Design and Implementation of NVM Express Controller

A Project Report

submitted by

MANTHA SHANMUKH ABHISHEK

in partial fulfilment of the requirements
for the award of the degree of

MASTER OF TECHNOLOGY

DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY, MADRAS.

May 2013

THESIS CERTIFICATE

This is to certify that the thesis entitled **Design and Implementation of NVM Express Controller**, submitted by **Mantha Shanmukh Abhishek**, to the Indian Institute of Technology, Madras, for the award of the degree of **Master of Technology**, is a bonafide record of the research work carried out by him under my supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Dr. V. Kamakoti
Research Guide
Professor
Dept. of Computer Science and Engineering
IIT-Madras, 600 036

Place: Chennai

Date:

ACKNOWLEDGEMENTS

The successful completion of this project work would not have been possible without the guidance, support and encouragement of many people. I take this opportunity to express my sincere gratitude and appreciation to them.

First of all I would like to express deepest sense of gratitude to my guide, **Dr. V. Kamakoti** whose energy, enthusiasm and constant support inspired me from the beginning till the end of my project.

I would like to express my sincere gratitude and thanks to Mr. G.S. Madhusudan who has provided me with lots of ideas and suggestions throughout the project. I am very much thankful for the valuable inputs that he has provided to the project, through the Industry experts.

I express my sincere gratitude to my fellow project mates Bhasker, Rahamthula, Chidhambaranathan, Suresh, Vikas, Debi , Avinash, Naveen and Neel who have been very supportive. I especially thank Avinash for his cooperation in designing the interface between the NVMe controller and his Nand Flash Controller module. This has led to a smooth integration of all the blocks in the NVM Subsystem.

I would like to thank my classmates who made my life in IIT a memorable experience.

Finally, I take this opportunity to thank my parents for their support and encouragement without which learning in and becoming a part of such a prestigious institution would not have been possible.

ABSTRACT

KEYWORDS: HDD, SSD, PCI Express, NVM Express, Nand Flash Controller

The use of SSDs with traditional disk-based I/O subsystem causes unnecessary latencies and translations in the Read Write commands. In order to completely utilize the performance of SSDs a Non Volatile Memory Subsystem was designed, based on the NVM Express Specification. The communication to this I/O subsystem is through PCI Express interface and the command set is based on NVMe 1.0c Specification. The designed subsystem typically consists of PCIe Core, PCIe controller , NVMe controller, NAND Flash Controller and several NAND Chips. The present project deals with the design and implementation of PCIe controller and the NVMe controller. The PCIe controller was designed as a generic bridge between any PCIe device and the PCIe Core. The NVMe controller was designed as a PCIe device which implements the NVMe Specification.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

NVM	Non Volatile Memory
NVMe	Non Volatile Memory Express
PCIe	Peripheral Component Interconnect Express
SATA	Serial Advanced Technology Attachment
AHCI	Advanced Host Controller Interface
SQ	Submission Queue
CQ	Completion Queue
ASQ	Admin Submission Queue
ISQ	I/O Submission Queue
ACQ	Admin Completion Queue
ICQ	I/O Completion Queue
MSI	Message Signalled Interrupt
ONFI	Open Nand Flash Interface
TLP	Transaction Level Packet
BAR	Base Address Register
SSD	Solid State Drives
MMIO	Memory Mapped Input Output
HBA	Host Bus Adaptor
FTL	Flash Translation Leyer

CHAPTER 1

Introduction

1.1 Overview of Processor Architecture

The overview of the processor architecture is shown in the figure 1.1

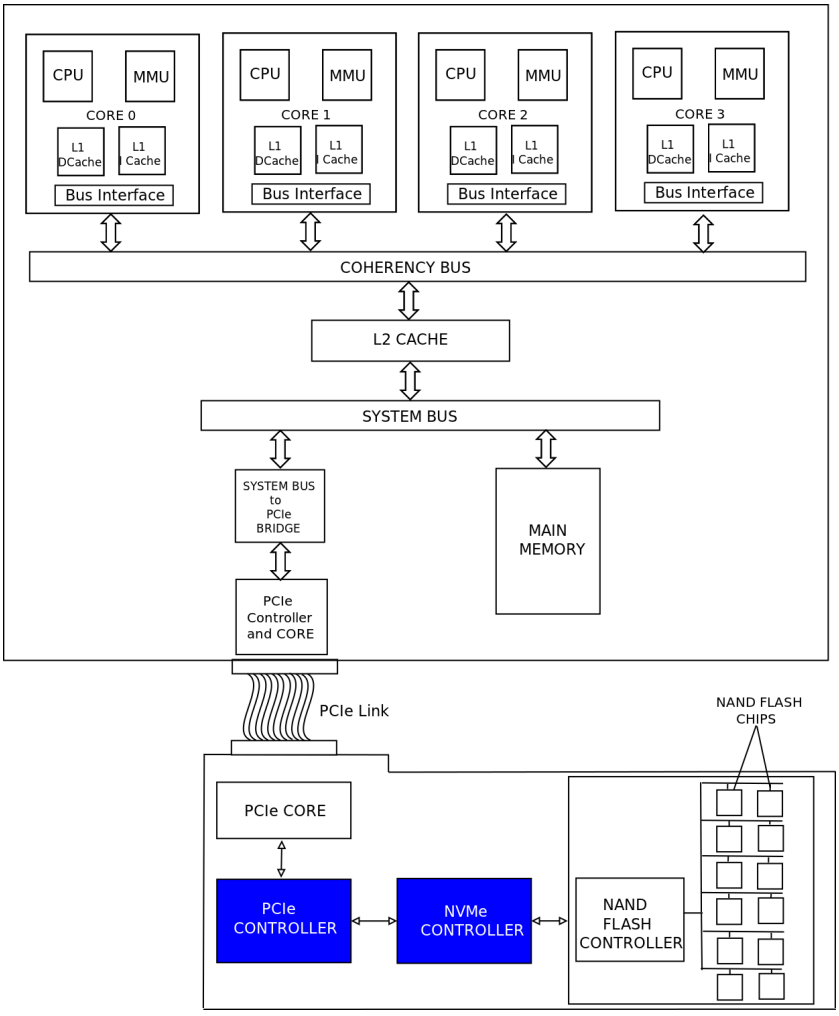

Figure 1.1: Overview of Processor Architecture

Overview of Processor

The processor design team in the Reconfigurable and Intelligent Systems Engineering (RISE) lab has been actively involved in the design of a high-end in-house processor for defence and security related applications. The figure 1.1 shows an overall view of the targeted processor. The processor system has a quad-core architecture, with a support for two levels of cache hierarchy and a single on-chip DRAM. It implements cache coherency at L1 cache level, with coherency bus. The CPU core is based on 64 bit PowerPC Instruction Set Architecture. It supports "dual-issue" and "out of order execution". A Memory Management Unit was designed for an efficient data transfer between the Processor Core and the Main Memory (DRAM). It supports NVM Express based I/O Subsystem with a PCI Express interface. This I/O subsystem typically implements the "File System" for the processor.

My Contribution

In order to store large amounts of data, an NVM Express based I/O subsystem was designed, which functions as a back end for On-chip DRAM.PCI Express has been used as the interface between the processor and the NVM Subsystem. The highlighted portions in the figure 1.1 shows my contribution to the NVM Subsystem.

1.2 Overview of contents

Chapter 2 explains the motivation for designing an NVM Express based I/O subsystem, with a PCIe interface to it. It explains the overall architecture of NVM subsystem and concludes with a brief description of the language used to code the design.

Chapter 3 describes the design and implementation of PCIe controller.

Chapter 4 gives a brief introduction to NVM Express 1.0c specification.

Chapter 5 includes the design and implementation of NVM Express controller.

Chapter 6 concludes with a short description of the future prospects of the designed subsystem.

Appendix A provides little more insight into the proposed modifications to the NVM Subsystem, with the inclusion of FTL unit.

CHAPTER 2

Background

2.1 PCIe as an SSD Interface

In the Enterprise and Client Systems, PCIe based SSDs are emerging as a back-end for DRAMS. The main factors for the wide-spread adoption of PCIe as an interface for SSDs are given below :

PCIe is High Performance

1. It is a full-duplex system , which can support multiple outstanding requests, and out of order processing.
2. It has a scalable port width ranging from single lane (x1) to sixteen lane (x16).
3. It also has a scalable link speed. Links speeds include 2.5GTps , 5GTps ,8GTps .
4. It is a direct attach to CPU subsystem hence, it has no HBA latency.

PCIe is Low Cost

1. It is a high volume commodity interconnect.
2. It has less number of pins. Hence lower area and hence lower cost.
3. It is a direct attach to CPU subsystem hence, it eliminates HBA cost.

PCIe also provides effective Power Management

1. Direct attachment to CPU subsystem eliminates HBA power.
2. It implements power budgeting and Dynamic Power Allocation [1].

2.2 Need For NVM Express

Standardized Interface

- A standardized interface is need for the easy adoption of PCIe based SSDs. NVM express specification provides this standard interface for PCIe SSDs.

Scalable

- NVM Express is scalable host controller interface standard, which is designed for Enterprise and Client systems that use PCI express SSDs.

Efficient Command Set

- It provides a simple, streamlined and efficient command set which eliminates the legacy HDD to SSD command conversion overhead.

Optimized Register Interface

- It provides an optimized register interface, that allows the host software to communicate with the non volatile memory subsystem.

Industry Support

- It was developed by industry consortium of 80+ members and hence enjoys a very wide industry support.

2.3 Key Attributes of NVM Express Controller

1. It does not require uncacheable/MIMO register reads in the command issue or command completion path. Each such access would otherwise take 2000 clock cycles.
2. A maximum of one MMIO register write is necessary in the command issue path.
3. Support for up to 64K I/O queues, with each queue supporting up to 64K commands.
4. Simplified command decoding and processing with fixed size(64B) command format.
5. All information to complete a 4KB read request is included in the 64B command itself, ensuring efficient small I/O operation.
6. Support for 2k MSI-X interrupts or 32 multiple message MSI.
7. Support for simple and efficient Interrupt Aggregation.

2.4 Non Volatile Memory Sub-system

The present developed Non Volatile Memory Sub-system consists of a PCIe Core , PCIe controller, NVMe Controller and Nand Flash Controller to access the NAND Flash Chips. The Sub-system is shown in the figure 2.1

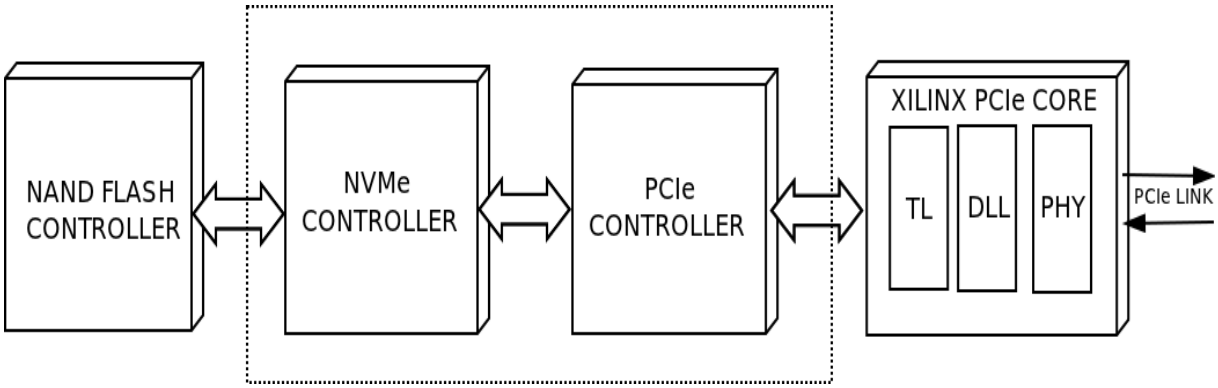

Figure 2.1: NVM Subsystem

The Xilinx 1-lane Integrated Endpoint Block was chosen as the PCI Express Core Architecture. It provides a user interface of width 32 bits. Link speeds of up to 2.5Gb/s is

supported. The core provides PCIe Express Base Specification compliant with v1.1 . The detail es about the core can be found in the Endpoint Block User Guide [2]

A PCIe controller was developed as a generic interface between any PCIe device and the Core. It also provides a device interface of width 32 bits.

The NVMe controller is implemented as a PCIe device. It provides a 32 bit interface on both PCIe side and Nand Flash Controller side. The controller is compliant to NVM Express 1.0c Specification.

An NVMe compliant Nand Flash Controller was developed in order to process the NAND Flash Commands. It provides a 32 bit interface on NVMe side and supports 16 bit interface to NAND Flash chips. Specifically Micron's MT29F2G16A Nand Flash chips are targeted.

2.5 BlueSpec System Verilog Language Background

BlueSpec System Verilog is a Hardware centric language based on industry standard SystemVerilog and Verilog [3]. It uses ***Rules*** and ***interface methods*** for behavioral description, which adds a powerful way to express complex concurrency and control :

- Across multiple shared resources.
- Across module boundaries.

Parallelization

- Concurrent behavior is expressed implicitly.
- It has a traditional hardware semantic model of cooperating FSMs.

- Rules express concurrent operations by simply describing the conditions under which the state element(s) are updated.
- Rules are implemented as unsequenced atomic transactions.
- Compiler introduces the scheduling and the muxing for the shared resources.

Level of Abstraction

- BSV provides significantly higher level of abstraction than Verilog, SystemVerilog, VHDL and SystemC.
- It provides Behavioral description through : Rules and Interface Methods.
- Various attributes for Structural description are:
  - High level abstraction types.
  - Powerful Static checking.
  - Powerful parametrization.
  - Powerful Static elaboration.

Standalone Function Libraries

BSV has a large set of function libraries, some of them are provided below.

- It has Data Containers such as FIFO, registers, BRAMs etc ..
- Circuits such LFSR and completion buffer [4].
- Interface types, GET, PUT transactors.
- Multiple Clock domain circuits such as synchronizers.
- Bus interfaces such as common data bus , Z-bus etc ..

CHAPTER 3

PCIe Controller Implementation

The PCIe Controller provides a medium for the PCIe Endpoint device (User Application) to communicate with the PCIe Core. The controller receives Transaction Level Packets(TLPs) from the Core, decodes them and sends required control and data signals to the Endpoint device. It receives control and data information from the device, forms TLPs and then sends to the Core. In this way it forms a bridge between any PCIe device and the PCIe Core. The Core side interface is designed to suit the interface of Xilinx PCIe Core. The device side interface is made generic for any endpoint device to use. In the present context the NVMe Controller is the endpoint device. The controller interfaces and module partitions are shown as a block diagram in the figure 3.1.

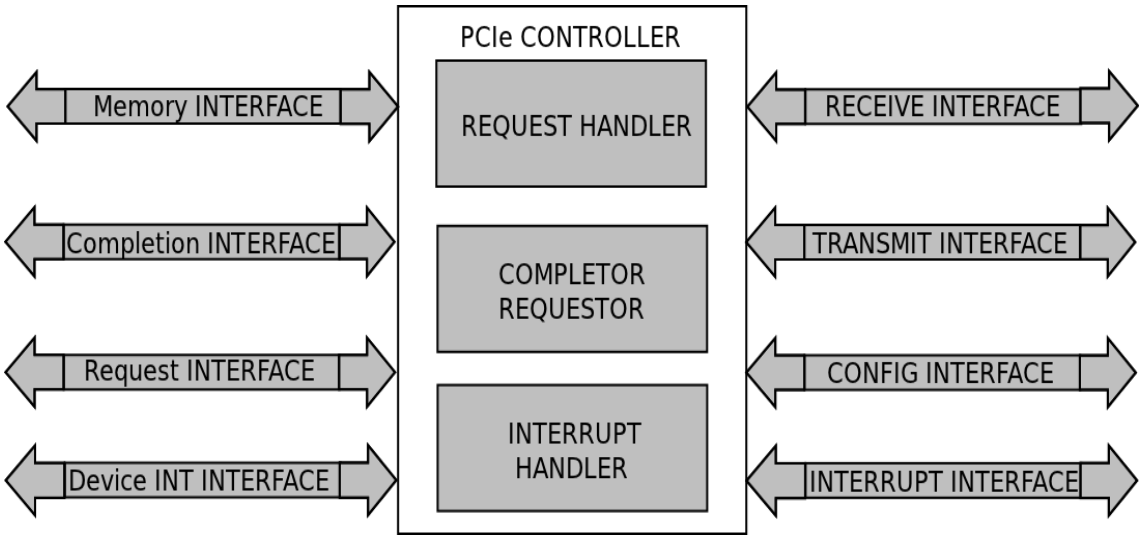

Figure 3.1: PCIe Modules and Interfaces

3.1 Interface Definitions

The controller provides several interfaces on the Core side to connect with the Xilinx PCIe core. It provides four interfaces on the device side to connect with the endpoint device. Core side and the device side interfaces are explained in this section.

3.1.1 Receive Interface

This interface provides a means for the controller to Receive TLPs from the Core's Transaction Layer. The Request Handler module utilizes this interface to get the TLPs from the core, process them and send the required data and control signals to the device. The device can implement as many as "seven" different memory mapped units with the help of the 7-bit Base Address Register. However, this controller is designed to support only one Memory unit. Receive interface signals are shown in the table 3.1.

