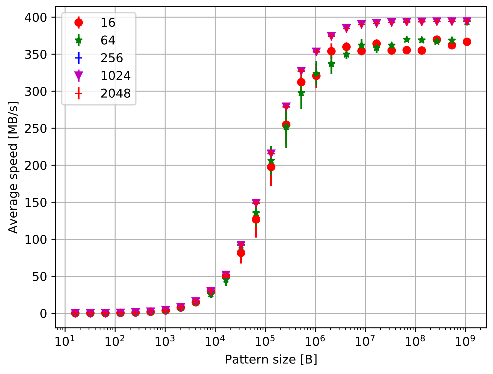


Figure 163. Speed results for FIFO depth value impact subtest in *32bit read* mode (*counter_8bit* pattern type and *shiftregister* FIFO memory type).

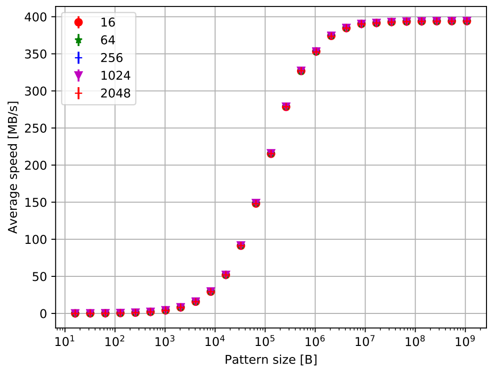


Figure 164. Speed results for FIFO depth value impact subtest in *32bit read* mode (*counter_32bit* pattern type and *blockram* FIFO memory type).

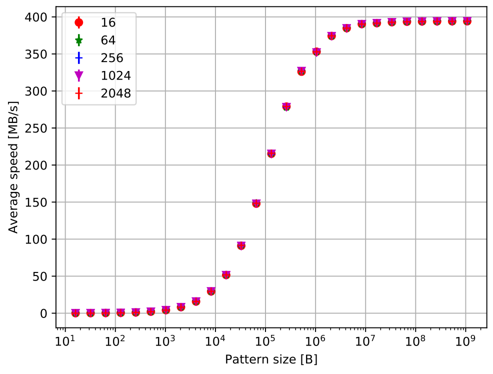


Figure 165. Speed results for FIFO depth value impact subtest in *32bit read* mode (*counter_32bit* pattern type and *distributedram* FIFO memory type).

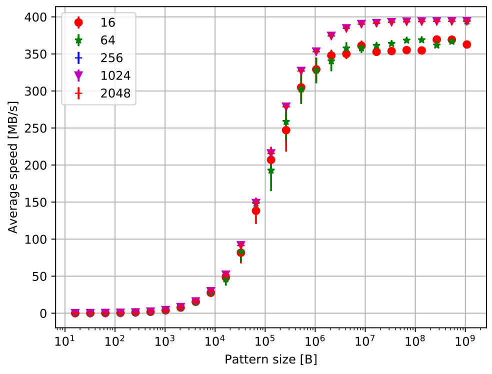


Figure 166. Speed results for FIFO depth value impact subtest in *32bit read* mode (*counter_32bit* pattern type and *shiftregister* FIFO memory type).

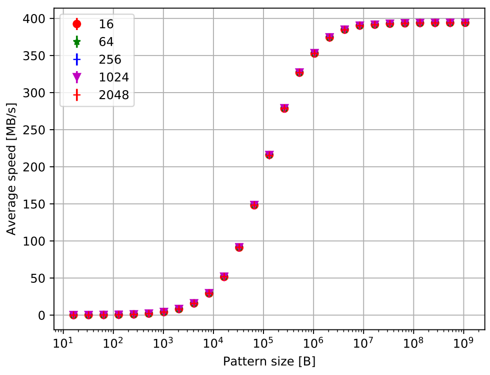


Figure 167. Speed results for FIFO depth value impact subtest in *32bit read* mode (*walking_1* pattern type and *blockram* FIFO memory type).

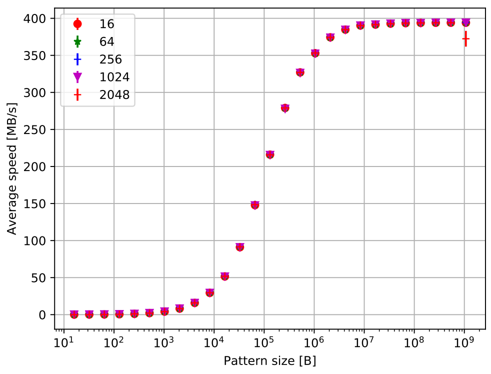


Figure 168. Speed results for FIFO depth value impact subtest in *32bit read* mode (*walking_1* pattern type and *distributedram* FIFO memory type).

