POLITECNICO DI TORINO

Master degree course in Electronic Engineering

Master Degree Thesis
in
Integrated System Architecture

Computational Storage Drive based on OpenSSD

Supervisor

Candidate
Giuseppe Dongiovanni
Mancino

Anno Accademico 2019-2020

Contents

List of Tables

List of Figures

Chapter 1

Computational Storage Project

In this chapter all the steps necessary to build an FPGA-based NVMe Computational Storage are going to be analyzed. However, due its complexity, it is necessary to first obtain the NVMe communication interface: an NVMe Controller. Therefore, the open source project Cosmos+ OpenSSD has been chosen as the basis for our goal due to matching the requirements.

1.1 Cosmos+ OpenSSD Project

The first step consists of analysis and adaptation of the OpenSSD project from the original custom board to the Xilinx Zynq-7000 SoC ZC706.

1.1.1 Overview

Cosmos OpenSSD is an open source and FPGA-based SSD controller project that has been developed since 2014 by the HYU ENC Lab of the Hanyang University in South Korea, with research and education purposes[1].

A first version of the project was based on Indilinx Barefoot, a SoC over SATA2.

The project version that will be analyzed is the Cosmos+ OpenSSD: developed in 2016, this version of the SSD controller supports the NVMe protocol. The project has been developed using Xilinx Developer Tools, Vivado Design Suite and SDK.

The custom Cosmos+ FPGA board has the following main features:

- FPGA Xilinx Zynq-7000 with a Dual ARM Cortex-A9 1GHz Core;
  - 1GB of DDR3;
  - AXI4-lite bus width of 32 bits;
  - AXI4 bus width of 64 bits;
- dual PCIe Gen2 x8 End-Points (Cabled PCIe Interface);

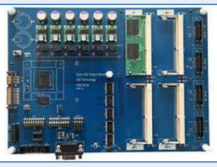
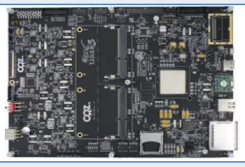
	Jasmine OpenSSD	Cosmos OpenSSD	Cosmos+ OpenSSD
Released in	2011	2014	2016
Main Board			
SSD Controller	Indilinx Barefoot (SoC)	HYU Tiger3 (FPGA)	HYU Tiger4 (FPGA)
Host Interface	SATA2	PCIe Gen2 4-lane (AHCI)	PCIe Gen2 8-lane (NVMe)
Maximum Capacity	128 GB (32 GB/module)	256 GB (128 GB/module)	2 TB (1 TB/module)
NAND Data Interface	SDR (Asynchronous)	NVDDR (Synchronous)	NVDDR2 (Toggle)
ECC Type and Strength	BCH, 16 bits/512 B	BCH, 32 bits/2 KB	BCH, 26 bits/512 B

Figure 1.1: OpenSSD project History - Cosmos+ OpenSSD 2017 Tutorial

- additional interfaces (JTAG,USB,Ethernet);
- up to 2 NAND Flash Modules, with 8 flash packages slot each.

In Figure 1.2 it is illustrated the internal system overview of the project.

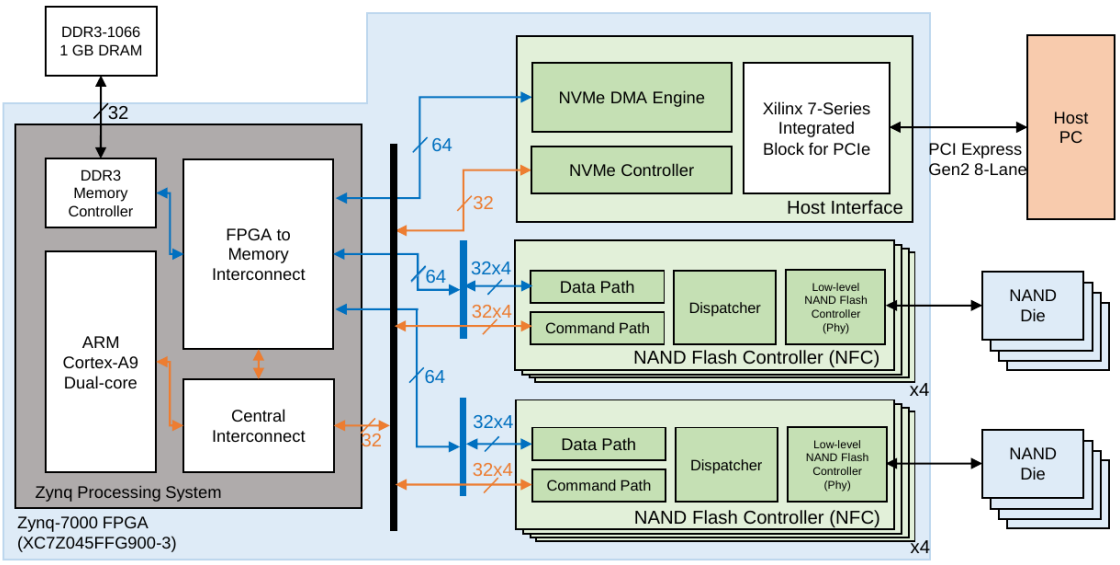


Figure 1.2: Cosmos+ OpenSSD project System Overview - Cosmos+ OpenSSD 2017 Tutorial

The Zynq processor is connected to the host through the Host Interface, called NVMe Host Controller, which is responsible of:

- handling of the data from the host to the buffer with a DMA engine;

- automated completion of the NVMe IO Command, without involving the Flash Transition Layer (FTL), that will be described later.

The NAND Flash Controller is the interface between the NAND Flash and the processor. It consists,as shown in the system block desing in Figure 1.3, of three different hardware IP blocks:

- Tiger4 NSC;
- Tiger4 Shared KES;
- V2NFC.

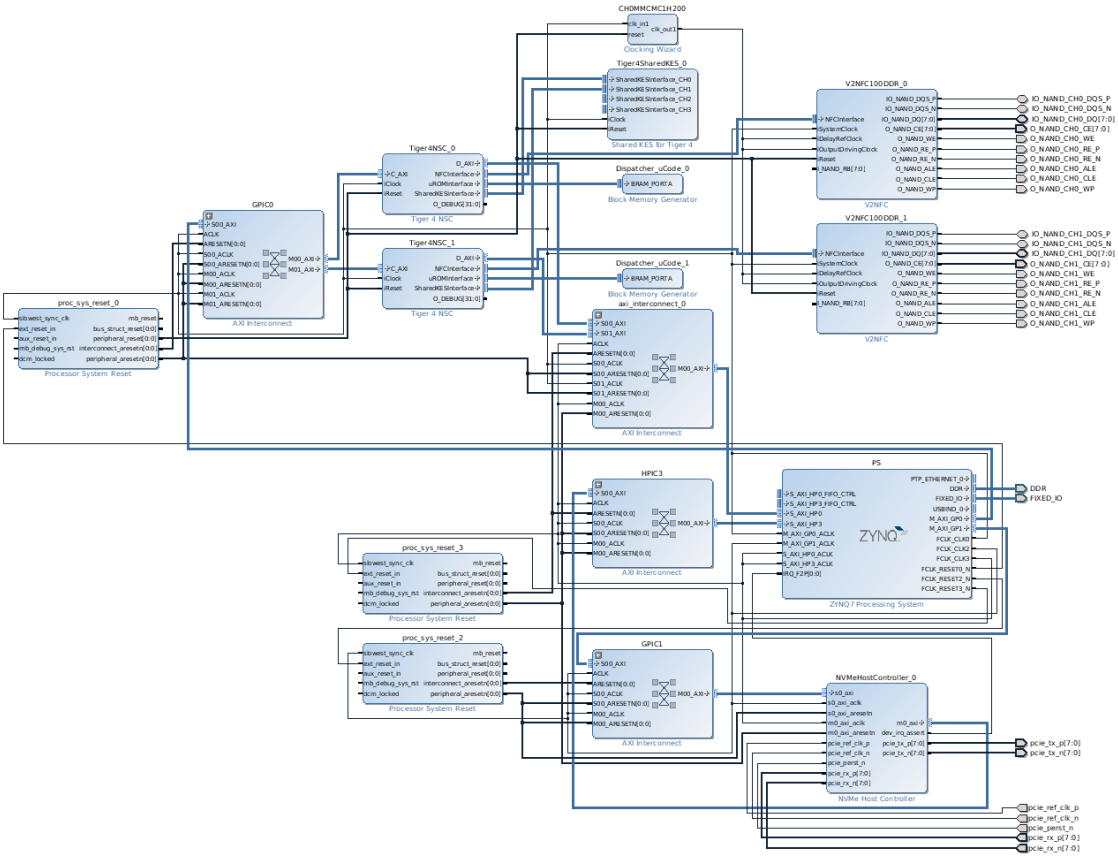


Figure 1.3: Cosmos+ OpenSSD project System Design

The IP Tiger4 NSC is the responsible of the handling of command and data from the processing system: the commands, consisting of information such as source and destination of the operation, are written by the firmware driver in the Tiger4 NSC registers and then elaborated.

The data, instead, undergo different manipulations: in particular they pass through a module responsible of the Error Detection and Correction, the Tiger4 Shared KES.

Finally, the data are handled by the V2NFC block, that physically performs the low-level I/O operations in the NAND Flash Modules.

The operations have to be scheduled in order to be performed on the right SSD package and die. As shown in Figure 1.4, each NAND module has up to 4 available channels, one every two packages, and each channel has 8 maximum ways, corresponding to the maximum number of connected dies. The number of channels is equal to the number of NAND Flash Controllers.

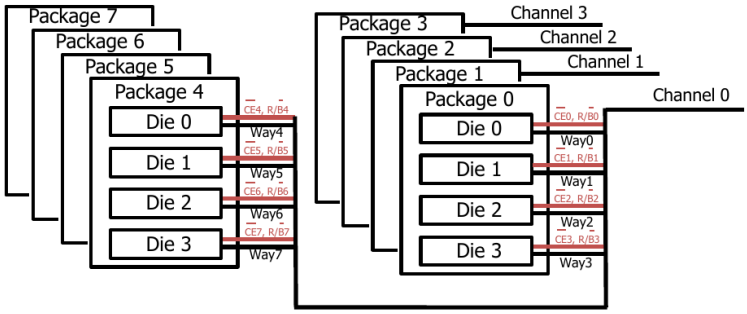


Figure 1.4: Nand Modules Organization - Cosmos+ OpenSSD 2017 Tutorial

In the first version of the project, Cosmos, the way scheduling was managed by the NFC block, while the channel one by the firmware Flash Transition Layer (FTL): with the Cosmos+ one, both channel and way scheduling is managed by the FTL, providing more flexibility.

The other main features of the FTL are:

- Least Recently Used (LRU) data buffer management;
- priority command scheduling, as shown in Figure 1.5, with the aim of enhancing the multi-channel and way parallelism;
- on demand garbage collection, triggered only when there is no more free user block in each die;

The garbage collection is needed to recover free blocks for write requests: a victim block with invalid data is selected, then the valid data are copied in a free block while the victim one is erased.

However, while supporting the garbage collection, the firmware does not support the wear leveling of the flash memory.

A schematic description of the firmware execution is shown in Figure 1.6.