Table 3.1: Receive Interface Signals

Signal	Direction	Description
wr_rx_tlast	In	Signals the end of packet. Valid only if rx_tvalid is also asserted.
wr_rx_tdata[31:0]	In	Packet data being received. Valid only if rx_tvalid
wr_rx_tvalid	In	Indicates that the core is presenting valid data on the rx_tdata[31:0]
rg_rx_tready	Out	Indicates that the device is ready to accept data on rx_tdata[31:0].
wr_bar_hit[6:0]	In	Indicates BAR(s) targeted by the current receive transaction. The designed controller supports only a single BAR. Hence, the valid value for this field is zero.

3.1.2 Transmit Interface

This interface provides a means for the controller to send the TLPs to the Core. The Completer-Requester module utilizes this interface to send the device requested Read Write or Completion TLPs to the core. The detailed description of the various signals in the interface is provided in the table 3.2.

Table 3.2: Transmit Interface Signals

Signal	Direction	Description
rg_tx_tdata[31:0]	Out	Packet data to be transmitted
rg_tx_tvalid	Out	Indicates that the controller is presenting Valid data on tx_tdata[31:0]
wr_tx_tready	In	Indicates that the core is ready to accept data. The simultaneous assertion of valid and ready, marks the successful transfer of one data beat on tdata[31:0].
rg_tx_tlast	Out	Signals the end of a packet. Valid only along with assertion of rg_tx_tvalid.
rg_src_dsc	Out	Source Discontinue Indicates that the device has discontinued the transaction
wr_tx_buf_av[5:0]	In	Indicates the number of transmit buffers available in the core. Each free transmit buffer can accommodate up to the supported Max Payload Size.
wr_tx_cfg_req	In	Configuration Request Asserted when the core is ready to transmit a Configuration Completion or other internally generated TLP.
rg_tx_cfg_gnt	Out	Configuration Grant Asserted by the device (controller) in respo-se to tx_cfg_req, to allow the core to transmit an internally generated TLP.

3.1.3 Configuration Interface

The PCIe controller uses the Configuration interface to inspect the state of the PCIe Configuration space. The controller provides a 10-bit configuration address, which selects one of the 1024 configuration space DWORD registers. The various signals used in this interface are defined in the table 3.3

Table 3.3: Configuration Interface Signals

Signal	Direction	Description
wr_cfg_do[31:0]	In	A 32-bit data output port used to obtain read data from the configuration space inside the core.
wr_rd_wt_done	In	Indicates a successful completion of the user configuration register access.
rg_cfg_dwaddr[9:0]	Out	Address port used to provide a configuration register DWord address during its access
rg_cfg_rd_en		Read enable for configuration register access
wr_cfg_bus_number b[7:0]	In	Provides assigned bus number for the device. Used in bus number field of out going TLPs.
wr_cfg_device_number b[7:0]	In	Provides assigned device number for the device. Used in device number field of out going TLPs.
wr_cfg_function_number b[7:0]	In	Provides assigned function number for the device. Used in function number field of out going TLPs.

3.1.4 Interrupt Interface - Core Side

The interface is used to access the core's interrupt related attributes. This interface is also a part of the Configuration interface, however, it has been separated from it for abstracting out the interrupt functionality. This interface also accesses the configuration space of the core. The interface signals are provided in the table 3.4.

Table 3.4: Interrupt Interface Signals - Core Side

Signal	Direction	Description
rg_cfg_interrupt	Out	Interrupt request signal to the Core. The controller asserts this to cause the selected interrupt mssg type to be transmitted by the core.
wr_cfg_interrupt_rdy	In	Interrupt grant signal from the Core.
rg_cfg_interrupt_do b[7:0]	Out	Lower five bits of the this data bus is used to transmit the interrupt vector number.
wr_cfg_interrupt_mmenable b[7:0]	In	Multiple Message Enable Signal. This indicates the number of messages the core can send as a part of multiple message MSI.
wr_cfg_msienable	In	Indicates that Message Signalling Interrupt (MSI) is enabled.

3.1.5 Completion data-to-device Interface

The TLPs received from the core are decoded by the Request-Handler module and the Completion-data payload is transmitted to the device through this interface. The description of the signals involved in the interface are provided in the table 3.5.

Table 3.5: Completion data-to-device Interface Signals

Signal	Direction	Description
rg_data_valid	Out	Indicates that the data transmitted on the data lines is valid
rg_data_out[31:0]	Out	Completion data received as payload is sent over this data bus
rg_last_DWord	Out	Indicates that this is the last Dword for the entire transaction. No more data with same tag
rg_tag_to_device[6:0]	Out	Tag value for this transaction

3.1.6 Memory Interface

The Memory interface provides a set of signals, for the Controller to access the Device Memory Space. The device typically implements its Register File (or Controller Registers) on this interface. The description of the signals involved in the interface are provided in the table 3.6

Table 3.6: Memory Interface Signals

Signal	Direction	Description
rg_address[31:0]	Out	Address of a Register in the Memory
rg_data_out[63:0]	Out	Data to be written into the Memory Can be used to access 32 bit register or a 64 bit register.
rg_byte_enable[3:0]	Out	4'b1111 indicates that all 64 bits on rg_data_out are valid 4'b0011 indicates that lower 32 bits on rg_data_out are valid
rg_read	Out	Memory Read operation
rg_write	Out	Memory Write Operation

3.1.7 Device-Request Interface

The Device-Request interface notifies the controller about the requests that are being made by the device. These requests trigger the Completer-Requester module in the controller to send Read, Write or Completion TLPs to the core. The interface also provides a 32-bit data bus for the device to send its data payload associated with the Write-TLP and Completion-TLP operation. The description of the signals involved in the interface are provided in the table 3.7.

Table 3.7: Device-Request Interface Signals

Signal	Direction	Description
wr_data_from_device[31:0]	In	Data bus to send data payload to pcie controller
wr_address_from_device[63:0]	In	Address bus to send address of the main memory location
wr_requested_tag[1:0]	In	Tag value requested for the Transaction
wr_payload_length[9:0]	In	Indicates the amount of payload being sent during write and completion and indicates the amount of payload requested during read operation
wr_send_completion_tlp	In	Request to send completion TLP
wr_send_write_tlp	In	Request to send write TLP
wr_send_read_tlp	In	Request to send read TLP
wr_64bit_address	In	If set it Indicates that all the 64 bits of the address lines are valid, else 32 bits are valid
wr_send_valid_data	Out	Indicates the device to send valid data payload. The device should send valid data on the data lines only if this signal is high
wr_data_valid	In	Indicates that the data on data lines is valid
wr_device_wait	In	Wait signalled by the device. Indicates the PCIe controller to wait
rg_wait	Out	Indicates the Device to wait. Device waits as long as this is asserted.

3.1.8 Interrupt Interface - Device Side

This interface enables the device to send MSI interrupt related information to the controller. The Interrupt Handler module then takes care of packing the messages (vector numbers) and sends them to Core. The description of the signals involved in the interface

are provided in the table 3.8.

Table 3.8: Interrupt Interface Signals - Device Side

Signal	Direction	Description
wr_interrupt_ready	In	Indicates that the interrupt vector number is Valid
wr_vector_number[4:0]	In	The value of the vector number for which the the interrupt is generated
wr_MSI_number[4:0]	In	Indicates the Device's request for number of messages in MSI interrupt

3.2 Modules Description

The complete functionality of the controller has been divided into three modules. The Request Handler takes care of the incoming requests from the Host-PCIe controller. The Completer-Requester takes the requests and completions from the device and sends them to the Host-PCIe controller. Interrupt Handler sends interrupt messages to core. The detailed implementation of each module is given in the following sub-sections.

3.2.1 Completer-Requester

This module takes the Read, Write, or Completion TLP requests from the device. It forms the required TLP and sends them over the transmit interface to the PCIe Core. Completion TLP and the Write TLP use the data supplied from the device to form the data payload, after the TLP headers. The details of how each request is handled is described below.

Completion TLP Request

The completion TLPs have three Dwords in header. The third DW is used to uniquely identifies the "Requester" of this transaction. As it was mentioned earlier, the Requester ID and the TAG are used to identify the "Requester". The Request Handler provides the information of the Requester by triggering an update signal to this module. This causes the DW 3 to get updated. The other two DWords are updated when the device requests for completion TLP. After updating the DWords, they are sent along with the data payload just as write TLPs. This process is shown as a flow diagram in the figure 3.2.

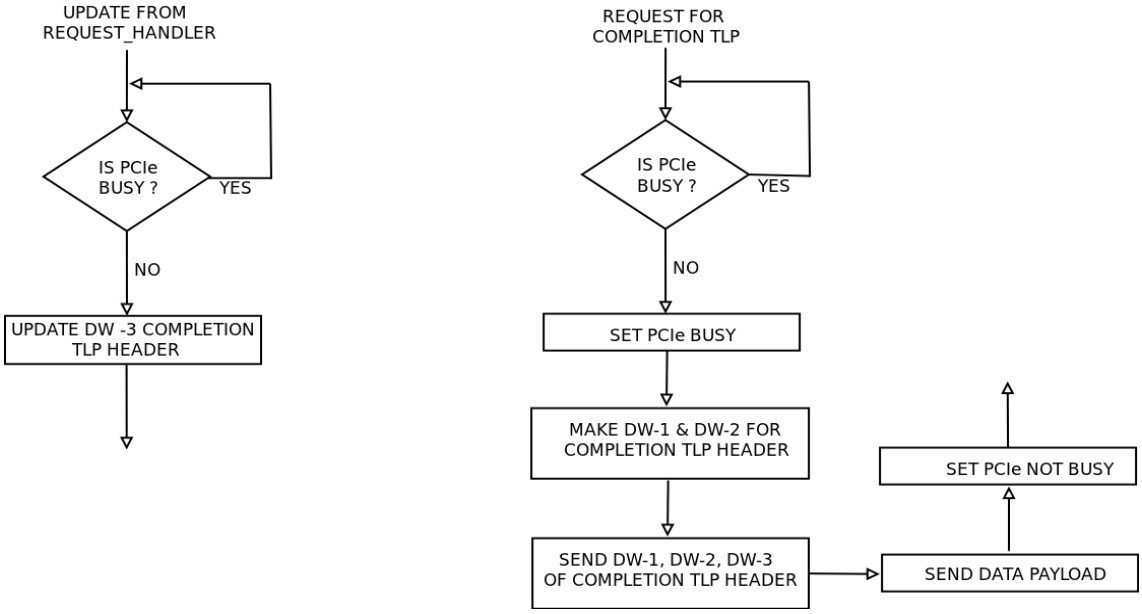

Figure 3.2: Completion TLP Flow

Read TLP Request

The controller starts to process the read TLP request, if it is not busy with any other TLP request processing. It first makes the DWord 1 and DWord 2 of the Read TLP header. The DWord 1 contains the information about the header type, and the requested payload size. The DWord 2 contains the Requester ID and the TAG value. Requester ID is obtained by combining the bus_number, device_number, and function_number from the configuration

interface. The Requester ID and the TAG value are used to uniquely define the particular transaction. After forming the DW 1 and DW 2, they are set to send along with the 32 bit lower order bits of the address through the transmit interface, each in different clock cycle. If the address is 64 bits then, the higher order bits of address are sent in the next cycle. The process is shown as a flow diagram in the figure 3.3.

Write TLP Request

The controller processes the write TLP similar to Read TLP request. The only additional operation is to send the data received from the device, at the end of address. The controller does not buffer any data. All the buffering is done in the device implementation only. The "valid" data sent from the device is just transferred to the Core through transmit interface. Write TLP process is shown as a flow diagram in the figure 3.3.

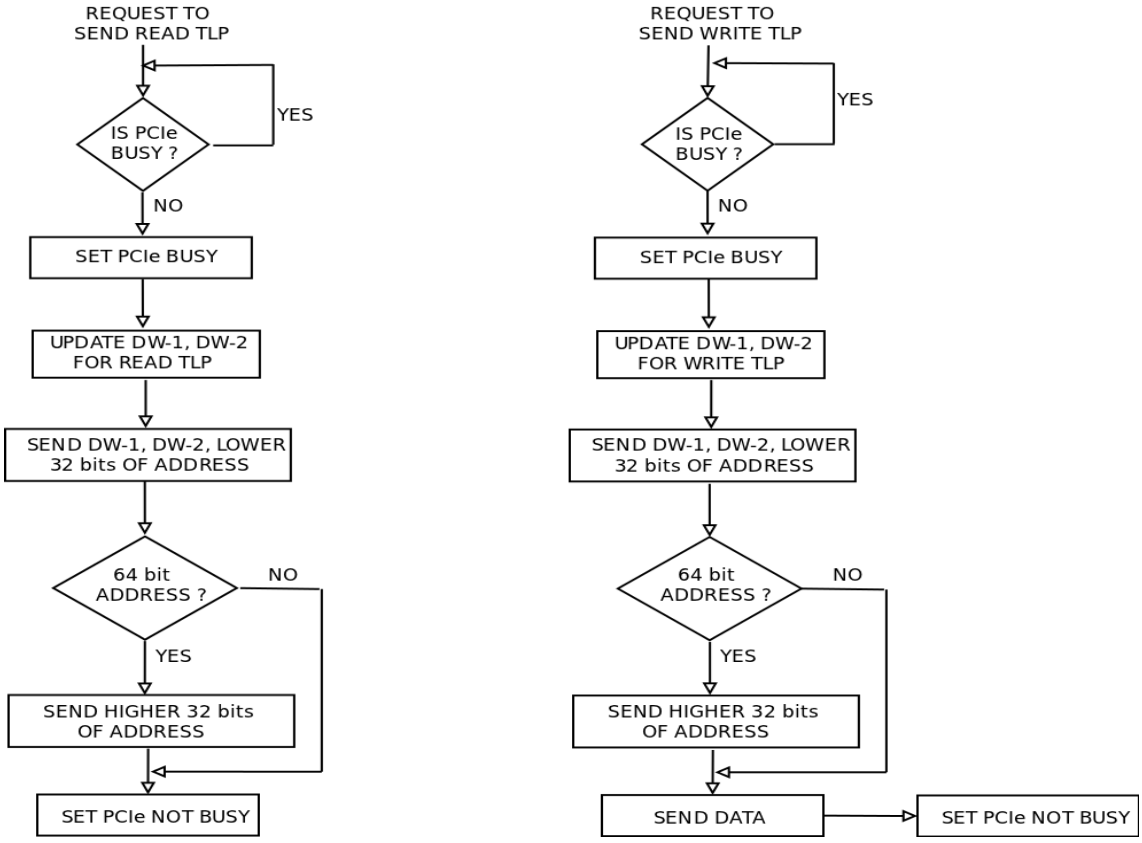

Figure 3.3: Read Write TLP Flow

Special issues while transmitting TLPs

Source Throttling

The controller can throttle back if it has no "valid" data present on the tx_tdata[31:0]. When this condition occurs, the the controller deasserts tx_tvalid, which instructs the core to disregard the data present on tx_tdata[31:0].

Destination Throttling

The core can throttle the controller if there is no space left for new TLP in the transmit pool buffer.

Transaction Discontinue

The controller can discontinue the transaction at any point of time. This is done by simultaneous assertion of valid, last and src_dsc lines. The core then discards the present transaction completely.

3.2.2 Interrupt Handler

The interrupt handler module takes the "valid" interrupt numbers from the Device, accumulates them and then sends them as a whole to the PCIe core. PCIe core sends these vectors as message TLPs to the Host - PCIe Core. The device requested number of interrupt messages and the Core's supported maximum number of messages are compared and the minimum of the two is chosen as the threshold limit for the number of vectors to accumulated. On every valid vector number, the vectors are accumulated. When the number of vectors equal the threshold value, then the module initiates messages to the Core.

This module functions only if MSI interrupts are enabled. This is notified by the core through the signal cfg_interrupt_msienable. The number of messages that the core supports is given by signal cfg_interrupt_mmenable.Two raised to the power of the value of the sig-

nal gives the maximum number of interrupts the Core can take at a time. For instance, if this signal value is zero, then it functions as a Single Message MSI interrupt.

3.2.3 Request Handler

This module receives the TLPs from the Core, through the Receive interface. It decodes the TLP, as per the PCI Express 1.0 spec [5]. The decoding of the TLP and the subsequent actions taken by the module are tabulated in the table 3.9

Table 3.9: TLP Decoding and Actions by the Request Handler Module

Fmt Field (**data[30:29]**)	**Typ Field** (**data[28:24]**)	**Description**	**Action taken**
2'b00	5'b00000	3 DWord Memory Read	1. Updates the Completer-Requester by supplying the Requester Tag value. 2. Sends read signal and read addrs through memory interface to device.
2'b01	5'b00000	4 Dword Memory Read	Same as 3DW Memory Read
2'b10	5'b00000	3 Dword Memory Write	1. Sends write signal to device memory. 2. Sets the byte enable to '1111 if it receives 2 Dword payload. Sets it to '0011 if it receives 1 DWord payload. 3. Puts the data on data bus.
2'b11	5'b00000	4 DWord Memory Write	Same as 3 Dword Memory write.
2'b00	5'b01010	Completion with No data	Not Applicable
2'b10	5'b01010	Completion with data	Sends the data payload on the Completion data-to-device interface

3.3 Verification

The verification of the PCIe Controller is carried out in QuestaSim simulator. Testbench was written for various test cases and the test bench was integrated to the actual module. The setup was translated to a verilog code and was simulated in QuestaSim. Controller was tested with the following test cases :

1. Test case for Transmitting Read, Write, Completion TLPs under normal conditions.
2. Test case for Transmitting Read, Write, Completion TLPs with "destination Throttling", Source Throttling, Transaction discontinue.
3. Test case for Receiving Read, Write, Completion TLPs under normal conditions.
4. Test case for Receiving Read, Write, Completion TLPs with "destination Throttling", Source Throttling, Transaction discontinue.
5. Receive Incomplete Payload.
6. Transmit Interrupt Vectors as Single Message MSI and Multiple Message MSI.