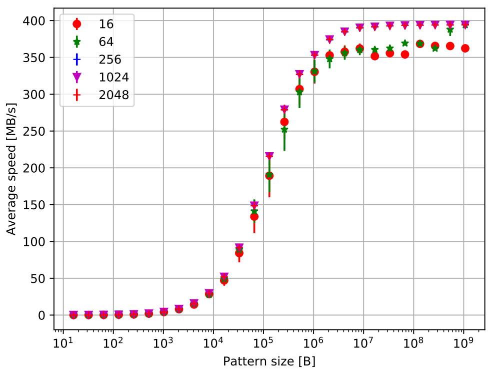


Figure 169. Speed results for FIFO depth value impact subtest in *32bit read* mode (*walking_1* pattern type and *shiftregister* FIFO memory type).

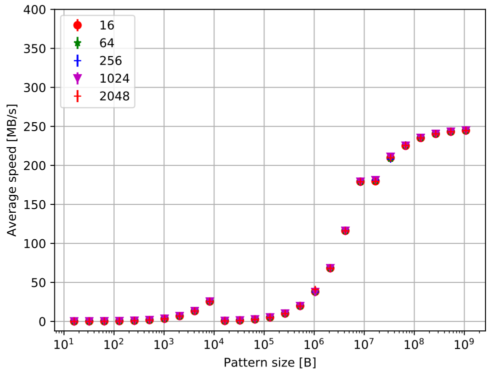


Figure 170. Speed results for FIFO depth value impact subtest in *32bit write* mode (*counter_8bit* pattern type and *blockram* FIFO memory type).

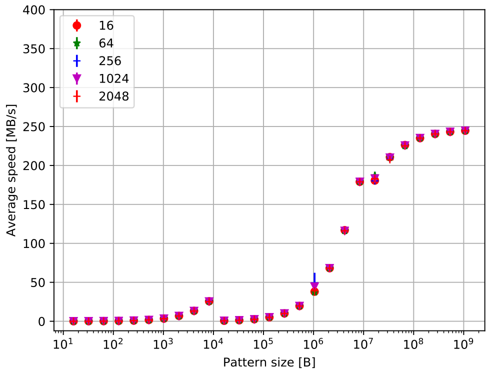


Figure 171. Speed results for FIFO depth value impact subtest in *32bit write* mode (*counter_8bit* pattern type and *distributedram* FIFO memory type).

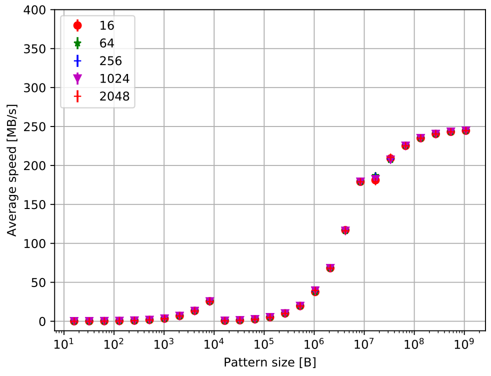


Figure 172. Speed results for FIFO depth value impact subtest in *32bit write* mode (*counter_8bit* pattern type and *shiftregister* FIFO memory type).

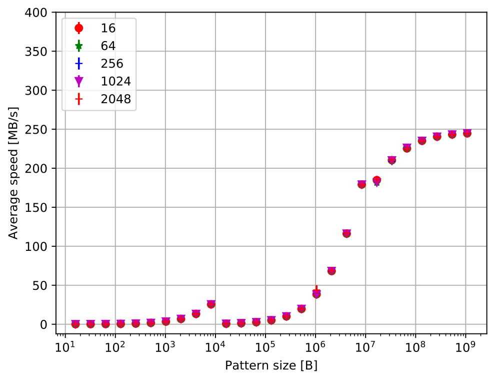


Figure 173. Speed results for FIFO depth value impact subtest in *32bit write* mode (*counter_32bit* pattern type and *blockram* FIFO memory type).

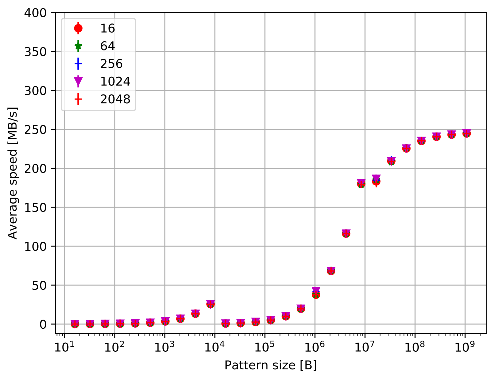


Figure 174. Speed results for FIFO depth value impact subtest in *32bit write* mode (*counter_32bit* pattern type and *distributedram* FIFO memory type).

