BEAM

# Dune NVMe Storage Design

*Live Document*

| | |
|---|---|
| Project | DuneNvme |
| Date | 2020-06-16 |
| Reference | DuneNvmeStorageDesign |
| Version | 1.0.0 |
| Author | Dr Terry Barnaby |

## Table of Contents

# 1. Introduction

This is a live document holding information and notes on the design of the Dune NVMe Storage System. The user level manual is DuneNvmeStorageManual which covers the core's interfaces and use of the FPGA core and testing of that core.

# 2. Requirements

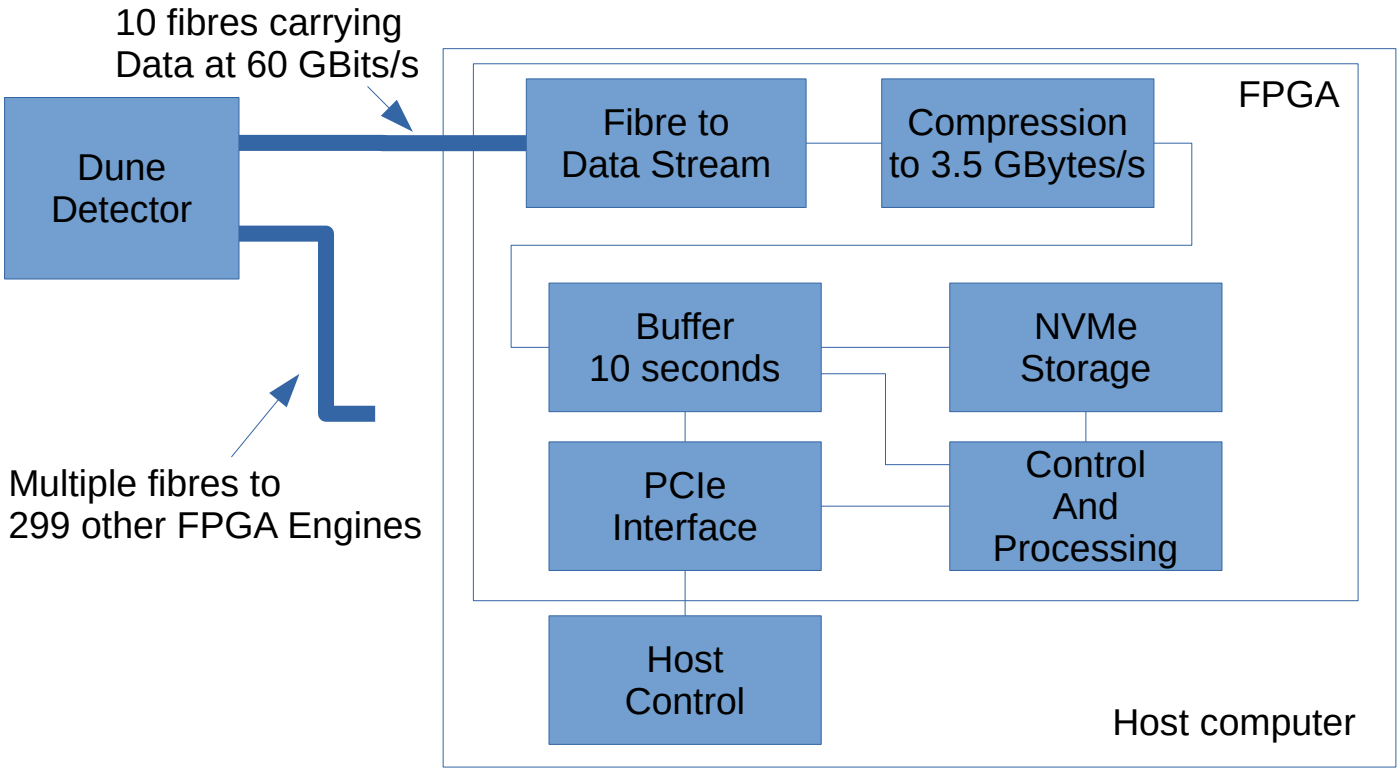
The Dune Neutrino experiment has two neutrino detection chambers. Each chamber has a matrix of wires and electronics to amplify and digitise signals captured from neutrino interactions in the chambers. The digitised data is passed through optical fibres to 300 FPGA data processing engines installed in conventional x86 computer servers. The fibres to each FPGA carry around 60 GBits/s of data plus extra bits for encoding overheads.

The Dune project will be designing a custom FPGA board to be installed into the multiple host server computers. This FPGA board will convert the optical data streams to electrical signals, compress and process this data. Currently a simple compression algorithm is proposed that provides around a 2:1 compression ratio. The resulting 3.5 GBytes/s, approx, data stream will be stored in a DDR4 RAM based buffer that can store up to 10 seconds worth of data.

The overall requirement of the NvmeStorage core is to store, real-time, the raw data stream from the Dune system and be able to, with a slower access rate, read that data. The full system is able to store 120 TeraBytes of incoming data at a rate of 600 GigaBytes per second across its multiple FPGA based data processing engines.

For each FPGA engine the source data consists of fixed sized chunks (bursts) of data that are from 20 to 200 GBytes in size. These data chunks are stored, round robin fashion, into the NVMe devices with a new chunk overwriting the oldest chunk written.

The host server computer controls the overall operation of the system via a PCIe interface. It is able to




lock a particular chunk in NVMe storage against subsequent writes if it contains data of interest. The data from this chunk can then be read out by the host computer.

## 2.1. Dune Raw Data

For information, the raw data from the Dune sensors consist of 2560 wires sampled with 12 bits resolution. A super packet containing 64 sets of this data will likely be generated, that includes a start time field and other control information. This will yield data packets of a size near 128 kBytes after compression.

The NvmeStorage core will store data with a block size granularity of 4 kBytes. If an NVMe error occurs, maybe due to a large block write latency or other error, the system will continue storing data, if it can, keeping the block numbers consistent. This means that there could be a set of blocks with in-valid data. Unwritten data blocks should have there contents set to 0, as long as the Nvme trim/deallocate function works correctly in the Nvme's used. So some method of determining if a packet of data is corrupt is needed. The compression algorithm used will provide variable length blocks. We suggest using a variable length packet with CRC checksum padded to a 4 kByte boundary. So a data packet for 64 * 2560 x 12 bit wire samples could look line:

| Item | Length | Notes |
|---|---|---|
| Length | 32 bits | The compressed data length in bytes (packet length is this field + timestamp + misc header length + CRC length). |

| | | |
|---|---|---|
| Timestamp | 64 bits | Timestamp for data |
| Misc | ? | Extra header information as needed |
| Compressed samples | ~122 kBytes | The compressed data for 64 samples of 2560 wires at 12 bit resolution. |
| CRC | 32 bits | CRC32 checksum |
| Padding | ? | Padding to next 4/8 kByte boundary |

The actual packet format is irrelevant to the NVMe storage system. It simply stores data in blocks to a 4 kByte boundary. However, due to possible latency issues, the NVMe may have to drop data and this needs to be handled in a way that allows the host software to use the resulting data stream and understand when data has been lost. To do this the data stream needs to be blocked into some sized packet with a suitable header and dropped at that chunk size.

Two possible methods are:

1. Store and drop at the super packet (~128 kByte) level.
2. Store and drop at the storage packet 4 kBytes level, the same as the underlying NvmeStorage block size. Ideally this 4 kByte block would have some header to allow the software to understand if some of these 4 k packets are missing. This could be two 16bit values: the super packet number and the 4k packet number within the super packet.

When a 4k block is sent across the AXI4-Stream, the tlast signal will need to be set in the last clock cycle of the block.

# 3. NVMe Storage Requirements

The overall requirements of the NVMe storage system are thus:

1. To be able to write up to 200 GByte chunks of data at 3.5 GBytes/s to NVMe storage. The data rate should match, or be better than the system buffers incoming data rate so that no incoming data is lost.
2. There will be some latency in writing the data to the NVMe storage. As well as processing latency, there will be some latency in the NVMe devices themselves caused by their block erasing and error recovering systems. It is not known what the peak latency will be as yet, this will be subject to experiments later in the project. However the system should limit the latency dropping data if necessary. The system main buffer will take up this latency. A figure for the maximum about of buffer space that can be used needs to be define. Lets say for now this will be 1 seconds worth of data so 10% of the buffer space.
3. The NVMe system should be able to store at least two separate 200 GByte chunks of data so that the system can be storing new data while reading.

4. The NVMe system should provide the ability to read the data from the inactive data chunk while the data write occurs. The reading of data is lower priority than writing the data and the read data rate is not critical.

# 4. NVMe Overview

NVMe FLASH storage devices use a register with DMA type API over a multi-lane PCIe physical interface. The NVMe devices thus look like a conventional PCIe device to a host system. The NVMe PCIe device presents the standard PCI configuration registers and a small set of its own registers for control. Data passing is accomplished by shared memory queues and data areas and a set of doorbell registers to indicate queue updates. The NVMe devices perform bus master access to read and write to these queue and data memory.

Current NVMe devices are based around the PCIe Gen3 physical interface (although there are some Gen4's becoming available) and generally implement the NVMe 1.3 protocol. We will base the Dune NVMe design around the PCIe Gen3/NVMe 1.3 interface standards although PCIe Gen4 and higher NVMe standards should work fine if the FPGA's physical PCIe lane interfaces used, NVMe devices and the PCB tracking support this.

NVNe API specification is in: NVM_Express_Revision_1.3.pdf
In essence:

- The NVMe implements a standard set of PCIe configuration registers.
- The NVMe has a set of controller registers.
- Most of the control is performed using shared memory based command submission and completion queues and uses a doorbell register for each queue to indicate the queues state. The command queues have 64 Byte fixed entry slots and can be configured for up to 64 k entries each. We will likely implement only a small number, say 8 entries, to reduce FPGA fabric overheads whilst still allowing the NVMe a queue of requests for its efficiency.