In order to be executed, a received IO command is first transformed in Slice Requests, which number depends on the number of logic blocks requested. Then

Command	Priority
LLSCommand_RxDMA	0
LLSCommand_TxDMA	0
V2FCommand_StatusCheck	1
V2FCommand_ReadPageTrigger	2
V2FCommand_BlockErase	3
V2FCommand_ProgramPage	4
V2FCommand_ReadPageTransfer	5

Figure 1.5: Command Priority - Cosmos+ OpenSSD 2017 Tutorial

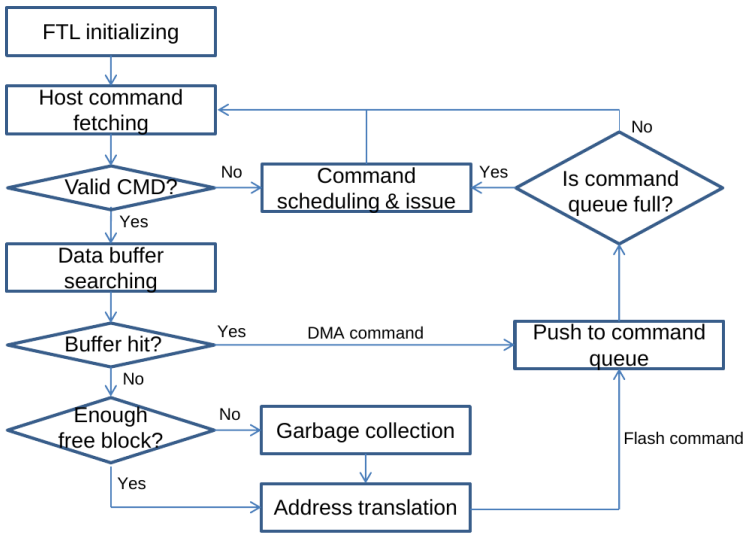


Figure 1.6: Firmware Overall Sequence - Cosmos+ OpenSSD 2017 Tutorial

each Slice Request undergoes a second transformation in DMA and, if there is no buffer hit for read operations, NAND Requests. These requests are organized in different queues: one for the free requests, three for the requests that are going to be executed, one for each aforementioned type of requests, and two for the blocked requests, either for buffer or row address dependencies. Then the requests are, if not blocked, finally scheduled and executed.

1.1.2 Project Adaptation

The available platform is the Xilinx ZC706: in comparison with the original custom board, it has:

- same FPGA Zynq-7000 SoC;

- 4-line Gen2 PCIe Connector, instead of the 8-lane of the custom board;
- no NAND module.

The first step to adapt the project is to modify the hardware[1]: there is no NAND module, therefore the entire NAND Flash Controller hardware is not necessary.

At the same time, the NVMe Host Controller has to be changed from the 8-lane PCIe to the 4-lane one: this can be done by modifying the configuration of the Xilinx PCIe Core IP. The result is shown in Figure 1.7. Likewise, the constraint files have to be modified or removed, in order to match the pinout of the Xilinx ZC706[2].

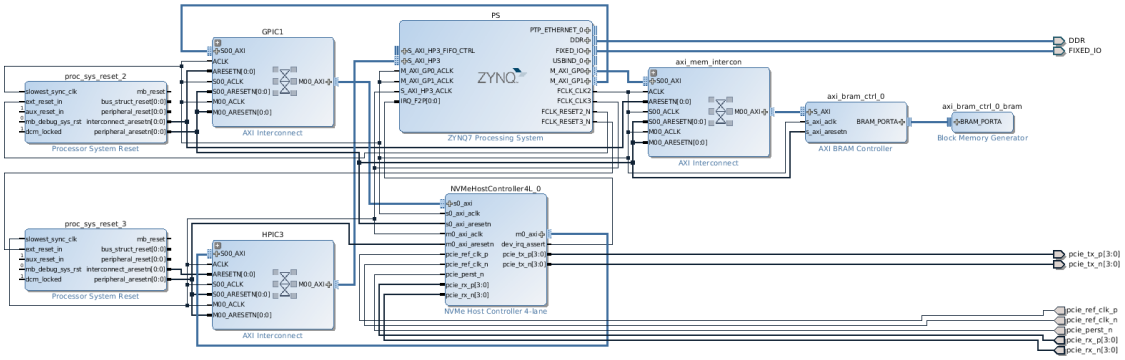


Figure 1.7: First Adaptation of the Cosmos+ OpenSSD project System Design

A BRAM Controller has been added to provide a destination address for the firmware channel, substituting the Tiger4NSC one: however, it has no active role in the operations.

After exporting the new hardware file (.hdf), it is necessary to create a new project with the new platform specifications.

Some modifications have also been made to the firmware:

- allocation of the memory arrays <MemSpace> in the DDR: the storage capacity is of 64 MB, due to the DDR already being used by the firmware FTL;
- different organization of the memory management unit (MMU) table and the memory mapping, given the presence of the memory array;
- variation to the memory dimensions, number of channel and way;
- bypass of the status check and ECC functions;

[1]This modified project and all the following ones have been uploaded on the GitHub repository <https://github.com/giuseppedongiovanni/nvme_comp_storage>

- replacement of the NAND operations, represented by the write operation of the commands in the Tiger4 registers, with the MemCpy function.

The memory array <MemSpace> is the replacement for the SSD: however, the available storage is much smaller than what the firmware expects. Varying the FTL configuration parameters, that should be left untouched, is necessary to avoid that important memory location useful for the firmware execution are corrupted.

1.1.3 Functionality Tests

The first test that has been carried out is the functionality one, consisting of a routine of power-up, partitioning and reset from both developer and host side, as shown respectively in Figures 1.8, 1.9 and 1.10. Then the data correctness, that corresponds to writing a file and reading it back, is verified in Figure 1.11 by the matching MD5 values (practically the digital fingerprint of a file).

Finally, it is possible to send some NVMe Admin Commands to verify the presence of the namespace and, as well, obtain information about the device. As it can be seen in Figure 1.13, the namespace id is set to 14740, equal to the storage capacity in terms of NVMe blocks (4096 bytes): this namespace, defined by the project creators, has been attached by the controller to the NSID 1, being the device seen as <nvme0n1>.

1.1.4 Performance Tests

Different performance tests are carried out: <dd> function and the software Iometer are used to evaluate IOPS and bandwidth; to measure the latency instead, timers are used both in the firmware and in a c-file on the host side.

The Figures 1.14 to 1.16 refer to the performed tests for the read operation.

The results obtained are unexpected: performing operations from a DDR to an host device through a PCIe bus should imply very high performance, in the order of GB/s for the bandwidth. Instead, the obtained one is around 100 MB/s, with only 7000 IOPS.

More information can be obtained from the analysis of the latency test results, shown in Figure 1.17.

The pie chart is divided in two main parts: the green one is related to the time spent in the device firware, while the orange one includes all the other contributions, in particular the host, the PCIe bus and the NVMeHostController of the device.

The firmware execution takes up the 73% of the total latency: in particular, the <MemCpy> function employs more than half of this time period to be performed, slowing the entire execution. The obtained speed is of about 150 MB/s, while a single-port DDR3, with a 32-bit bus-width at 533 MHz, has a maximum theoretical bandwidth of 4 GB/s.

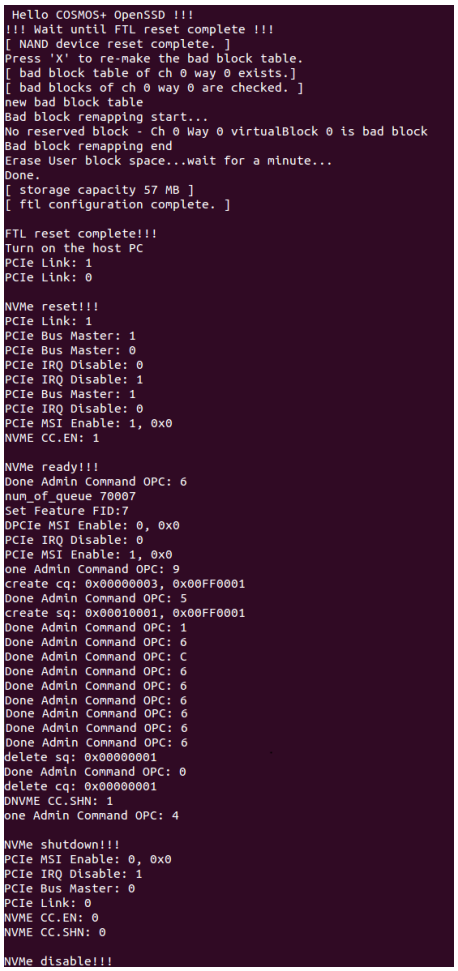


Figure 1.8: Functionality test - Power-up, Partitioning and Reset - Development PC

The cause of this problem has to be found in the project itself: both the processing system and the NVMeHostController IP are running with a heavy memory access loading through the same port of the DDR, that for this reason has become the bottleneck of the project.

On the other hand, no write test are available: if the device is stressed with long or continuous write operations, the DMA of the NVMe Host Controller freezes, resulting in a Timeout Abort Error on the host.

In Figure 1.18 different variables were printed in the terminal to backtrack the cause of the error: in particular, head.autoDmaRx is the hardware counter of the completed DMA request, while tail.autoDmaRx is the software counter of the submitted DMA request: when the two counters coincide, the DMA operation is completed. It is possible to see that the DMA is stuck at head.autoDmaRx = 0xAA, although other 2 requests are present in the queue, being tail.autoDmaRx = 0xAC.

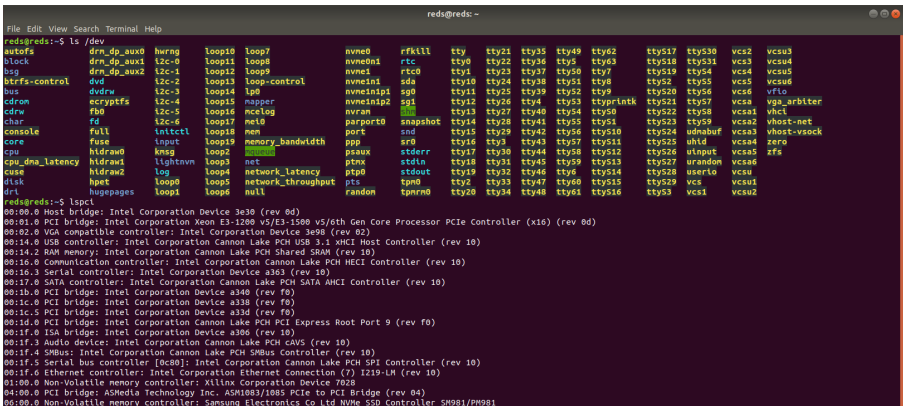


Figure 1.9: Functionality test - Power-up - Host PC

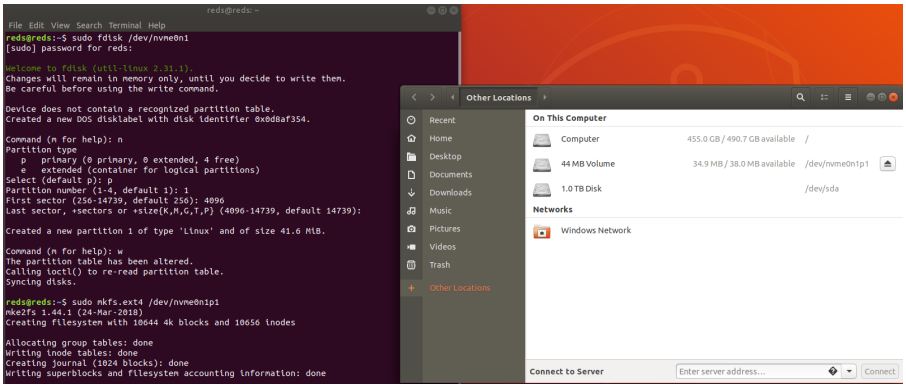


Figure 1.10: Functionality test - Formatting operation - Host PC

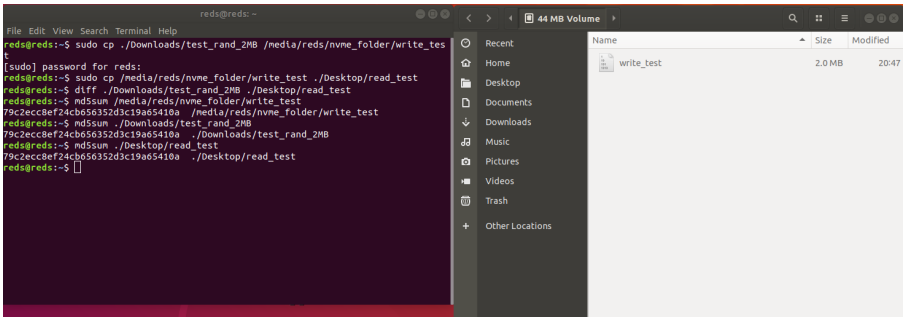


Figure 1.11: Functionality test - Data Correctness - Host PC

As it can be seen in Figures 1.19 and 1.20, the counter X of the submitted request is incremented up to 0x30, however the last increment of head.autoDmaRx is due to the count 0x2e.