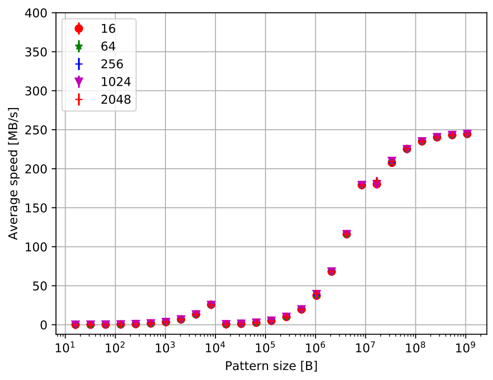


Figure 175. Speed results for FIFO depth value impact subtest in *32bit write* mode (*counter_32bit* pattern type and *shiftregister* FIFO memory type).

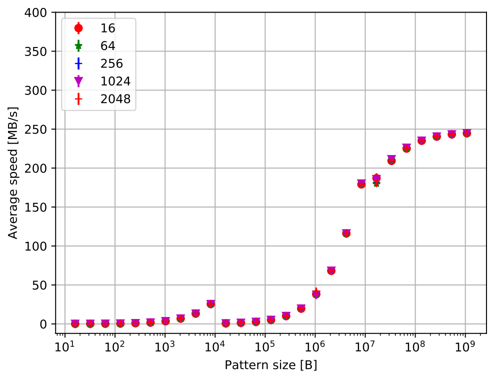


Figure 176. Speed results for FIFO depth value impact subtest in *32bit write* mode (*walking_1* pattern type and *blockram* FIFO memory type).

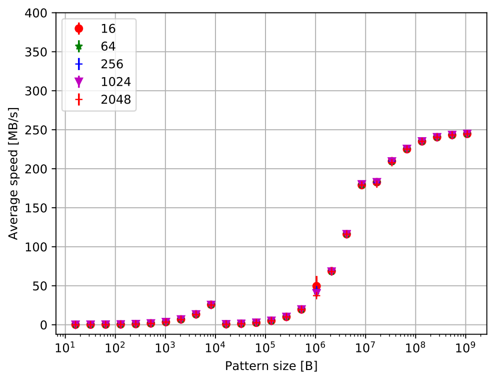


Figure 177. Speed results for FIFO depth value impact subtest in *32bit write* mode (*walking_1* pattern type and *distributedram* FIFO memory type).

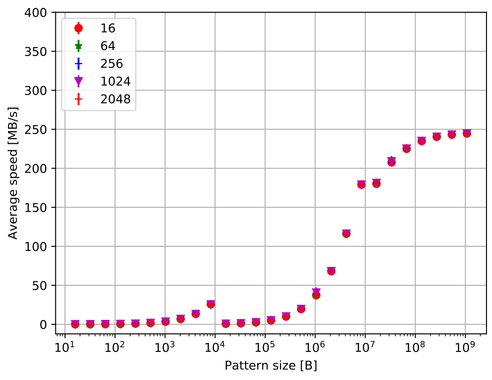


Figure 178. Speed results for FIFO depth value impact subtest in *32bit write* mode (*walking_1* pattern type and *shiftregister* FIFO memory type).

Table 32. *The fastest depth values (speeds in square brackets are in MB/s) compared between memory types (32bit mode, read direction, counter_8bit pattern type).*

Pattern size [B]	blockram	distributedram	shiftregister
16	64 *[0.066]*	2048 *[0.066]*	1024 *[0.066]*
32	1024 *[0.132]*	2048 *[0.132]*	2048 *[0.132]*
64	64 *[0.264]*	2048 *[0.263]*	2048 *[0.264]*
128	64 *[0.528]*	2048 *[0.526]*	2048 *[0.527]*
256	1024 *[1.054]*	256 *[1.056]*	1024 *[1.053]*
512	64 *[2.111]*	2048 *[2.107]*	2048 *[2.107]*
1024	256 *[4.196]*	2048 *[4.200]*	1024 *[4.203]*
2048	64 *[8.249]*	256 *[8.249]*	2048 *[8.254]*
4096	1024 *[15.924]*	2048 *[15.913]*	256 *[15.919]*
8192	256 *[29.621]*	2048 *[29.598]*	256 *[29.620]*
16384	256 *[51.932]*	64 *[51.938]*	2048 *[51.947]*
32768	2048 *[91.806]*	64 *[91.560]*	1024 *[91.683]*
65536	256 *[148.730]*	2048 *[148.649]*	256 *[148.781]*
131072	64 *[216.011]*	2048 *[215.843]*	2048 *[215.546]*
262144	256 *[278.965]*	2048 *[278.975]*	1024 *[278.995]*
524288	256 *[327.101]*	2048 *[327.229]*	256 *[327.212]*
1048576	256 *[353.209]*	2048 *[353.201]*	1024 *[353.185]*
2097152	64 *[374.407]*	256 *[374.434]*	2048 *[374.468]*
4194304	64 *[384.869]*	256 *[384.898]*	2048 *[384.840]*
8388608	16 *[390.351]*	2048 *[390.351]*	1024 *[390.341]*
16777216	1024 *[391.583]*	256 *[391.590]*	1024 *[391.585]*
33554432	16 *[392.863]*	2048 *[392.873]*	256 *[392.868]*
67108864	16 *[393.517]*	2048 *[393.512]*	256 *[393.516]*
134217728	1024 *[393.835]*	256 *[393.833]*	256 *[393.844]*
268435456	1024 *[393.982]*	64 *[393.995]*	1024 *[393.990]*
536870912	64 *[394.080]*	1024 *[394.074]*	2048 *[394.078]*
1073741824	16 *[394.109]*	16 *[394.106]*	256 *[394.097]*
Most frequent parameter	64	2048	2048