3.4 Simulation Results

The Simulation Results for some of the mentioned test cases are shown in the following figures.

Normal Write Operation

The simulation result for normal write operation is shown in figure 3.4

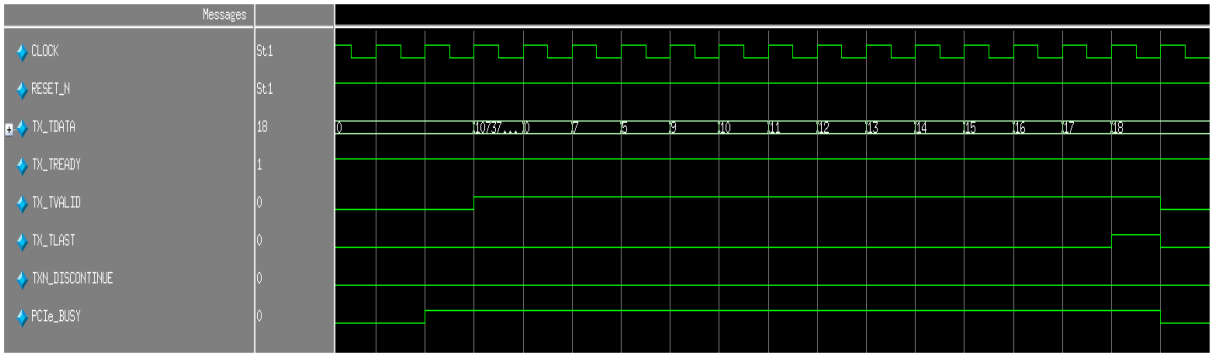

Figure 3.4: Normal-Write TLP waveform

Write Operation with Destination Throttling

The simulation result for write operation with destination throttle is shown in figure 3.5

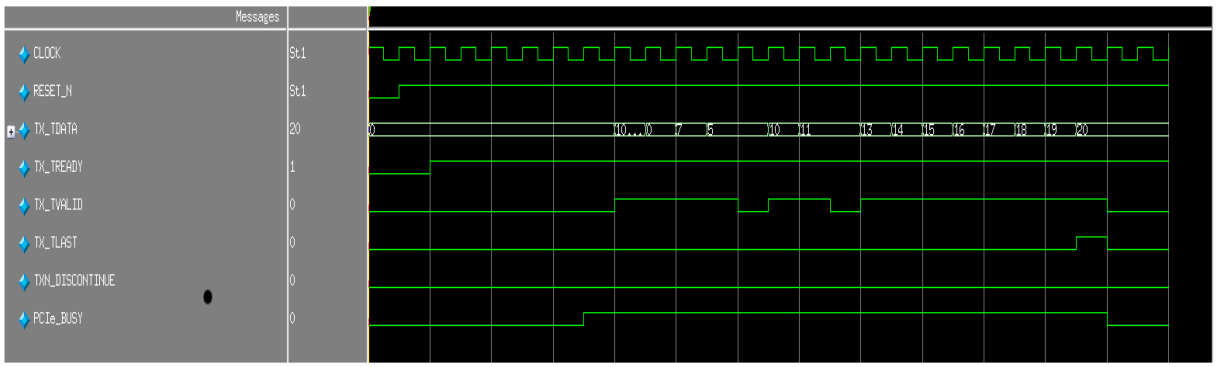

Figure 3.5: Write TLP - with Destination Throttle waveform

Write Operation with Source Throttling

The simulation result for write operation with source throttle is shown in figure 3.6

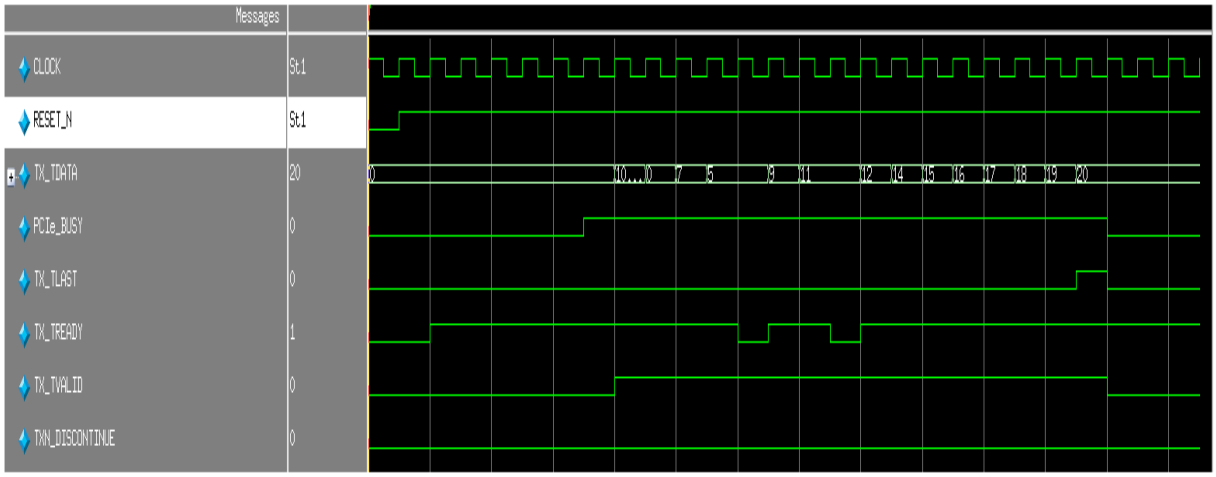

Figure 3.6: Write TLP - with Source Throttle waveform

Write Operation with Transaction Discontinue

The simulation result for write transaction discontinue is shown in figure 3.7

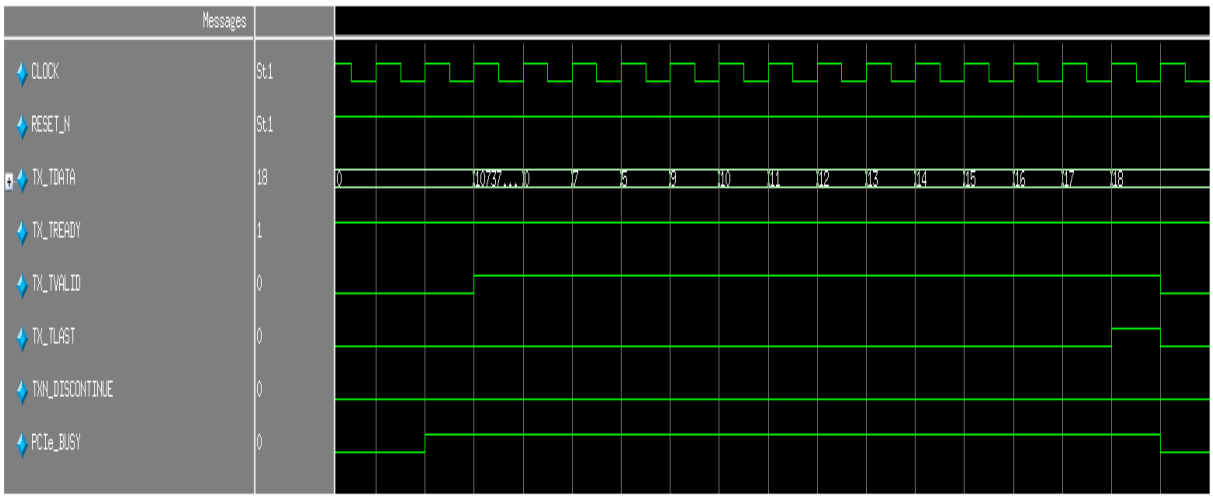

Figure 3.7: Write TLP Transaction Discontinue waveform

Normal Completion TLP Operation

The simulation result for normal write operation is shown in figure 3.8

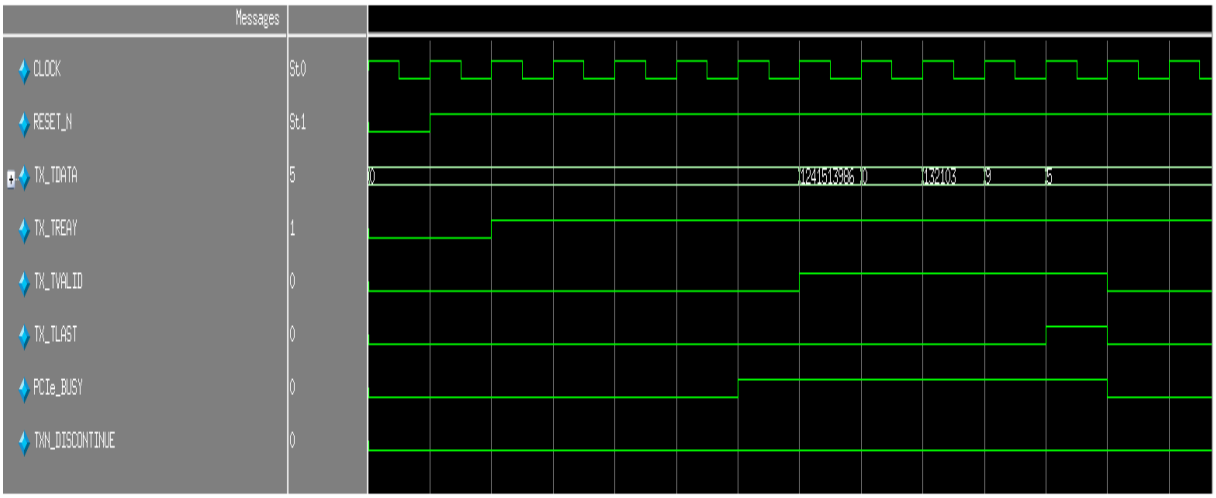

Figure 3.8: Normal-Completion TLP waveform

Read Operation with Incomplete Payload

The simulation result for read operation receiving payload as two transactions is shown in figure 3.9

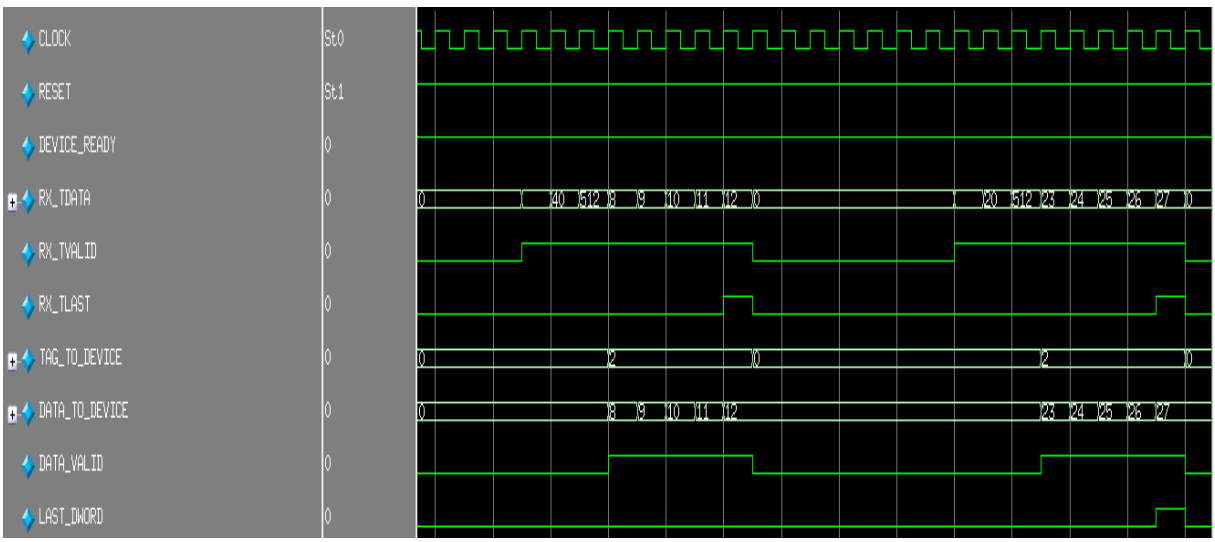

Figure 3.9: Read Incomplete Payload

Interrupt Mechanism with multiple MSI

The simulation result for normal write operation is shown in figure 3.10

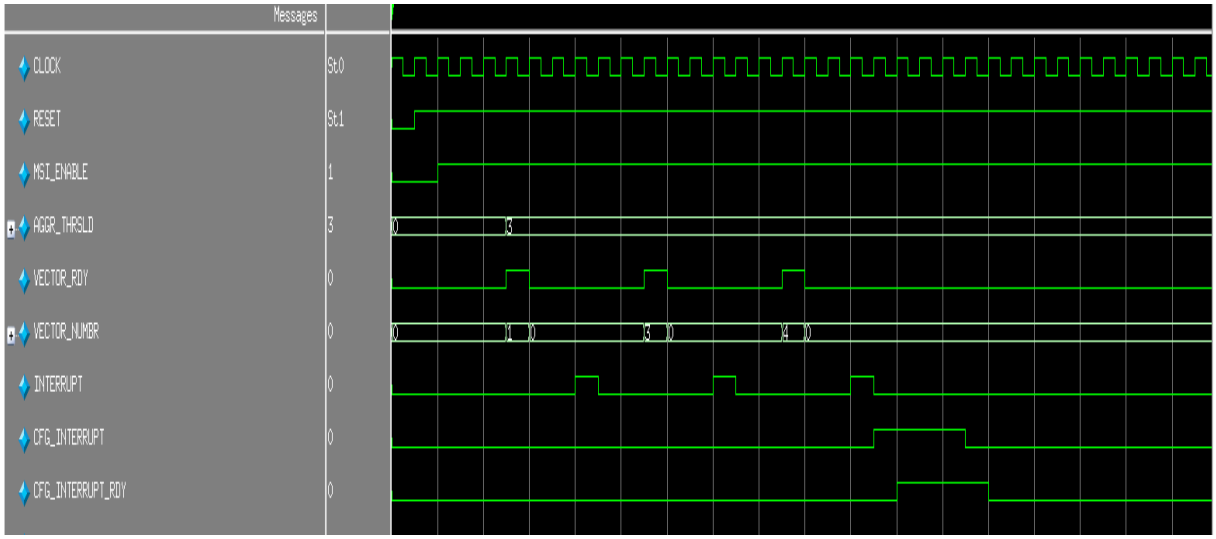

Figure 3.10: Multiple Message MSI waveform

3.5 Synthesis Report

The PCIe Controller was synthesized on the device *XC6VLX240T-FF1156*. The Slice utilization and timing summary has been provided below.

Number of Slice Registers used : 720

Number of Slice LUTs used : 626

Number of LUT-FF Pairs used : 432

Maximum Frequency of operation : 295 MHz

CHAPTER 4

NVM Express Controller

This chapter explains the entire working of the NVM Express controller. It explains the basic steps involved in Controller Initialization, Controller Shutdown,and Command Processing. It introduces the Command Structure, Command Set, and Command Completion Structure. All the steps and structures mentioned here are as per the NVM Express 1.0c Specification [6].

4.1 Command Processing

This section describes the steps involved in issuing and completing a command. Figure 4.1 shows the steps. The steps are:

- Step 1 : The host creates a command for execution within the appropriate Submission Queue in the host memory.
- Step 2 : The host updates the Submission Queue Tail Doorbell register with the new value of the Submission Queue Tail entry pointer. This indicates to the controller that a new command(s) is ready for processing.
- Step 3 : The controller arbitrates for the command queue and fetches it. Round Robin arbitration is used.
- Step 4 : Controller executes the command, which has been fetched.
- Step 5 : After the command has completed execution, the controller writes a completion queue entry to the associated Completion Queue.
- Step 6 : The controller optionally generates an interrupt to the host to indicate that there is a completion queue entry to process. Here, multiple message MSI interrupt is used as interrupt mechanism.

- Step 7 : The host processes the completion queue entry in the Completion Queue.
- Step 8 : The host writes the Completion Queue Head Doorbell register to indicate that the completion queue entry has been processed. The host may process many entries before updating the associated CQHDBL register.