Given multiple factors, between which the difficulty to easily reproduce and backtrack the error and the altered timing due to the modifications, a solution to

```
reds@reds:~$ sudo nvme id-ctrl -H /dev/nvme0
NVME Identify Controller:
vid     : 0x1edc
ssvid   : 0x1edc
sn      : SSDD515T
mn      : Cosmos+ OpenSSD
fr      : TYPE0005
rab     : 0
ieee    : 5cd2e4
cmic    : 0
  [2:2] : 0     PCI
  [1:1] : 0     Single Controller
  [0:0] : 0     Single Port

mdts    : 8
cntlid  : 9
ver     : 0
rtd3r   : 0
rtd3e   : 0
oaes    : 0
  [8:8] : 0     Namespace Attribute Changed Event Not Supported

ctratt  : 0
  [0:0] : 0     128-bit Host Identifier Not Supported

oacs    : 0
  [8:8] : 0     Doorbell Buffer Config Not Supported
  [7:7] : 0     Virtualization Management Not Supported
  [6:6] : 0     NVMe-MI Send and Receive Not Supported
  [5:5] : 0     Directives Not Supported
  [4:4] : 0     Device Self-test Not Supported
  [3:3] : 0     NS Management and Attachment Not Supported
  [2:2] : 0     FW Commit and Download Not Supported
  [1:1] : 0     Format NVM Not Supported
  [0:0] : 0     Security Send and Receive Not Supported

acl     : 3
aerl    : 3
frmw    : 0x3
  [4:4] : 0     Firmware Activate Without Reset Not Supported
  [3:1] : 0x1   Number of Firmware Slots
  [0:0] : 0x1   Firmware Slot 1 Read-Only

lpa     : 0
  [2:2] : 0     Extended data for Get Log Page Not Supported
  [1:1] : 0     Command Effects Log Page Not Supported
  [0:0] : 0     SMART/Health Log Page per NS Not Supported

elpe    : 8
npss    : 0
avscc   : 0
  [0:0] : 0     Admin Vendor Specific Commands uses Vendor Specific Format
```

Figure 1.12: Functionality test - NVMe Identify Command - Host PC

```
reds@reds:~$ sudo nvme get-ns-id /dev/nvme0n1
nvme0n1: namespace-id:14740
reds@reds:~$ sudo nvme id-ns /dev/nvme0n1
NVME Identify Namespace 14740:
nsze    : 0x3994
ncap    : 0x3994
nuse    : 0x3994
nsfeat  : 0
nlbaf   : 0
flbas   : 0
mc      : 0
dpc     : 0
dps     : 0
nmic    : 0
rescap  : 0
fpi     : 0
nawun   : 0
nawupf  : 0
nacwu   : 0
nabsn   : 0
nabo    : 0
nabspf  : 0
noiob   : 0
nvmcap  : 0
nguid   : 00000000000000000000000000000000
eui64   : 0000000000000000
lbaf  0 : ms:0   lbads:12 rp:0x2 (in use)
```

Figure 1.13: Functionality test - NVMe Namespace List - Host PC

```
                                                        reds@reds: ~
File  Edit  View  Search  Terminal  Help
reds@reds:~$ sudo dd if=./Downloads/test_rand_8MB of=/dev/nvme0n1 bs=4096
[sudo] password for reds:
2000+0 records in
2000+0 records out
8192000 bytes (8.2 MB, 7.8 MiB) copied, 0.0530143 s, 155 MB/s
reds@reds:~$ sudo dd if=/dev/nvme0n1 of=./Desktop/read_test bs=4096 count=1500
1500+0 records in
1500+0 records out
6144000 bytes (6.1 MB, 5.9 MiB) copied, 0.0644825 s, 95.3 MB/s
```

Figure 1.14: Performance test - dd function, write and read - Base Version

the problem was not found.

1.1.5 Optimized Firmware Version

At first, the cause for the poor performance has been attributed to the scheduling of the NAND requests, having taken for granted that the <MemCpy> function was extremely fast. Therefore an optimized version of the firmware was developed in which the converted slice requests are directly executed, not creating any NAND request.

Even if working correctly, performance were just slightly better than the ones

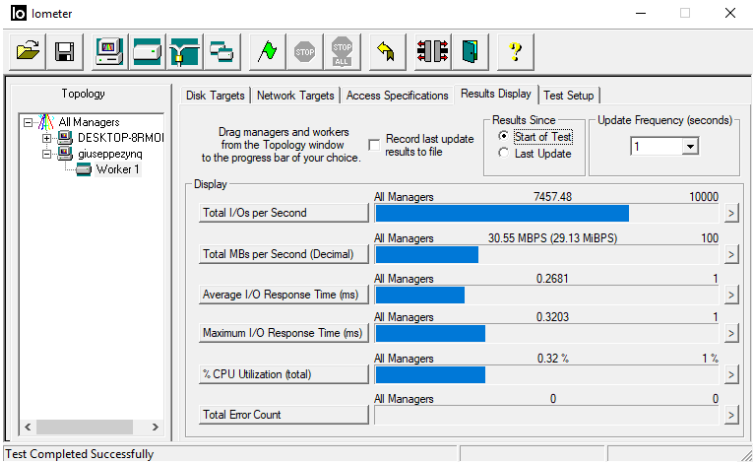


Figure 1.15: Performance test - Iometer Sequential Read transfer size = 4k, block size = 4k - Base Version

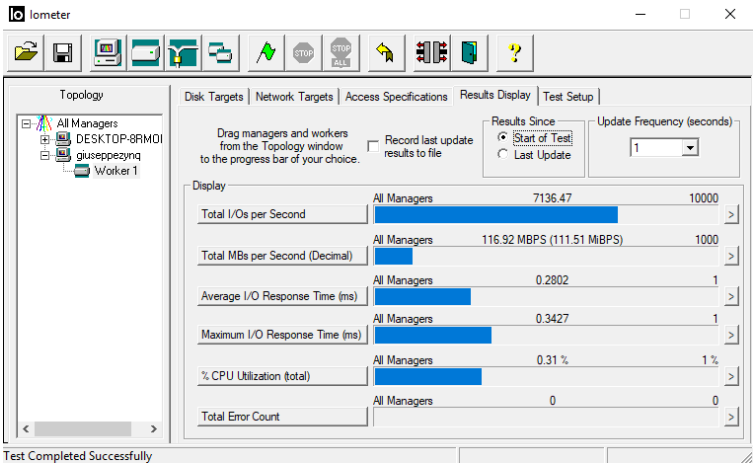


Figure 1.16: Performance test - Iometer Sequential Read transfer size = 16k, block size = 4k - Base Version

of the original firmware, as shown in Figures 1.21,1.22 and 1.23: this helped to discover the real problem, but has lead to no significant improvements. For this reason and to modify the original code the least possible to preserve stability, this version was discarded and no further optimization was carried out.

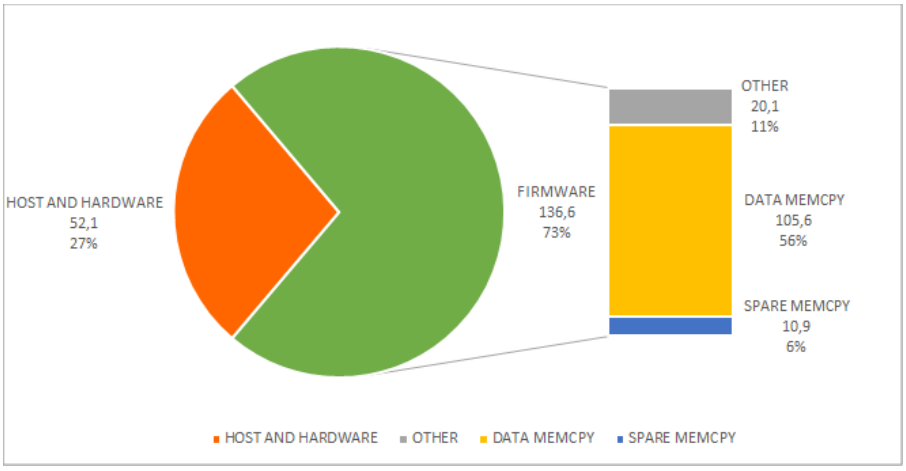


Figure 1.17: Performance test - Read Latency - Base Version

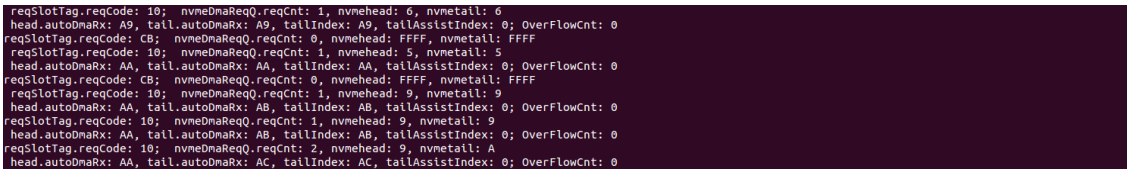


Figure 1.18: Terminal Error - head.autoDmaRx stuck

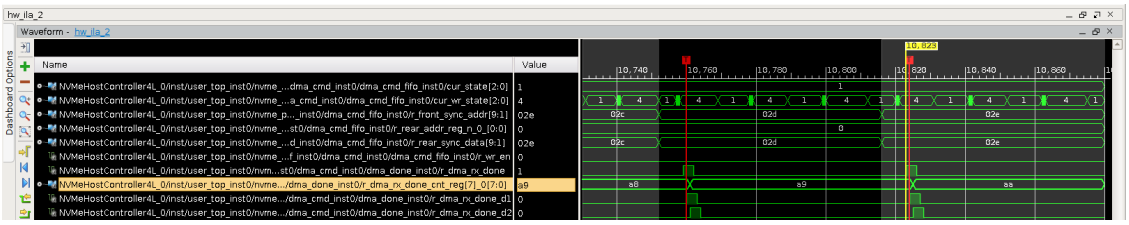


Figure 1.19: Waveform Error - head.autoDmaRx stuck

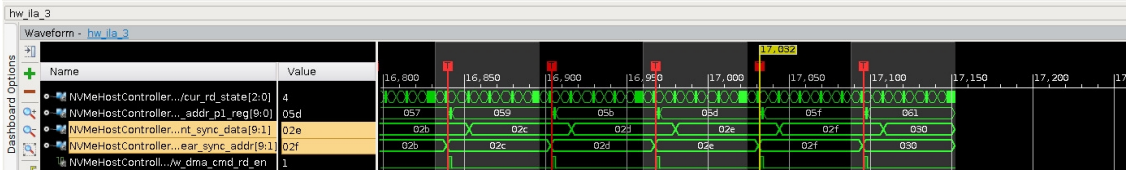


Figure 1.20: Waveform Error - Submitted DMA Request

Figure 1.21: Performance test - dd function, write and read - Optimized Version

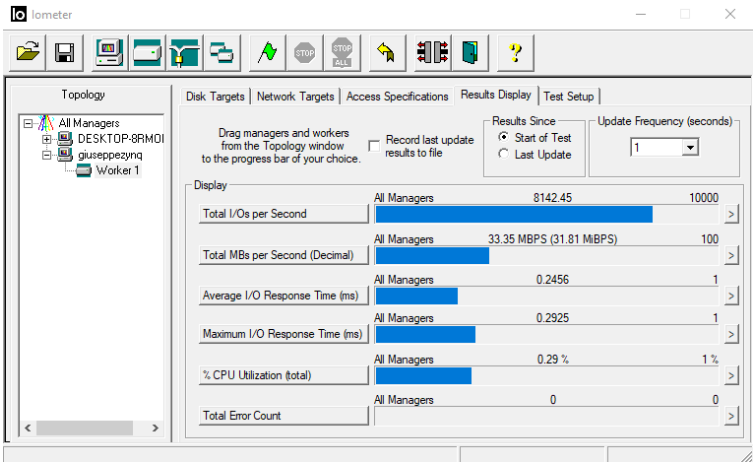


Figure 1.22: Performance test - Iometer Sequential Read transfer size = 4k, block size = 4k - Optimized Version