Table 33. *The fastest depth values (speeds in square brackets are in MB/s) compared between memory types (32bit mode, read direction, counter_32bit pattern type).*

Pattern size [B]	blockram	distributedram	shiftregister
16	64 [*0.066*]	2048 [*0.066*]	2048 [*0.066*]
32	256 [*0.132*]	64 [*0.132*]	256 [*0.132*]
64	64 [*0.263*]	2048 [*0.264*]	2048 [*0.264*]
128	256 [*0.527*]	256 [*0.527*]	2048 [*0.528*]
256	256 [*1.053*]	256 [*1.057*]	1024 [*1.054*]
512	1024 [*2.106*]	64 [*2.108*]	256 [*2.107*]
1024	64 [*4.207*]	2048 [*4.204*]	2048 [*4.204*]
2048	64 [*8.249*]	256 [*8.244*]	1024 [*8.253*]
4096	1024 [*15.908*]	2048 [*15.919*]	256 [*15.885*]
8192	64 [*29.643*]	2048 [*29.613*]	1024 [*29.607*]
16384	16 [*51.933*]	2048 [*51.776*]	1024 [*51.991*]
32768	256 [*91.614*]	64 [*91.664*]	1024 [*91.670*]
65536	1024 [*148.776*]	64 [*148.720*]	256 [*148.732*]
131072	1024 [*215.868*]	2048 [*215.991*]	1024 [*215.740*]
262144	64 [*279.063*]	2048 [*279.038*]	256 [*278.770*]
524288	256 [*327.203*]	1024 [*326.884*]	1024 [*327.009*]
1048576	16 [*353.151*]	16 [*353.030*]	2048 [*353.153*]
2097152	64 [*374.437*]	256 [*374.423*]	2048 [*374.409*]
4194304	256 [*384.872*]	256 [*384.863*]	2048 [*384.844*]
8388608	16 [*390.341*]	2048 [*390.336*]	1024 [*390.343*]
16777216	16 [*391.590*]	16 [*391.569*]	1024 [*391.571*]
33554432	64 [*392.871*]	2048 [*392.860*]	256 [*392.873*]
67108864	16 [*393.518*]	64 [*393.513*]	1024 [*393.512*]
134217728	256 [*393.817*]	256 [*393.841*]	256 [*393.840*]
268435456	1024 [*394.002*]	16 [*393.999*]	2048 [*394.003*]
536870912	2048 [*394.078*]	256 [*394.068*]	2048 [*394.075*]
1073741824	256 [*394.103*]	64 [*394.102*]	256 [*394.101*]
Most frequent parameter	64, 256	2048	1024

Table 34. *The fastest depth values (speeds in square brackets are in MB/s) compared between memory types (32bit mode, read direction, walking_1 pattern type).*