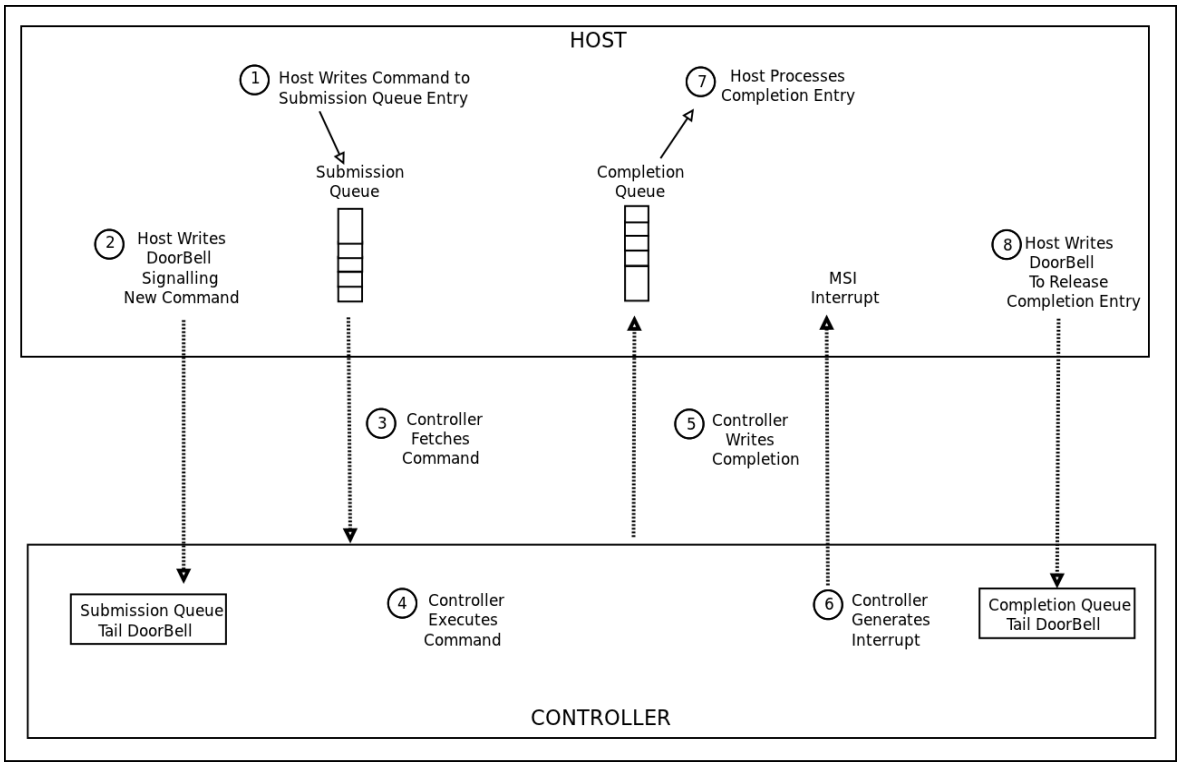

Figure 4.1: Command Processing

4.2 Command Structure

Each command is 64 bytes in size. The 64 byte command format for the Admin Command Set and NVM Command Set is defined in the figure 4.2.

Command Dword 0, Namespace Identifier, Metadata Pointer, PRP Entry 1, and PRP Entry 2 have common definitions for all Admin commands and NVM commands. Command DWord 10 to 15 have definitions that are specific to a command.

Bytes	Description
63:60	**Command Dword 15 (CDW15):** This field is command specific Dword 15.
59:56	**Command Dword 14 (CDW14):** This field is command specific Dword 14.
55:52	**Command Dword 13 (CDW13):** This field is command specific Dword 13.
51:48	**Command Dword 12 (CDW12):** This field is command specific Dword 12.
47:44	**Command Dword 11 (CDW11):** This field is command specific Dword 11.
43:40	**Command Dword 10 (CDW10):** This field is command specific Dword 10.
39:32	**PRP Entry 2 (PRP2):** This field contains the second PRP entry for the command or if the data transfer spans more than two memory pages, then this field is a PRP List pointer.
31:24	**PRP Entry 1 (PRP1):** This field contains the first PRP entry for the command or a PRP List pointer depending on the command.
23:16	**Metadata Pointer (MPTR):** This field contains the address of a contiguous physical buffer of metadata. This field is only used if metadata is not interleaved with the logical block data, as specified in the Format NVM command. This field shall be Dword aligned.
15:08	Reserved
07:04	**Namespace Identifier (NSID):** This field specifies the namespace that this command applies to. If the namespace is not used for the command, then this field shall be cleared to 0h. If a command shall be applied to all namespaces on the device, then this value shall be set to FFFFFFFFh.
03:00	**Command Dword 0 (CDW0):** This field is common to all commands and is defined in Figure 4.3

Figure 4.2: Command Format - Admin and NVM Command Set

Bit	Description
31:16	**Command Identifier (CID):** This field specifies a unique identifier for the command when combined with the Submission Queue identifier.
15:10	Reserved
09:08	**Fused Operation (FUSE):** In a fused operation, a complex command is created by "fusing" together two simpler commands.This is an optional field and is not supported. This filed is always set to 00, which indicates a Normal operation.
07:00	**Opcode (OPC):** This field specifies the opcode of the command to be executed.

Bits	Definition
00b	Normal operation
01b	Fused operation, first command
10b	Fused operation, second command
11b	Reserved

Figure 4.3: Command DWord 0

4.3 Command Set

The NVMe controller defines two sets of commands namely Admin Command Set and the NVM Command Set. The Admin Command Set takes care of all the administrative tasks related to the controller. This includes setting up the controller, creating or deleting Queues, aborting commands etc. The NVM Command set is responsible for the actual data transfer between the Host and the Nand Flash Device. The list of Admin Commands and NVM commands supported by the designed controller is shown in the figure 4.4

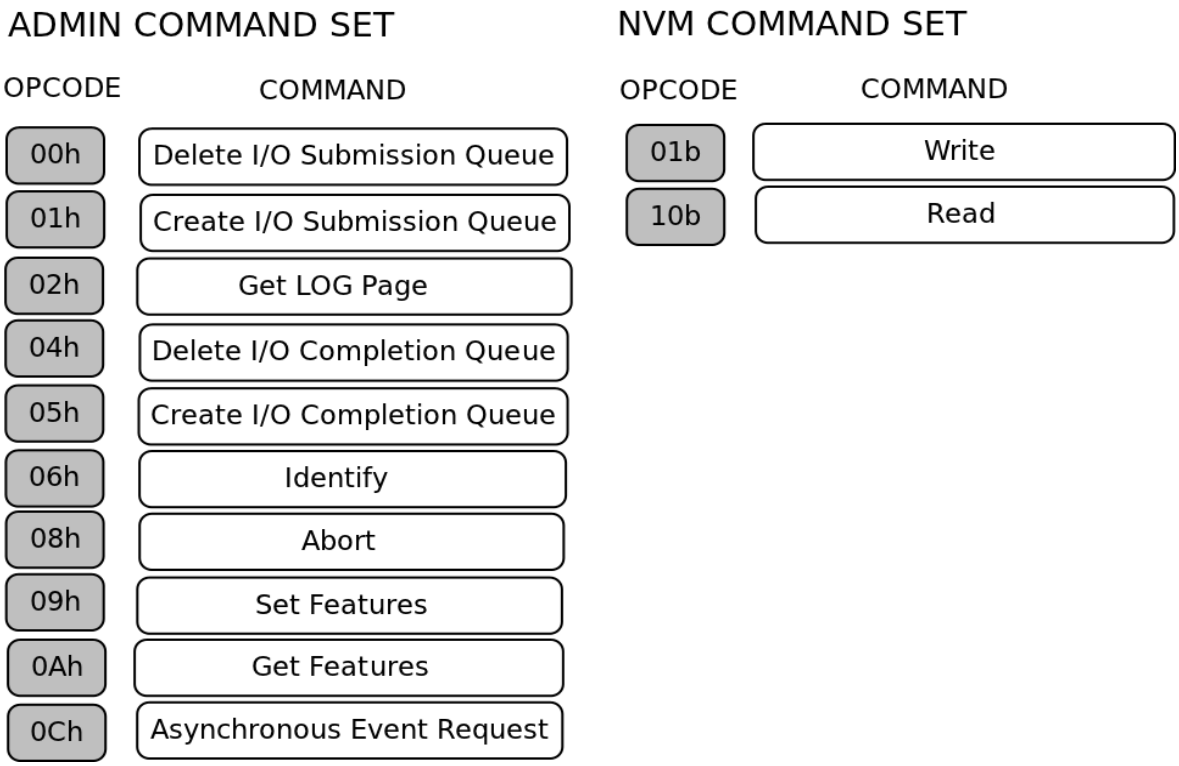

Figure 4.4: Command Set - Admin and NVM

4.4 Controller Registers

Controller registers are memory mapped to the Host memory. The amount of memory required for these registers is advertised to the Host during PCIexpress link setup. Once the link is setup, the controller registers can be accessed by the Host by simple Load-Store operations to its memory. These operations are eventually transformed into PCIe Transactions for Read and Write to the Controller Registers(or Register File). All the registers are to be accessed in their native width. Unaligned register accesses are not supported and are handled at the PCI Express layer. The following figure describes the register map for the controller.

Start	End	Symbol	Description
00h	07h	CAP	Controller Capabilities
08h	0Bh	VS	Version
0Ch	0Fh	INTMS	Interrupt Mask Set
10h	13h	INTMC	Interrupt Mask Clear
14h	17h	CC	Controller Configuration
18h	1Bh	Reserved	Reserved
1Ch	1Fh	CSTS	Controller Status
20h	23h	Reserved	Reserved
24h	27h	AQA	Admin Queue Attributes
28h	2Fh	ASQ	Admin Submission Queue Base Address
30h	37h	ACQ	Admin Completion Queue Base Address
38h	EFFh	Reserved	Reserved
F00h	FFFh	Reserved	Command Set Specific
1000h	1003h	SQ0TDBL	Submission Queue 0 Tail Doorbell (Admin)
1000h + (1 * (4 << CAP.DSTRD))	1003h + (1 * (4 << CAP.DSTRD))	CQ0HDBL	Completion Queue 0 Head Doorbell (Admin)
1000h + (2 * (4 << CAP.DSTRD))	1003h + (2 * (4 << CAP.DSTRD))	SQ1TDBL	Submission Queue 1 Tail Doorbell
1000h + (3 * (4 << CAP.DSTRD))	1003h + (3 * (4 << CAP.DSTRD))	CQ1HDBL	Completion Queue 1 Head Doorbell
1000h + (4 * (4 << CAP.DSTRD))	1003h + (4 * (4 << CAP.DSTRD))	SQ2TDBL	Submission Queue 2 Tail Doorbell
1000h + (5 * (4 << CAP.DSTRD))	1003h + (5 * (4 << CAP.DSTRD))	CQ2HDBL	Completion Queue 2 Head Doorbell
...	...	...	...
1000h+ (2y * (4 << CAP.DSTRD))	1003h + (2y * (4 << CAP.DSTRD))	SQyTDBL	Submission Queue y Tail Doorbell
1000h + ((2y + 1) * (4 <<	1003h + ((2y + 1) * (4 <<	CQyHDBL	Completion Queue y Head Doorbell

Figure 4.5: Register map for the controller

4.5 Controller Initialization

This section describes the procedure for initializing the controller. The following actions are performed by the host, in sequence, to initialize the controller and begin executing commands : The steps are:

- Step 1 : The PCI Express registers are set appropriately based on the system configuration.
- Step 2 : The Admin Queue is configured. This is done by setting the Admin Queue

Attributes (AQA), Admin Submission Queue Base Address (ASQ), Admin Completion Queue Base Address (ACQ) to appropriate values.

- Step 3 : The controller settings should be configured. Specifically :
  - The controller arbitration should be selected in CC.AMS.
  - The memory page size should be initialized in CC.MPS.
  - The IO command set that is to be used should be selected in CC.CSS.
- Step 4 : Controller is enabled by setting CC.EN to 1.
- Step 5 : The host waits for the controller to indicate it is ready to process commands. the controller is ready to process the commands when CSTS.RDY is set to 1.
- Step 6 : Host determines the configuration of the controller by issuing the Identify Command, specifying the controller data structure. The host then determines the configuration of each namespace by issuing the identify command for each namespace, specifying the Namespace data structure.
- Step 7 : Host then determines the number of I/O Submission Queues and I/O Completion Queues supported using the Set Features command with the Number of Queues feature identifier. After determining the number of I/O Queues, the MSI registers are configured.
- Step 8 : The host allocates the appropriate number of I/O Completion Queues based on the number required for the system configuration and the number supported by the controller. The I/O Completion Queues are allocated using the Create I/O Completion Queue command.
- Step 9 : The host allocates the appropriate number of I/O Submission Queues based on the number required for the system configuration and the number supported by the controller. The I/O Submission Queues are allocated using the Create I/O Submission Queue command.

4.6 Controller Shutdown

This section describes how the controller is shutdown. The controller can be shutdown down in one of the two ways, namely Normal shutdown or abrupt shutdown.

The host performs the following actions in sequence for normal shutdown:

- Step 1 : The host stops issuing any new I/O commands to the controller and allow any outstanding commands to complete.

- Step 2 : The host then deletes all I/O Submission Queues. A result of successful completion of the Delete I/O Submission Queue command is that any remaining commands outstanding are aborted.
- Step 3 : The host deletes all I/O Completion Queues.
- Step 4 : The host sets the Shutdown Notification (CC.SHN) field to 01b to indicate a normal shutdown.
- Step 5 : The Controller indicates when shutdown processing is complete by updating the Shutdown Status (CSTS.SHST) field to 10b .

The host performs the following actions in sequence for abrupt shutdown:

- Step 1 : The host stops issuing any new I/O commands to the controller and allow any outstanding commands to complete.
- Step 2 : The host sets the Shutdown Notification (CC.SHN) field to 10b to indicate a abrupt shutdown.
- Step 3 : The Controller indicates when shutdown processing is complete by updating the Shutdown Status (CSTS.SHST) field to 10b .

CHAPTER 5

Implementation of NVM Express

This chapter explains how the NVM express controller is implemented in hardware. The controller is designed according to the NVMe Spec 1.0c [6]. Controller interface signals and their brief description is provided along with the detailed description of all the logical modules involved in the design. The chapter also provides various test cases that are used to verify the functionality of the design. Synthesis results for the controller are provided at the end. The controller interfaces and logical module partitions are shown as a block diagram in the figure 5.1.

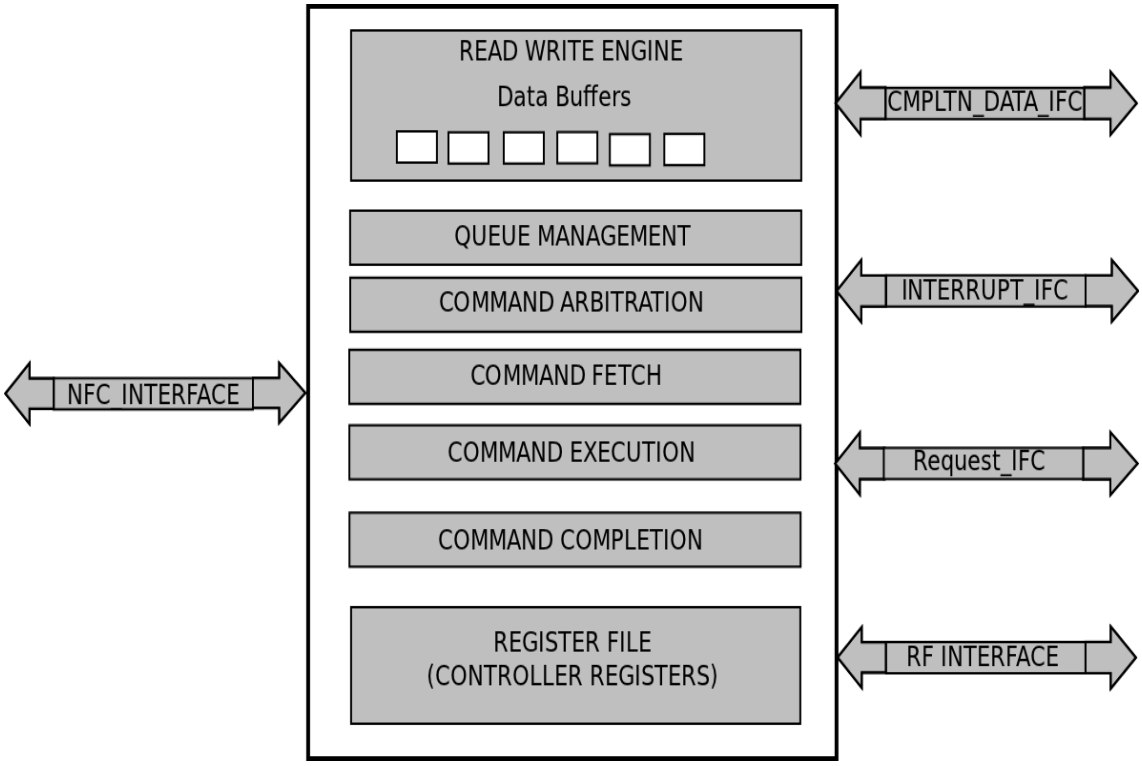

Figure 5.1: NVMe Modules and Interfaces

5.1 Controller Interfaces

The controller has five interfaces to connect PCIe controller on one side and Nand Flash Controller on the other. The description of the interfaces and the signals involved in each interface are given in the following sub-sections.