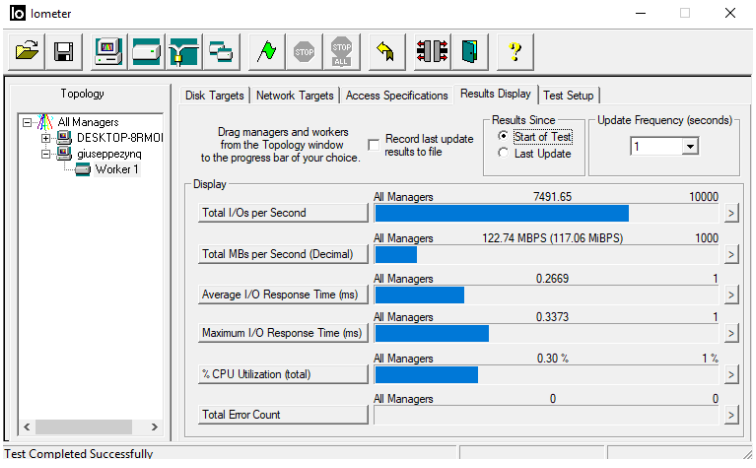


Figure 1.23: Performance test - Iometer Sequential Read transfer size = 16k, block size = 4k - Optimized Version

1.2 Computational Storage Development

The objective is to develop an hardware accelerator consisting of a wrapper that can host different types of acceleration core. However, before starting with the development of the hardware accelerator, it is necessary to modify the project with the addition of the AXI DMA IP, in order to match the general architecture of a FPGA-based Computational Storage Drive 1.24.

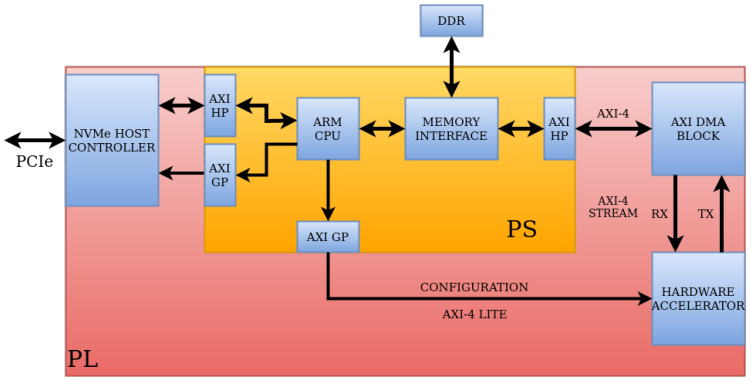


Figure 1.24: General Architecture of a FPGA-based Computational Storage

1.2.1 32-bit AXI DMA

At first, the AXI DMA has been used as a replacement for the MemCpy function, being simply closed on a loop-back. The only firmware modifications concern the configuration of the DMA and, as mentioned above, the substitution of the MemCpy with a DMA transfer.

Some preliminary performance tests are carried out to compare this version to the one with <MemCpy> function.

In Figure 1.26 the <dd> test has been performed: however, due to the small amount of data transferred, the results are not accurate. Instead, as it can be seen in Figures 1.27 to 1.29, despite an increment in the latency, there is a slight increase in both IOPS and bandwidth due to the introduction of the DMA, that reduces the workload of the processing system and favors the increase of throughput.

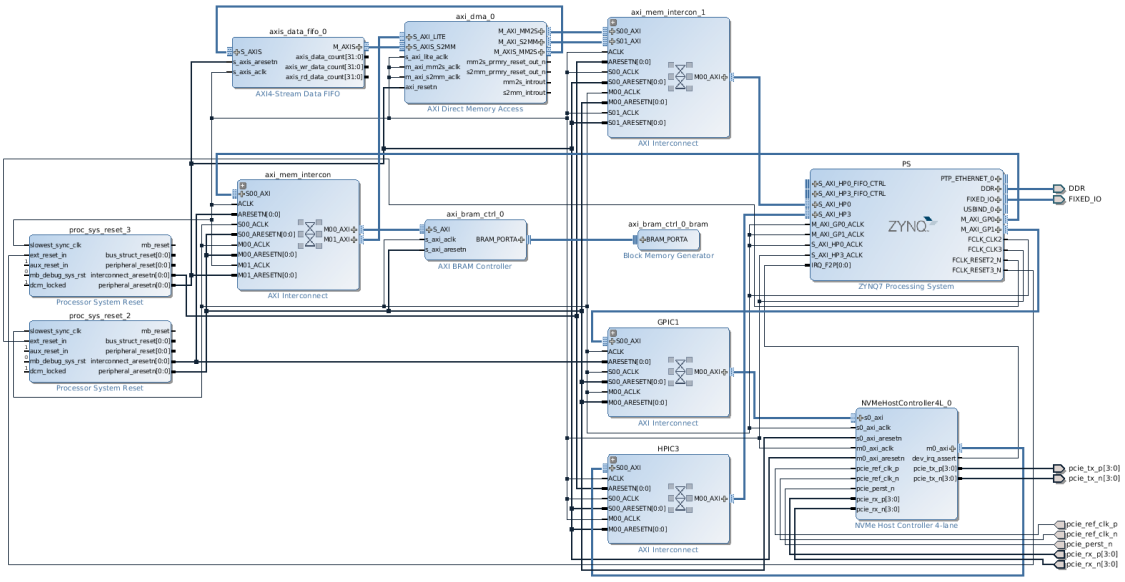


Figure 1.25: 32bit AXI DMA Adaptation of the Cosmos+ OpenSSD project System Design

Figure 1.26: Performance test - dd function, write and read - 32bit AXI DMA Version

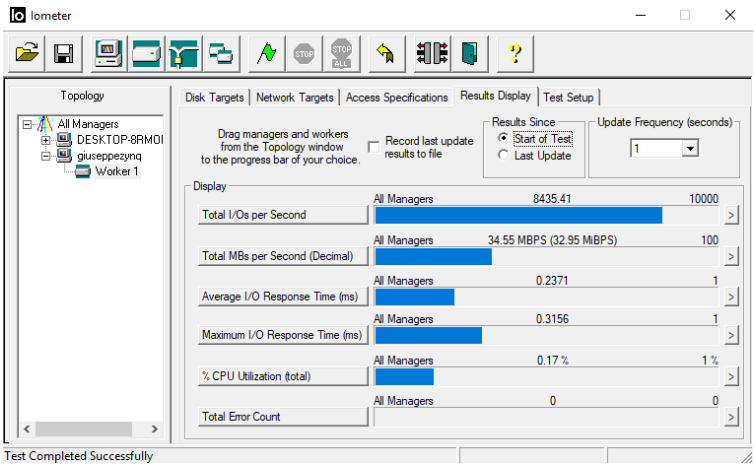


Figure 1.27: Performance test - Iometer Sequential Read transfer size = 4k, block size = 4k - 32bit AXI DMA Adaptation

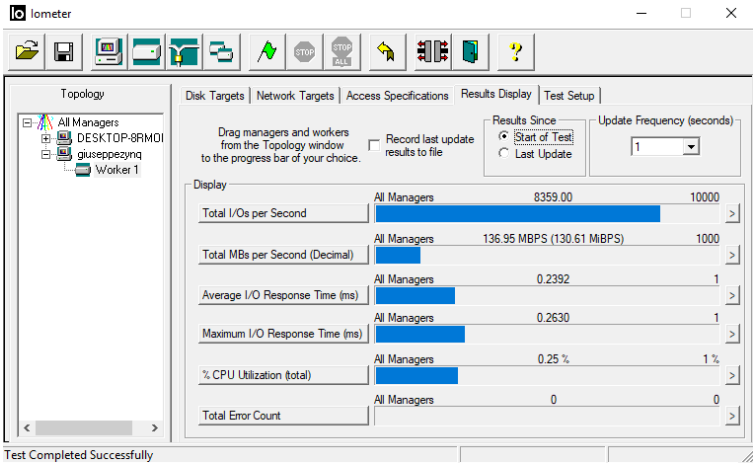


Figure 1.28: Performance test - Iometer Sequential Read transfer size = 16k, block size = 4k - 32bit AXI DMA Adaptation

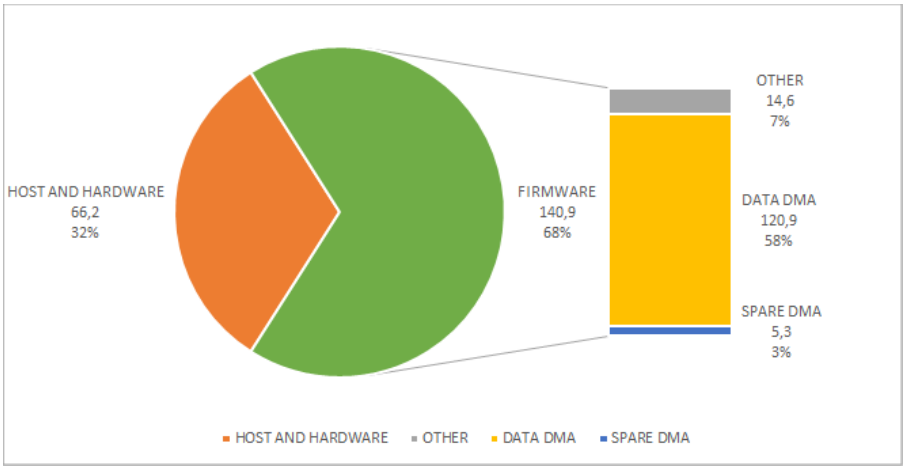


Figure 1.29: Performance test - Read Latency - 32bit AXI DMA Adaptation

1.2.2 First Prototype

The first prototype of hardware accelerator is a simple block which adds the value of a parameter to the data that need to be processed. Its primary goal was to provide a better understanding of the AXI-4 lite and stream protocols and the NVMe Admin Command.

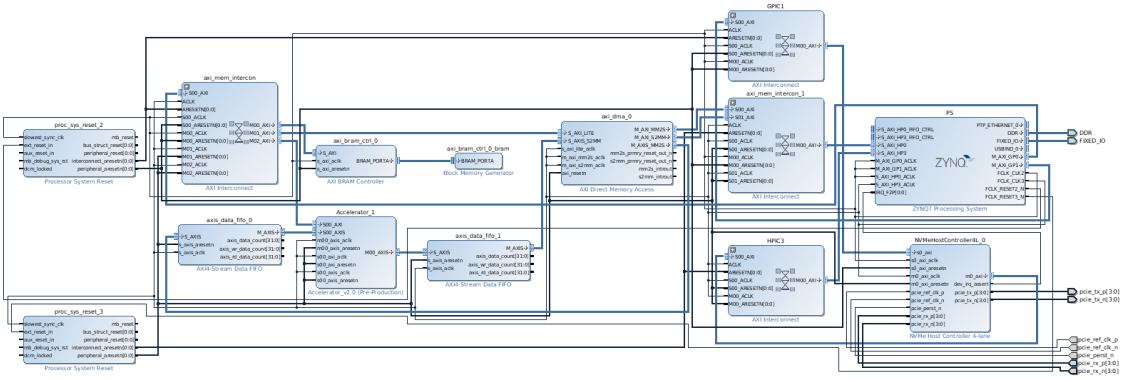


Figure 1.30: System Design of the first version of the Computational Storage based on the Cosmos+ OpenSSD project

As shown in Figure 1.30, the accelerator is enclosed by two Data FIFO, in order to decouple the DMA transfer from the computation.