**PCIe Configuration registers**

For the most part we will not need to configure any of the NVMe's PCIe registers. The only register we need to change is the command register to enable memory accesses and bus master accesses. NVMe devices normally have one BAR address range. We can leave this set to 0. If they have multiple BAR addresses we might have to set the second BAR although this is normally only used for IO rather than Memory based accesses.

| Register | Setting |
|---|---|
| PCIe Command register | 0x06 |

The NVMe PCIe root complex implemented by the Xilinx PCIe hard block doesn't need any configuration space registers set for our usage.

*Do we need to set some MSI-X registers for interrupts ?*

**NVMe Queues**

The NVMe operation protocol involve the use of request and reply queues. there can be up to 64000 separate command submission and completion queues, although particular NVMe devices limit this. Each request queue entry is a fixed 64 Bytes in size, the reply/completion queue has 16 Byte fixed size entries. The submission queue entry defines the command and the pointer to data etc. For the Dune system we only need to use two or three queues:

- Queue0: Admin control queue. The FPGA will need to use this to initialise the IO queues needed. We can use FPGA ROM to send the initialisation sequence of packets or allow the host CPU access to these queues so host software can set these up.
- Queue1: Data write queue. NVMe write and trim (deallocate) commands will be sent over this queue. NVMe read commands could also use this queue if wanted. It might be worth using a separate queue for the trim/deallocate commands.
- Queue2: Data read queue. NVMe read requests to fetch the data would be sent over this queue. Note that queue priority is based on the queue number with queue 1 (the write queue) having the highest priority.

In each NVMe command, the data location is defined as a direct 64 bit memory page pointer or a scatter gather list. For our simple streaming needs, the direct 64 bit memory page pointer is sufficient. We may be able to use simple FIFO's rather than random access memory for some data.

The NVMe devices can execute the given requests in any order and some requests may be delay with respect to others. So the reply mechanism needs to be used to determine request completion and the data used by the request needs to be present for the duration of the request.

Current NVMe devices can support around a 2 GByes/s write speed over Gen3 PCIe. Very new NVMe devices are reported to support up to 4 GBytes/s over Gen4 PCIe. We will implement the system for Gen3 PCIe devices with a maximum write speed of 2 GBytes/s. In this case we will need to write to 2 NVMe devices in parallel to support the 3.5 GBytes/ data rate required. The system will be designed so that future Gen4 PCIe, 4 GByte/s devices can be used.

One major issue with SSD storage as used in NVMe devices is that they wear out when writing data to them. The devices are normally specified with an endurance value to define this. With the very fast and large streaming data writes the Dune system uses the NVMe endurance has to be carefully considered. There are three basic structures of SSD device available: SLC, MLC and TLC. SLC stores 1 bit per cell, MLC normally 2 bits per cell and TLC 3 bits per cell. The SLC has a greater endurance than the MLC and in turn is greater than the TLC type. The overall lifetime of the NVMe storage is thus defined by:

1. The amount of data per second that is written to the drive on average. If the data chunk size is 200 GBytes and this is written once per day this is 200 GBytes/per day.
2. The size of the storage drive. Larger storage means each storage cell will be written to less often.
3. The type of SSD cell is use SLC, MLC or TLC.
4. Temperature of the NVMe may be a factor. Cooling the NVMe's has to be considered.

So the NVMe type and size will need to be carefully chosen for the expected chunk duty cycle.

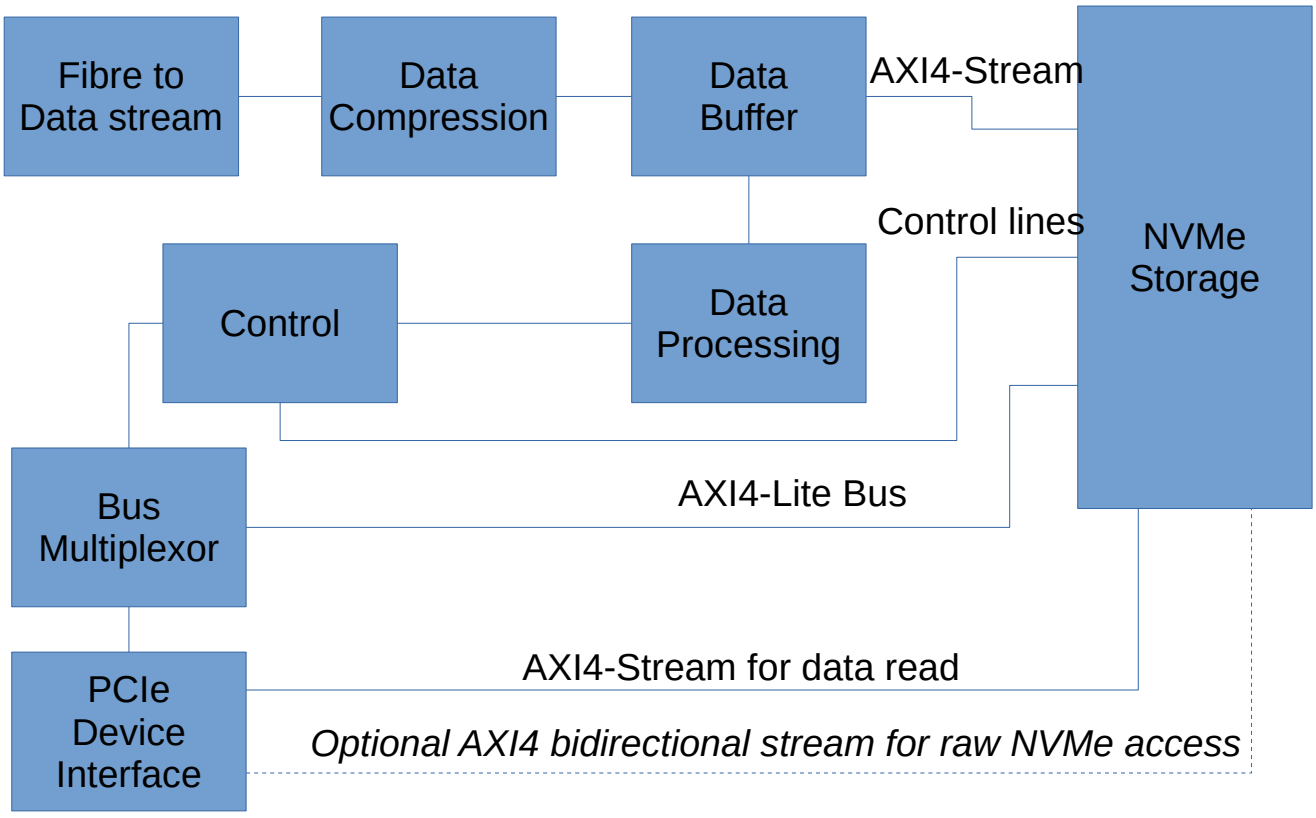
For a typical commodity Samsung 970 Pro MLC 512 GByte device the wear rating is 600 TByte. This can store roughly 600T / 200G = 3000 data chunks before failure. So the lifetime of these drives will be approximately:

- One data chunk per day = 8 Years.
- One data chunk an hour = 125 days
- Continuous data storage = 3 days

If instead 1T Byte NVMe's are used (so that 4 x 200 GByte data chunks can be stored) then the NVMe lifetime would be doubled.

## 5. Dune NVMe Storage FPGA Firmware Design

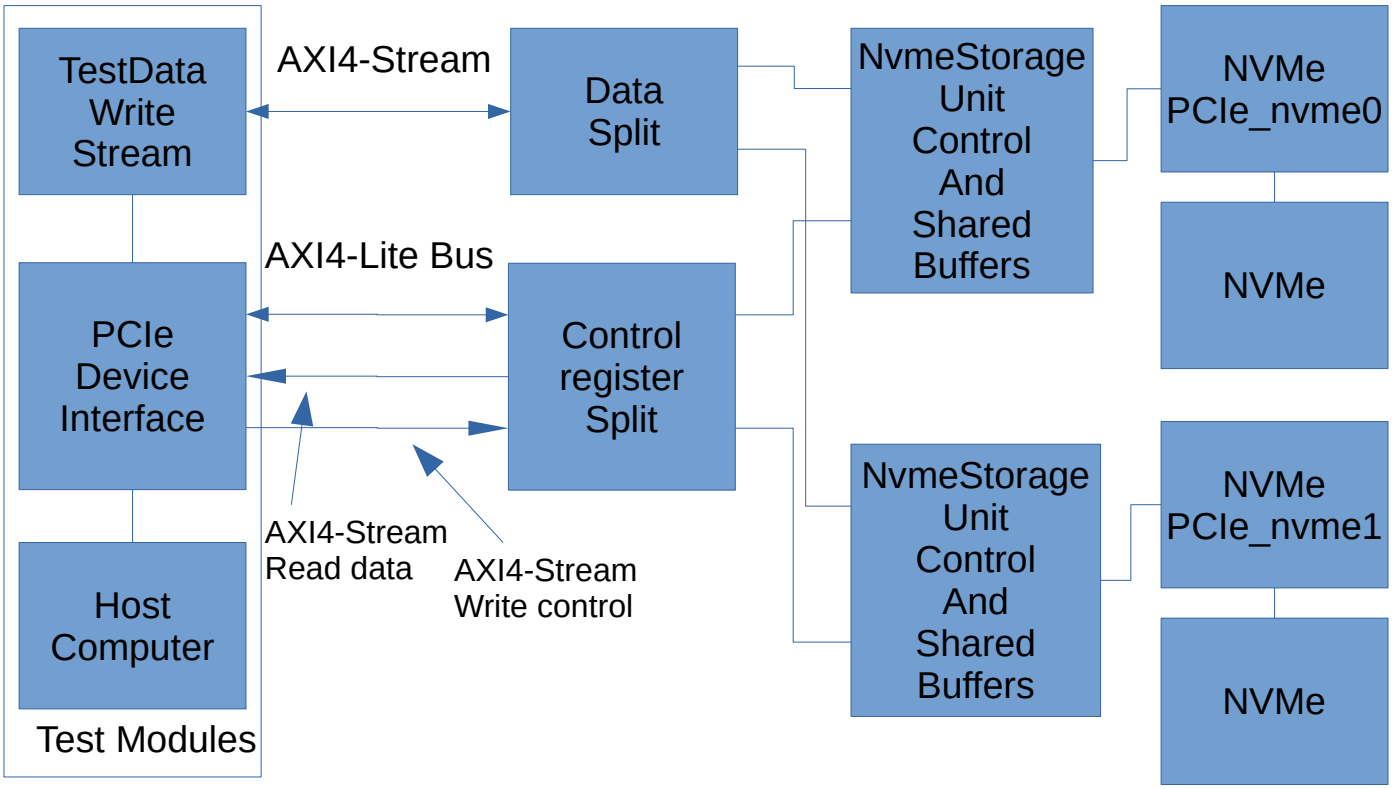
The expected overall FPGA firmware design for the system is shown below. This aspect is being designed and developed by other parts of the Dune project.




