ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

NVMe on Enzian

Semester Project

Shashank Anand

February 10, 2023

Advisors: Prof. Dr. Timothy Roscoe, Dr. Michael Giardino
Department of Computer Science, ETH Zürich

Abstract

No longer is the CPU the sole computation engine in a computer. Higher bandwidths, larger storage, and limited power budgets have lead to the development of what is known as *near memory* or *near storage computation*. With extremely power efficient CPUs and reprogrammable logic like FPGAs, more complicated operations can be pushed further towards storage. The Enzian research computer was developed to answer interesting architectural questions of future computing systems. It is imperative to provide the FPGA in Enzian access to the NVMe storage device to explore research questions related to future storage systems. This involves multiple steps including creating a PCIe root complex in the FPGA design, linking the PCIe root complex with the NVMe endpoint, and allowing the Microblaze CPU on the FPGA to interface with this root complex. We show that the NVMe device is operational. Preliminary tests show successful discovery of an Intel Optane SSD 900P Series NVMe device and a Geforce 9500GT GPU

Contents

Chapter 1

Introduction

Modern computers are undergoing a significant evolution, with a shift from traditional central processing unit (CPU) based architectures to more distributed architectures that utilize multiple computational units. One such trend is the move towards near memory processing, where computation is performed close to the memory where data is stored, instead of within the CPU.

There are several types of near memory processing units, including graphics processing units (GPUs), field-programmable gate arrays (FPGAs), and application-specific integrated circuits (ASICs). These units can be designed to perform specific tasks more efficiently than a general-purpose CPU and can be used in parallel with the CPU to offload certain types of computations.

One of the key advantages of FPGAs is that they can be located close to the storage and memory, allowing for computation to be performed in close proximity to the data. This can significantly reduce the time and energy required to move data between different parts of the system, leading to improved performance and energy efficiency.

FPGAs can also be designed to perform specific types of computations more efficiently than a general-purpose CPU, making them well suited for tasks such as image and video processing, cryptography, and machine learning. In these applications, FPGAs can provide the performance benefits of custom hardware, while still retaining the flexibility and programmability of software. The introduction of new memory technologies (e.g. Optane, HBM) as well as different models of communication (e.g. CXL), leads to many interesting questions about the architecture of such systems.

Enzian [1] is a novel research computer designed and developed by the Systems Group at ETH Zurich to answer important research questions of the future. Enzian is a two socket server in which one socket has a Cavium/Marvell 48-core ThunderX (ARMv8) and the other socket has a Xilinx Ultrascale+

FPGA. The two sockets are connected by ECI a 30 GB/s coherence link that runs a protocol compatible with the ThunderX's native cache coherence protocol. Both sides have ample DRAM (128 GiB on CPU and up to 1 TiB on FPGA), network bandwidth (2 x 40 Gbps on CPU and 4 x 100 Gbps on FPGA). Interesting for this project are the NVMe ports available on both CPU and FPGA.

The first step to implementing a platform for examining smart storage on Enzian is to enable the NVMe on the FPGA. The objective of this report is to describe in detail the engineering challenges encountered in the process of enabling the NVMe device on the FPGA.

In chapter 2, we discuss the background knowledge required and terminology used in the remainder of the report. Chapter 3 discusses the steps involved in enabling the NVMe and provides a detailed documentation to facilitate reproduction of this work. Finally, chapter 4 concludes with a discussion of potential future work. Reference code used in the report can be found in the appendix.

Chapter 2

Background

In this chapter we provide a brief discussion of the necessary background knowledge for terminology used throughout the remainder of the report.

2.1 FPGA

Field-Programmable Gate Arrays (FPGAs) are a type of reconfigurable integrated circuit (IC) that can be programmed to perform a wide range of digital logic functions. They offer a high degree of flexibility and adaptability, as their digital logic circuits can be dynamically reconfigured to perform specific computation tasks, making them a powerful tool for various applications.