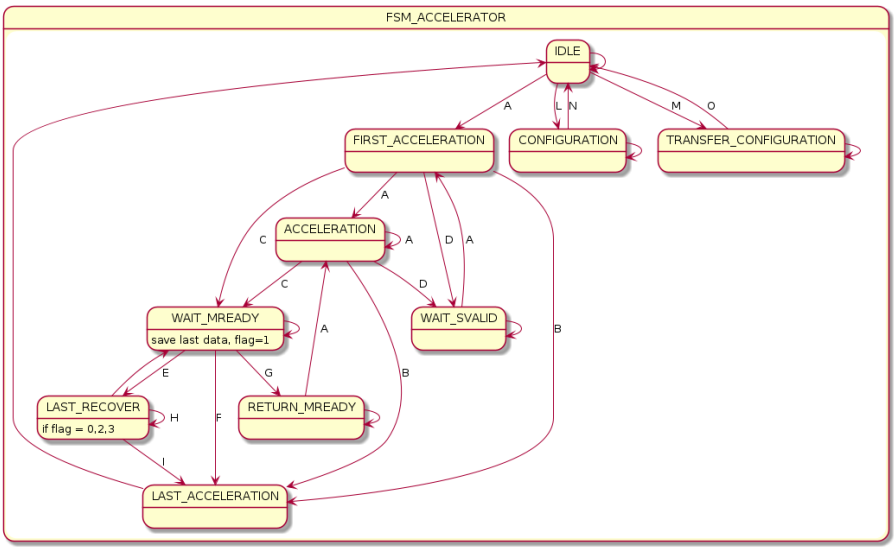
The designed Finite State Machine (FSM) is shown in Figure 1.31.

The Configuration an Transfer_Configuration state handle respectively the modification and the reading of the configuration and parameter registers through the Set Parameters NVMe Admin Commands. The feature identifiers of the commands, a field that indicates for what feature the attribute are being specified for, are chosen between the group of the vendor specific ones[3], being the registers custom.

On the acceleration branch, the different states are used to take care of all the combination of the AXI-4 stream protocol signals and to recover any lost input data in the occurrence of particular conditions (e.g. output not ready on the last element).

The Accelerator has two configurations: the first is a simple pass-through, necessary for the power-on procedure, while the second is the acceleration configuration that, as previously said, is a simple addition of a parameter to the data. It is possible to change the values of the configuration and parameter registers through the NVMe Set Parameter Command, as shown in Figure 1.32.

The test in Figures 1.33 to 1.36 shows a degradation in performance for this solution due to the insertion of FIFOs and Accelerator block.



Letter	Signals
A	m_axis_tready && s_axis_tvalid
B	m_axis_tready && !s_axis_tvalid && flag_tlast
C	!m_axis_tready
D	!s_axis_tvalid && !flag_tlast
E	m_axis_tready && ((s_axis_tlast && s_axis_tvalid)
F	m_axis_tready && flag_tlast
G	m_axis_tready
H	m_axis_tready && flag
I	m_axis_tready && !flag
L	s_axi_awvalid && s_axi_wvalid
M	s_axi_arvalid
N	s_axi_bready
O	s_axi_rready

Figure 1.31: FSM of the first version of the Hardware Accelerator

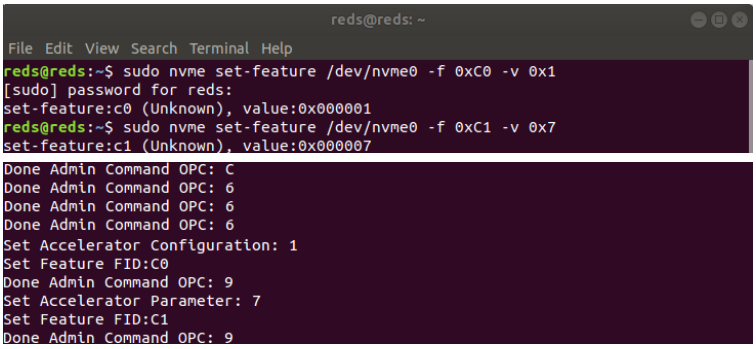


Figure 1.32: NVMe Set Parameter Admin Command - Addition of 0x7 to a 32-bit sequence - Host and Developer PCs

1.2.3 128-bit AXI DMA

As it will be treated in the next section, the chosen acceleration core works on 128bit data packet: in order not to add complexity to the accelerator and achieve higher performance, it is necessary to change the AXI DMA bus width from 32 to 128 bits.

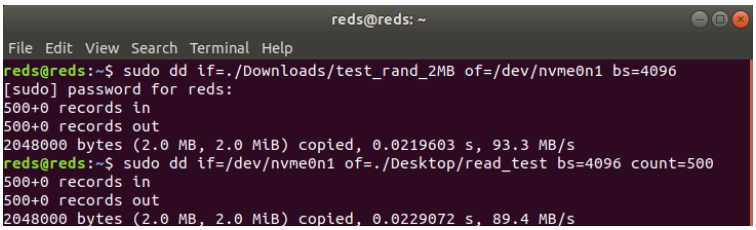


Figure 1.33: Performance test - dd function, write and read - Computational storage first prototype

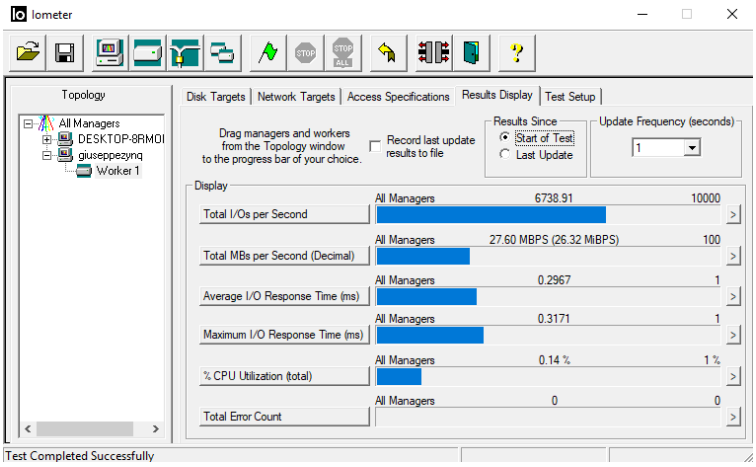


Figure 1.34: Performance test - Iometer Sequential Read transfer size = 4k, block size = 4k - Computational storage first prototype

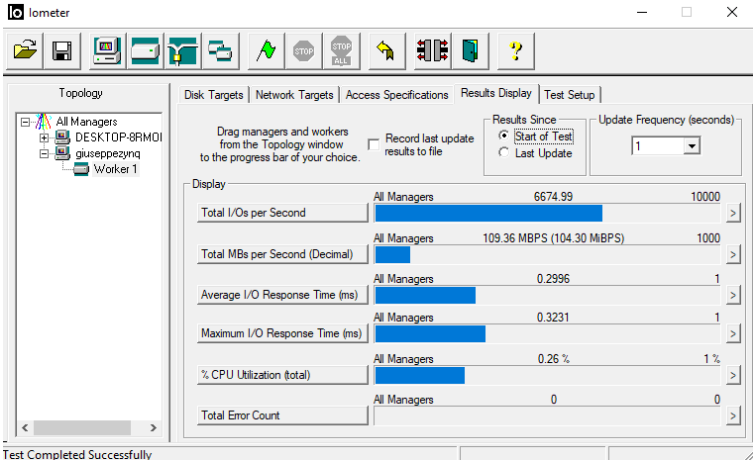


Figure 1.35: Performance test - Iometer Sequential Read transfer size = 16k, block size = 4k - Computational storage first prototype

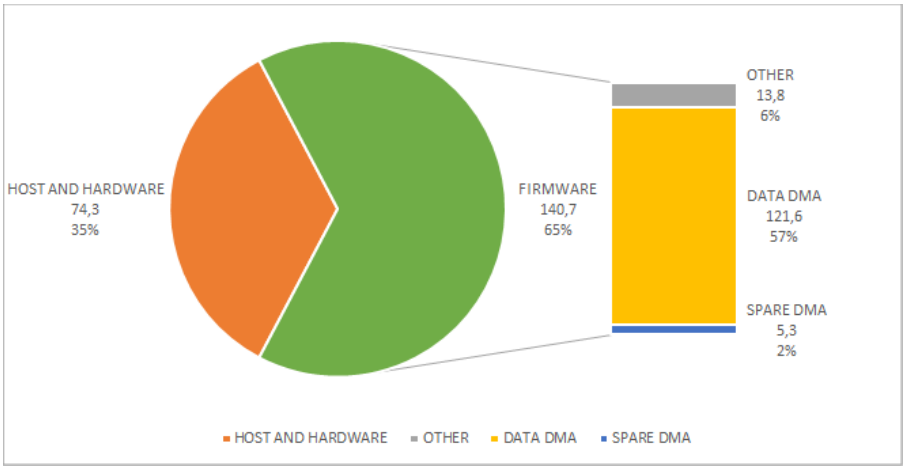


Figure 1.36: Performance test - Read Latency - Computational storage first prototype

Only two firmware modifications has to be carried out in order to perform the 128-bit DMA transfer:

- allocation of a new memory array for the spare data, called <SpaceArray>, to avoid conflicts for consecutive transfers;
- use of an attribute to specify the minimum alignment for the memory arrays, that has to be set to 16 bytes.

As for the previous versions, some preliminary tests are carried out to evaluate the performance of the new configuration.

Figure 1.37: Performance test - dd function, write and read - 128bit AXI DMA Version

The achieved performance, in terms of bandwidth, IOPS and latency, for the 128-bit AXI DMA version are better than those of the 32-bit one thanks to the higher speed of the data transfer from and to the PS. However, as seen in section 1.1.1, the PS can provide only a 64-bit AXI interface, limiting the possible performance increase.

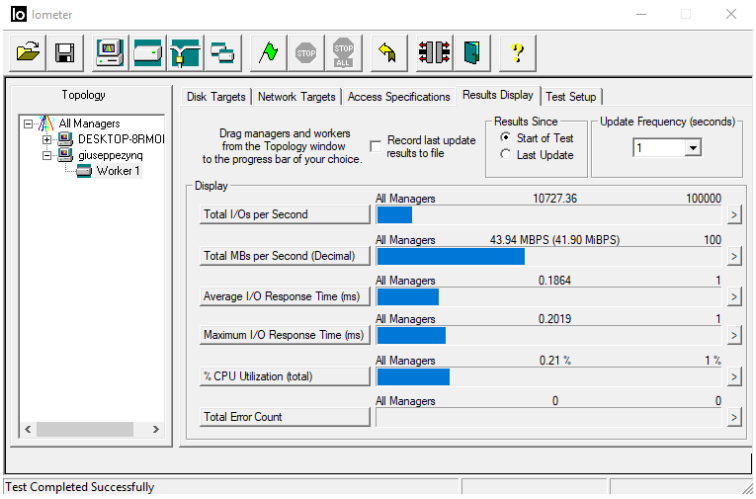


Figure 1.38: Performance test - Iometer Sequential Read transfer size = 4k, block size = 4k - 128bit AXI DMA Adaptation

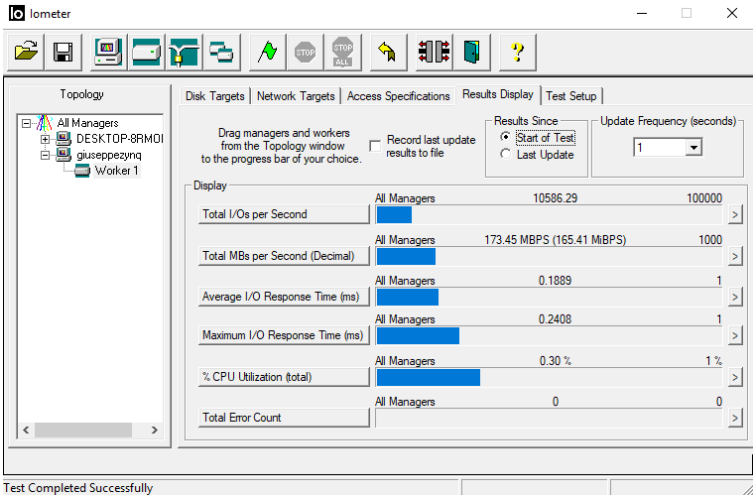


Figure 1.39: Performance test - Iometer Sequential Read transfer size = 16k, block size = 4k - 128bit AXI DMA Adaptation

1.2.4 Second Prototype

The first prototype is not very suitable as general wrapper because the AXI-4 stream protocol makes the management of cores with different timing difficult. Therefore, the previous external FIFOs are then included in the hardware of this second prototype, in order to separate the input and the output AXI-4 stream signals, as shown in Figure 1.41.