The main blocks of interest here are:

- PCIe Device interface. PCIe interface to host computer. Provides PCI configuration and FPGA control registers. Will likely be based on the Wupper PCIe core but we have used the Xilinx DMA/Bridge Subsystem's IP core and the Beam BFpga Linux driver on the host side.
- Bus Multiplexor. Provides access to both the general FPGA system control registers and the NVMe control registers by simple address based segmentation.
- NVMe storage. This implements the NVMe Storage system that this project is involved with.

The following is a basic block diagram of the NvmeStorage block:




The Test modules implement a test interface to closely match the final FPGA system. The host computer controls the sending of test data and provides access to the NVMe devices via the NvmeStorage core.

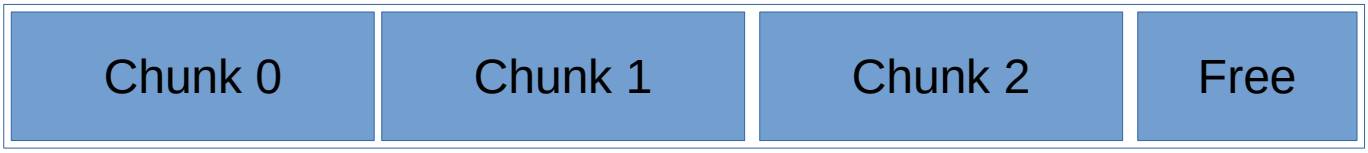
The NVMe storage subsystem uses two separate NVMe devices to handle the 3.5 GBytes/s overall data rate. The incoming data is split into two paths for the two NVMe devices into 4k blocks. Each NVMe device receives alternate data blocks.

- Data Split: Splits the incoming AXI4-Stream into two sending alternate blocks to the two NVMe devices.
- Control Split: Performs overall control register accesssplit to the two NVMe Storage cores. This uses a set of registers driven through an AXI4-Lite bus interface. A one way AXI4-Stream interface is used for data read allowing for DMA read of the NVMe storage over the standard Xilinx PCIe DMA/XDMA system.
- NvmeStorageUnit, Control and Shared Buffers: This implements the actual NVMe control system.

- NVMe PCIe_nvme[01]: This implements the PCIe interface to the NVMe devices. It will likely use a Xilinx PCIe UltraScale Devices Gen3 Integrated Block core and 4 x PCIe physical lanes to communicate with the NVMe device. However a PCIe PHU softcore interface based on the Xilinx PCIe PHY IP could be considered.

## 5.1. NVMe Storage Access

The NvmeStorage uses a block size of 4 kBytes. The NVMe storage use a simple integer block number to




address the individually stored blocks. The host's software will likely split the data storage area into DataChunk areas of between 20 and 200 GBytes in size. It is recommended to leave a certain amount of storage at the end of the NVMe reserved for additional free blocks to assist with the NVMe's wear bad blocks system and probably reduce write latency. The NVMe devices manage the actual physical "blocks" used for each logical block and having a percentage that have not been written to provides it with a greater number of free blocks to use. Note that these blocks should not be written to or they should be trimmed/deallocated so that the NVMe drives are free to use them.

The data chunks will align to a 4 kByte boundary. It might be good to align them to an NVMe erase block boundary (possibly 1 MByte depending on NVMe), we will determine this later in the project.

The NvmeStorage modules registers are used to indicate the starting block and how many blocks to write.

## 5.2. NVMe Storage Module Interface

This is detailed in the DuneNvmeStorageManual. This section contains some notes on the interfaces.

We assume here that the FPGA systems master clock is 250 MHz, possibly generated from the hosts 100 MHz PCIe card slots clock. Each of the NvmeStorageUnit modules takes its clock from the external NVMe PCIe 100 Mhz clock/clocks that also drive the two external NVMe devices. The PCIe_nvme[01] root complex hard blocks produce a 250 MHz user clock for the NvmeStorageUnit's logic. The system has been based on that 250 MHz clock frequency and the timings have been targeted for that clock speed. It maybe that the FPGA systems master clock will run at a different speed to this in the future. The system has some flexibility in clock speeds.

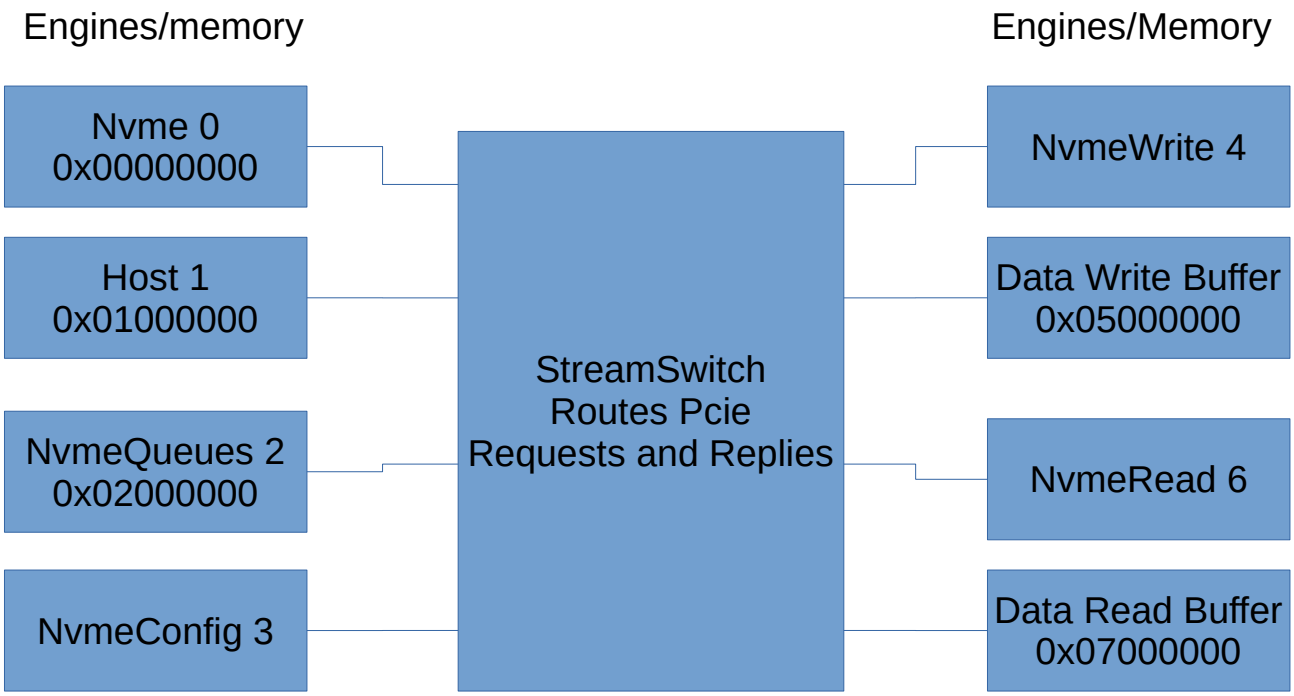
The NVMe FPGA module presents the following interfaces to the physical interface and the rest of the FPGA fabric.

# 6. NvmeStorage NVMe operation

Following experimentation with the NVMe devices and the Xilinx PCIe blocks we have developed an access design as follows. All of the communications with the NVMe device is performed using PCIe packets. The Xilinx PCIe root hard block provides 4 streams with its own style PCIe packet headers in

order to send and receive packets over these streams. We could use the simpler Xilinx PCIe PHY block in the future that would just deliver raw PCIe TLP packets.

The architecture uses a PCIe packet switch to route request and reply packets to/from the various transaction engines, memory and the NVMe device. This modularises and simplifies the system somewhat. It also allows the host computer system full access to the Nvme device while the FPGA is using it for writing data, This allows Nvme status and logs as well as Nvme data read and writes to be performed.




## 6.1. PCIe TLP packets

When communicating with the NVMe over the Xilinx PCIe Gen3 core, PCIe packets are sent and received with a particular packet header. These headers are translated to/from genuine PCIe TLP headers by the Xilinx PCIe Gen3 core. The Xilinx PCIe Gen3 manual provides details of these headers. A request header is 16 Bytes long (4 x 32 bit DWords) and a reply packet's header is 12 Bytes long (3 x 32 bit DWords).

The PCIe protocol uses simple TLP packets to send requests and receive replies from the Nvme devices. There was a choice over what style of packets to use within the NvmeStorage system. It would be nice to be able to use the standard PCIe TLP packets as these are quite suited for our use. Unfortunately the Xilinx PCIe Gen3 hard core implements its own style of PCIe packets translating them to/from genuine PCIe TLP packets. PCIe TLP packets are well suited to bi-directional communications over a single bi-directional stream. The Xilinx PCIe Gen3 packets are designed to use 4 separate streams, two for the host requests and replies and two for the Nvme requests and replies. Note if we use the Xilinx PCIe Phy ip core we would be using standard PCIe TLP packets.