FPGAs are implemented using a combination of programmable digital logic circuits, such as look-up tables (LUTs) and programmable interconnect points (PIPs), and programmable routing resources. The digital logic circuits can be programmed to perform specific computation tasks, while the routing resources can be used to route signals between different parts of the circuit.

2.2 Xilinx Virtex Ultrascale+ FPGA

The Virtex Ultrascale+ FPGA offered by Xilinx[2] is a high-performance, highly flexible programmable logic device. It is part of the Xilinx Ultrascale+ FPGA family and is designed to meet the demands of advanced computing applications, such as high-performance computing, artificial intelligence, and data center acceleration.

Of most relevance for us are the capabilities of the FPGA to instantiate a PCIe root complex.

2.3 Intellectual Property (IP) Core

An Intellectual Property (IP) core, is a pre-designed digital logic circuit that can be integrated into an FPGA or ASIC design to perform a specific function or set of functions. IP cores are widely used in digital system design, as they provide a convenient and efficient way to add functionality to a system without having to design the circuit from scratch.

2.4 Microblaze

The MicroBlaze CPU is a 32-bit soft processor core developed by Xilinx[3] for use in their FPGA devices. It is particularly useful as it is implemented using a small number of FPGA logic elements and memory. In this project, we primarily use the microblaze to run test programs and interface with other IPs.

2.5 I2C

I2C (Inter-Integrated Circuit) is a serial communication protocol used for communication between devices in an electronic system. It allows for the transfer of data between devices over a two-wire bus, with the wires referred to as SCL (serial clock) and SDA (serial data).

2.6 GPIO

GPIO (General-Purpose Input/Output) is a type of digital input/output pin found on microcontrollers, single-board computers, and other digital devices. GPIO pins are used to interface with external devices and circuits, allowing the microcontroller to control or receive information from these devices.

2.7 PCIe

PCIe (Peripheral Component Interconnect Express)[4] is a high-speed serial computer expansion bus standard that is used to connect peripheral devices to a computer's motherboard. PCIe uses lanes to transfer data from one device to another. A lane consists of two differential pairs used to transmit or receive data. PCIe supports 1, 2, 4, 8 and 16 lane configurations. PCIe also supports multiple transfer speeds at 2.5, 5, 8, 16 or 32 GT/s

2.7.1 Root Complex

The PCIe root complex is a central component in a PCIe system that acts as the interface between the processor or chipset and the PCIe bus. The

root complex is responsible for managing the communication between the processor or chipset and the attached devices on the PCIe bus.

2.7.2 Link Training

PCIe devices are initialized by the root complex by a process known as link training. Link training negotiates lane width and link speed, and discovers lane reversals and lane swaps.

2.7.3 PCIe Enumeration

Enumeration is the process of discovering and initializing PCIe devices. During PCIe enumeration, scans the PCIe bus for attached devices, identifies the type of each device, and assigns resources (such as memory ranges and I/O addresses) to each device.

2.8 NVMe

NVMe (Non-Volatile Memory Express) is a high-performance, scalable, and efficient interface specification for accessing non-volatile memory (NVM) such as flash storage devices. It was designed specifically for use with solid-state drives (SSDs) and provides a new command set and feature set optimized for NVM. NVMe devices communicate over the PCIe bus.

Chapter 3

Process

In this section we describe the process of bringing up and configuring the PCIe IP block, and subsequently enumerating the connected devices. We discuss major problems faced during the process and engineered solutions.

The process begins with programming a PCIe Root Complex into the FPGA, as there is none. We then add a Microblaze CPU to communicate with the NVMe device via the PCIe Root Complex. We also required communicating with the IO Extender over I2C. Finally, we run a test program and show that the NVMe device is enumerated successfully.

3.1 Vivado Project Setup

Create a new Vivado RTL Project with no sources. Select the part **xcvu9p-flgb2104-3-e**. Create a new block design.

3.2 PCIe XDMA

We start by instantiating the DMA for PCI Express (PCIe) Subsystem (also called PCIe XDMA) [5]. This block will act as the PCIe root complex and will talk to the NVMe device acting as a PCIe endpoint.