In this second prototype, there are two FSMs, as shown in Figure 1.42: the modification and the reading of the registers through NVMe Admin Commands is

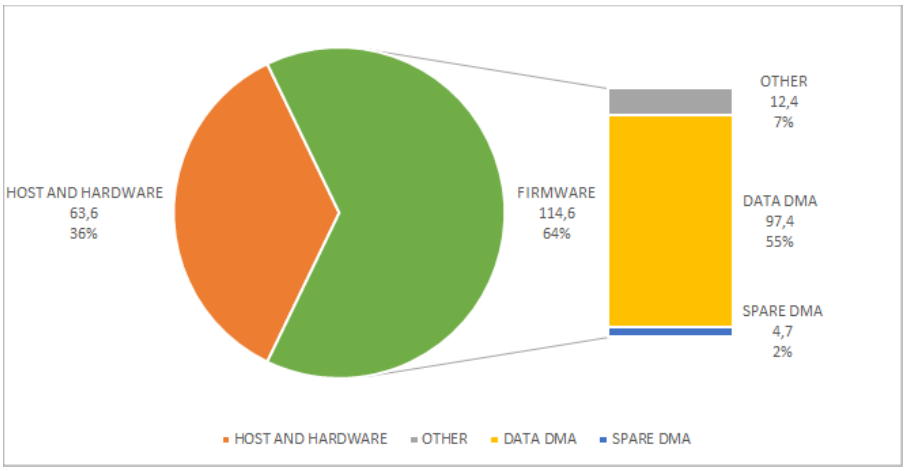


Figure 1.40: Performance test - Read Latency - 128bit AXI DMA Adaptation

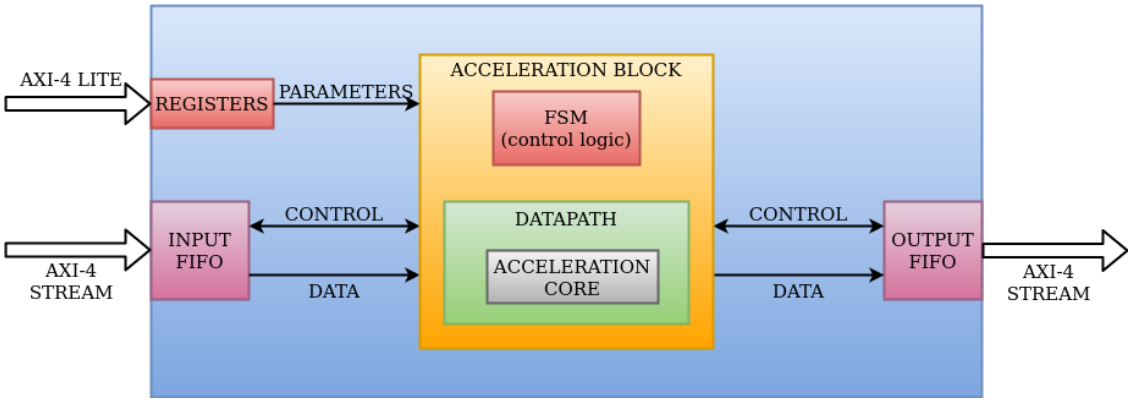


Figure 1.41: General diagram of the Hardware Accelerator

now divided from the acceleration branch.

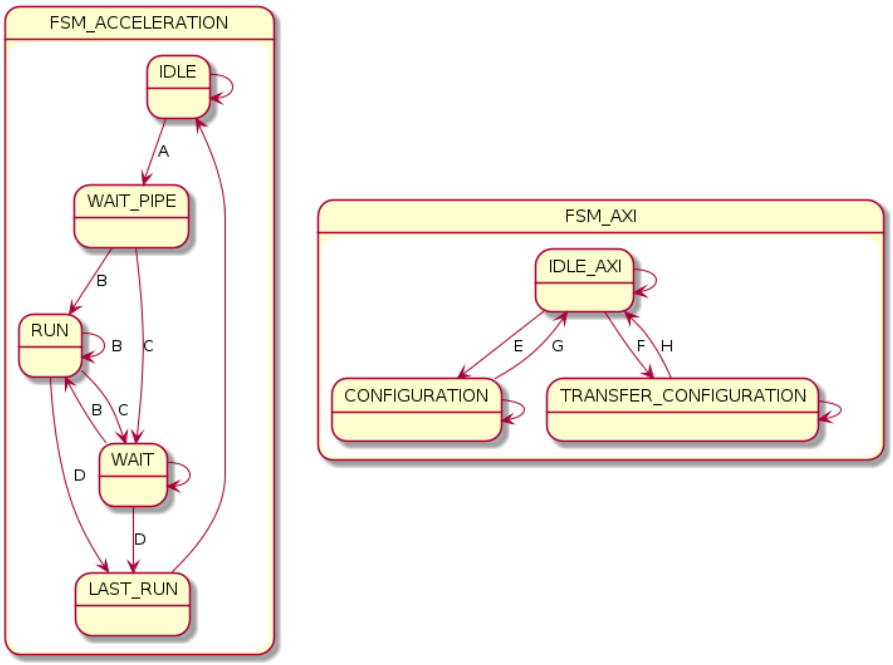
The acceleration branch is simpler than in the first prototipe, due to the acceleration process being indipendent from the AXI-4 stream protocol. The WAIT_PIPE state is used to wait for the core to be ready, in the case its output is not immediately available.

AES Core and CTR Configuration

In order to verify the correct functioning of the block, it was necessary to choose the acceleration function and, consequently, the core to be inserted.

The sought acceleration function has to be:

- widespread and standard;
- format-preserving: for construction limits (DMA transfer) the output data



Letter	Signals
A	!empty & !full
B	valid & !empty & !full
C	valid & !empty & !full
D	valid & !empty & !full & last
E	s_axi_awvalid && s_axi_wvalid
F	s_axi_arvalid
G	s_axi_bready
H	s_axi_rready

Figure 1.42: FSMs of the second version of the Hardware Accelerator

format has to be the same of the input one;

- used with a data stream: to achieve high performance.

Therefore, the chosen core is an encryption one based on the Advanced Encryption Standard (AES)[4], called "tiny_aes". The AES encryption algorithm, that is a specific implementation of the Rijndael one, has been adopted and standardized by the National Institute of Standards and Technology (NIST) and the US FIPS PUB in 2001 and is accepted all over the world. The AES is a symmetric encryption algorithm, using a single key for both encryption and decryption. It is a format-preserving encryption algorithm that works on a single fixed-length data block of 128-bit. On the other hand, the key size can be 128,192 or 256 bit.

The encryption process consists of different rounds of transformations, which number varies depending on the length of the key.

Five modes of operation were defined[5]:

- Electronic Code Book (ECB) mode: the simplest mode and, for this reason,

generally not recommended. All the input data blocks, called plain text, are encrypted with same key, allowing parallel encryption but resulting in low level of security;

- Cipher Block Chaining (CBC) mode: the input block is xored with an initialization vector (IV) and then encrypted. The resulting cipher text is used as IV for the next block;
- Cipher FeedBack (CFB) mode: the key and the IV are the input of the encryption block, which result is xored with the plain text. As in the CBC, the resulting cipher text is used as IV for the next block;
- Output FeedBack (OFB) mode: as in the CFB, the key and the IV are the input of the encryption block, which result is xored with the plain text to give the cipher text. However the IV for the next block is given by the result of the encryption block;
- Counter (CTR) mode: in this mode the IV is a counter. As in the CFB, the key and the IV are the input of the encryption block, which result is xored with the plain text to give the cipher text. However the IV for the next block is given by the incremented counter.

Apart from the ECB, each mode has advantages and disadvantages: the choice on which to use depends on the application. Given the requirements for the accelerator, the best choice is the CTR mode: in fact in this mode the encryption and decryption can be prepared in advance and the parallel computing is supported, providing in this way the possibility to achieve high speed while ensuring security. Then the hardware accelerator is modified according to the scheme shown in Figure 1.43, in which each block is equivalent of a pipeline stage.

It is fundamental that a data block uses the same value of the counter in both encryption and decryption. Otherwise, the operation is not completed correctly.

As described in the project specification[4], there are three cores available, one for each key size: due to the limit in FPGA resources, only the AES-128 and AES-256 are included in the hardware accelerator.

Functionality Tests

To verify the correct behavior of the developed accelerator, the NIST provides test vectors[5] for both encryption and decryption operations.

By construction of the cores, input and output of the core have to be reversed in order to be processed. This occurs for the opposite orientation of the software and hardware vectors.

Accounting NIST test vectors to be in a reversed state, only the output of the core is inverted: the key has to be inserted as it is. As well, to verify that results match, both plain-text and cipher-text have to be reversed.

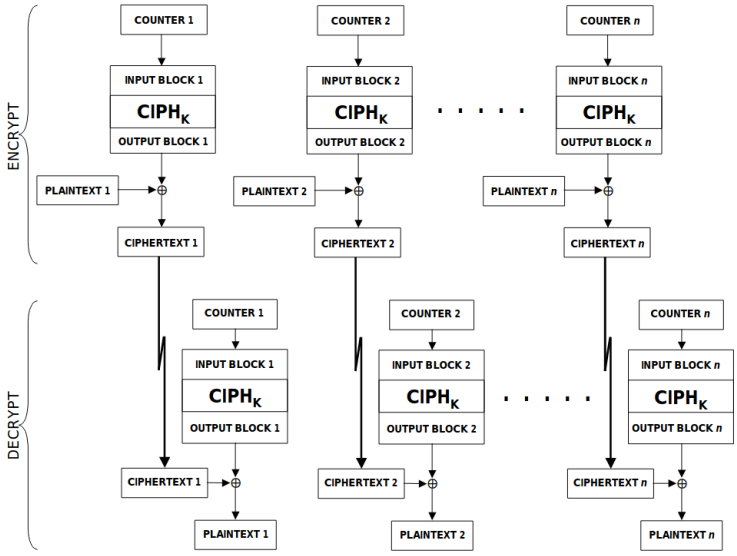


Figure 1.43: CTR Mode - NIST, Recommendation for Block Cipher Modes of Operation: Methods and Techniquesm

The test are performed in three different ways:

- Module verilog test-bench;
- Module execution;
- General execution.

As shown in Figures the first vectors of the test-bench waveform are, respectively, plain-text blocks, initial counter and key from NIST test vectors.

The plain-text is reversed in the <data_in_pipe> vector to ensure the correct behavior: the match of results can be verified by the vector <reverse_data>, inversion of the accelerator output.

For the module execution test, the comparison between the obtained cipher-text and NIST test vectors is directly executed by the software. In this type of test the configuration still takes place with direct writing of configuration and parameter registers.

Figures 1.48 and 1.51 shows the messages printed by the software, as a consequence of the correct completion of the operations, for both configurations.

In Figures 1.49, 1.50, 1.52 and 1.53 instead, the complete execution waveforms for the parameters configuration and the encryption phase only are shown.

As mentioned previously, it is possible to notice the difference between the time passed in the state WAIT_PIPE (code 2) in the two encryption waveforms: the process takes more rounds of transformations to complete the operations in the case of the AES-256 core rather than the AES-128 one.

