BEAM

DuneNvmeTest

Dune NVME FPGA Tests

Project	DuneNvme
Date	2020-04-12
Reference	DuneNvmeTest
Author	Dr Terry Barnaby

1. Introduction

This directory contains a simple NVMe test environment that allows experimentation with the low level PCIe NVMe interfaces as available on a Xilinx FPGA environment.

The directory contains the source code, simulation environment and build environment for the Nvme test FPGA firmware as well as the nvme_test host software.

2. Directories

src	The main VHDL source code
ip-core	IP cores generated with Vivado
sim	The simulation environment
vivado	The build environment
test	Host test programs accessing the FPGA firmware
docsrc	Source for the documentation
doc	Output for the documentation

3. Test Program

The test Linux host program is in the test directory and is called nvme_test. This program communicates with an NVMe device through the FPGA connected to the host machines PCIe bus. This program allows experimentation with the NVMe low level PCIe interface.

4. Building the FPGA bit file and programming

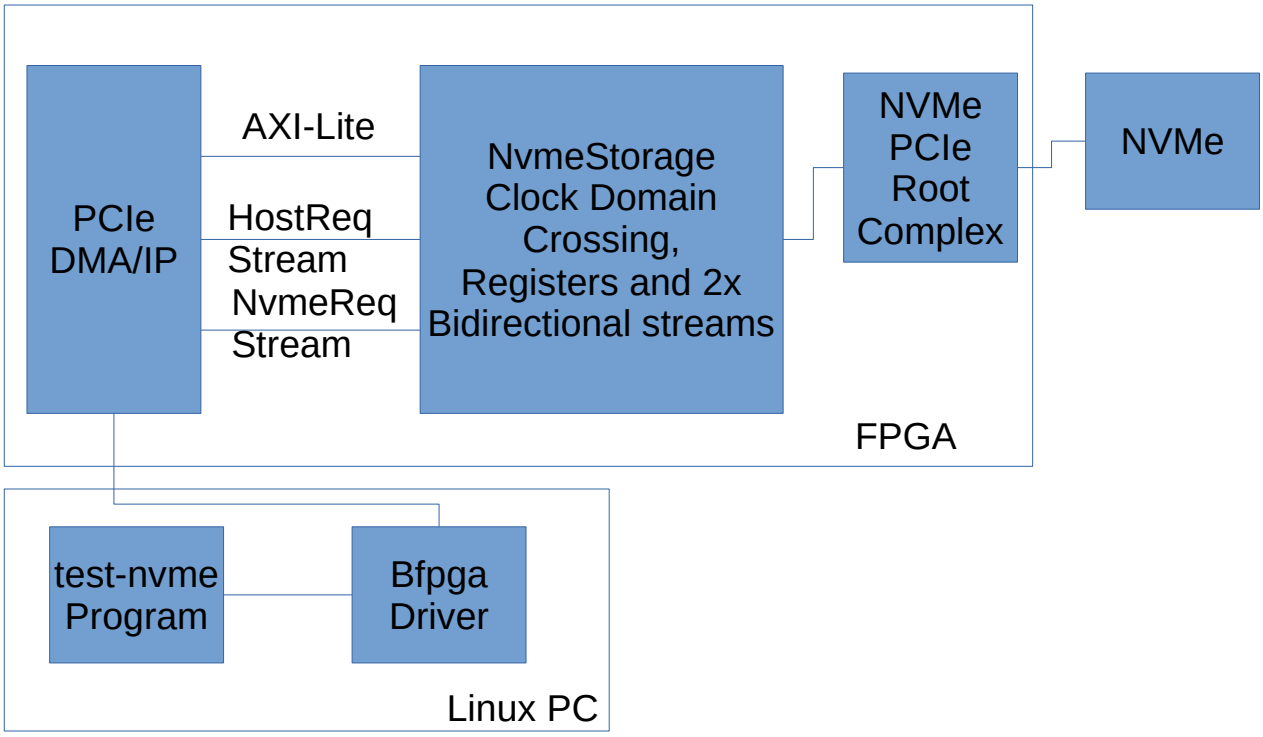
1. cd vivado
2. make clean
3. make all
4. make program

5. To Simulate FPGA

1. cd sim
2. Edit Makefile and testbench files for simulation required
3. make
4. make view

6. NVMe Experimentation

We setup a simple test system to allow us to communicate with the FPGA connected NVMe device to allow us to easily experiment with the protocols to configure and access the NVMe. This test system is in the DuneNvmeStorageTest software directory. In here there is a simple FPGA firmware design that allows



the NVMe to be accessed from a Linux program over the hosts PCIe bus. The FPGA firmware has been designed to mimic the final system to a degree. It 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 user clock from this.
2. **NvmeStorage:** This simple core has an API similar to the final systems **NvmeStorage** core. It implements a simple set of test registers driven from the AXI-lite bus and de-multiplexes/multiplexes single bidirectional DMA stream through to the two Axi streams used by the NVMe PCIe root complex. It performs the necessary clock domain crossings to do this.
3. **Nvme PCIe Root Complex:** This implements a PCIe root complex that the external NVMe device is connected to. It only uses 4 PCIe lanes at a relatively slow speed to simplify the system.

This is clocked from the 100 MHz NVMe carrier boards FMC connector and provides a user clock from this.

4. **NVMe device:** One of the example NVMe's was installed in the carrier board.
5. **Bfpga Driver:** This is a simple Linux driver of our own design. It is based on our driber/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.
6. **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.

6.1. 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 the NVMe PCIe Root Complex and provides a user clock for the NvmeStorage core. The NvmeStorage core performs the necessary clock domain crossing logic for the data paths.
- The system reset is driven from one of two sources. The hosts PCIe connector and a timed pulse from FPGA design start. 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.
- When the Linux host boots the FPGA PCIe design is not present. The Linux host has to be rebooted so the BIOS and Linux kernel allocates the PCIe resources correctly. 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:
  - Program the FPGA over the USB-JTAG cable using Vivado.
  - Unload the bfpga driver. "rmmod bfpga";
  - 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 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.

6.2. 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.

6.3. 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.