Add the IP from the catalog. Run block automation. Rename the created *diff_clock_rtl_0* port to *pcie*.

The XDMA IP is configured as follows. The options not mentioned in the following list must be unchanged from their default configuration.

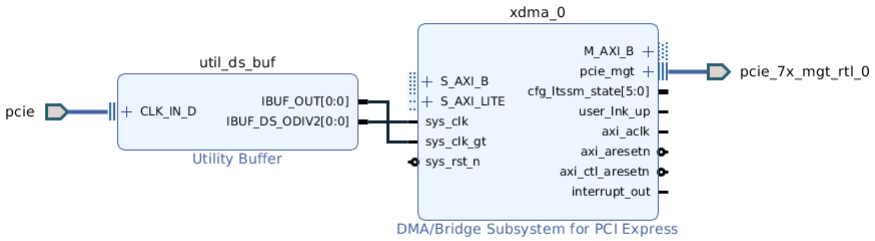
1. Set *Function Mode* to **AXI Bridge** and *Mode* to **Advanced**
2. Set *Device / Port Type* to **Root Port of PCI Express Root Complex**
3. Set *PCIe Block Location* to **X1Y4**. Check *Enable GT Quad Selection* and select *GT Quad* as **GTY Quad 229**

4. Set *Lane Width* to **x4** and *Maximum Link Speed* to **8.0GT/s**
5. In the *PCIe BARs* tab, enable **64 bit** and **Prefetchable**

Steps 1 and 2 are done to set our PCIe XDMA block to act as a root complex as the block also allows endpoint mode of operation. Step 3 points to the exact GT that the block is configured with. This information is gathered from the Enzian board specifications document. Step 4 refers to settings of the NVMe port (NVMe communicates over a 4 lane PCIe bus). Finally Step 5 is required because the PCIe XDMA drivers for the microblaze required prefetchable 64 bit BARs.

At the end of these steps, the block diagram looks like Figure 3.1

Figure 3.1: PCIe XDMA IP



3.3 Microblaze

Now, we add the microblaze to this block diagram to interface with the PCIe block.

Add the microblaze to the block design. Run block automation and check *Interrupt Controller*. Run connection automation, and rename the external clock port to **clk_100**.

Replace the AXI Interconnect with the AXI Smartconnect. We use the AXI SmartConnect as it saves us the trouble of manually connecting the clock signal for each AXI Slave or Master connection separately. Connect the clock output of the newly created clock wizard and the **axi_aclk** output of the PCIe XDMA to the AXI Smartconnect.

Configure the Microblaze Debug Module and enable *JTAG UART*. Run connection automation, and connect the debug module to the AXI SmartConnect.

Finally, add a Concat from the catalog, and connect the **interrupt_out** of the PCIe XDMA to the input of the Concat block. Connect the output of this Concat to the **intr** input of the interrupt controller. We use a Concat now because in the upcoming steps we will be connecting more interrupts to the interrupt controller.

We now have a design with a Microblaze module and a PCIe XDMA which are connected together and can interface with each other. At the end of this step, the block diagram looks like Figure 3.3

3.4 Problems

This design however fails to discover the connected NVMe device. From debug information, we observe that the failure happens during link training of the PCIe Root Complex and the NVMe endpoint, at the very first **Detect** state. We conclude from this information that as link training is never successfully completed, the link is not configured and as a result we are unable to detect the NVMe endpoint device.

From the link training steps, we see that the PCIe endpoint must cycle the PCIe Reset signal [4] to signal to the PCIe Root Complex that the endpoint is ready to initiate link training. In the Enzian mainboard, this is the *FN_PERST_N* signal for the NVMe, and the *F_PCIE16_PERST_N* signal for the x16 PCIe. However, we observe from the schematics that this signal is not accessible directly by the FPGA. So, the reset signal is never toggled and the Root Complex does not initiate link training. The PCIe reset signal is connected to the *GPIO_Extender* component, which is in turn connected to the FPGA via a level shifter.