5.1.1 Completion Data Interface

This interface is used to connect the controller to the PCIe controller. It provides interface signals necessary to receive the incoming data from the PCI express. This data is received as a completion data payload by the PCIe Controller and then transferred to the NVMe controller. In the context of NVMe, the data could either be a 4KB data page or a 64B command. The interface signals are shown in the table 5.1

Table 5.1: Completion Data Interface Signals

Signal	Direction	Description
wr_data_valid	In	Indicates that the data received on the data lines is valid
wr_data_in[31:0]	In	Completion data received from the PCIe controller
wr_last_DWord	In	Indicates that this is the last Dword for the entire transaction No more data will be received with the same tag value.
wr_tag[6:0]	In	Tag value for this transaction The details of how the controller uses the tag is given in table 5.6

5.1.2 Interrupt Interface

This interface provides signals necessary to send interrupt vector information to the PCIe controller. This vector information is eventually sent as multiple message MSI over the PCIe channel, by the PCIe controller. The interface signals are shown in the table 5.2

Table 5.2: Interrupt Interface Signals

Signal	Direction	Description
rg_interrupt_ready	Out	Indicates that the interrupt vector number is Valid
rg_vector_number[4:0]	Out	The value of the vector number for which the the interrupt is generated
rg_MSI_number[4:0]	Out	Indicates the NVMe controller's request for number of messages in MSI interrupt

5.1.3 Request Interface

This interface is also a part of the connection between the NVMe controller and PCIe controller. It provides interface signals necessary to Request the PCIe controller to transmit Read, Write or Completion TLPs over the PCIe channel. This interface also helps the NVMe controller to transmit the data, as a payload to Write and Completion TLPs. This data could be 4KB data page or 16 byte command completion or 64 bit registers. Data page and command completions are sent as Write TLP payloads and the registers are sent as Completion TLP payload. The interface signals are shown in the table 5.3

Table 5.3: Request Interface Signals

Signal	Direction	Description
rg_data_to_pcie[31:0]	Out	Data bus to send data payload to pcie controller
rg_address_to_pcie[63:0]	Out	Address bus to send address of the main memory location
rg_requested_tag[1:0]	Out	Tag value requested for the Transaction
rg_payload_length[9:0]	Out	Indicates the amount of payload being sent during write and completion and indicates the amount of payload requested during read operation
rg_send_completion_tlp	Out	Request to send completion TLP
rg_send_write_tlp	Out	Request to send write TLP
rg_send_read_tlp	Out	Request to send read TLP
rg_64bit_address	Out	If set it Indicates that all the 64 bits of the address lines are valid, else 32 bits are valid
wr_send_valid_data	In	Indicates the NVMe controller to send valid data payload. The controller sends valid data on the data lines only if this signal is high
rg_data_valid	Out	Indicates that the data on data lines is valid
rg_nvm_wait	Out	Indicates the PCIe controller to wait
wr_wait	In	Indicates the NVMe controller to wait

5.1.4 Register File Interface

This interface provides a Read Write access to the Controller Registers. PCIe controller can write into registers through this interface. However it can only request to read through this interface. The read data from the register file is sent as completions through the NVMe Request interface. The interface signals are shown in the table 5.4

Table 5.4: Register File Interface Signals

Signal	Direction	Description
wr_regFile_address_in[31:0]	In	Register File Address
wr_regFile_data_in[63:0]	In	Data bus for writing into Register File
wr_byte_enable[3:0]	In	4'b1111 indicates 64 bit register access 4'b0011 indicates 32 bit register access
wr_read	In	Register File Read operation
wr_write	In	Register File Write operation

5.1.5 Nand Flash Controller Interface

This interface is used to connect the controller to the Nand Flash Controller. It provides necessary signals to access the memory mapped register file of the Nand Flash Controller. It has a 32-bit data bus to transfer the data to or from Nand Flash Controller buffers. Nand Flash Chips in general support either a 16 bit bus or a 8 bit bus. However, the data bus to NFC is made 32 bits in order to transfer the data at a much faster rate and hence the controller will ready to process other commands. The interface signals are shown in the table 5.5

Table 5.5: Nand Flash Controller Interface Signals

Signal	Direction	Description
rg_address_to_nand[11:0]	Out	Address to NAND's memory mapped register file and buffers
rg_data_to_nand[31:0]	Out	Data to NAND's memory mapped register file and buffers
wr_data_from_nand[31:0]	In	Data from NAND
rg_chip_enable	Out	Active low signal to indicate Chip Enable to NAND
rg_write_enable	Out	Active low signal to indicate Write Enable to NAND
rg_output_enable	Out	Active low signal to indicate Output Enable to NAND
wr_interrupt	In	Interrupt from NAND indicating Read data Ready in buffers
wr_ready_busy	In	Indicate NFC is Ready or Busy

5.2 Modules Description

The entire design of the controller is divided into seven logical modules as shown in figure 5.1. Six state machines are defined to assist the modules in deriving their respective functionality. Detailed block diagram showing the modules and state machines is shown in the figure 5.2. State machines are highlighted in italics.

5.2.1 Read-Write Engine

This module is designed to facilitate data transfer between PCIe controller and NVMe controller, and also between NVMe controller and Nand Flash controller. Five buffers are used in this module. A brief description of each buffer is given below.

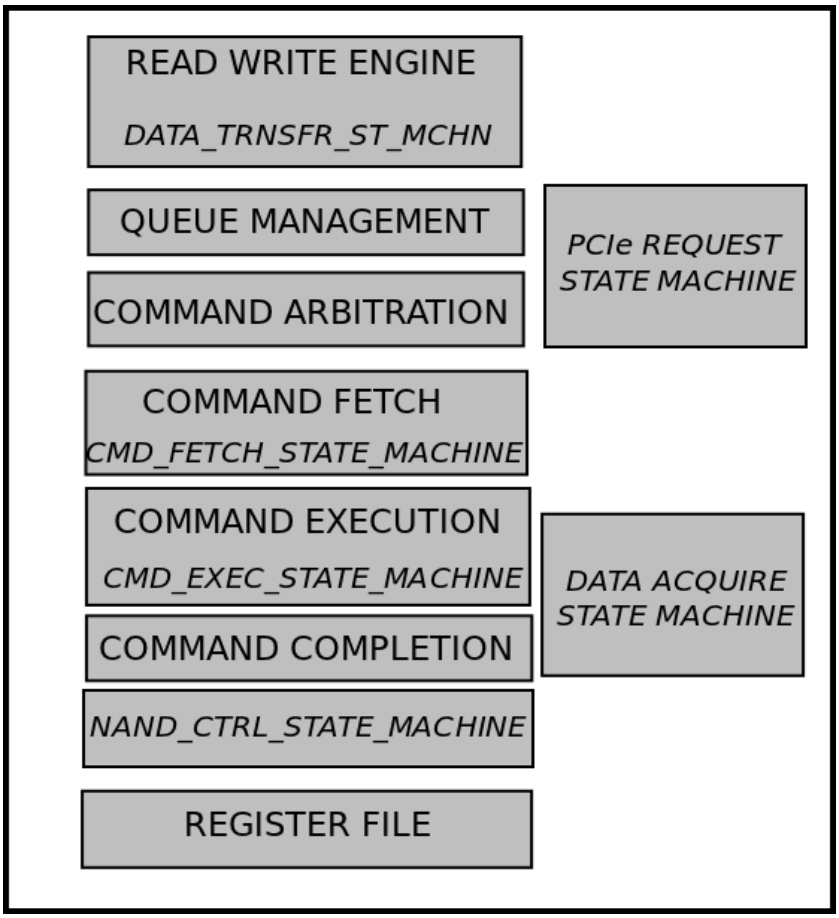

Figure 5.2: NVMe Modules and State Machines

- **Data Buffer** : This buffer is 4K Byte in size. This is used to store the data page that is received from the Host main memory. Hence the data page that is to be written to the Nand Flash, is first stored in this intermediate buffer and then a write operation is initiated to the Nand Flash. However, the data page transferred from the Nand Flash to the Host memory, as a part of Read operation, is not stored in this buffer. The Data-Transfer State machine helps in getting access to the PCIe Controller, through the PCIe request - State Machine , to fill the buffer with data page and to notify the controller about the same.

- **Controller Data Structure Buffer** : This buffer is 4K Bytes in size. This buffer is used to store the controller data structure. This is a Read-Only data structure, which is initialized on controller reset. The data structure defines the entire functionality of the controller. PCIe Request - State Machine helps to gain access to PCIe controller and transfer the data from NVMe to PCIe.

- **Name Space Data Structure Buffer** : This buffer is 4K Bytes in size. This buffer is used to store the namespace data structure. This is a Read-Only data structure, which is initialized on controller reset. The data structure defines the name space related

attributes. PCIe Request - State Machine helps to gain access to PCIe controller and transfer the data from NVMe to PCIe.

- **Controller Features Buffer** : This buffer is 4K Bytes in size. This buffer is used to store the features supported by the controller. This is a Read-Only data structure, which is initialized on controller reset. PCIe Request - State Machine helps to gain access to PCIe controller and transfer the data from NVMe to PCIe.
- **Error logs** : This buffer is 64 bytes in size. There are five such buffers. These buffers are used to update error logs for as many as five commands at a time. Any errors in the incoming commands are reported to the host by updating the status information in the command completions. However, if it is desired to notify the host more about the error in the command then the error logs can be used.

5.2.2 Queue management

This module maintains the Tail and Head Pointers of the Submission Queues and Completion Queues. It also signals Queue-Empty and Queue-Full conditions. The generic structure of the queue and its Empty and Full conditions are shown in the figure 5.3.

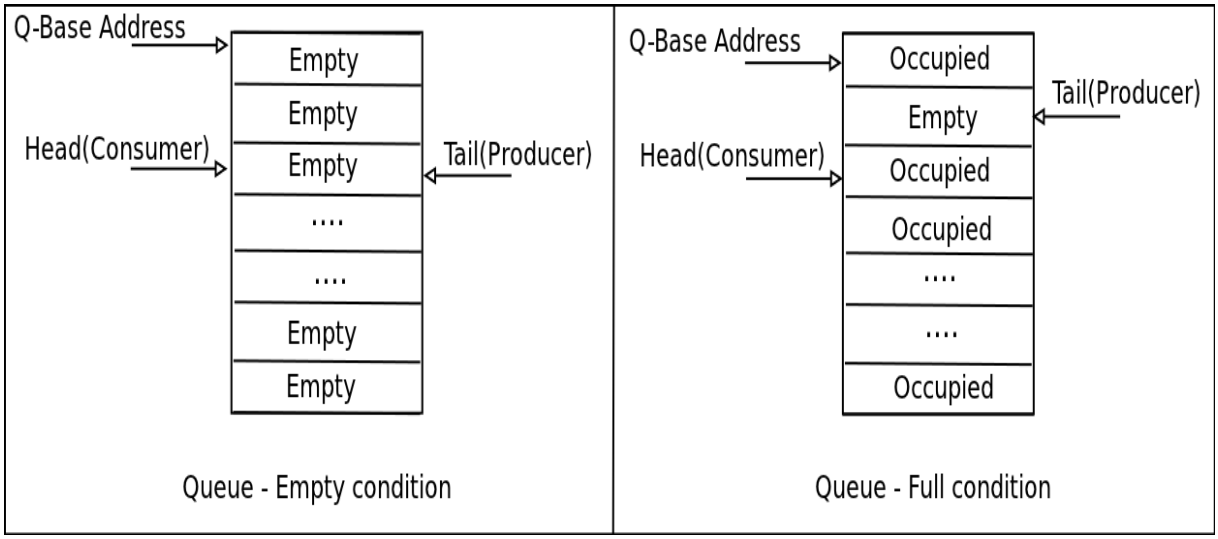

Figure 5.3: Queue Empty and Full Conditions

Queue-Full :

The queue full condition is alerted to the controller when the Head pointer equals one more

than the Tail pointer. The total number of entries in a queue when full is one less than the queue size.

Queue-Empty :

The queue empty condition is alerted to the controller when the Head pointer equals the Tail pointer.

Queue-Wrap :

Whenever the pointers (Tail or Head) reach the end of the Queue (i.e. Max queue size) , then the pointers are brought to top of the queue. In essence it is a circular-queue.

Submission Queue :

For the submission queue, the Host is the producer and controller is the consumer. Hence, the Tail pointer is updated by the host by writing into SQ Tail Pointer Register in the Register File. Whenever a new command is fetched by the controller, the corresponding SQ Head Pointer is incremented by one.

Completion Queue :

For the completion queue, the Controller is the producer and the Host is the consumer. Hence, the Tail pointer is updated by the controller and the head pointer is updated by the Host by writing into the CQ Head Pointer Register in the Register File. Tail pointer in the controller is updated whenever a new completion is sent to the host.

Queue Size :

The minimum size for any queue (upon creation) is two. The maximum size for I/O SQ or I/O CQ is 64K entries. However, the CAP.MQES field in the controller registers limits the maximum limit. The maximum size for the Admin SQ and CQs is defines as 4K entries.

Queue Identifier :

Admin SQ Identifier and CQ Identifier value is zero. I/O SQ and CQ are identified through a 16-bit ID value that is assigned to the queue when it is created.

5.2.3 Command Arbitration

A simple Round-Robin arbitration mechanism is used to decide the Queue from which to Fetch the command. Bluespec's predefined functional block "Arbiter" has been used to implement the design. The Arbiter functions on a Client Request-Grant mechanism. Each Submission Queue is given a client number. Admin SQ has a client number 0, I/O SQ 0 has a client number 1 and similarly I/O SQ `x` has a client number `x+1`. Any number of clients can request the Arbiter at a time. Access is granted on a round robin priority basis. The priority order is Client 0 , Client 1 , Client 2 .. so on i.e. Admin SQ , ISQ 0 , ISQ 1 , so on .

Eligibility to request :

A Submission Queue gains the eligibility to request under the following conditions :

- The controller is enabled.
- The corresponding queue is created .
- The queue is not empty . This is signalled by the queue-management rule.
- The queue is not full . This is also signalled by the queue-management rule.

Grant :

After completing the requests from the highest priority client, the next highest client,which is requesting, is given the grant.

Hold :

The Submission Queue holds the request until all the commands in its queue are completely fetched by the controller. Once the SQ gets empty, it looses the eligibility to request and hence the next SQ is granted.

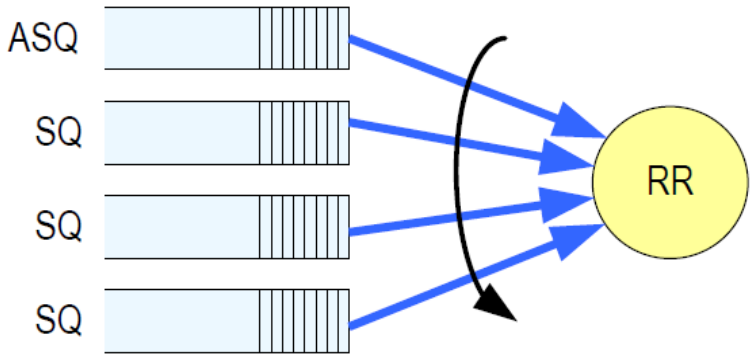

Figure 5.4: Round Robin Arbiter

5.2.4 Command Fetch

This logical module fetches the commands from the Submission Queue that has got the grant from the Arbiter. This module is implemented as a state machine, named Command-Fetch State Machine. The state machine is shown in the figure 5.5.

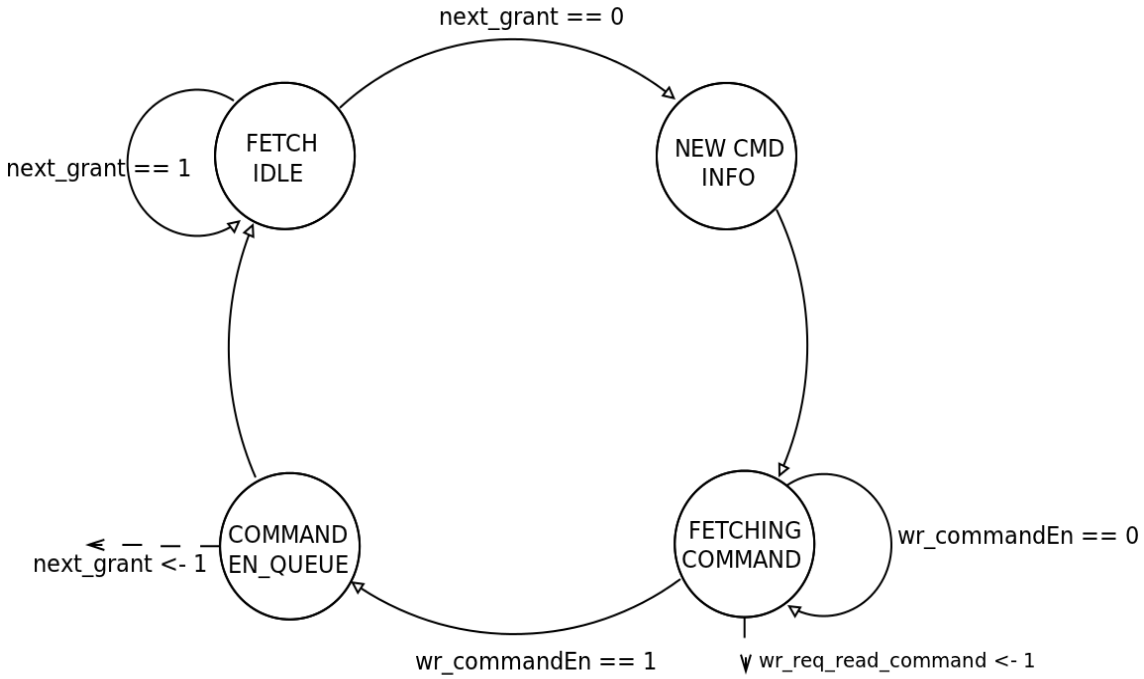

Figure 5.5: Command Fetch State Machine

Command Fetch State Machine Description :

FETCH IDLE STATE :

This is the idle state for the State machine, where it waits for new submission queue to be granted. Once the SQ is granted, the `next_grant` signal is lowered to indicate to the Arbiter not to grant to any other queue until, the present command is fetched. This lowering of the signal triggers the state machine to transition to next state to start fetching the command.