Pattern size [B]	**blockram**	**distributedram**	**shiftregister**
16	256 [*0.066*]	64 [*0.066*]	2048 [*0.066*]
32	64 [*0.132*]	64 [*0.132*]	1024 [*0.132*]
64	64 [*0.264*]	2048 [*0.263*]	2048 [*0.263*]
128	1024 [*0.528*]	64 [*0.526*]	2048 [*0.527*]
256	64 [*1.054*]	64 [*1.053*]	2048 [*1.054*]
512	64 [*2.103*]	2048 [*2.103*]	1024 [*2.107*]
1024	64 [*4.206*]	256 [*4.208*]	2048 [*4.210*]
2048	1024 [*8.259*]	2048 [*8.246*]	2048 [*8.256*]
4096	1024 [*15.928*]	2048 [*15.915*]	1024 [*15.907*]
8192	64 [*29.635*]	2048 [*29.614*]	256 [*29.657*]
16384	64 [*51.995*]	2048 [*51.898*]	256 [*51.920*]
32768	256 [*91.875*]	64 [*91.623*]	256 [*91.676*]
65536	256 [*148.762*]	256 [*148.760*]	256 [*148.623*]
131072	16 [*216.013*]	16 [*215.960*]	256 [*215.823*]
262144	256 [*279.096*]	2048 [*279.068*]	256 [*279.032*]
524288	256 [*327.124*]	16 [*327.012*]	1024 [*327.098*]
1048576	1024 [*353.194*]	2048 [*353.215*]	2048 [*353.198*]
2097152	1024 [*374.452*]	256 [*374.457*]	2048 [*374.417*]
4194304	64 [*384.863*]	2048 [*384.868*]	2048 [*384.881*]
8388608	16 [*390.338*]	2048 [*390.341*]	1024 [*390.286*]
16777216	16 [*391.583*]	2048 [*391.584*]	256 [*391.583*]
33554432	16 [*392.870*]	256 [*392.867*]	1024 [*392.869*]
67108864	64 [*393.515*]	2048 [*393.518*]	256 [*393.515*]
134217728	2048 [*393.827*]	16 [*393.838*]	2048 [*393.829*]
268435456	64 [*393.998*]	2048 [*393.999*]	1024 [*393.989*]
536870912	256 [*394.076*]	2048 [*394.072*]	1024 [*394.080*]
1073741824	256 [*394.104*]	64 [*394.099*]	2048 [*394.106*]
Most frequent parameter	64	2048	2048

Table 35. *The fastest depth values (speeds in square brackets are in MB/s) compared between memory types (32bit mode, write direction, counter_8bit pattern type).*

Pattern size [B]	**blockram**	**distributedram**	**shiftregister**
16	256 *[0.054]*	256 *[0.054]*	64 *[0.054]*
32	64 *[0.107]*	256 *[0.108]*	64 *[0.108]*
64	64 *[0.215]*	256 *[0.215]*	64 *[0.215]*
128	2048 *[0.430]*	16 *[0.430]*	64 *[0.430]*
256	2048 *[0.859]*	1024 *[0.860]*	64 *[0.860]*
512	64 *[1.718]*	256 *[1.716]*	16 *[1.718]*
1024	64 *[3.439]*	16 *[3.439]*	64 *[3.434]*
2048	2048 *[6.847]*	256 *[6.847]*	1024 *[6.851]*
4096	64 *[13.340]*	16 *[13.336]*	16 *[13.354]*
8192	64 *[25.691]*	16 *[25.680]*	64 *[25.730]*
16384	64 *[0.689]*	256 *[0.705]*	16 *[0.701]*
32768	16 *[1.312]*	256 *[1.358]*	2048 *[1.344]*
65536	256 *[2.589]*	64 *[2.617]*	64 *[2.616]*
131072	1024 *[5.148]*	64 *[5.147]*	1024 *[5.201]*
262144	64 *[10.061]*	256 *[10.166]*	2048 *[10.060]*
524288	16 *[19.833]*	64 *[19.628]*	64 *[19.629]*
1048576	2048 *[41.010]*	256 *[49.369]*	2048 *[39.485]*
2097152	16 *[68.183]*	2048 *[68.182]*	256 *[68.182]*
4194304	1024 *[116.136]*	16 *[116.909]*	16 *[116.976]*
8388608	64 *[179.112]*	64 *[179.136]*	2048 *[179.098]*
16777216	256 *[181.196]*	64 *[186.456]*	64 *[186.764]*
33554432	1024 *[211.038]*	16 *[210.713]*	2048 *[210.642]*
67108864	256 *[226.216]*	16 *[226.209]*	64 *[225.626]*
134217728	64 *[235.332]*	2048 *[235.185]*	256 *[235.096]*
268435456	256 *[240.466]*	2048 *[240.492]*	256 *[240.481]*
536870912	256 *[243.239]*	256 *[243.252]*	1024 *[243.279]*
1073741824	1024 *[244.690]*	256 *[244.699]*	2048 *[244.697]*
Most frequent parameter	64	256	64

Table 36. *The fastest depth values (speeds in square brackets are in MB/s) compared between memory types (32bit mode, write direction, counter_32bit pattern type).*