Therefore, the next step is to add the relevant components to our FPGA design to communicate with the GPIO extender and cycle the PCIe reset signal.

3.5 GPIO Extender

The Texas Instruments TCA6424ARGJR [6] IO extender is used in the Enzian board to expand the available GPIO pins. It provides 28 output ports. Communication with the GPIO extender happens over I2C. By sending specific command and data bytes via I2C, we set and reset the output signals.

3.5.1 GPIO Extender Protocol

The GPIO extender requires specifying the operation using a command byte [6]. The command byte specifies the register being accessed, and the type of operation (Read/Write) being performed.

If a read operation is being performed, a command byte with the appropriate register information and the R/W bit set is sent. Then, the value of the register requested is received.

For a write operation, a command byte with the register information and the R/W bit cleared is sent. Then, a byte which contains the value of the register to be written is also sent.

3.5.2 Block Design

We start by adding the required components to our design to communicate over I2C.

Add the *AXI IIC* block from the IP catalog and run connection automation. A new external port **iic_rtl_0** is created, and the AXI IIC block is connected to the clock wizard and to the AXI SmartConnect. Additionally, the interrupt out of the AXI IIC block should be connected to the Concat block created in section 3.3.

Finally, an external port is created for the GPO pin in the AXI IIC block. This is because, the reset of the GPIO extender has to be toggled explicitly to enable the component. The reset is referred to by *F_I2C5_RESET_N* in the Enzian mainboard schematics.

The final block design is shown in figure 3.4.

3.6 Toggle PCIe Reset

Now we are ready to communicate with the GPIO extender and toggle the PCIe reset signal. We communicate with the I2C block over AXI using the microblaze CPU. We describe how to run the test program on the microblaze.

We first generate the bitstream in Vivado, and subsequently export and save the hardware platform. Create a new project in Vitis, and import the saved hardware platform. Using Vitis we program the microblaze to communicate with AXI IIC block using the Xilinx IIC driver [7].

From the Enzian schematics, we see that *FN_PERST_N* is connected to **P26** of the GPIO extender. From the datasheet [8], the 7th bit of register 2 contains **P26**. The command bytes to access the configuration and output registers are `0x06` and `0x0E` respectively. Therefore, we read the values of the config and output registers, and write them back with the updated values of the 7th bit. The process is as follows:

1. Read the value of Configuration Port 2
2. Set the 7th bit of the byte to be `1` and write back to Configuration Port 2

3. Read the value of Output Port 2
4. Set the 7th bit of the byte to be `0` and write back to Output Port 2
5. Read the value of Output Port 2
6. Set the 7th bit of the byte to be `1` and write back to Output Port 2

To streamline this process, we have written a driver to communicate with the GPIO extender. The code can be found in the Appendix A.

Using debug information from the VIO core [9], we notice that as soon as the reset is cycled, link training successfully completes and the link is up.

3.7 Enumeration

We are finally ready to enumerate the NVMe device. We use the PCIe enumeration program as provided by Xilinx [10]. We first cycle the PCIe reset to complete link training, and then we perform enumeration of the devices. We observe that the NVMe device is successfully discovered.

```
Root Complex IP Instance has been successfully initialized
xdma_pcie:
PCIeBus is 00
PCIeDev is 00
PCIeFunc is 00
xdma_pcie: Vendor ID is 10EE
Device ID is 9134
xdma_pcie: This is a Bridge
xdma_pcie: bus: 0, device: 0, function: 0: BAR 0, ADDR: 0x0 size : 0K
xdma_pcie:
PCIeBus is 01
PCIeDev is 00
PCIeFunc is 00
xdma_pcie: Vendor ID is 8086
Device ID is 2700
xdma_pcie: This is an End Point
xdma_pcie: bus: 1, device: 0, function: 0: BAR 0, ADDR: 0x20000 size : 0K
xdma_pcie: bus: 1, device: 0, function: 0: BAR 2 is not implemented
xdma_pcie: bus: 1, device: 0, function: 0: BAR 3 is not implemented
xdma_pcie: bus: 1, device: 0, function: 0: BAR 4 is not implemented
xdma_pcie: bus: 1, device: 0, function: 0: BAR 5 is not implemented
xdma_pcie: End Point has been enabled
Successfully ran XdmaPcie rc enumerate Example
```

Figure 3.2: NVMe device enumerated

The device vendor ID is report as 8086 which corresponds to Intel. The device ID is 2700 which correctly refers to an Intel Optane SSD 900P Series device [11].