NEW COMMAND INFO STATE :

This is an intermediate state, where the required information to fetch the command is gathered. This information regarding the command includes : Base Address of the SQ, Offset in the Queue, and also the SQ ID for the command. The SQ ID for the command is actually en-queued into a FIFO `sqID_fifo`. This sq ID is used during the command execution state. Base Address plus the Offset gives the actual address from where the command has to be fetched from the Main Memory.

Obtaining Base Address and Offset :

Case 1 : **SQ is Contiguous in the Main Memory**

This condition is indicated by the host, by setting the cd11.pc bit to 1, while creating the SQ. In this case the base address is directly indicated by the PRP1 value in the command structure. The value of the offset is incremented on each command fetch, and is *wrapped* around to zero when it reaches the size of the queue.

Case 2 : **SQ is Not Contiguous in the Main Memory**

This condition is indicated by the host, by setting the cd11.pc bit to 0, while creating the SQ. In this case, the PRP1 value points to a list of page addresses. The list contains

base addresses for the pages which contain the SQ. Each base address points to a 4KB page containing 64 commands of the SQ. Hence, upon fetching the 64 Commands in the Queue, the Base address needs to be changed to next value in the List. The offset value increments on each command fetch and wraps to zero if its value is 64. Upon reaching the end of queue condition, the offset is made zero and base address points to first value in the list.

FETCHING COMMAND STATE :

This is the state where the controller fetches the command. In order to fetch the command the controller has to request the PCIe controller to send Read TLP, with the address obtained from the previous state. The access to the PCIe controller is obtained by requesting the PCIe Request State Machine, by raising the wire `wr_req_read_command` high. PCIe Request State Machine acknowledges the Fetch State Machine by raising the register `rg_req_read_command_accepted` high for one clock cycle. The controller then sends the Read TLP request to the PCIe controller and waits for the command to be obtained from the PCIe controller. The wire `wr_CommandEn` is raised high to indicate that the command fetched is ready in the wire `wr_Command_In`.

COMMAND EN-QUEUE STATE :

In this state, the command obtained in the wire `wr_Command_In` is en-queued into the internal fifo `internal_exeQ_fifo`. The wire `next_grant` is raised high to indicate the Arbiter that it can start issuing grants to eligible queues. State then transitions to IDLE state.

5.2.5 Command Execution

This module executes the Admin and NVM Commands present in the internal execution queue . When the queue is not Empty, the command is fetched and is executed. The SQ ID for the command to be executed is taken from the fifo `sqID_fifo`. The command is identified as Admin SQ command or NVM Command on the basis of this SQ ID information. Admin SQ will have SQ ID to be zero. Admin Command Execution State Machine is started if it SQ ID is zero else , NVM Command Execution State Machine is started. Both the state machines have the same idle state however, they are explained as two different state machines for the ease of explanation.

Command Execution State Machine :

IDLE STATE :

The command execution state machine is in idle state as long as the internal command execution queue is empty. Whenever it is not empty, the state machine jumps to either Admin Command Execution States or NVM Command Execution States on the basis of SQ ID value.

Admin Command Execution States :

This is a part of the Command Execution State Machine , that executes Admin Commands. Admin Commands are executed if the Submission Queue ID value associated with the command is zero. If the Submission Queue ID value is non-zero, then they are executed by the NVM Command Execution states. The Admin Command States are shown in the figure 5.6.

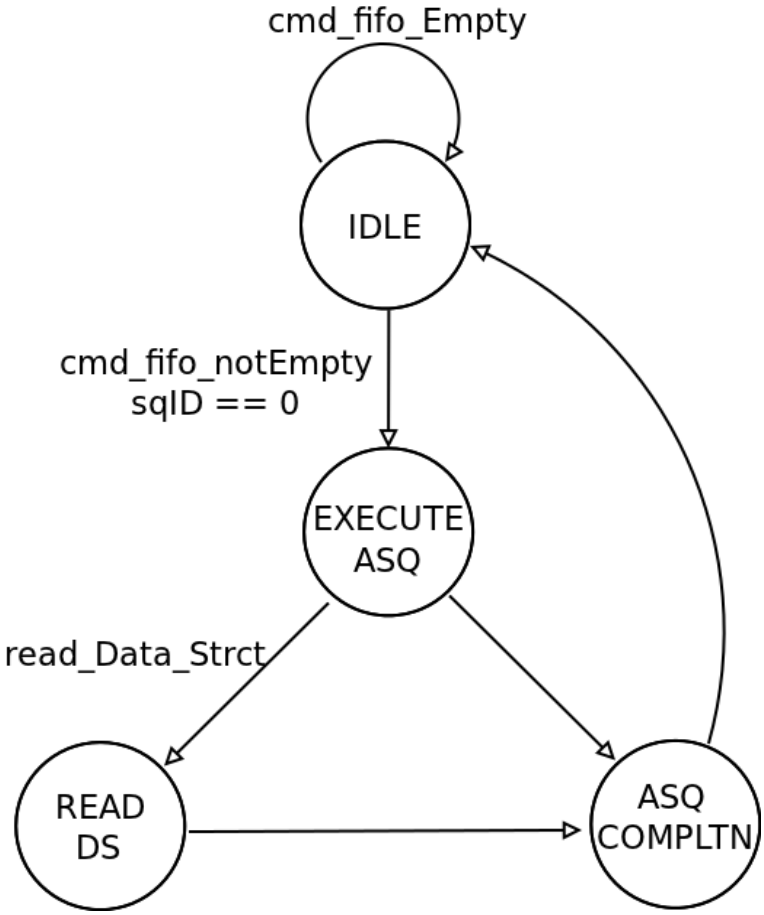

Figure 5.6: Admin Command Execution States

ASQ COMMAND EXECUTE STATE :

This state performs both the decode and execution of the command. Depending upon the opcode, the command execution is performed. “The get_features“ command and the ”identify” commands need the controller to access the data-structure buffers. Hence these commands cause the controller to transition to another state called Read Data Structure State. All the other commands are executed in this state itself.

Support for Non-Contiguous allocation of Queues in the Main Memory

The execution of *Create I/O CQ Command* and *Create I/O SQ Command* depends on weather the host has allocated the Queues Contiguously or Non-Contiguously in the Main Memory.

Case 1: **Contiguous allocation**

This is indicated by setting the CD11.PC bit to 1. Under this condition, the controller just enables the particular queue and the PRP1 field is treated as the *Base Address* of the queue.

Case 2 : **Non-Contiguous allocation**

This is indicated by setting the CD11.PC bit to 0. Under, this condition, the controller performs the following steps, before enabling the particular queue:

1. Requests the PCIe Request State Machine to fetch the PRP list by specifying the *tag* value associated with this queue.
2. Waits for the PRP List to be fetched. The rule `rl_prp_list_acquire` fetches the PRP List and notifies about its completion.
3. Enables the Queue after fetching the List.

The tag values associated with various operations inside the controller is shown in the table 5.6

Table 5.6: Tag values for various transactions

tag [1:0]	**tag [2]**	**tag [6:3]**	**Description**	**Txn Initiating Unit**	**Txn Receiving Unit**
00	X	XXXX	Not Used	None	None
01	X	XXXX	Acquire Command	Fetch Unit	Command-Data Acquire State Machine
10	X	XXXX	Acquire Data	Data transfer Unit	Command-Data Acquire State-Machine
11	0	0000 to 1111	Acquire CQ 0 to CQ 15 PRP List	Execution Unit	PRP-List Acquire Rule `rl_prp_list_acquire`
11	1	0000 to 1111	Acquire SQ 0 to SQ 15 PRP List	Execution Unit	PRP-List Acquire Rule `rl_prp_list_acquire`

READ DATA STRUCTURE STATE :

This state writes one of the requested data structure into the required memory location. This state requests the PCIe Request State Machine to get access to the PCIe Controller. This is done by raising the wire `wr_req_write_data` high. The PCIe Request State Machine asserts the signal `rg_req_write_data_accepted` to acknowledge the execution unit. The controller then sends Write TLP request to the PCIe controller. The PCIe controller later acknowledges by raising the wire `wr_send_valid_data` high. As long as this wire is logic high, the controller keeps sending the data from the datastructure buffer in units of 32bits every cycle. After sending the 4KB data the `wr_send_valid_data` is lowered. Later the command execution machine jumps to Admin command completion state.

ADMIN COMMAND COMPLETION STATE :

This state sends the completions to the main memory in the form of Write TLPs. As soon as the command is executed, the controller updates the status of the command either it is a success or a failure with some error. If there is any error in the command, then it also specified as a part of the command completion. The details of command completion are specified in the logical module Command Completion . Refer to section 2.2.6

NVM Command Execution States :

This is a part of the Command Execution State Machine , that executes the two NVM Commands, namely Read from NAND and Write to NAND. This set of states are executed if the Submission Queue ID for the command to be executed is not zero. The states are shown in the figure 5.7.

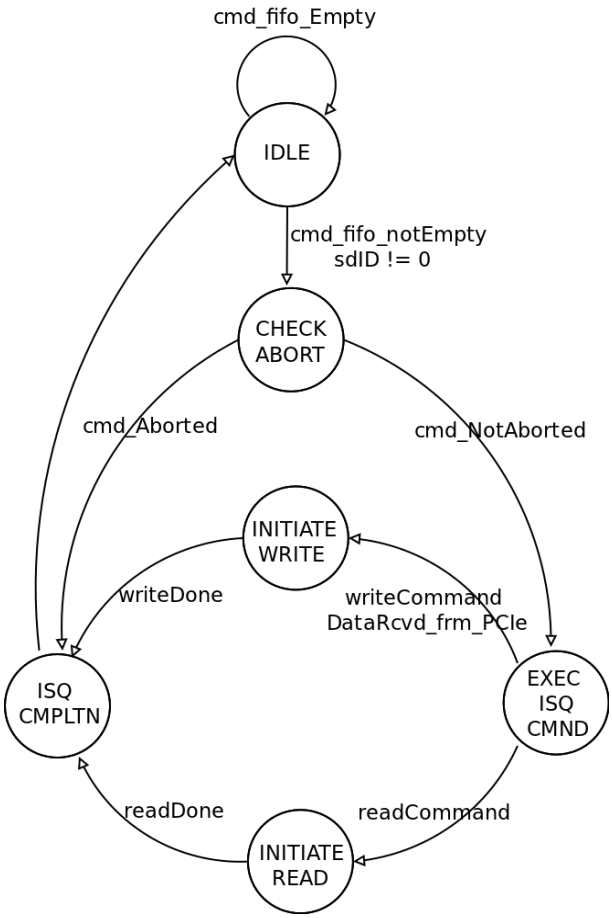

Figure 5.7: I/O Command Execution States

ISQ COMMAND CHECK ABORT STATE :

This state checks if the command is to be aborted or not. The Abort command in the Admin Command set decides which command is to be aborted. The command to be aborted is uniquely specified by the command ID and the Submission Queue ID. There is an "Abort Command List", with five entries to enable as many as five outstanding commands to be aborted. A sample list is shown on the table below.

If the "*Valid bit*" is marked Valid, then the corresponding command ID and Submission Queue IDs are compared with that of the present commands command ID and SQ ID respectively. If they match then the Command is aborted and the Valid-bit is marked InValid. Later the controller directly transitions to ISQ Completion State and sends the Completion

with status indicating that the command is Aborted. This check is performed with each of the entries which are marked Valid. If none of the entries match then the Command is ready to be executed and the controller transitions to ISQ Command Execution State.

Table 5.7: Sample Abort Command List for Check Abort State

Valid-bit	Command ID	SQ ID
Valid	20	1
In-Valid	18	3
In-Valid	45	1
Valid	57	5
Valid	59	4

ISQ COMMAND EXECUTION STATE :

This State executes the NVM Commands, namely Read and Write. If it is a read command then it immediately transitions to Initiate Read State. However, if it is a Wirte command then the controller raises the wire `wr_req_read_data`. The PCIe request State Machine acknowledges by raising the signal `rg_read_data_accepted`. The controller send request to send read TLP to the PCIe controller, and waits for the data to be filled in the data buffer. Once, the data buffer is filled, execute state machine transitions to Initiate Write State Machine.

INITIATE WRITE STATE

This state initiates the write operation to the Nand Flash Controller. The controller starts the Write operation in the Nand Flash Control State Machine by asserting the signal `rg_initiate_write`. Nand Flash Control state machine acknowledges the execution machine after sending the data in the NVMe buffer to the Nand Flash Controller buffer. Later the controller transitions to ISQ Command Completion State.

INITIATE READ STATE

This state initiates the read operation to the Nand Flash Controller. The controller asserts the signal `rg_initiate_read`, to start the Read operation in the Nand Flash Control State Machine. Nand Flash Control State Machine sends the read command to NFC and then comes to its default or idle state. When the NFC is ready with the data to be Read in its Data buffers, then it generates an interrupt to the controller. Upon reception of the interrupt, the Data Transfer State Machine is triggered to transfer the data from the NFC buffers to the Host Memory in the form of Write TLP payload. Controller then transitions to ISQ Completion State.

ISQ COMMAND COMPLETION

This state sends ISQ Command Specific Completions to the Host memory. The detailed explanation for command completion is provided in the next section.

5.2.6 Command Completion

Command Completions are a means by which the controller notifies the host about the status of each of the commands that it has issued. This logical module takes the status for every command, forms the command completion structure and then sends it to the host. There are four DWords in the completion structure.

- Completion DWord 0 is used only by Asynchronous Event Request Command to return asynchronous event information, and used by GetFeatures to return Feature information. This Dword is not used by any other command.
- Completion Dword 1 is reserved.
- Completion Dword 2 contains the Submission Queue ID and Submission Queue Head pointer value.

- Completion Dword 3 contains the command specific status value. This is tabulated in the table 5.8.

Table 5.8: Command Completion DWord 3 for each command

Command	**DWord 3**
Abort	success , Abort limit exceed
Asynchronous Event Request	NA
Create I/O CQ	success, invalid Q ID, Invalid Interrupt Vector, Max Q Size exceeded
Create I/O SQ	success, invalid CQ ID, Invalid Q ID, Max Q Size exceeded
Delete I/O CQ	success, invalid Q ID
Delete I/O SQ	success, invalid Q ID
Get Features	success, invalid Feature ID
Get Log Page	success, invalid LOG Page
Identify	success Invalid attributes
Set Features	success, invalid Feature ID
Read	success, Invalid attributes
Write	success, Invalid attributes

After sending the Completion the host is interrupted by sending a MSI interrupt. The details about the interrupt mechanism is explained in the next section.

5.2.7 Interrupts

The present adopted interrupt architecture allows for efficient reporting of interrupts such that the host may service the interrupts through the least amount of overhead.
According to the NVMe Specification, the controller can be choose one of the four ways to report interrupts. They are : pin-based interrupt, single message MSI, multiple message MSI, and MSI-X. The present controller implements multiple message MSI architecture. Multiple message MSI is a PCie feature, where a set of interrupt vectors (not more than 32) can be sent in a single message TLP to the Host. The details of MSI are given in the

PCIe controller implementation.

Interrupt aggregation is a means to mitigate host interrupt overhead by reducing the rate at which interrupts are generated by the controller. This reduced host overhead typically comes at the expense of increased latency.

INTERRUPT ARCHITECTURE

The architecture basically consists of a set of registers to facilitate the detection of an interrupt. It can generate up to 32 different interrupt vectors as per MSI. It also implements interrupt aggregation on a per vector basis.

Interrupt Registers

Interrupt Mask Register (IM) :

It is a 32 bit register. Each bit corresponds to one interrupt vector. If a bit is set high then that particular interrupt vector is "masked", else it is unmasked. Interrupt Mask Set (INTMS) and Interrupt Mask Clear (INTMC) registers in the controller register file are used by the host to Set or Clear a particular bit in the Interrupt Mask Register. For instance if the 29th bit in the INTMC register is set high then the 29th bit in the Interrupt Mask Register is cleared to zero.