We could do one of the following:

1. Use PCIe TLP packets in our system and translate to/from Xilinx PCIe Gen3 packets at the PCIe hard blocks interface.
2. Use the Xilinx PCIe Gen3 packets in our system, but modify the usage of some of the bits to suit our bi-directional system. We would multiplex/de-multiplex the dual b-directional streams to one pair for the Xlinix PCIe Gen3 hard block to simplify this interface.

### 6.1.1. Using PCIe TLP Packets

We would use PCIe TLP packets as they are. The Requester ID field and the address fields 4 bits will be used to indicate the particular request/reply stream in use. The address bits are used for requests so that this will work for the Nvme's bus master accesses. The TLP headers fmt and type fields would be used to indicate packet type including if the packet is a request or a reply.

If we are using a Xilinx PCIe hard-core we would have to translate the PCIe TLP packets to/from Xilinx PCIe hard-core packets. This looks relatively easy to do.

### 6.1.2. Using Xilinx PCIe Gen3 Core Packets

We would need to include information on if the packet is a request or a reply as there is no information in a standard location for this. We could use bit 31 of DWord 2 for this setting it high for replies.

The Requester ID field and 4 highe end bits of the address field will be used to indicate the particular request/reply stream in use. The address bits are used for requests so that this will work for the Nvme's bus master accesses.

If we are using a Xilinx PCIe PHY core we would have to translate the Xilinx PCIe Gen3 Core packets to/from PCIe TLP packets. This looks relatively easy to do.

Actually most of the fields are actually redundant for our usage and we could use a simpler/smaller packet header.

The packet headers will thus be like:

**Request header**

| DWord | Value | Description |
|---|---|---|
| 0 | 0xNSAAAAAA | N: Nvme number, S: Stream, A: Address |
| 1 | 0 | Not used (Address upper 32 bits. We could put the N and S fields in here if wanted) |
| 2 | 0x000SRCCC | S: Stream, R: request type and CCC: DWord count (request is 5 bits count is 11 bits) |
| 3 | 0x000000TT | TT: Request tag |
| 4 -n | ... | The following DWords contain the data |

**Reply Header**

| DWord | Value | Description |
|---|---|---|
| 0 | 0x0BBBEAAA | B: Byte count, E: Error, A: Address |
| 1 | 0x000SUCCC | S: Stream, U: Status, C: DWord count |
| 2 | 0x000N00TT | N: Nvme number, T: Request tag |

Note that as we have a 64bit address, we could use bits in the upper 32bits of the address to define the routing of the packets thus preserving a greater 64bit address range.

### 6.1.3. Alternate Packet Structure

Note we could implement our own headers, perhaps two words so one 128 bit cycle would be used for both requests and replies including the first 2 x 32 bit words of data. A translator block could then convert between this and either the Xilinx PCIe Gen3 core protocol or direct PCIe TLP packets. This could be like:

**Possible Request header**

| Address | Value | Description |
|---|---|---|
| 0 | 0xNSAAAAAA | N: NVMe number, S: Stream, A: Address |
| 1 | 0xTTR00CCC | T: Request tag, R: Request type (0 = read, 1 = write), C: DWord count |

**Possible Reply Header**

| Address | Value | Description |
|---|---|---|
| 0 | 0xNS000BBB | N: NVMe number, S: Stream, B: Byte count, top bit set to 1 for replies |
| 1 | 0xTTREECCC | T: Request tag, R: Request type (0 = read, 1 = write), E: Error, C: DWord count |

Using this structure would mean that most packets would fit in a single 128bit word. Instead of the NS fields we could put these in the tag: Bit 7: NVMe device, Bits 6-4: stream, Bits: 3-0: queue entry.

### 6.1.4. Packet Structure used

We have chosen to use the Xilinx PCIe Gen3 packets in our system as it simplified the code and testing when using the Xlinix PCIe Gen3 hard block. We did experiment with using genuine PCIe TLP packets and it would be possible to convert to this style with a bit of work. This would be nice as we would be keeping to a major standard rather than the Xilinx specific standard for the packets but otherwise would not have any benefits and would increase the development and testing burden.

## 6.2. Packet Streams

To allow a modular NvmeStorage design, we have implemented separate Pcie packet processing engines sending and receiving packets over AXI streams and have a simple switch to route these packets to the appropriate processing blocks. Each engine will has a TX and RX AXI stream channel and the packets use the address field for routing requests and the requesterId for routing replies. In effect we have implemented a simple PCIe “bus” like structure within the FPGA.

The address fields bits 24-27 are used to route request packets and the PCIe RequesterId fields are used to route replies. Note the Xilinx Pcie Gen3 core seems to only support 3bits for the RequesterId. For simplicity we have used the same number for both the request and reply stream addresses on the requester blocks.

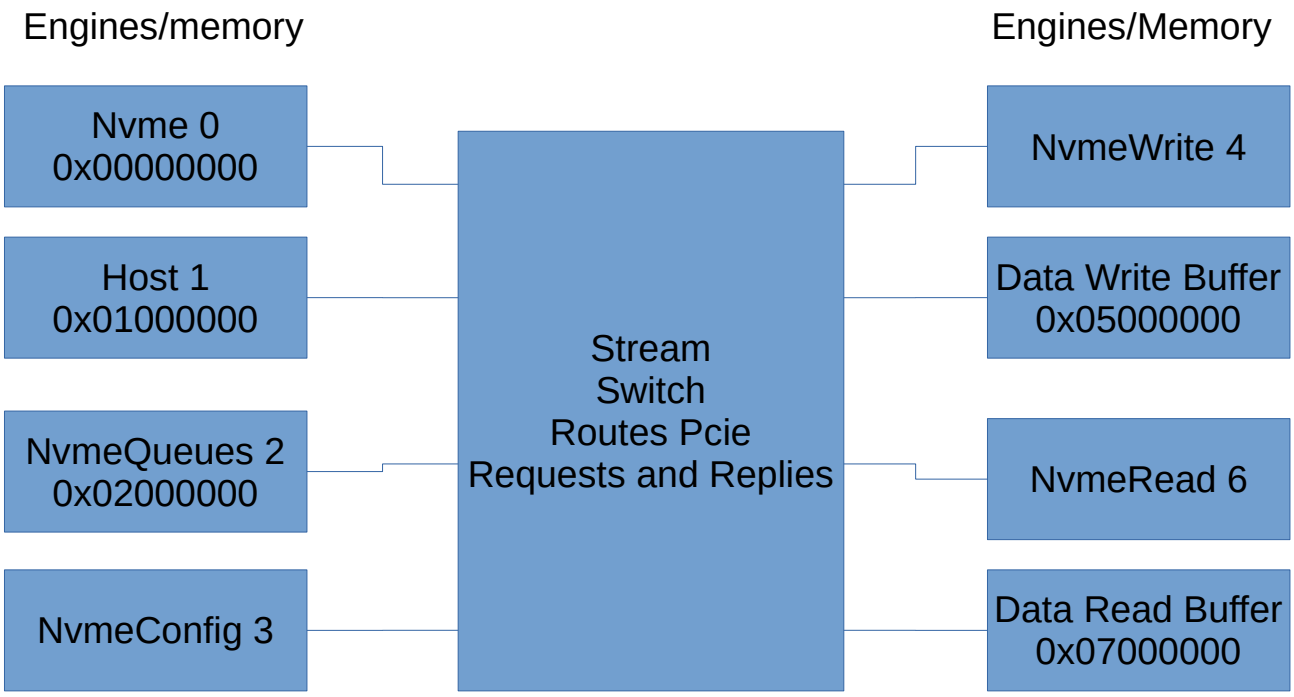
There are 6 requester engines:

| RequesterId (3 bits) | Description |
|---|---|
| 0 | Nvme |
| 1 | Host computer |
| 2 | NvmeQueues queue engine |
| 3 | NvmeConfig configuration engine |
| 4 | NvmeWrite data write engine |
| 6 | NvmeRead data read engine |

The address locations are as follows:

| Address | Description |
|---|---|
| 0x00000000 | Nvme (Register area) |
| 0x01000000 | Host computer |
| 0x02RQ0000 | NvmeQueues queue engine (Address bits 16 and 17 are queue number and bit 20 indicates reply queue) |
| 0x05000000 | NvmeWrite data input FIFO (For bus master reads) |
| 0x07000000 | NvmeRead data output FIFO (For bus master writes (not used as yet)) |

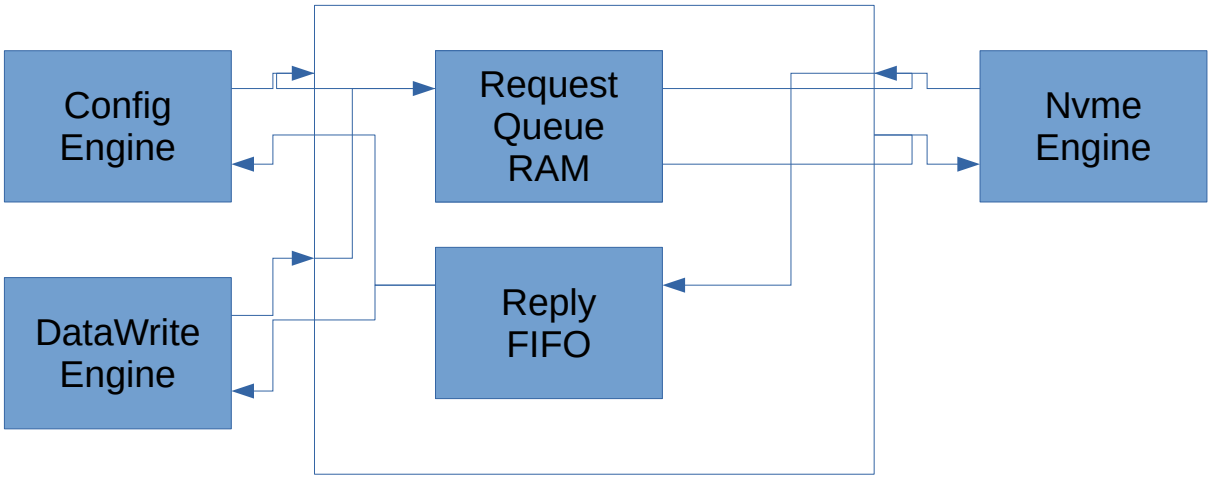
The top most 4 bits of the address are used to indicate communication with the Nvme0 or Nvme1 storage modules.