Figure 3.3: Microblaze connected to the PCIe XDMA IP

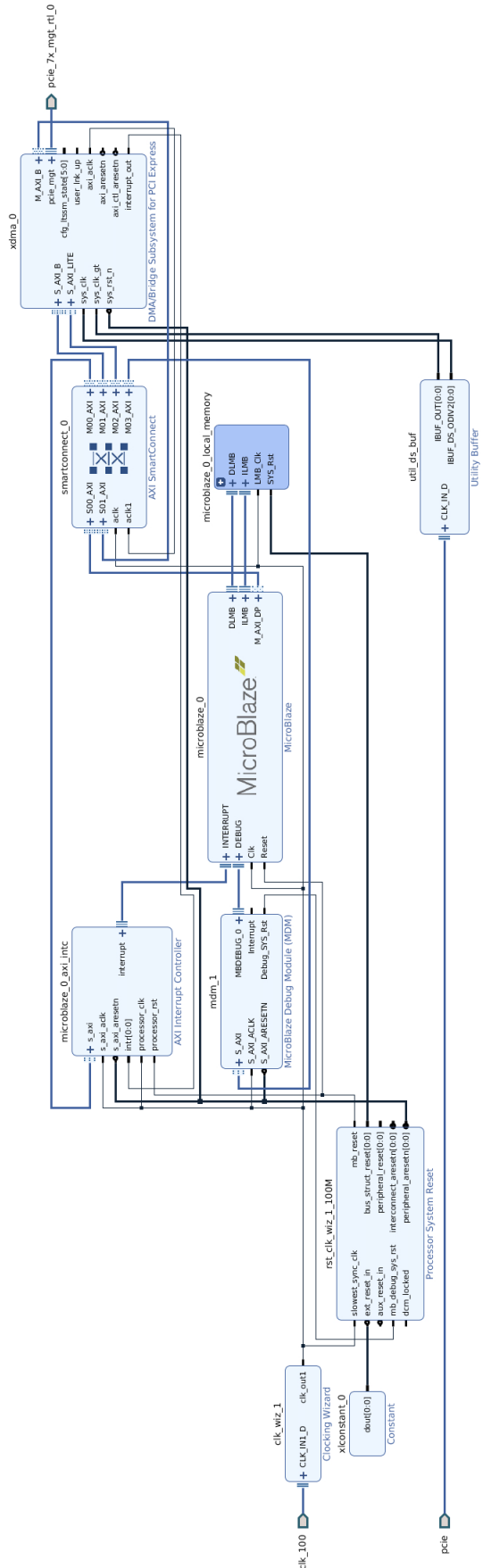
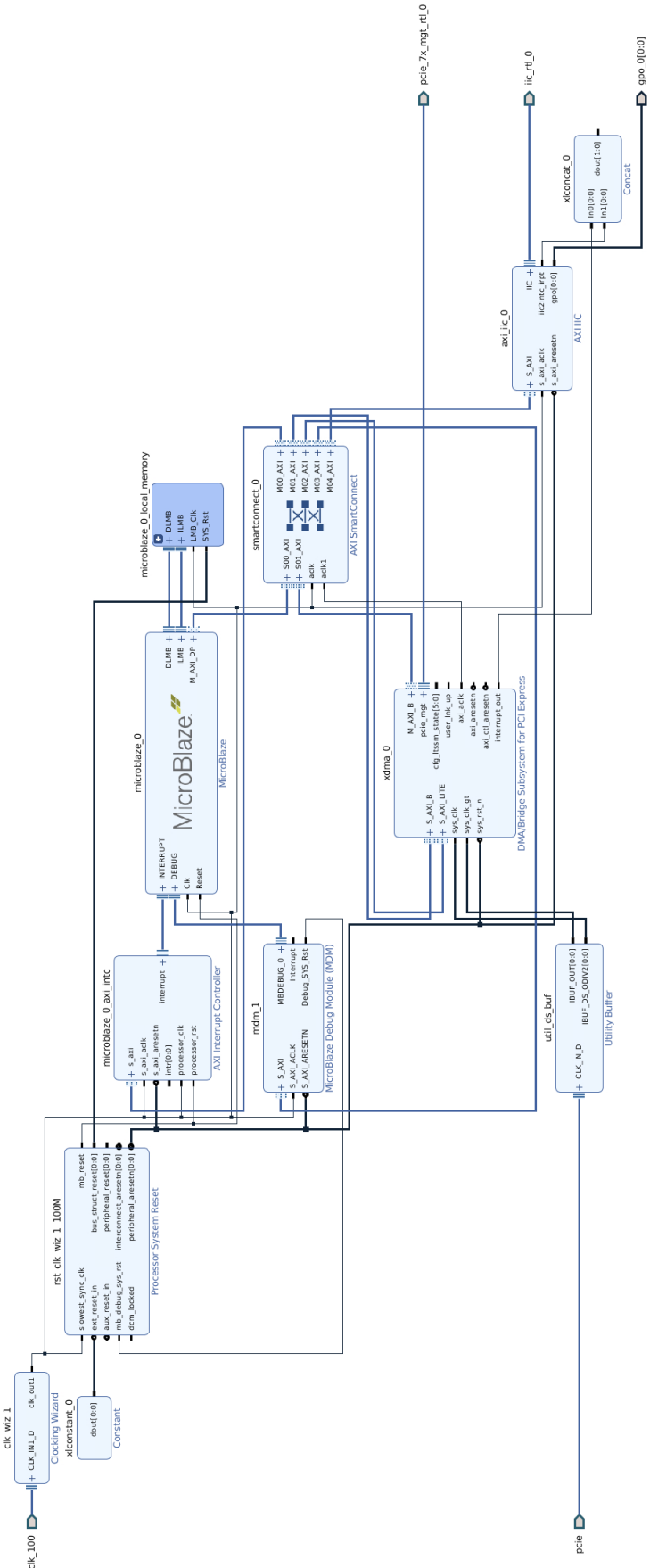