Interrupt Status Register (IS) :

It is a 32 bit register and each bit corresponds to one interrupt vector. A particular bit in the register is set under the following conditions :

- There is one or more unacknowledged completion queue entries in a Completion Queue that utilizes this vector.
- The CQ has interrupts "enabled" in the "Create I/O Completion Queue" command.
- Corresponding interrupt vector is unmasked. If it is masked then there is no need to interrupt.

Interrupt Event

It is the mechanism which causes the controller to send an interrupt vector to the PCIe controller. A positive edge transition on the status register bit is in essence detected as an event, if the particular interrupt is unmasked. The situations where an interrupt event is detected is shown in the table 5.9.

Table 5.9: Interrupt Event

Mask register bit	Status Register bit	Event	Description
Steady 1	Will be steady 0	No Event	Interrupt Masked
Steady 0	Steady 0	No Event	No Interrupt
Transition from 1 to 0	Will be steady 0	No Event	No interrupt
Steady 0	Transition from 0 to 1	Event	Interrupt Sent

Interrupt Aggregation

The controller generates interrupt for every command completion. However, it can reduce the host overhead by aggregating a set of command completions onto a single vector. The host decides the number of completions to aggregate on a per vector basis, by indicating in the "Aggregation Threshold field". This is implemented by counting the number

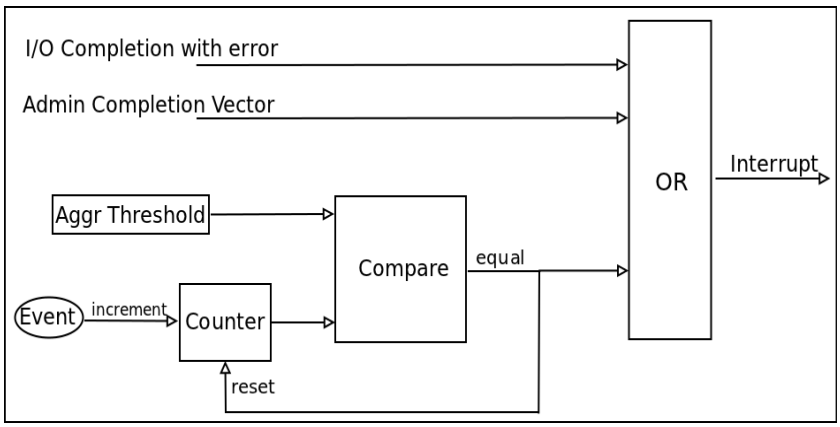

Figure 5.8: Interrupt Mechanism

of events on a single vector, and sends the interrupt when the count equals the aggregation threshold for that vector. Interrupt Aggregation is not applied to Admin Completion Queues and to those I/O Queues that complete in error. The figure 5.8 shows interrupt mechanism for a single vector.

5.2.8 Register File

The Register File implements the Controller registers as defined in the section 4.4. It provides read-write interface to the controller registers. All the Registers are accessed in their native widths. A 64 bit register is written as a 64 bit value. And when it is read, it sends 64 bits in two cycles with 32 bits in each cycle.

Register File Write

The Register File Interface provides control signals, address(32 bit) and data bus (64 bit wide) for write operation. To access a 64 bit register, all the 64 bits on the data bus should carry valid data. While writing a 32 bit register only 32 bits of the data bus are used by the controller, the other higher order bits are ignored. When the wire `wr_write` is high, the controller expects the address and data to be valid on the wries `wr_address` and `wr_data_in`.

Register File Read

The Register File Interface provides only the control signals and address (32 bit) for read operation. When the wire `wr_read` is set high, the controller sends the required register value through a Completion TLP. The completions are sent through the Request Interface.

5.2.9 Supporting State Machines

This subsection explains all the state machines that are necessary to support the logical modules that are described in the previous sections. These state machines primarily facilitate data transfer amongst PCIe, NVMe and the Nand Flash Controller.

1. Data Transfer State machine:

This state machine has the following functionality :

- Requests the PCIe Controller through the PCIe Request State Machine, to get the data page into the data buffers. This data is read into the data buffer as a part of write operation to the Nand Flash.
- The data received from the Nand Flash Controller is sent to the PCIe controller as Write TLP.

Various states that are involved in the State Machine are described below.

IDLE STATE:

As long as the `send_data_request` and `receive_data_request` as lowered the State Machine remains in the IDLE state. If `send_data_request` signal is asserted then, it transitions to REQ_DATA_FROM_PCIe state. If `receive_data_request` signal is asserted then, it transitions to REQ_PCIe_To_RECEIVE_DATA State.

REQ_PCIe_To_RECEIVE_DATA STATE

This state raises the signal `wr_req_write_data` high and waits for the acknowledgement from the PCIe Request State machine. PCIe Request State Machine acknowledges by raising the signal `rg_req_write_data_accepted` high. The state machine then transitions to WRITE_DATA_TO_PCIe State.

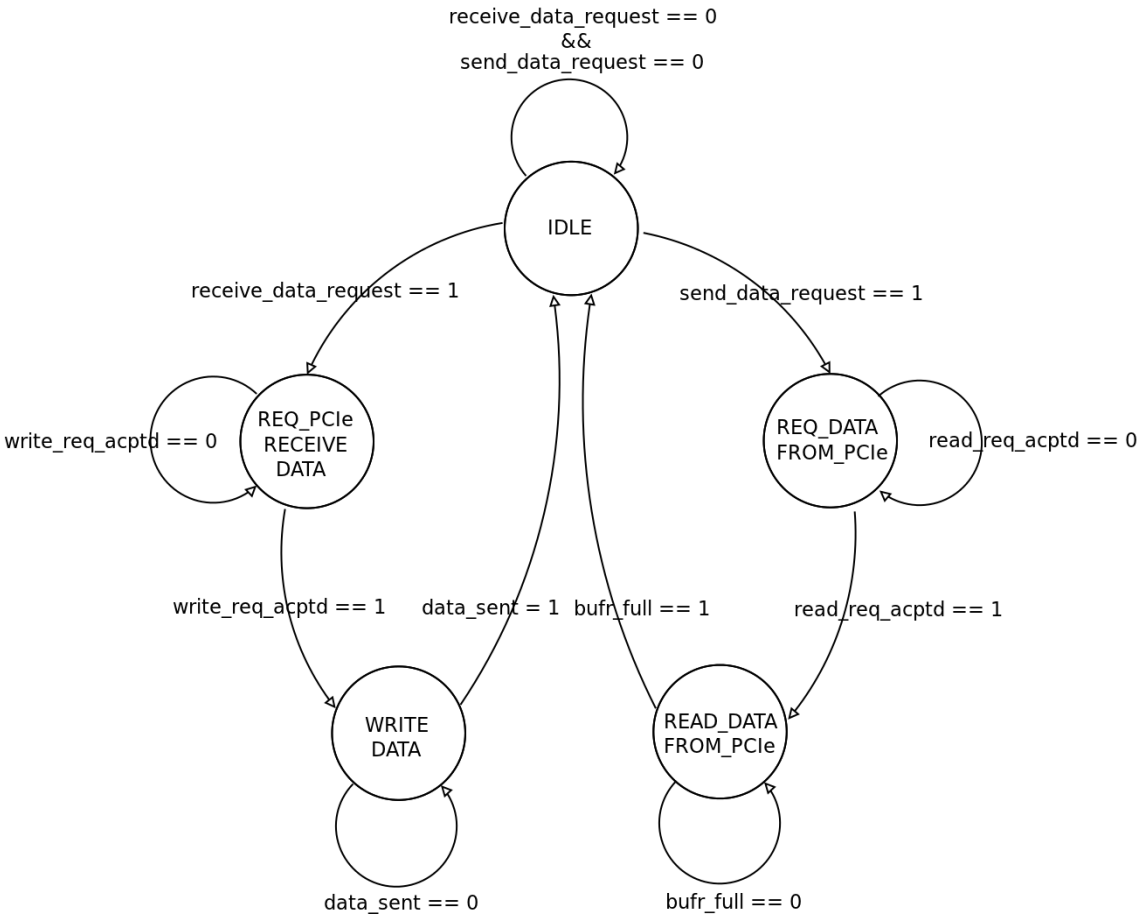

Figure 5.9: Data Transfer State Machine

WRITE_DATA_TO_PCIe STATE

The controller fetches the data from the NFC buffer on every cycle and sends it to the PCIe controller. This is done as long as the signal wr_send_valid_data from the PCIe Controller is high. The State machine transitions to IDLE State after sending the 4KB data.

REQ_DATA_FROM_PCIe STATE

This state raises the signal `wr_req_read_data` high and waits for the acknowledgement from the PCIe Request State machine. PCIe Request State Machine acknowledges by raising the signal `rg_req_read_data_accepted` high. The state machine then transitions to READ_DATA_FROM_PCIe State.

READ_DATA_FROM_PCIe STATE

This state keeps track of the status of the data buffer. The state machine remains in this state as long as the data buffer is not full. As soon as the data buffer is filled, `rg_buffer_full` is raised and the state machine transitions to IDLE State.

2. Pcie_Request_State_Machine

The controller generates various requests to the PCIe Controller. They include Read Data request, Write data request, Read Command Request, Write Completion and Send Completion TLP request. Any number of requests can be raised at a time by the controller. To arbitrate amongst the various requests, this state machine is implemented. The state machine takes various requests and acknowledges one of the requesting state machine based on the assigned priority.

IDLE STATE

This state waits for the requests, from within the controller, to access the PCIe controller. When multiple requests arise then the following priority is assigned :
"*write_completion* > *send_completion_tlp* > *write_data* > *read_data* > *read_command* > *read_prp*".

The next state is WRITE_COMPLETION_INFO if write_completion request is accepted. State transition to WRITE_DATA_INFO state occurs if write_data request is accepted. State transition to READ_DATA_INFO state occurs if read_data request is accepted. State transition to READ_COMMAND_INFO state occurs if read_command request is high. State transition to SEND_COMPLETION_TLP_INFO state occurs if send-cmpltn request is accepted. State transitions to READ_PRP_INFO state if read_prp_list request is high.

READ_COMMAND_INFO STATE

This state sends a read TLP request to the PCIe controller, with the following information : `read_tlp` is set high, `requested_tag` is set to 1 indicating it is expecting a command, `payload_length` is set to 16 as the command has 16 Dwords. It expects the state machine that has initiated this request to send the address information, from where to fetch the command, in this cycle. The state machine then jumps to wait state to give an extra cycle to send the request to the PCIe controller.

READ_DATA_INFO STATE

This state sends a read TLP request to the PCIe controller, with the following information : `read_tlp` is set high, `requested_tag` is set to 2 indicating it is expecting data , `payload_length` is set to 1024 as the data has 1024 Dwords (4KB). It expects the state machine that has initiated this request to send the address information, from where to fetch the data, in this cycle. The state machine then jumps to wait state to give an extra cycle to send the request to the PCIe controller.

WRITE_COMPLETION_INFO STATE

This state sends a write TLP request to the PCIe controller, with the following information : `write_tlp` is set high, `requested_tag` is set to 0 (not applicable for write), `payload_length` is set to 4 as the command completion is 4 Dwords. It expects the state machine that has initiated this request to send the address information, where to send the completion to, in this cycle. State transition to WRITE_COMPLETION state occurs.

WRITE_DATA_INFO STATE

This state sends a write TLP request to the PCIe controller, with the following infor-

mation : write_tlp is set high, requested_tag is set to 0 (not applicable for write), payload_length is set to 1024 as the data is 1024 Dwords (4KB). It expects the state machine that has initiated this request to send the address information, which points to the data-buffer location in the Main Memory. The state machine then jumps to WRITE_DATA state.

WRITE_DATA STATE

During this state the state machine which has requested to write data is expected to send the data to the PCIe controller. Upon sending the data, send_data_done signal is asserted and the state machine transitions to PCIe_REQ_IDLE state.

WRITE_COMPLETION STATE

During this state the state machine which has requested to write completion is expected to send the data to the PCIe controller. Upon sending the data, send_completion_done signal is asserted and the state machine transitions to PCIe_REQ_IDLE state.

SEND_COMPLETION_TLP_INFO

This state sends completion TLP request to the PCIe controller with following information : completion_tlp is set high, no tag information, payload_length is as specified by the register file (1 or 2). This is an intermediate state between IDLE and SEND_COMPLETION_TLP state and is not shown in the figure.

READ_PRP_INFO STATE

This state sends a read TLP request to the PCIe controller, with requested_tag , payload_length and address as specified by the *execution unit*.

SEND_COMPLETION_TLP

During this state the Register File send the completion data i.e. the register information. If register is 32 bit it sends in one cycle and if it is 64 bits it sends in two cycles. The assertion of the signal `cmpltn_tlp_sent` causes a state transition to IDLE state.

Figure 5.10: PCIe Request State Machine

3. Data and Command Acquire State Machine :

PCIe controller may not send the data (or command) completely in one transaction. It may take several transactions before the data is completely received. This state machine keeps a track of weather the data is completely received or not. This is the state machine that actually fills the “data buffer” and “command buffer” . The state machine is shown in the figure 5.11.

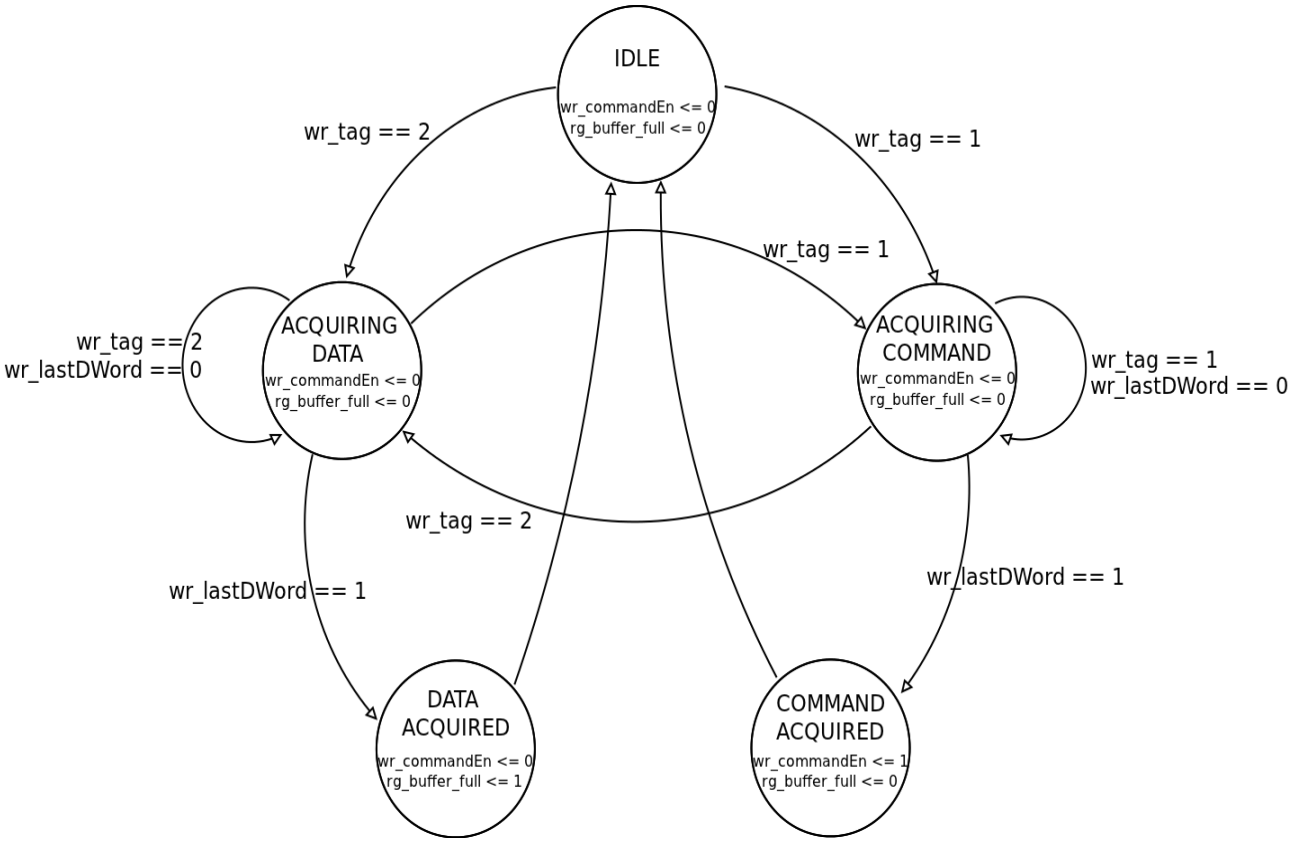

Figure 5.11: Data and Command Acquire State Machine

IDLE STATE

This states waits for the completions to be received from the PCIe controller. As long as the signal `wr_tag` is zero, it indicates that there are is no data from the PCIe controller. If `wr_tag` is 1 then it indicates that command is being sent by the PCIe, else if it is 2 then it indicates that data is being sent.

ACQUIRING_COMMAND STATE

This state is reached when wr_tag is 1. During this state the controller keeps buffering the command dword received, into the command buffer. The assertion of the signal `lastDWord` indicates that the transaction is complete and the command is completely received, the machine jumps to COMMAND_ACQUIRED state. If the `lastDword` is

not asserted and the wr_tag is changed to 2, then the state machine jumps to ACQUIRING_DATA state as the tag value 2 indicates data.