Pattern size [B]	blockram	distributedram	shiftregister
16	64 *[0.054]*	256 *[0.054]*	64 *[0.054]*
32	64 *[0.108]*	256 *[0.108]*	64 *[0.108]*
64	2048 *[0.215]*	16 *[0.215]*	64 *[0.215]*
128	2048 *[0.429]*	16 *[0.430]*	16 *[0.430]*
256	2048 *[0.860]*	256 *[0.859]*	16 *[0.860]*
512	64 *[1.717]*	256 *[1.715]*	64 *[1.717]*
1024	64 *[3.436]*	16 *[3.437]*	16 *[3.436]*
2048	2048 *[6.846]*	16 *[6.853]*	16 *[6.851]*
4096	64 *[13.355]*	16 *[13.352]*	16 *[13.333]*
8192	64 *[25.713]*	256 *[25.706]*	16 *[25.709]*
16384	64 *[0.687]*	256 *[0.709]*	16 *[0.687]*
32768	2048 *[1.598]*	1024 *[1.330]*	256 *[1.413]*
65536	16 *[2.616]*	1024 *[2.800]*	1024 *[2.644]*
131072	64 *[5.203]*	2048 *[5.147]*	16 *[5.147]*
262144	16 *[10.061]*	1024 *[10.061]*	64 *[10.061]*
524288	2048 *[19.829]*	2048 *[19.627]*	2048 *[19.629]*
1048576	2048 *[43.821]*	1024 *[42.366]*	2048 *[40.008]*
2097152	2048 *[68.766]*	64 *[68.772]*	1024 *[68.177]*
4194304	2048 *[116.953]*	2048 *[116.142]*	256 *[116.132]*
8388608	16 *[179.115]*	1024 *[180.997]*	256 *[179.111]*
16777216	2048 *[185.262]*	1024 *[186.452]*	256 *[184.802]*
33554432	2048 *[210.343]*	256 *[211.014]*	1024 *[209.960]*
67108864	1024 *[226.194]*	64 *[225.831]*	2048 *[225.824]*
134217728	16 *[235.169]*	2048 *[235.271]*	64 *[235.218]*
268435456	256 *[240.484]*	1024 *[240.495]*	2048 *[240.465]*
536870912	256 *[243.278]*	2048 *[243.272]*	2048 *[243.272]*
1073741824	16 *[244.689]*	64 *[244.765]*	256 *[244.756]*
Most frequent parameter	2048	256, 1024	16

Table 37. *The fastest depth values (speeds in square brackets are in MB/s) compared between memory types (32bit mode, write direction, walking_1 pattern type).*

Pattern size [B]	**blockram**	**distributedram**	**shiftregister**
16	256 *[0.054]*	16 *[0.054]*	64 *[0.054]*
32	64 *[0.108]*	256 *[0.108]*	64 *[0.108]*
64	2048 *[0.215]*	16 *[0.215]*	16 *[0.215]*
128	64 *[0.430]*	16 *[0.430]*	16 *[0.430]*
256	64 *[0.860]*	256 *[0.859]*	64 *[0.861]*
512	64 *[1.718]*	16 *[1.717]*	16 *[1.714]*
1024	2048 *[3.437]*	256 *[3.437]*	16 *[3.439]*
2048	2048 *[6.839]*	16 *[6.844]*	1024 *[6.854]*
4096	64 *[13.347]*	16 *[13.351]*	16 *[13.345]*
8192	64 *[25.717]*	256 *[25.706]*	64 *[25.740]*
16384	64 *[0.678]*	16 *[0.710]*	16 *[0.714]*
32768	1024 *[1.340]*	256 *[1.326]*	256 *[1.534]*
65536	16 *[2.589]*	1024 *[2.650]*	256 *[2.651]*
131072	64 *[5.147]*	256 *[5.092]*	16 *[5.092]*
262144	16 *[10.062]*	64 *[10.168]*	256 *[10.168]*
524288	16 *[19.630]*	64 *[19.629]*	1024 *[19.629]*
1048576	2048 *[42.179]*	16 *[49.550]*	1024 *[40.664]*
2097152	16 *[68.187]*	256 *[68.769]*	64 *[68.186]*
4194304	16 *[116.129]*	2048 *[116.151]*	64 *[116.135]*
8388608	1024 *[180.109]*	1024 *[180.022]*	2048 *[179.116]*
16777216	16 *[187.065]*	64 *[182.744]*	2048 *[182.219]*
33554432	1024 *[211.339]*	256 *[209.704]*	64 *[211.025]*
67108864	1024 *[226.019]*	64 *[225.438]*	2048 *[226.376]*
134217728	16 *[235.094]*	2048 *[235.181]*	256 *[235.350]*
268435456	64 *[240.438]*	256 *[240.535]*	2048 *[240.523]*
536870912	2048 *[243.283]*	2048 *[243.277]*	256 *[243.259]*
1073741824	256 *[244.698]*	256 *[244.711]*	1024 *[244.709]*
Most frequent parameter	64	256	64, 16