Figure 3.4: Design extended with AXI IIC



Chapter 4

Conclusion

We have successfully programmed and configured PCIe root complex in the FPGA and enabled the Intel Optane SSD 900P Series NVMe device. Additionally, by following the same process as outlined we have also managed to discover an Nvidia Geforce 9500GT GPU as seen in figure 4.1.

```
Root Complex IP Instance has been successfully initialized
xdma_pcie:
PCIeBus is 00
PCIeDev is 00
PCIeFunc is 00
xdma_pcie: Vendor ID is 10EE
Device ID is 9121
xdma_pcie: This is a Bridge
xdma_pcie: bus: 0, device: 0, function: 0: BAR 0, ADDR: 0x0 size : 0K
xdma_pcie: bus: 0, device: 0, function: 0: BAR 1 is not implemented
xdma_pcie:
PCIeBus is 01
PCIeDev is 00
PCIeFunc is 00
xdma_pcie: Vendor ID is 10DE
Device ID is 0640
xdma_pcie: This is an End Point
xdma_pcie: bus: 1, device: 0, function: 0: BAR 0, ADDR: 0x20000 size : 0K
```

Figure 4.1: GPU enumerated[12]

The debugging process while summarized in a few paragraphs in the report was long and arduous. Debugging problems in the FPGA was performed using devices such as the oscilloscope, and Xilinx tools such as the VIO[9], the IBERT[13], and the ILA[14].