COMMAND_ACQUIRED STATE

During this state, the controller is notified that the command has been completely received and is present in wr_Command_in. The wire wr_CommandEn is raised high. State transition to IDLE state occurs.

ACQUIRING_DATA STATE

This state is reached when wr_tag is 2. During this state the controller keeps buffering the data dwords received, into the data buffer. The assertion of the signal lastDWord indicates that the transaction is complete and the data is completely received, and the machine jumps to DATA_ACQUIRED state. If the lastDword is not asserted and the wr_tag is changed to 1, then the state machine jumps to ACQUIRING_COMMAND state as the tag value 1 indicates command.

DATA_ACQUIRED STATE

During this state, the controller is notified that the data has been completely received and is present in the data buffer. The register rg_buffer_full is raised high.And the State transition to IDLE state occurs.

4. Nand Control State Machine

This state machine generates control and address signals to access the Nand Flash Controller. It writes read/write commands to the memory mapped controller registers of the Nand Flash Controller to initiate read/write operations respectively.

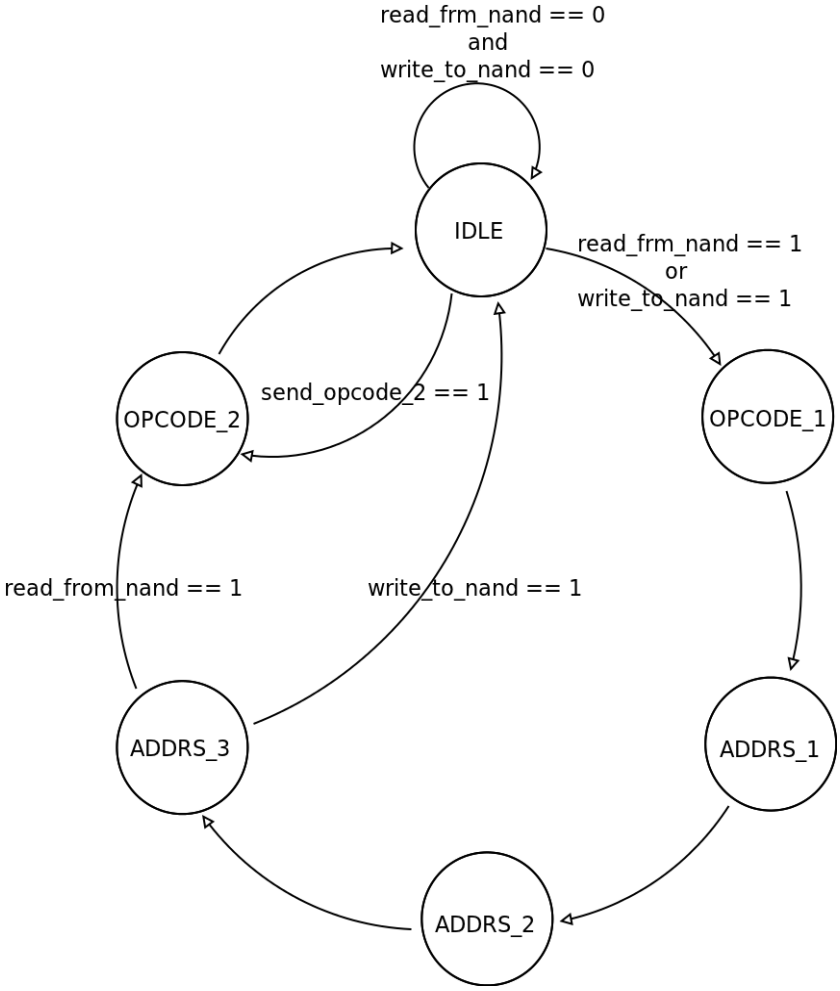

Figure 5.12: Nand Control State Machine

IDLE STATE

The state machine waits for the signals `read_frm_nand` or `write_to_nand` to be asserted . When one of the signals is asserted, state transition to OPCODE1 state occurs. There is no possibility for both the signals to be asserted at the same time.

OPCODE_1 STATE

The nand flash controller needs a start command and end command to initiate either read or write operation. Nand Chip enable and Write Enable are asserted and the opcode is written into the nand control register by supplying address of the “command register” in

the address lines and opcode in the data lines.

ADDRESS_1 STATE

Nand Flash requires the "page address" to be sent in three clock cycles. This state provides the first part of the page address.

ADDRESS_2 STATE

This state provides the second part of the page address.

ADDRESS_3 STATE

This state provides the third part of the page address. In the sequence of read operations the next state is OPCODE_2, to supply the End Command which initiates the read operations in the Nand Flash Controller. NVMe controller should then wait until the Nand Flash Controller gets the required data into its Data Buffers. Nand Flash Controller interrupts the NVMe controller indicating that it is ready with the requested page in its buffers. In the sequence in of Write operations, the state machine directly jumps to IDLE state after supplying the third part of address. The controller should then send the data to be written to the NFC buffers in the subsequent cycles, followed by the second opcode or end-command.

OPCODE_2

This state provides the End Command for read and write operations.

5.3 Design Challenges

Some of the challenges faced design decisions taken during the design are explained below. This section will be very helpful while improving the features of the present controller for future use.

Multiple Requests to PCIe controller from within the NVMe controller :

Because of the parallel execution of the Fetch, Command Execution and Register File units, there is a possibility for simultaneous requests for read-write data,read command, write completion and send completion TLP. In order to resolve this issue a PCIe Request State Machine was developed which responds to one of the requests on the basis of assigned priority. The decision to assign the following priority was taken with a view to provide more priority to complete existing commands than to acquire new commands for execution. The order is as follows : write_completion $>$ completion_tlp $>$ write_data $>$ read_data $>$ read_command.

Sequence of operations for Command Fetch and Command Arbitration :

Commands can be fetched and then arbitrated or first arbitrated and then fetched. If the commands are fetched into a buffer and then arbitrated, then we require additional buffers to store the fetched commands. However, if they are arbitrated and fetched, then the number of additional buffer spaces for storing the commands are minimized. Hence this option was chosen.

Abort Command Limits :

Every NVM Command has to be checked if it is to be aborted or not. This comparison adds a lot of hardware and hence limits the number of outstanding commands to be aborted. A decision to support five outstanding commands is taken.

5.4 Verification

The entire verification of the controller was carried out in the "*BlueSim* " simulator. The NVMexpress controller was integrated with Nand Flash Controller, which is also developed in BlueSpec, and the whole system was verified. The test setup used and the verified test cases are described in the following sub-sections.

5.4.1 Verification Setup

The verification setup for verifying the NVMe controller is shown in the figure 5.13.

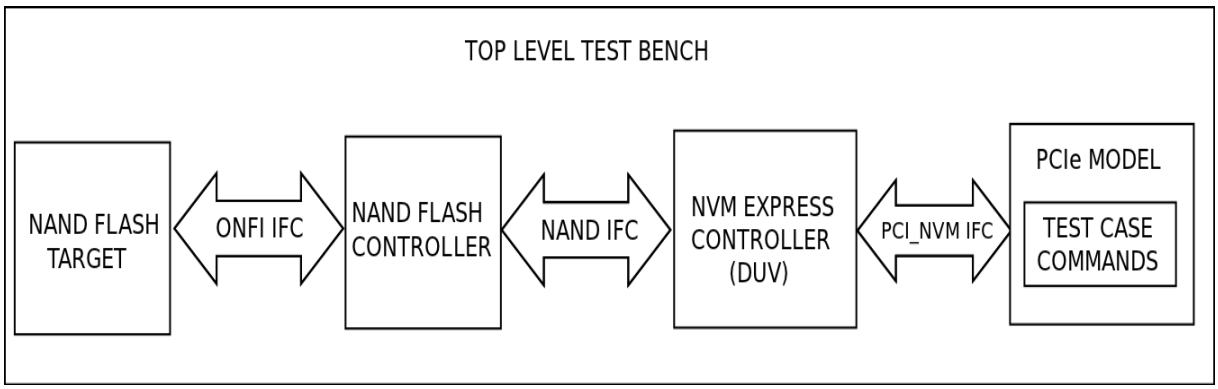

Figure 5.13: Verification Setup

NVM EXPRESS CONTROLLER

This is the present controller, which is the Design Under Verification (DUV).

NAND FLASH CONTROLLER

The NVMe controller was integrated with the Nand Flash Controller, in order to verify NVMe Controller's functionality. The Nand Flash Controller is a fully developed system with the source code in BlueSpec System Verilog.

NAND FLASH TARGET :

This is a functional model written in BSV to simulate the functionality of “NAND FLASH CHIPS” , with ONFI Interface.

PCIe MODEL :

This is a functional model written in BSV to abstract the functionality of the PCIe Controller and the PCIe Core in the form of Interface level Transactions . It has write data buffers which are used to store the data coming from the NVMe. This is used to verify the correctness of write operation. It also has a read buffer, which is initialized with some random data at the reset of the system. This can be used to verify the correctness of the Read command.

TEST CASE COMMANDS :

This is the memory model in BSV, which stores the required commands for NVM Express. This is accessed just as the Host Main Memory is accessed for commands from the submission queues.

5.4.2 Verification Test Cases

The controller has been verified for the following test cases :

1. Test cases for the I/O command set, that includes Write and Read commands.
2. Test cases for all the commands in the Admin Command set.
3. Test case to verify the Controller Initialization responses.

4. Test case to verify the Controller Shutdown responses. This includes both “Abrupt Shutdown” and “Normal Shutdown” .

5. Test case to verify the Read-Write operation to the Register File.

6. Test case to verify the “Arbitration Mechanism”.

7. Test cases to verify interrupt mechanism. This includes verification for Interrupt Aggregation for Successful I/O Commands.

8. Test cases to verify that the Aggregation is not applied to Admin Commands and to I/O commands that complete in error.

9. Test case for abort command with “Abort Command Limit Exceeded” error.

10. Creation of I/O CQ with :

    1. Invalid Q ID.
    2. Queue size greater than Max Queue size exceeded.
    3. Invalid Interrupt vector.

11. Creation of I/O CQ with :

    1. Invalid Q ID.
    2. Non-existent associated CQ.
    3. Max Queue size exceed.

12. Delete a CQ that does not exist.

13. Delete a SQ that does not exist.

5.5 Synthesis Report

The design was synthesized for the device *Virtex 6 XC6VLX240T-FF1156*. The slice utilization and timing summary is provided below.

Number of Slice Registers used : 4773

Number of Slice LUTs used : 6803

Number of LUT Flip Flop Pairs used : 2851

Number of Block RAMs used : 13

Max Frequency of operation : 204 MHz.

CHAPTER 6

Conclusions and Future Work

The NVM Express 1.0c Specification compliant controller was designed with all the basic functionality and mandatory command sets. As it was stated earlier that the NVMe Controller is designed as a PCIe device, hence a PCIe core and PCIe controller were also implemented. The PCIe controller was designed as a bridge between the Xilinx PCIe Endpoint Block and the NVMe controller. Both the controllers were individually verified for their functionality. PCIe controller was verified as a standalone controller between any PCIe device and Core. NVMe controller was verified with the implemented Nand Flash controller and the Transactional model of PCIe core and controller. Finally all the three controllers were integrated. The core and the three controllers form a basic working Non Volatile Memory system.

The NVM subsystem can be improved in the following ways :

1. NVMe command set can be enriched with some optional commands, if required.
2. An FTL(Flash Translation Layer) processor can be added to the subsystem to carry-out Flash Management activities at software level [7]. A brief detail about FTL and its necessity is provided in the appendix.

APPENDIX A

NVM Subsystem with FTL Processor

Flash Translation Layer is a very essential part of any SSD based Non Volatile Memory subsystem. There are three fundamental limitations for any SSD based system [7]. FTL helps in overcoming these limitations. The three limitations and the steps taken by FTL to overcome it or to hide it from the host are detailed below.

Limitation 1 : Erase-before-write

One of the limitations of the Flash memories is their "Erase before write" for re-writ ting into an already written location. This means that a page data can be written only into a location which is in erased state. FTL is primarily used to manage these activities by hiding them to the Host processor. The typical steps performed by the FTL while performing an over-write operation is shown below :

1. Takes the incoming logical page address, checks if the location pointed by the address in Nand Chip is in "erase" state or "written" state.
2. If it is in erase state then the page is written to that location.
3. If it is in "written" state then, it looks for a "new" location in the same block which is in "erased" state and maps its address to the present logical address. This new address is the actual physical address of the page. The incoming data page is written to this physical page address. There are various algorithms to perform this mapping operation.
4. Whenever the host gives a read command with the previous logical page address, the FTL fetches the page from its "mapped physical page address". In this way the host is unaware of the address translation.

Limitation 2 : Flash can be written in pages but can only be erased as a Block of pages

Flash memories can be written in units of pages, but they can be erased only in larger units called “blocks” of pages. If a particular set of pages are no longer required and are to be erased then the FTL performs a sequence of steps called “*Garbage Collection*”. The steps are detailed below.

1. If a set of pages in a block are no longer needed (stale pages) , then the FTL first reads the other “valid“ pages in that block.
2. Re-writes these valid pages into another ”erased“ block .
3. Then erases the previous block completely.

This is also implemented by the FTL unit, and there are various algorithms to perform it.

Limitation 3 : Flash can be ”Read“ or ”Written” only for a certain number of times

If a particular block were erased and programmed repeatedly without writing to any other blocks, the one block would wear out before all the other blocks, thereby prematurely ending the life of the SSD. For this reason, FTL implements a technique called “*wear leveling*“ to distribute writes as evenly as possible across all the flash blocks in the SSD.

Suggested Modifications to the present NVM Subsystem

The modified NVM Subsystem with the inclusion of the FTL processor is shown in the figure A.1

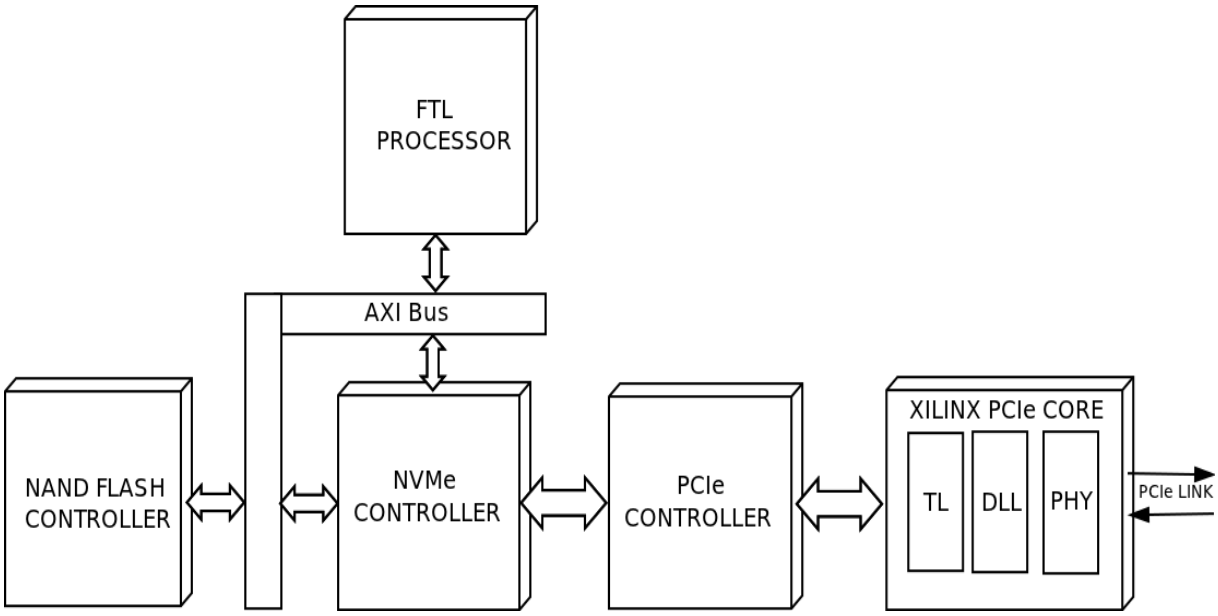

Figure A.1: Future Work on NVM Subsystem

The modifications that could be made are :

1. AXI wrappers could be put around the NVMe controller and Nand Flash Controller, so as to communicate with the FTL processor.
2. FTL processor takes the Logical Address from the NVMe controller and performs the Flash Management operations and sends the ”Control signals“ to the Nand Flash Controller and then initiate the required data transfer between NVMe controller and Nand Flash Controller.

REFERENCES

[1] J. S. Adam Wilen and R. Thornburg, *Introduction to PCI Express: A Hardware and Software Developer's Guide*. Intel Press, 2003.

[2] Xilinx, *Spartan-6 FPGA Integrated Endpoint Block for PCI Express*, ug672 ed., January 18, 2012.

[3] Bluespec, Inc, *Bluespec System Verilog Reference Guide*, revision: 17 ed., 2012.

[4] R. S. Nikhil and K. Czeck, *BSV by Example*. Bluespec, Inc, 2010.

[5] "Pci express base specification," April 29, 2002.

[6] A. Huffman, "Nvm express," February 16, 2012.

[7] L. C. Rino Micheloni and A. Marelli, *Inside NAND Flash Memories*. Springer, 2010.