A.4. Graphical presentation of results from the data pattern type impact investigation subtest in pseudo-duplex mode

Table 38. *The fastest pattern types (values in square brackets are in MB/s) compared between block sizes (duplex mode).*

Pattern size [B]	**16**	**64**	**256**	**1024**
16	walking_1 [*0.035*]	counter_32bit [*0.035*]	counter_8bit [*0.035*]	counter_8bit [*0.035*]
32	counter_8bit [*0.035*]	counter_32bit [*0.070*]	counter_32bit [*0.070*]	counter_8bit [*0.069*]
64	counter_8bit [*0.035*]	counter_8bit [*0.139*]	counter_8bit [*0.140*]	counter_32bit [*0.139*]
128	counter_32bit [*0.035*]	counter_8bit [*0.139*]	counter_32bit [*0.279*]	walking_1 [*0.280*]
256	counter_32bit [*0.035*]	counter_8bit [*0.139*]	counter_32bit [*0.557*]	counter_8bit [*0.555*]
512	counter_32bit [*0.035*]	counter_8bit [*0.139*]	counter_8bit [*0.557*]	walking_1 [*1.111*]
1024	walking_1 [*0.035*]	walking_1 [*0.139*]	walking_1 [*0.558*]	walking_1 [*2.227*]
2048	walking_1 [*0.035*]	counter_32bit [*0.139*]	counter_8bit [*0.556*]	counter_32bit [*2.232*]
4096	counter_8bit [*0.035*]	counter_8bit [*0.139*]	counter_32bit [*0.556*]	walking_1 [*2.230*]
8192	counter_8bit [*0.035*]	counter_8bit [*0.139*]	walking_1 [*0.557*]	counter_32bit [*2.210*]
16384	counter_8bit [*0.035*]	counter_32bit [*0.138*]	counter_32bit [*0.553*]	counter_8bit [*2.209*]
32768	counter_8bit [*0.035*]	walking_1 [*0.138*]	counter_32bit [*0.553*]	counter_8bit [*2.212*]
131072	walking_1 [*0.035*]	counter_8bit [*0.138*]	walking_1 [*0.554*]	counter_32bit [*2.208*]
262144	counter_32bit [*0.035*]	walking_1 [*0.138*]	walking_1 [*0.554*]	walking_1 [*2.213*]
524288	walking_1 [*0.035*]	walking_1 [*0.138*]	walking_1 [*0.553*]	counter_8bit [*2.209*]
1048576	counter_32bit [*0.035*]	counter_8bit [*0.138*]	counter_8bit [*0.551*]	counter_32bit [*2.204*]
2097152	walking_1 [*0.035*]	counter_32bit [*0.138*]	counter_8bit [*0.553*]	walking_1 [*2.208*]
4194304	walking_1 [*0.035*]	counter_32bit [*0.138*]	counter_8bit [*0.553*]	counter_8bit [*2.207*]
8388608	counter_32bit [*0.035*]	counter_8bit [*0.138*]	counter_32bit [*0.552*]	counter_8bit [*2.202*]
16777216	walking_1 [*0.035*]	counter_8bit [*0.138*]	counter_32bit [*0.553*]	walking_1 [*2.211*]
33554432	counter_8bit [*0.035*]	counter_8bit [*0.138*]	counter_8bit [*0.553*]	walking_1 [*2.208*]
67108864	counter_32bit [*0.035*]	walking_1 [*0.138*]	walking_1 [*0.553*]	counter_8bit [*2.209*]
134217728	counter_32bit [*0.035*]	walking_1 [*0.138*]	counter_32bit [*0.553*]	walking_1 [*2.209*]
268435456	counter_8bit [*0.035*]	counter_8bit [*0.138*]	counter_32bit [*0.553*]	counter_32bit [*2.208*]
536870912	walking_1 [*0.035*]	counter_8bit [*0.138*]	walking_1 [*0.553*]	walking_1 [*2.208*]
1073741824	walking_1 [*0.035*]	walking_1 [*0.138*]	walking_1 [*0.553*]	counter_8bit [*2.208*]
Most frequent parameter	walking_1	counter_8bit	counter_32bit	counter_8bit

A.5. Graphical presentation of results from the best block size value research in pseudo-duplex mode

Table 39. *Speed transfer (in MB/s) depending on the block size for counter_8bit pattern type.*