## 6.3. NvmeQueues Nvme Request/reply Queue Management

We needed some way of handling the shared memory Nvme request/reply queues. From the Nvme side we can simply use straight memory read/write accesses. However from the FPGA control side it would be nice/simpler to automate the request and reply queues to simplify things. So we have implemented a NvmeQueue engine to operate the Nvme request/reply queues.

- When a Pcie write request is received the NvmeQueue engine ignores the lower bits of the write address and simply appends the packet's contents to a particular numbered queue sending a Nvme register write request to the appropriate Nvme doorbell register with the queue entry number. The various engines then use this to submit queue entries without having to know where to write the queue's entry. The upper 8 bits of the queued request's Command Identifier field is set to the requestersId so the final reply packet can re routed to the engine that made the request.
- When a reply is sent from the Nvme device to be written into the Queue ram, instead of actually writing it to RAM the NvmeQueue engine re-directs it to the appropriate requester as a PcieRequest packet so that the requesting module can respond to it based on the Command Identifier field's to most 8 bits. The address of 0x0X100000 in this packet indicates it was an Nvme queue reply. The NvmeQueues engine writes to the Nvme reply doorbell register for the particular queue to indicate reception.




At the moment the NvmeQueue engine is simply linked by two stream channels to the StreamSwitch. It would be possible to add additional stream channels directly to the Nvme device and/or another Stream switch to allow simultaneous read and writes to the queues.

Some more detail:

On receiving a Pcie write packet the NvmeQueue engine will:

- The queue entries CommandIdentifier should have its top 8 bits set to the requesterId, by the engine posting the queue entry.
- The queue data in the Pcie packet is written into next queue location in RAM and the queue pointer updated.
- Sends a PcieWrite request to set the Nvme's doorbell register for the appropriate queue to update the queue pointer location.

On receiving a Pcie reply packet the NvmeQueue engine will:

- Create a Pcie write packet header with the address field set to the appropriate engine based on the reply queue's CommandIdentifier field's upper 8 bits.
- Sends the Pcie write packet on the PcieWrite command to the appropriate engine with the reply queue entry as a payload.
- Note that the system buffers the entire reply queue packet (2 x 128 bit words) so that the StreamSwitch is not locked up.
- The engine receiving the Pcie write request would look at the address to see if this is a queue reply (address bits) and then check the status field to determine if an error occurred.
- The NvmeQueue engine sends a PcieWrite request to set the Nvme's doorbell register for the appropriate reply queue to the reply queue pointer location.

## 6.4. StreamSwitch

This is the heart of the system but simple in operation. It performs the routing of request and reply packets sent by processing engines to a destination based on the request's address and the replies requesterId.

At the moment it has a simple priority system in that lower numbered streams have higher priority. We might want to increase the priority of particular streams or maybe just change their numbering order as needed.

The simple switch only allows one packet at a time to be sent through it. It could be modified to support multiple packet transfers simultaneously.

The Stream switch currently has a 1 cycle latency. It could be possible to reduce this.

## 6.5. NvmeConfig NVMe Configuration

Each NVMe device is connected to a separate PCIe 4 lane Xilinx hard IP PCIe block configured as a root complex. As there is only one device per root complex little PCIe enumeration and configuration needs to be done. Most NVMe devices have a single BAR memory address range and so we have kept that at the default address of 0. Some NVMe's have a second IO BAR address range, we don't use this.

The only PCIe configuration needed is to set the PCIe command register to enable memory accesses and bus mastering accesses (Write 0x0006).

Following this the NVMe's registers need to be configured and the admin. data write and data read shared memory queue initialised.

The NvmeConfig engine performs this work using a simple ROM structure containing the Pcie write requests to be performed in order to configure the various registers and queues. There is a FPGA module parameter that instructs the core to automatically perform this configuration after reset. If this is parameter is not set, then the host CPU can instruct the NvmeConfig engine to perfom its work via the modules control register or it can manually perform the configuration itself over the Control request and reply streams. See the DuneNvmeStorageManual for the register operation.

## 6.6. NvmeWrite Data Write

This is the heart of the module. It needs to perform efficient writing of the data stream to the NVMe devices to obtain the 4 GByte/s average data transfer rate with as low a latency as possible. It uses the Nvme trim/deallocate system to inform the Nvme of free blocks prior to performing the data write.

For efficiency it is able to buffer and write up to 8 x 4k Blocks at a time (configurable).

Some notes:

- There is an NVMe queue for sending write requests. This is by default configured with a queue size of 16 (DataWriteQueueNum) entries. This allows 8 concurrent data block writes whilst performing block trim/deallocate operation. We have found that writing 8 x 4k blocks at a time is

sufficient on the Nvme's we have to obtain the performance needed. It would be possible to write larger numbers of blocks at once to possibly obtain higher data rates.

- The multiple block write queue entries allow the NVMe to continue writing while one or more block write requests have a high latency, perhaps due to physical block reallocation in the Nvme.
- Input data is buffered in a 128 bit wide RAM.
- When at least one blocks worth of data is available in the DataBuffer the NvmeWrite state machine sends the data to the NVMe:
  - Send a Nvme IO write command to the WriteData request queue (NvmeQueues engine) with the DataBuffer's address in the 64byte queue entry. In actual fact most of this queue entry is static for all writes apart from the source address and the NVMe block number.
  - The NvmeQueue engine will "send" this queue entry to the Nvme by writing the queue request into RAM and writing to the WriteData queue doorbell register. The Nvme will perfoem a "bus master" read request for this queued request and perform the necessary work.
  - The NvmeWrite engine responds to Nvme read data requests from the DataBuffer for the actual data to write to the Nvme.
  - Wait for a WriteData reply from the Nvme via the NvmeQueue engine. If there is an error status set an error. This will wait for the NVMe to write to the WriteData reply queue a status message with the same tag as the request. The status is in this packet. The data buffer for this block is now freed for input.

When sending requests or replies to the NVMe they have a corresponding Xilinx PCIe request or reply header added.

**A Write data request**

The actual data to write to the NVMe is sent in 128 Byte chunks to the NVMe because of PCIe packet size limitations. This could likely be increased to 512/1024 Bytes with appropriate settings and when the particular Nvme supports it (The Xilinx PCIe Gen3 block allows greater sizes).

| Host/Queue/Data | Direction | Nvme |
|---|---|---|
| Data is written into in DataBuffer | | |
| NvmeWrite: When a 4k block is available a DataWrite request queue entry is added with the DataBuffer's address | -> | NvmeQueue queues this request and writes to the Nvme request queue doorbell register |
| NvmeQueue: Process readqueue request | <- | Send readqueue request for queue entry |
| NvmeQueue: Send readqueue reply head and data | -> | Process readqueue reply header |
| NvmeWrite: Process readdata request | <- | Send readdata request for writedata |
| NvmeWrite: Send readdata chunk reply head and data | -> | Process readdata reply header |
| NvmeWrite: Send readdata chunk reply head and data | -> | Process readdata reply header |
| NvmeWrite: Send readdata chunk reply head and data | -> | Process readdata reply header |
| | ... | |

| Host/Queue/Data | Direction | Nvme |
|---|---|---|
| NvmeWrite: Process writequeue reply | <- | Send reply queue entry |

### 6.6.1. NvmeWrite Implementation

There are n x DataBuffers implemented in a singe RAM area each buffering a block (could be more) of data to be written. For each DataBuffer there is a status register and an area of BlockRAM to store the data block.

- **Input data process**: When at least a block of data is available this process looks for a free buffer. When available it marks this buffer as in use, stores the block number of the data, stores the start time and starts feeding the input data stream into the BlockRAM. When this BlockRAM is full (on the stream's last signal) the DataBuffer number is written to a process queue.
- **Process requests**: This process accepts DataBufffer numbers from the process queue. It then sends an Nvme write request into the Nvme's write queue.
- **Process replies**: This process receives the replies from the Nvme write queue. The replies indicate the status of the request. The status is stored if there is an error and the statistics timers are updated. The DataBuffer used is marked as available.
- **Process Nvme Requests**: This processes Nvme requests to read the data from the buffers.

### 6.6.2. NvmeWrite Trim

As well as writing the blocks of data to the Nvme, the NvmeWrite engine sends trim/deallocate requests to the Nvme drive to clear the usage state of blocks in the Nvme. This is used to reduce write latency and wear in the drive.

As soon as a DataChunk process is started the NvmeWrite process will send deallocate requests to the Nvme drive. Up to 4 deallocate requests are sent at a time and each one is set to deallocate up to 32768 Nvme blocks (512 Bytes) at a time. The deallocate requests are sent interleaved with the block write requests, but the first deallocate request is sent before the first block write.

## 6.7. Error handling

The system has basic error handing capability. There are a few ways to determine an error has occurred:

### 6.7.1. Errors During data capture

1. If the NVMe engine returns an error status in its Queued reply this is written into the error register of the appropriate NVMe. The NvmeStorage engine carries on capturing regardless. At the end of the data capture the host should read both error registers. An error status of 0 is Ok. The meaning of the error status can be found in the NVMe Express 1.3 protocol document.
2. If the NvmeStorage's NumBlocks register does not reach the designed number of blocks within the expected time, then a timeout of a block write has occurred.

### 6.7.2. Errors during read

At the moment the NvmeStorage module ignores any errors during read. However the host will be able to determine that an error occurred if a read block is missing or it times out reading all of the blocks it requested.