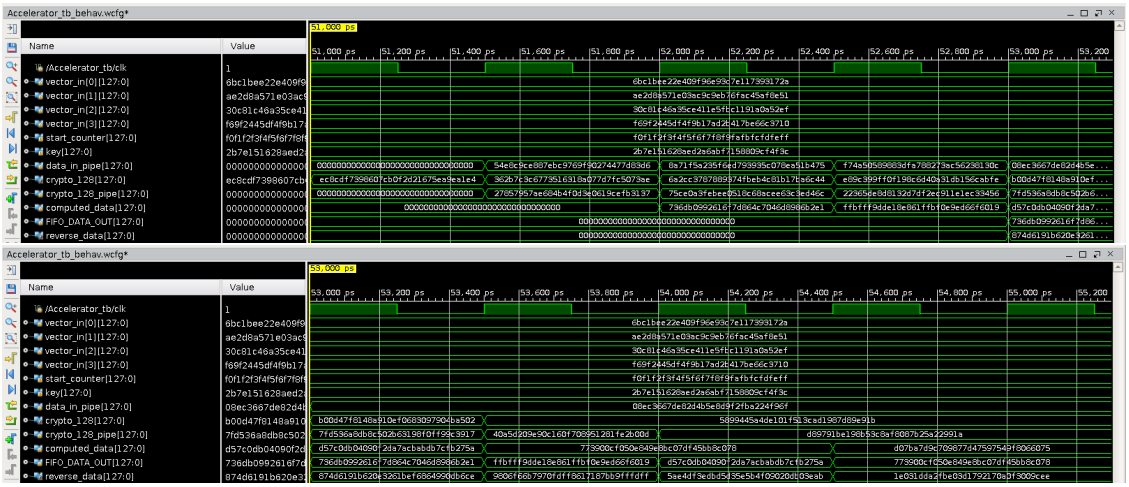


Figure 1.44: Verilog Test-bench - AES-128 Encryption

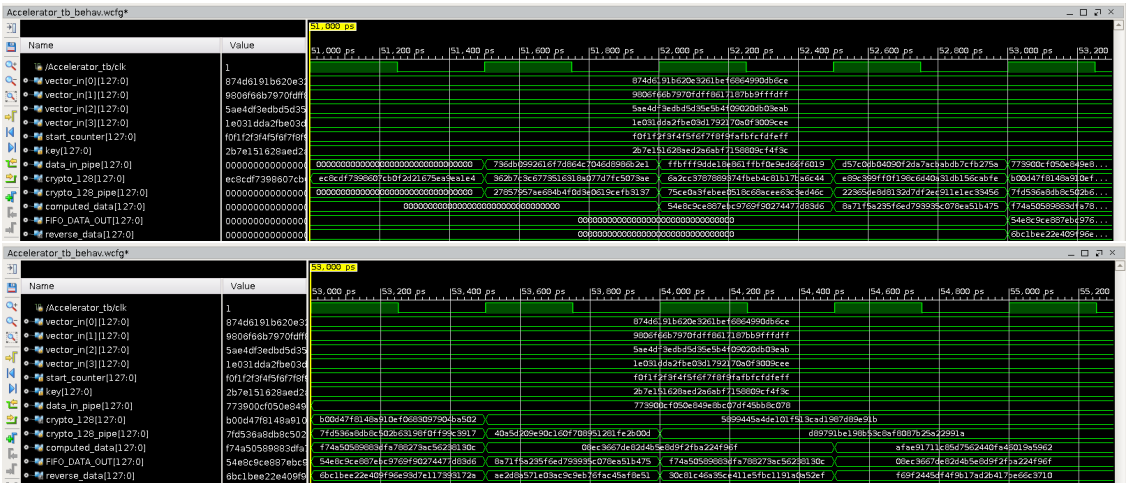


Figure 1.45: Verilog Test-bench - AES-128 Decryption

Before performing the general test, is necessary to define in the firmware the Feature Identifiers of the Set Feature command. Their assignation to the configuration and parameters registers is shown in Table 1.1.

Finally, it is possible to perform the general test. Figures 1.54 and 1.55 show, from both host and developer PC sides, the configuration phase and write operation to send the test vectors: <test_128.bin> is a binary file in which the NIST vectors have been reversed. The printed output, which are the hexadecimal values of the byte saved in the MemSpace array, are equivalent to test vectors reversed.

The write operations are always referred to a page (16kB): therefore, as it can be seen in Figure 1.31, the amount of computed data is much larger than the 64 bytes of binary file.

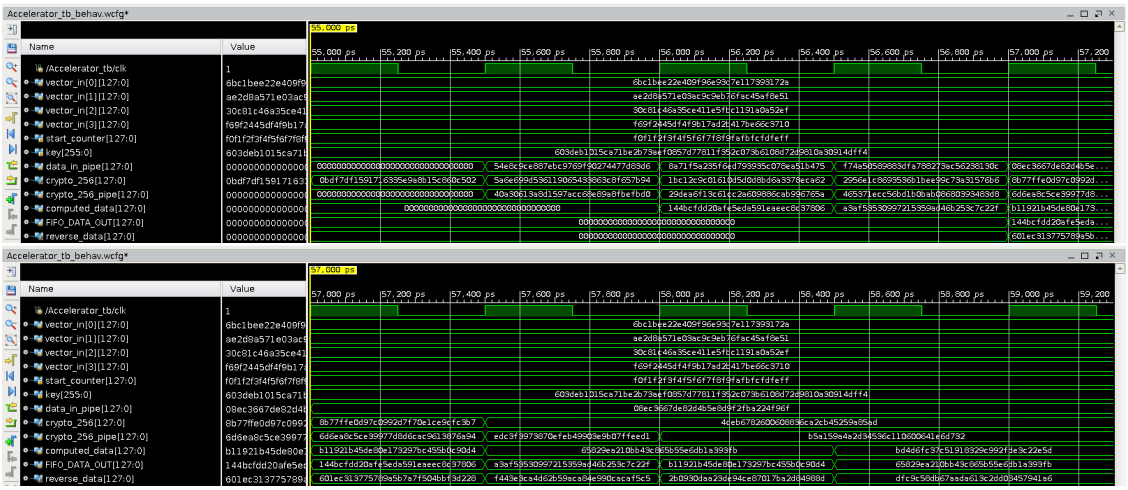


Figure 1.46: Verilog Test-bench - AES-256 Encryption

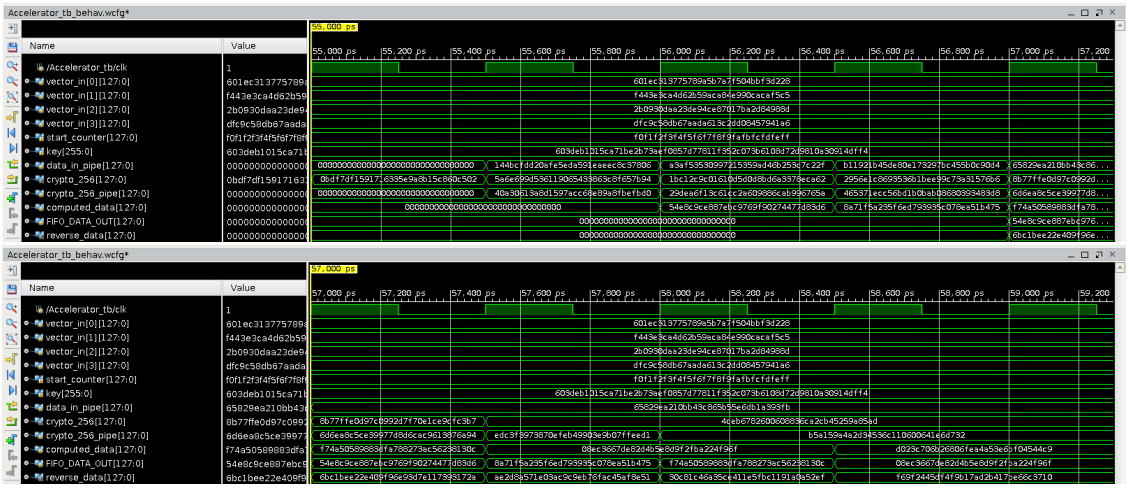


Figure 1.47: Verilog Test-bench - AES-256 Decryption

```
Terminal ready
◆[!] MMU has been enabled.

 Hello COSMOS+ OpenSSD !!!
Configuration - Press 'X' to start
Configuration: 02; Key 128bit = 2B7E1516-28AED2A6-ABF71588-9CF4F3C; IV: F0F1F2F3-F4F5F6F7-F8F9FAFB-FCFDFEFF
Encryption-Decryption - Press 'X' to start
transfer Dev-DMA successful
transfer DMA-Dev successful
Encryption completed successfully
transfer Dev-DMA successful
transfer DMA-Dev successful
Decryption completed successfully
done
```

Figure 1.48: Module execution - AES-128 Terminal

Same operations are performed for the AES-256 core in Figures 1.57 to 1.59.

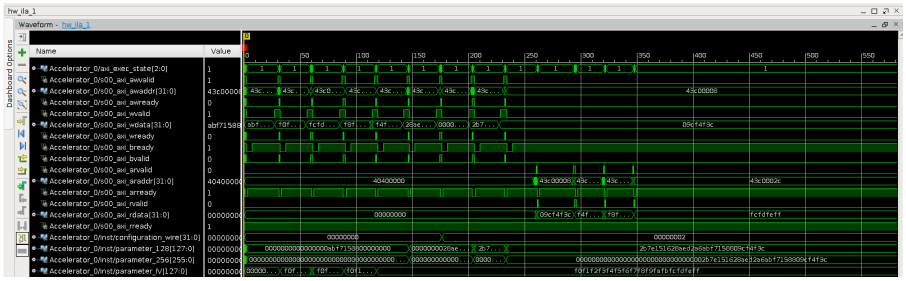


Figure 1.49: Module execution - AES-128 Configuration

Figure 1.50: Module execution - AES-128 Encryption

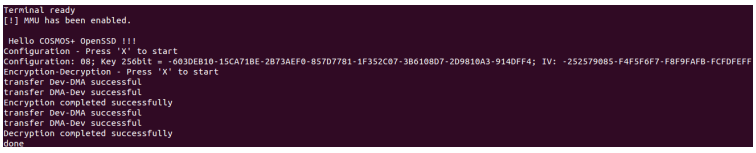


Figure 1.51: Module execution - AES-256 Terminal

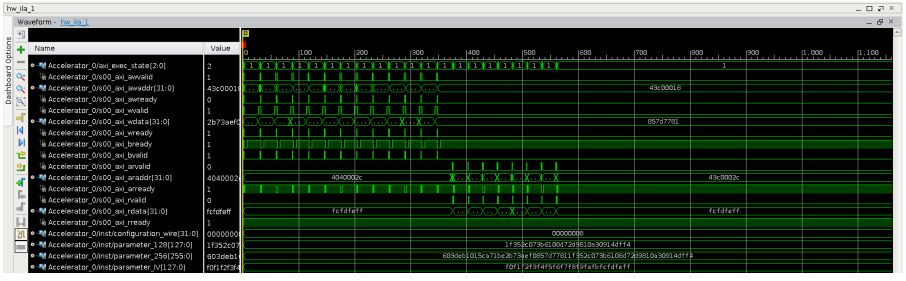


Figure 1.52: Module execution - AES-256 Configuration

Figure 1.53: Module execution - AES256 Encryption

Feature Identifier	Register Name	Notes
0xC0	Configuration Register	0 = Pass through; 1 = Adder; 2 = AES-128; 8 = AES-256
0xC1	Parameter 1 Register	LSW of the key vector
0xC2	Parameter 2 Register	
0xC3	Parameter 3 Register	
0xC4	Parameter 4 Register	MSW of the 128-bit key vector
0xC5	Parameter 5 Register	
0xC6	Parameter 6 Register	
0xC7	Parameter 7 Register	
0xC8	Parameter 8 Register	MSW of the 256-bit key vector
0xCA	Parameter A Register	LSW of the IV vector
0xCB	Parameter B Register	
0xCC	Parameter C Register	
0xCD	Parameter D Register	MSW of the IV vector