Pattern size [B]	**16**	**64**	**256**	**1024**
16	0.0346	0.0348	0.0348	0.0348
32	0.0349	0.0696	0.0694	0.0693
64	0.0349	0.1394	0.1398	0.1379
128	0.0346	0.1393	0.2783	0.2788
256	0.0346	0.1392	0.5376	0.5547
512	0.0345	0.1388	0.5569	1.1092
1024	0.0347	0.1383	0.5558	2.2161
2048	0.0346	0.1382	0.5555	2.2142
4096	0.0347	0.1389	0.5537	2.2080
8192	0.0346	0.1387	0.5542	2.2057
16384	0.0346	0.1379	0.5525	2.2088
32768	0.0346	0.1381	0.5526	2.2125
65536	0.0345	0.1381	0.5505	2.2114
131072	0.0345	0.1382	0.5525	2.2063
262144	0.0346	0.1383	0.5526	2.2107
524288	0.0346	0.1382	0.5521	2.2094
1048576	0.0346	0.1382	0.5512	2.2012
2097152	0.0345	0.1383	0.5526	2.2048
4194304	0.0346	0.1382	0.5531	2.2073
8388608	0.0345	0.1382	0.5519	2.2022
16777216	0.0345	0.1382	0.5523	2.2078
33554432	0.0346	0.1382	0.5527	2.2071
67108864	0.0346	0.1382	0.5525	2.2093
134217728	0.0346	0.1382	0.5525	2.2083
268435456	0.0346	0.1382	0.5527	2.2062
536870912	0.0346	0.1382	0.5526	2.2073
1073741824	0.0346	0.1382	0.5526	2.2077

Table 40. *Speed transfer (in MB/s) depending on the block size for counter_32bit pattern type.*

Pattern size [B]	**16**	**64**	**256**	**1024**
16	0.0347	0.0348	0.0347	0.0347
32	0.0346	0.0696	0.0698	0.0670
64	0.0348	0.1388	0.1384	0.1391
128	0.0347	0.1373	0.2789	0.2713
256	0.0347	0.1382	0.5567	0.5407
512	0.0347	0.1385	0.5458	1.0910
1024	0.0346	0.1388	0.5579	2.2079
2048	0.0346	0.1385	0.5540	2.2317
4096	0.0346	0.1383	0.5559	2.2203
8192	0.0346	0.1386	0.5529	2.2097
16384	0.0346	0.1381	0.5532	2.2019
32768	0.0346	0.1381	0.5530	2.2084
65536	0.0345	0.1381	0.5530	2.2078
262144	0.0346	0.1383	0.5539	2.2115
524288	0.0345	0.1383	0.5523	2.2077
1048576	0.0346	0.1381	0.5510	2.2041
2097152	0.0345	0.1383	0.5525	2.2041
4194304	0.0345	0.1383	0.5526	2.2042
8388608	0.0346	0.1382	0.5523	2.2012
16777216	0.0345	0.1381	0.5525	2.2094
33554432	0.0346	0.1382	0.5526	2.2055
67108864	0.0346	0.1382	0.5519	2.2063
134217728	0.0346	0.1383	0.5529	2.2077
268435456	0.0346	0.1382	0.5527	2.2084
536870912	0.0346	0.1382	0.5526	2.2075
1073741824	0.0346	0.1382	0.5526	2.2075

Table 41. *Speed transfer (in MB/s) depending on the block size for walking_1 pattern type.*

Pattern size [B]	**16**	**64**	**256**	**1024**
16	0.0348	0.0340	0.0348	0.0348
32	0.0341	0.0695	0.0693	0.0691
64	0.0344	0.1358	0.1391	0.1357
128	0.0346	0.1393	0.2783	0.2796
256	0.0346	0.1391	0.5561	0.5547
512	0.0346	0.1381	0.5569	1.1110
1024	0.0347	0.1391	0.5580	2.2273
2048	0.0346	0.1385	0.5515	2.2230
4096	0.0345	0.1385	0.5543	2.2296
8192	0.0346	0.1384	0.5570	2.2065
16384	0.0346	0.1380	0.5532	2.2065
32768	0.0345	0.1382	0.5529	2.2096
65536	0.0345	0.1382	0.5533	2.2112
131072	0.0345	0.1380	0.5536	2.2045
262144	0.0346	0.1383	0.5541	2.2126
524288	0.0346	0.1384	0.5529	2.2070
1048576	0.0345	0.1380	0.5507	2.1999
2097152	0.0346	0.1383	0.5522	2.2084
4194304	0.0346	0.1382	0.5528	2.2068
8388608	0.0345	0.1382	0.5516	2.2020
16777216	0.0346	0.1381	0.5524	2.2109
33554432	0.0346	0.1382	0.5526	2.2080
67108864	0.0346	0.1382	0.5527	2.2083
134217728	0.0346	0.1383	0.5526	2.2086
268435456	0.0346	0.1382	0.5525	2.2083
536870912	0.0346	0.1382	0.5526	2.2076
1073741824	0.0346	0.1382	0.5527	2.2073