The enabling of the NVMe on the FPGA provides scope for examining smart storage systems on Enzian. The next step would be to provide access to the NVMe from the CPU across ECI[15]. Future work may also involve benchmarking the performance of accessing the NVMe through the FPGA.

Appendix A

Reference Code

Listing A.1: "gpio_ext.h"

```
/*
 * Driver to interface with the gpio extender TCA6424A
 */

#include "xiic.h"
#include "xintc.h"
#include "xil_exception.h"
#include "xil_printf.h"

int Gpio_Ext_Init(XIntc *InterruptController, u16 IicDeviceId, u8 IicIntId, int GpioExtAddr);
int Gpio_Ext_Stop(void);
int Gpio_Ext_Set_Gp(u8 Val);
int Gpio_Ext_Read_Reg(u8 CommandByte, u8 *Val);
int Enzian_Initialize_Cycle_PCIe_RST(XIntc *InterruptController, u16 IntDeviceId,
                u16 IicDeviceId, u8 IicIntId);
```

Listing A.2: "gpio_ext.c"

```
/*
 * Driver to interface with the gpio extender TCA6424A
 */

#include "gpio_ext.h"

#define ENZIAN_GPIO_EXT_ADDR     0x22

XIic IicInstance;

volatile u8 TransmitComplete;
volatile u8 ReceiveComplete;

/*
 * Handler invoked when IIC send completes
 */
static void SendHandler(XIic *InstancePtr)
{
        TransmitComplete = 0;
}

/*
 * Handler invoked when IIC receive completes
 */
static void ReceiveHandler(XIic *InstancePtr)
{
        ReceiveComplete = 0;
}

/*
```

```
 * Handler invoked when IIC event is received
 */
static void StatusHandler(XIic *InstancePtr, int Event)
{
        xil_printf("IIC got event 0x%x\n", Event);
}

/*
 * helper function to setup interrupts
 * NOTE: Does NOT initialize or start the interrupt controller
 */
static int SetupInterruptSystem(XIntc *InterruptController, XIic *IicInstPtr, u8 IicIntId)
{

        int Status;

        //Connect interrupt handler
        Status = XIntc_Connect(InterruptController, IicIntId,
                               (XInterruptHandler) XIic_InterruptHandler,
                               IicInstPtr);
        if (Status != XST_SUCCESS) {
                return XST_FAILURE;
        }

        //Enable the interrupts for the IIC device.
        XIntc_Enable(InterruptController, IicIntId);

        //Initialize the exception table.
        Xil_ExceptionInit();


        //Register the interrupt controller handler with the exception table.
        Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
                                (Xil_ExceptionHandler) XIntc_InterruptHandler,
                                InterruptController);

        //Enable non-critical exceptions.
        Xil_ExceptionEnable();

        return XST_SUCCESS;
}


/*
 * Initializes the GpioExtender driver. Starts the IIC device
 * NOTE: It is up to the caller to initialize the interrupt controller
 */
int Gpio_Ext_Init(XIntc *InterruptController, u16 IicDeviceId, u8 IicIntId, int GpioExtAddr) {

        int Status;
        XIic_Config *ConfigPtr; /* Pointer to configuration data */

        //Initialize the IIC driver so that it is ready to use.
        ConfigPtr = XIic_LookupConfig(IicDeviceId);
        if (ConfigPtr == NULL) {
                return XST_FAILURE;
        }

        Status = XIic_CfgInitialize(&IicInstance, ConfigPtr,
                                                ConfigPtr->BaseAddress);
        if (Status != XST_SUCCESS) {
                return XST_FAILURE;
        }


        //Setup the Interrupt System.
        Status = SetupInterruptSystem(InterruptController, &IicInstance, IicIntId);
        if (Status != XST_SUCCESS) {
                return XST_FAILURE;
        }

        //Set the Transmit, Receive and Status handlers.
        XIic_SetSendHandler(&IicInstance, &IicInstance,