### 6.7.3. Errors During host NVMe access

When the host is communicating directly with the NVMe devices it is sending and receiving PCIe packets. The following errors can occur:

1. If the host is expecting a PCIe reply packet and it does not appear within an expected time, the host should timeout with a timeout error.
2. The host should inspect the status and error fields in the Xilinx style PCIe replies for any error status. Se the Xilinx PCIe Gen3 hard blocks documentation for information on these errors.
3. In the case of an NVMe queued reply, the error status field should be interrogated. An error status of 0 is Ok. The meaning of the error status can be found in the NVMe Express 1.3 protocol document.

## 7. NVMe Development and Test System

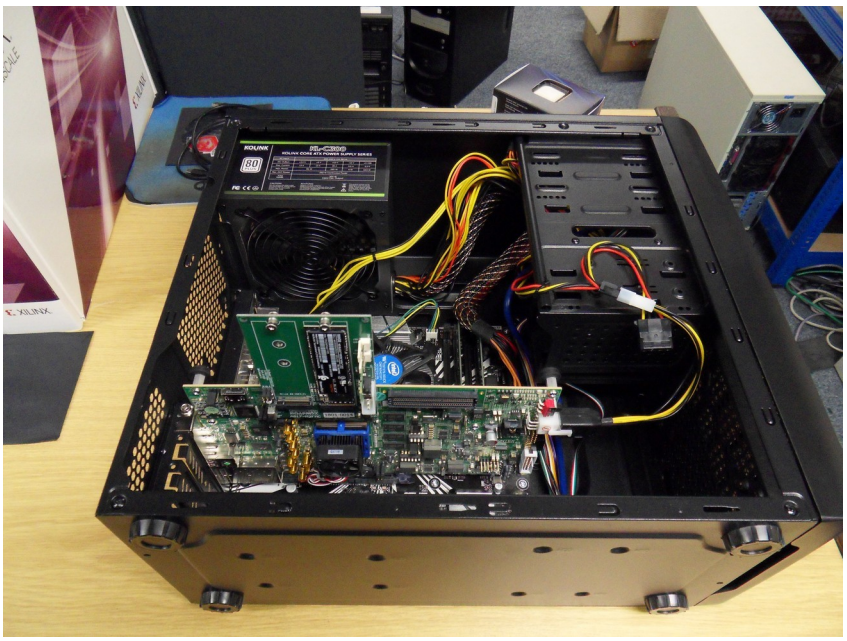
The test system comprises of:

- Xilinx evaluation board KCU105 with XCKU040-2FFVA1156E FPGA. All switches and jumpers to default except SW15.1 was set to on so the Xilinx FPGA would only be configured over the USB JTAG lead. This was powered from one of the PC's 4-pin disk power connectors using the Xilinx supplied adapter power lead. NOTE the standard PC 6-pin PCI power connector is NOT compatible its pin-out is different and will likely damage the FPGA board !!!
- Xilinx NVMe adapter board. Either an AB17-M2FMC or Ospero OP47, with two NVMe M2 locations. For testing two NVMe drives were installed. If one was used, on the AB17-M2FMC, it was installed into the Drive1 connector nearest the auxiliary 12V power connector. The AB17-M2FMC board was set as defaults, then the 4pin jumper was set: 1-2 and 3-4 so that the FPGA generates NVMe resets. The Ospero OP47 board has no configurable jumpers/switches.
- 6 x 500 GByte NVMe devices (2 installed at a time in final system each with around 2000 MBytes/s write speed). (Each set of two is from a different vendor and and has a different NVMe controller type).


These were installed into a simple PC unit built using the following components:

- Motherboard: Asus PRIME Z390M-PLUS LGA 1151 DDR4 mATX
- CPU: Intel Core i5 9400F 6 Core 2.9 GHz Processor
- RAM: 2 x Crucial 8GB DDR4-2666 UDIMM
- DISK: WD Blue SN550 250GB NVME M.2 2280 PCIe Gen3 SSD
- Power: Kolink Core Series 300W 80 Plus Certified Power Supply

- Case: AvP Viper Mini Tower Black USB 3.0 case



## 7.1. System Software

We used the KDE spin of Linux Fedora31 as the host OS setup with all of the packages that BEAM normally uses. The system had all updates applied to the date of testing. The Fedora system needs some of the development packages installed for GCC, G++ make etc. The “installPackages” make target in the test software Makefile should install all of the main software packages required (“dnf install @development-tools gcc-c++ kernel-devel”). We can supply a shell script to install all packages that our test system has if needed.

The Xilinx Vivado 2019.2 toolset should also be installed or at least some way of programming the KCU105 with a bit file should be provided.

## 7.2. KCU105 Setup

This is configured as per standard factory defaults (All jumpers, switches etc) except SW15.1 was set to on so the Xilinx FPGA would only be configured over the USB JTAG lead.  It is plugged into a PCIe 16 lane graphics slot on the PC motherboard. Power is supplied using the PC 4 way disk power connector to KCU105 adapter lead (NOT directly from a PC’s 6 way PCI power connector as it has a different pin-out!!).

The FMC VADJ1V8 voltage was set to 1.8V using the KCU105 System Controller v1.0 system via the serial port interface.

## 7.3. AB17-M2FMC Setup

This is configured as per standard factory defaults (All jumpers, switches etc). For testing one NVMe unit was installed into the Drive1 connector nearest the auxiliary 12V power connector. The board was set as

defaults, then the 4pin jumper was set: 1-2 and 3-4 so the FPGA generates NVMe resets. Power for the unit comes from the KCU105 but an extra 4 way PC disk power lead can be connected as well although we have not found this necessary for 2 x NVMe usage.

The board should be plugged into the KCU105's HPC connector once the NVMe's have been installed on it.

## 7.4. Ospero OP47 Setup

There is no configuration on this board. It should be plugged into the KCU105's HPC connector once the NVMe's have been installed on it.

## 7.5. Test FPGA resources

The test environment uses the following FPGA resources on the KCU105 and AB17-M2FMC boards:

| Item | Value | Notes |
|---|---|---|
| Host PCIe Gen3 block | PCIE_3_1_X0Y0 | Host PCIe interface |
| Host PCIe Clk | MGTREFCLK0P_225_AB6<br>MGTREFCLK0P_225_AB6 | Host PCIe 100 MHz clock |
| Host PCIe Lanes | MGTHTXP0_225_AH6<br>MGTHTXN0_225_AH5<br>MGTHRXP0_225_AH2<br>MGTHRXN0_225_AH1<br>MGTHTXP1_225_AG4<br>MGTHTXN1_225_AG3<br>MGTHRXP1_225_AF2<br>MGTHRXN1_225_AF1<br>MGTHTXP2_225_AE4<br>MGTHTXN2_225_AE3<br>MGTHRXP2_225_AD2<br>MGTHRXN2_225_AD1<br>MGTHTXP3_225_AC4<br>MGTHTXN3_225_AC3<br>MGTHRXP3_225_AB2<br>MGTHRXN3_225_AB1 | PCIe 4 lanes |
| | | |
| NVMe0 PCIe block | PCIE_3_1_X0Y2 | NVME CN2 |
| NVMe0 PCIe Clk | MGTREFCLK0P_228_K6<br>MGTREFCLK0N_228_K5 | FMC HPC Connector J22, D4,D5 |
| NVMe0 PCIe Reset | FMC_HPC_LA00_CC_P, H11 | FMC HPC Connector J22, G6 |
| NVMe0 PCIe Lanes | MGTHTXP0_228_F6<br>MGTHTXN0_228_F5 | R0: C6,C7, R1: A2,A3, R2: A6,A7, R3: A10,A11<br>T0: C2,C3, T1: A22,A23, T2: A26,A27, T3: |

| | | |
|---|---|---|
| | MGTHRXP0_228_E4<br>MGTHRXN0_228_E3<br>MGTHTXP1_228_D6<br>MGTHTXN1_228_D5<br>MGTHRXP1_228_D2<br>MGTHRXN1_228_D1<br>MGTHTXP2_228_C4<br>MGTHTXN2_228_C3<br>MGTHRXP2_228_B2<br>MGTHRXN2_228_B1<br>MGTHTXP3_228_B6<br>MGTHTXN3_228_B5<br>MGTHRXP3_228_A4<br>MGTHRXN3_228_A3 | A30,A31 |
| | | |
| NVMe1 PCIe block | PCIE_3_1_X0Y1 | NVME CN3 |
| NVMe1 PCIe Clk | | FMC HPC Connector J22, D4,D5 |
| NVMe1 PCIe Lanes | MGTHTXP0_227_N4<br>MGTHTXN0_227_N3<br>MGTHRXP0_227_M2<br>MGTHRXN0_227_M1<br><br>MGTHTXP2_227_J4<br>MGTHTXN2_227_J3<br>MGTHRXP2_227_H2<br>MGTHRXN2_227_H1<br><br>MGTHTXP1_227_L4<br>MGTHTXN1_227_L3<br>MGTHRXP1_227_K2<br>MGTHRXN1_227_K1<br><br>MGTHTXP3_227_G4<br>MGTHTXN3_227_G3<br>MGTHRXP3_227_F2<br>MGTHRXN3_227_F1 | R0: A14,A15, R1: A18,A19, R2: B16,B17, R3: B12,B13<br>T0: A34,A35, T1: A38,A39, T2: B36,B37, T3: B32,B33<br>Note that lanes 1 and 2 are swapped relative to normal MGT lane numbering on the KCU105 to the FMC. The Vivado Pcie gen3 wizard creates the lanes in normal MGT order and this needs to be overridden in the constraints file. |

The test environment uses the following FPGA resources on the KCU105 and Ospero OP47 boards:

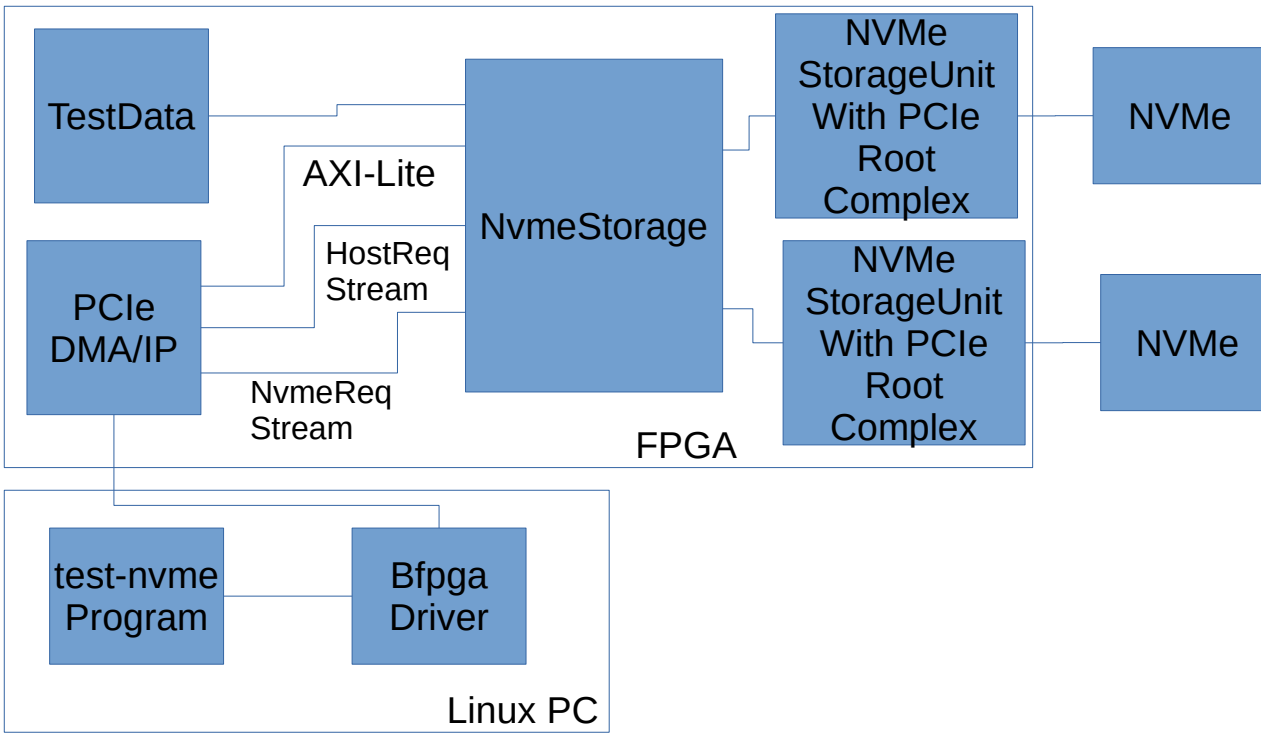
| Item | Value | Notes |
|---|---|---|
| Host PCIe Gen3 block | PCIE_3_1_X0Y0 | Host PCIe interface |
| Host PCIe Clk | MGTREFCLK0P_225_AB6<br>MGTREFCLK0P_225_AB6 | Host PCIe 100 MHz clock |
| Host PCIe Lanes | MGTHTXP0_225_AH6 | PCIe 4 lanes |

| | | |
|---|---|---|
| | MGTHTXN0_225_AH5<br>MGTHRXP0_225_AH2<br>MGTHRXN0_225_AH1<br>MGTHTXP1_225_AG4<br>MGTHTXN1_225_AG3<br>MGTHRXP1_225_AF2<br>MGTHRXN1_225_AF1<br>MGTHTXP2_225_AE4<br>MGTHTXN2_225_AE3<br>MGTHRXP2_225_AD2<br>MGTHRXN2_225_AD1<br>MGTHTXP3_225_AC4<br>MGTHTXN3_225_AC3<br>MGTHRXP3_225_AB2<br>MGTHRXN3_225_AB1 | |
| | | |
| NVMe0 PCIe block | PCIE_3_1_X0Y2 | NVME CN2 |
| NVMe0 PCIe Clk | MGTREFCLK0P_228_K6<br>MGTREFCLK0N_228_K5 | FMC HPC Connector J1D, D4,D5 |
| NVMe0 PCIe Reset | FMC_HPC_LA00_CC_P, H11 | FMC HPC Connector J1G, G6 |
| NVMe0 PCIe Lanes | MGTHTXP0_228_F6<br>MGTHTXN0_228_F5<br>MGTHRXP0_228_E4<br>MGTHRXN0_228_E3<br>MGTHTXP1_228_D6<br>MGTHTXN1_228_D5<br>MGTHRXP1_228_D2<br>MGTHRXN1_228_D1<br>MGTHTXP2_228_C4<br>MGTHTXN2_228_C3<br>MGTHRXP2_228_B2<br>MGTHRXN2_228_B1<br>MGTHTXP3_228_B6<br>MGTHTXN3_228_B5<br>MGTHRXP3_228_A4<br>MGTHRXN3_228_A3 | R0: C6,C7, R1: A2,A3, R2: A6,A7, R3: A10,A11<br>T0: C2,C3, T1: A22,A23, T2: A26,A27, T3: A30,A31 |
| | | |
| NVMe1 PCIe block | PCIE_3_1_X0Y1 | NVME CN3 |
| NVMe1 PCIe Clk | MGTREFCLK1P_228_H6<br>MGTREFCLK1N_228_H5 | FMC HPC Connector J1B, B20,B21 |
| NVMe0 PCIe Reset | FMC_HPC_LA04_P, L12 | FMC HPC Connector J1H, H10 |
| NVMe1 PCIe Lanes | MGTHTXP0_227_N4<br>MGTHTXN0_227_N3<br>MGTHRXP0_227_M2 | R0: A14,A15, R1: A18,A19, R2: B16,B17, R3: B12,B13<br>T0: A34,A35, T1: A38,A39, T2: B36,B37, T3: |

| | | |
|---|---|---|
| | MGTHRXN0_227_M1<br><br>MGTHTXP2_227_J4<br>MGTHTXN2_227_J3<br>MGTHRXP2_227_H2<br>MGTHRXN2_227_H1<br><br>MGTHTXP1_227_L4<br>MGTHTXN1_227_L3<br>MGTHRXP1_227_K2<br>MGTHRXN1_227_K1<br><br>MGTHTXP3_227_G4<br>MGTHTXN3_227_G3<br>MGTHRXP3_227_F2<br>MGTHRXN3_227_F1 | B32,B33<br>Note that lanes 1 and 2 are swapped relative to normal MGT lane numbering on the KCU105 to the FMC. The Vivado Pcie gen3 wizard creates the lanes in normal MGT order and this needs to be overridden in the constraints file. |

# 8. NVMe Experimentation

We setup a simple test system to allow us to communicate with the FPGA connected NVMe devices to allow us to easily experiment with the protocols to configure and access the NVMe. The DuneNvme test system provides this ability. In this source code tree there is a simple FPGA firmware design that allows




the NVMe to be accessed from a Linux program over the host's PCIe bus. The FPGA firmware uses the following components:

1. **PCIe DMA/IP:** This is the standard Xilinx PCIe endpoint IP configured for an AXI-Lite bus and a bidirectional DMA stream. This is clocked from the 100 MHz PCIe connector and provides a 250 MHz user clock from this.
2. **TestData:** This simple module produces a test data stream comprising of an incrementing number in consecutive 32bit values over a 256 bit AXI4 stream.
3. **NvmeStorage:** This provides the top level of the NvmeStorage FPGA firmware It implements a set of registers driven from the AXI4-lite bus and de-multiplexes/multiplexes single bidirectional DMA stream through to the two AXI4 streams used for direct communication with the NVMe's
4. **NvmeStorageUnit:** This implements a complete NVMe management engine for a single NVMe device. It has a PCIe root complex that the external NVMe device is connected to. It uses 4 PCIe lanes at Gen3 speeds. This is clocked from the 100 MHz NVMe carrier boards FMC connector and provides a 250 MHz user clock from this.
5. **NVMe device:** One of the example NVMe's was installed in the carrier board.
6. **Bfpga Driver:** This is a simple Linux driver of our own design. It is based on our driver/IpCore system but has been modified to work with the Xilinx XDMA IpCore. It supports memory mapped access to the AXI-lite bus and up to 4 x blocking, bi-directional DMA channels accessed via simple read/write calls. The Xilinux XDMA driver could be used, but we found this has issues with supporting the latest 5.x Linux kernels, possible bugs and was generally very complex for our requirements where we wanted to make sure other components of the system over the FPGA were not causing unintended issues.
7. **Test-nvme:** This is a simple test program that allows us to communicate with the NVMe drive over the FPGA fabric. It configures the NVMe's PCIe configuration registers, the NVMe registers and bus master queues and performs operations on them. It allows us to test the most simple method of NVMe configuration and access prior to implementing this on the FPGA.

## 8.1. Some points on the test system

- There are two clock sources used: The host's 100 MHz PCIe bus clock for the PCIe DMA/IP that provides a user clock for the AXI-lite bus and streams, and the NVMe card's 100 MHz PCIe bus clock that drives both of the NVMe PCIe Root Complexes and provides user clocks for the NvmeStorageUnit cores. The NvmeStorageUnit cores perform the necessary clock domain crossing logic for the data paths.
- The system reset is driven from the hosts PCIe connector. This reset is also used to reset the Nvme PCIe Root Complex and the external NVMe devices. This reset can be driven from the Linux host software.
- When the Linux host boots the FPGA PCIe design is likely not present. The Linux host has to be rebooted so the BIOS and Linux kernel allocates the PCIe resources correctly once the FPGA design has been loaded. Once done the FPGA PCIe design can re reloaded again and again. It might be possible to not do the reboot on some Motherboards.