Table 1.1: Set Parameter - Feature Identifiers Assignation

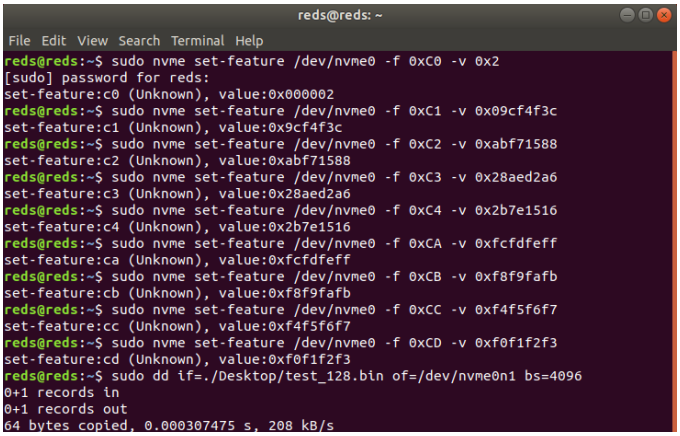


Figure 1.54: General Execution - AES-128 Host PC Terminal

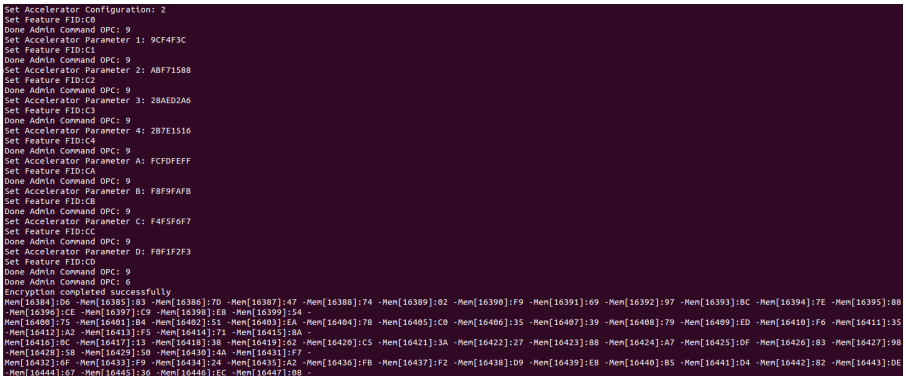


Figure 1.55: General Execution - AES-128 Developer PC Terminal

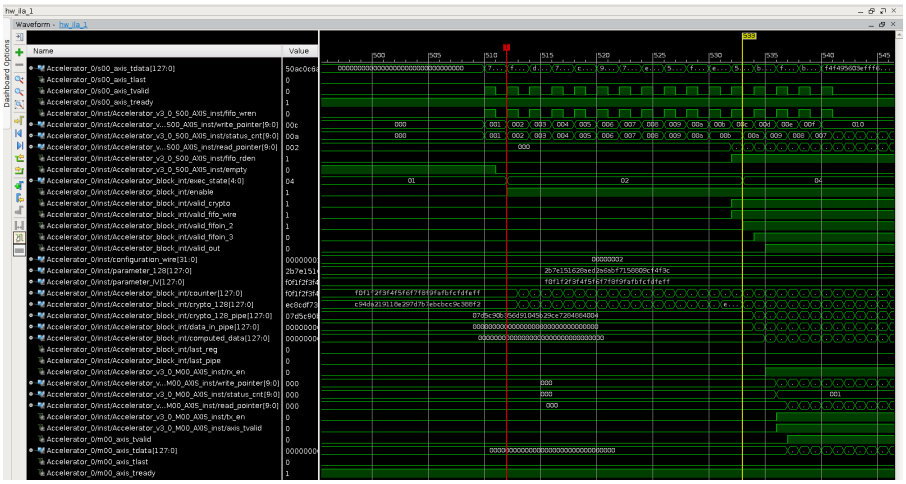


Figure 1.56: General Execution - AES-128 Encryption Waveform

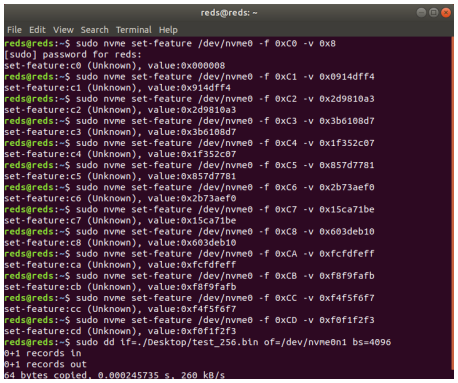


Figure 1.57: General Execution - AES-256 Host PC Terminal

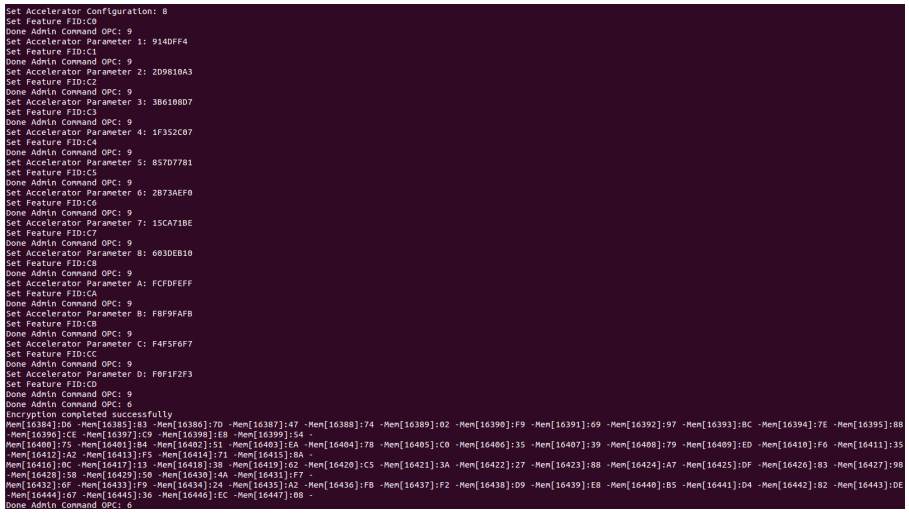


Figure 1.58: General Execution - AES-256 Developer PC Terminal

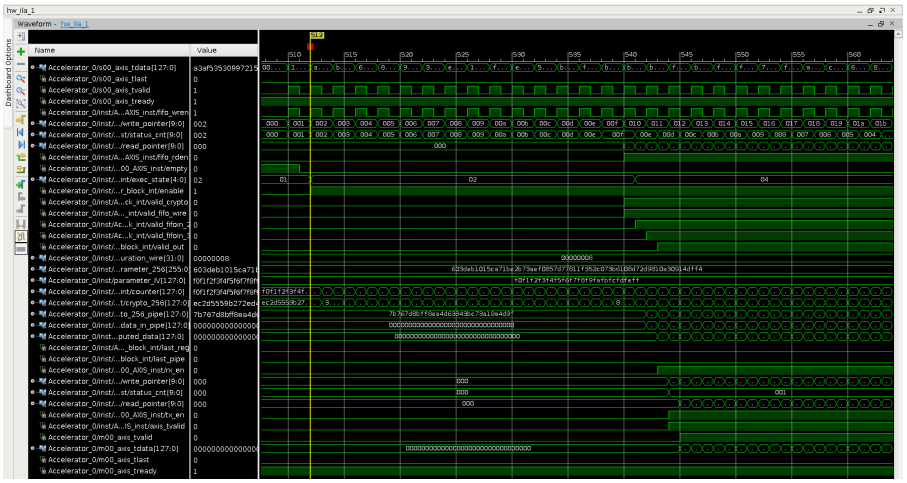


Figure 1.59: General Execution - AES-256 Encryption Waveform

Performance Tests

As in the previous cases, performance tests are carried out to evaluate IOPS, bandwidth and latency for the final computational storage configured to perform the cryptography.

As it can be seen in Figures 1.60 to 1.63, there is almost no degradation in performance after the insertion of the hardware accelerator.

Figure 1.60: Performance test - dd function, write and read - 128bit AXI DMA Version

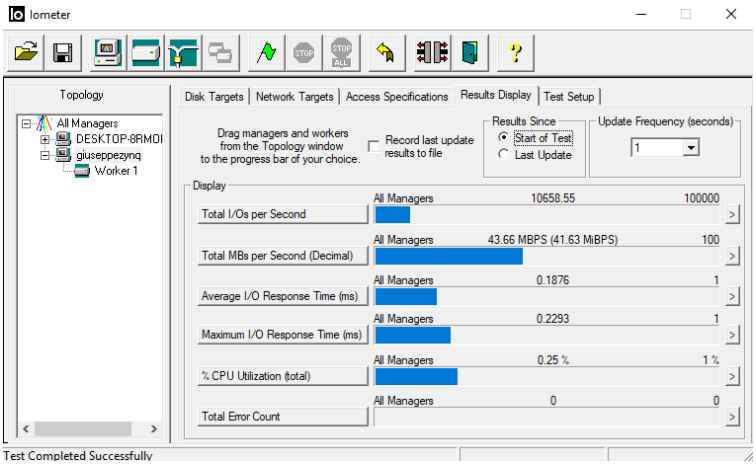


Figure 1.61: Performance test - Iometer Sequential Read transfer size = 4k, block size = 4k - Computational Storage

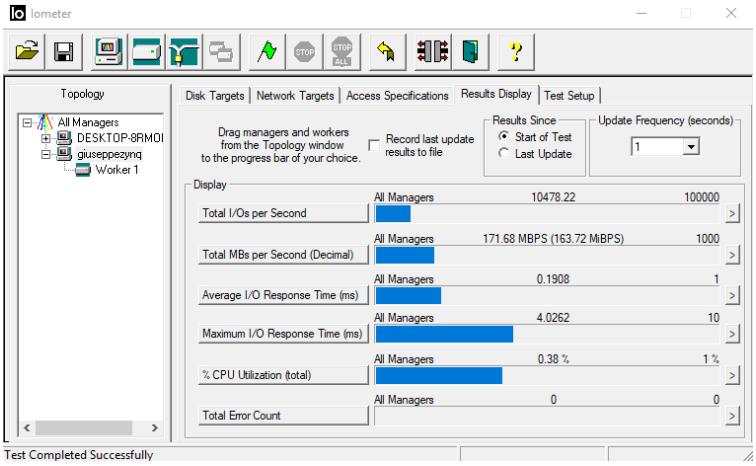


Figure 1.62: Performance test - Iometer Sequential Read transfer size = 16k, block size = 4k - Computational Storage

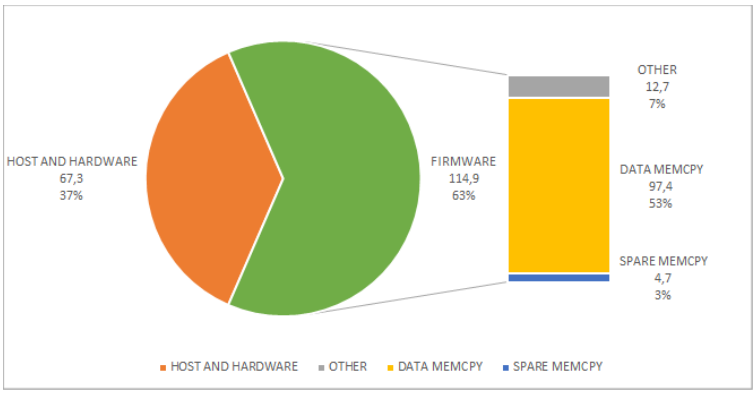


Figure 1.63: Performance test - Read Latency - Computational Storage

Bibliography

[1] Yong Ho Song, *Cosmos+ OpenSSD 2017 Tutorial*, HYU ENC Lab of Hanyang University, 2017.

[2] Xilinx, *ZC706 Evaluation Board for the Zynq-7000 XC7Z045 SoC, User Guide*, Page 46-47, 2019.

[3] NVM Express, *NVM Express™Base Specification Revision 1.4*, Page 206-207, June 2019.

[4] Homer Hsing, *AES Core Specification*, OpenCores, February 2013, Accessed June 2020, <https://opencores.org/projects/tiny_aes>

[5] Morris Dworkin, *Recommendation for Block Cipher Modes of Operation: Methods and Techniquesm*, NIST Special Publication 800-38A, December 2001.