- When re-loading the FPGA bit file you should:
  - Unload the bfpga driver. “rmmod bfpga”;
  - Program the FPGA over the USB-JTAG cable using Vivado.
  - Rescan the PCIe bus: “echo 1 > /sys/bus/pci/rescan”
  - Reset the PCIe device: “echo 1 > /sys/bus/pci/devices/0000:01:00.0/reset”
  - Re-install the bfpga driver: “insmod bfpga.ko”
  - The command “make driver_load” in the test software directory will perform these commands.
- The nvme-test program will communicate with the NvmeStorage core and NVMe using the bfpga driver. It accesses the NvmeStorage cores registers and sends PCIe transactions back and forth over the DMA links.

Basic tests include:

- Reload bfpga Linux device driver resetting FPGA: “make driver_load”
- Capture 20GBytes of data: “./test_nvme -d 2 -s 0 -n 5242880 capture”
- Read and validate some of this data: “./test_nvme -d 2 -s 0 -n 1000 read”
- Trim/Deallocate NVMe blocks: “./test_nvme -d 2 -s 0 -n 5242880 trim”

There is a test.sh BASH script that shows some example test commands.

The DuneNvmeStorageTestSoftware document describes the test_nvme program in more detail.

## 8.2. Notes on test system’s NVMe accesses

The main documentation for this is:

- NVM_Express_Revision_1.3.pdf
- pg156-ultrascale-pcie-gen3.pdf
- pg195-pcie-dma.pdf

1. The Xilinx PCIe 3.1 hard block adds a protocol layer above the PCIe TLP packets. This involves using different packet headers to the standard TLP packets and also it separates the host requests/completions from the NVMe requests/completions providing two bi-directional streams to deal with. It also then has tags and other signals to manage where the packets go and the byte/dword enables bits etc. All of this complicates things for us. A simple PCIe interface block that provided the raw PCIe TLP packets would have been much better and simpler like the Virtex-6’s implementation. It might be possible and better to use the Xilnx soft PHY IP core instead.
2. The Xilinx PCIe 3.1 hard block’s s_axis_rq_tuser signals need to provide the last_be signals at the start of the packet and they need to be set to “0000” when only one or no DWORD’s are sent. The NvmeStorage module peeks at the first 128 bits of the PCIe TLP packet to determine the numWords in the packet and sets this accordingly.

3. The maximum packet size appears to be 128 bytes in this system. This means there is a fair degree of overhead with the packet header size. It might be possible to increase this. The Xilinx PCIe hard block supports up to 1024 Byte packets. This needs to be looked at as the packet size should have been 512 bytes, but TLP header size may be within this 512 bytes. There is both the host's PCIe and the NVMe's interfaces to be looked at wrt to this. When a block write of 512 bytes is made this results in 4 separate packets of 128 bytes (data) each with 12 bytes of header (10% data overhead + 4 x write processing overhead).
4. The NVMe protocol is based around DMA reads and writes. However this can be extrapolated to simple command and data FIFO's.
5. When an error occurs, the error handling and reporting is not that good. Some commands return a status value, but often this is just a standard error value. Asynchronous error reporting is available but this requires an Asynchronous read command to be called prior to getting the error message. We have also found that often a command such as a write, will return a status of say 6 (internal error) but there is no asynchronous error response and other information as to what actually the error was.

## 8.3. NVMe Configuration and usage

From our testing, the simplest and most basic configuration and usages is as follows:

1. Most of the accesses are by an address. Although 64bit addresses are used we only use the lower 32bits and use the top 4 bit nibble (bits 28-31) to define where the data is to be read from/received from for simplicity. This could link to a memory address or a FIFO queue.
2. Write to the PCIe command register the value 0x06 to enable memory accesses and bus master accesses.
3. Setup the NVMe control register to set the IO queue request and reply entry sizes (64 and 16 bytes).
4. Setup an address to Admin Queue request fetches. Say 0x10000000.
5. Setup an address to Admin Queue reply storage. Say 0x20000000.
6. Start the controller running by setting bit 0 of the control register. Note that clearing bit 0 and setting it to 1 appears to do a soft reset of some things.
7. Setup one or more IO request/reply queues by sending commands via the admin request/reply queues. Their memory addresses can be set to 0x20000000, 0x30000000, ...
8. Optionally send a GetAsynchronousEvent message so that asynchronous events will be reported.
9. Optionally find the block size and number of blocks in namespace 1. Namespace 1 seems to be present on new NVMe devices.
10. When the NVMe device has sent something into its request queue it should send a MSI-X interrupt message.

11. When responding to memory read requests, if the required data is greater than 128 bytes (including packet headers?) then return multiple responses with 128 bytes in each. Note the usage of the nbytes parameter within the packet headers to accomplish this.
12. Queued requests are 64 bytes or 16 Dwords (32 bits) in size. Queued replies are 12 bytes or 3 Dwords in size.
13. The NVMe queues and data reads could be implemented using block ram organises as random accessed memory or simpler FIFO queues. The FIFO queues sound like the simplest solution as the data needs to be send in a stream anyway.
14. If we use the Xilinx PCIe 3 hard block we will have to handle/implement the packet headers that this system uses and multiplex across the two bidirectional streams. If we use a PCIe PHY soft core we can just send/receive PCIe TLP packets across a single bidirectional stream.

# 9. Linux Host Testing

## 9.1. Basic Tests

This is for inspecting NVMe drives installed in a Linux host. Install package nvme-cli for these commands. See the manual on “nvme” for more options and information.

| | |
|---|---|
| lspci -s 01:00.0 -vvvx | Show PCIe details of a specific NVMe device |
| nvme list | List NVMe devices |
| nvme list-subsys | List NVMe sub systems |
| nvme id-ctrl /dev/nvme0 | Identify controller |
| nvme id-ns /dev/nvme0n1 | Identify Namespace, display structure |
| nvme show-regs -H /dev/nvme0 | Display Nvme regs. Note this does not work with recent Linux kernels due to security changes. |
| Nvme raw data read command | nvme io-passthru /dev/nvme0 --opcode=0x02 --namespace-id=0x1 --data-len=512 --read --cdw10=0 --cdw11=0 --cdw12=0 -b > blk0 |

**PCIe Interface**

01:00.0 Non-Volatile memory controller: Samsung Electronics Co Ltd NVMe SSD Controller SM981/PM981/PM983 (prog-if 02 [NVM Express])
Subsystem: Samsung Electronics Co Ltd Device a801
Flags: bus master, fast devsel, latency 0, IRQ 16, NUMA node 0
Memory at f7c00000 (64-bit, non-prefetchable) [size=16K]
Capabilities: [40] Power Management version 3
Capabilities: [50] MSI: Enable- Count=1/1 Maskable- 64bit+
Capabilities: [70] Express Endpoint, MSI 00
Capabilities: [b0] MSI-X: Enable+ Count=33 Masked-
Capabilities: [100] Advanced Error Reporting

Capabilities: [148] Device Serial Number 00-00-00-00-00-00-00-00
Capabilities: [158] Power Budgeting <?>
Capabilities: [168] Secondary PCI Express <?>
Capabilities: [188] Latency Tolerance Reporting
Capabilities: [190] L1 PM Substates
Kernel driver in use: nvme
Kernel modules: nvme

# 10. Notes

- The addresses used and request types indicate the types of transaction required. BlockRAM sizes are: 1k x 36 so for 128 bits wide would need a minimum of 4 blocks which is a minimum of 16 kBytes. If single write/read ports are used we could use just 2 x blocks. The KU040 FPGA we are using for test has 600 such 36kbit blocks. We will need to experiment with the DataBuffer's size. 32k (8 BlockRam's) would allow 8 concurrent writes of 4 kBytes.

# 11. Ideas for Future Work

This lists some of the additional work and/or changes that we think could be done.

Main Items:

1. Instead of an AXI4-lite register interface, implement a separate NvmeStorage module with a register interface that would be easier to connect to a Wishbone bus.
2. Improve host data read performance by collating the 128 Byte PCIe packets into a larger packet (4k or 8k) when returning data to the host.
3. Provide a better arbitration between data capture and read data performance. At the moment read access is held while a data capture is in progress.
4. Experiment with the DataSet Management Command to deallocate blocks and set performance parameters.

Possible Future Items:

1. Configuring the NVMe's for a 4k block size (rather than the default 512 Bytes) may yield some performance improvements. The benefits would likely come from within the NVMe's themselves rather than in the NvmeStorage logic.
2. Increasing the number of blocks, actually overall data size, sent in each request to the NVMe's might help performance, again from within the NVMe's.
3. Increasing the PCIe packet size from 128 to 256 Bytes. This might have a slight effect, but note one of the NVMe's we have has a maximum PCIe packet size of 256 so anything larger is unlikely to work.
4. Manage the NVMe device parameters so they are not hard coded. These are the NvmeBlockSize, NvmeTotalBlocks and NvmeRegStride parameters. The simplest way would be to use registers for

these values and have the host CPU to interrogate the NVMe's and configure these. It would also be possible, using additional logic resources, to get the NvmeStorage firmware to do this instead during the configuration phase.

5. Modify the code to use 256 bit width streams rather than 128 bit streams so that it would be directly compatible with Gen4 PCIe data rates allowing a single NVMe to be used when ones off sufficient performance are available. This would entail modifying the various state machines to use this data width and would use additional FPGA resources.
6. Create a simple test design for running just one NVMe device.
7. The host computer currently determines that a data capture or read operation has finished by polling the status and number of blocks registers. It would be possible to send an interrupt to the host when this happened or some other status changes. This would require the logic external to the NvmeStorage block to generate the necessary PCIe interrupt.
8. If the NvmeWrite module has a timeout when sending a block to the Nvme (does not get a reply from the Nvme) then it carries on but has one less block write queue slot and will complete with the NumBlocks register set to one less than required. If more than one block request does not get a reply eventually the engine will slow down and stop. We could add support for timeouts on replies. Note we have not seen this happen so far during testing.
9. More error handling and internal timeout support could be added.
10. We have only supplied test software for the system. A proper NvmeStorage access software API library for the host could be produced for production usage. This host API library could provide NVMe statistics information such as SMART data etc.