                                (XIic_Handler) SendHandler);
```

```
        XIic_SetRecvHandler(&IicInstance, &IicInstance,
                            (XIic_Handler) ReceiveHandler);
        XIic_SetStatusHandler(&IicInstance, &IicInstance,
                              (XIic_StatusHandler) StatusHandler);

        //Set the Address of the Slave.
        Status = XIic_SetAddress(&IicInstance, XII_ADDR_TO_SEND_TYPE, GpioExtAddr);
        if (Status != XST_SUCCESS) {
                return XST_FAILURE;
        }

        Status = XIic_Start(&IicInstance);
        if (Status != XST_SUCCESS) {
                return XST_FAILURE;
        }

        //This line is really important. Even the first message doesn't work without this
        //DO NOT REMOVE
        IicInstance.Options = XII_REPEATED_START_OPTION;

        return XST_SUCCESS;
}

/*
 * Stop the IIC device
 * Must be called after usage completion
 */
int Gpio_Ext_Stop(void) {

        int Status;

        Status = XIic_Stop(&IicInstance);
        if(Status != XST_SUCCESS) {
                return XST_FAILURE;
        }

        return XST_SUCCESS;
}

/*
 * Set Gpio connected to the IIC IP.
 * This should be connected to the IIC reset to wake the device up from reset.
 */
int Gpio_Ext_Set_Gp(u8 Val) {

        int Status;

        Status = XIic_SetGpOutput(&IicInstance, Val);
        if (Status != XST_SUCCESS) {
                return XST_FAILURE;
        }

        return XST_SUCCESS;
}
/*
 * Read an 8 bit register from the device
 * Blocking
 */
int Gpio_Ext_Read_Reg(u8 CommandByte, u8 *Val) {

        int Status;

        TransmitComplete = 1;
        Status = XIic_MasterSend(&IicInstance, &CommandByte, 1);
        if (Status != XST_SUCCESS) {
                return XST_FAILURE;
        }

        while(TransmitComplete);

        ReceiveComplete = 1;
        Status = XIic_MasterRecv(&IicInstance, Val, 1);
        if (Status != XST_SUCCESS) {
                return XST_FAILURE;
```

```
        }

        while(ReceiveComplete);

        return XST_SUCCESS;
}

/**
 * Write an 8 bit register to the device
 * Blocking
 */
int Gpio_Ext_Write_Reg(u8 CommandByte, u8 Val) {

        int Status;

        u8 WriteBuffer[2];
        WriteBuffer[0] = CommandByte;
        WriteBuffer[1] = Val;

        TransmitComplete = 1;
        Status = XIic_MasterSend(&IicInstance, WriteBuffer, 2);
        if (Status != XST_SUCCESS) {
                return XST_FAILURE;
        }

        while(TransmitComplete);

        return XST_SUCCESS;
}

/**
 * Function to save time.
 * Initializes and starts the interrupt controller, starts the device,
 * cycles PCIe Reset (of both the NVMe and the 16x) and stops the device
 */
int Enzian_Initialize_Cycle_PCIe_RST(XIntc *InterruptController, u16 IntDeviceId,
                u16 IicDeviceId, u8 IicIntId) {

        int Status;

        Status = XIntc_Initialize(InterruptController, IntDeviceId);
        if (Status != XST_SUCCESS) {
                return XST_FAILURE;
        }

        //Initialize with known gpio extender device address
        Status = Gpio_Ext_Init(InterruptController, IicDeviceId, IicIntId, ENZIAN_GPIO_EXT_ADDR);
        if (Status != XST_SUCCESS) {
                return XST_FAILURE;
        }

        Status = XIntc_Start(InterruptController, XIN_REAL_MODE);
        if (Status != XST_SUCCESS) {
                return XST_FAILURE;
        }

        //Wake up the IIC device from reset. On the enzian board the IIC reset is inverted
        Status = Gpio_Ext_Set_Gp(1);
        if (Status != XST_SUCCESS) {
                return XST_FAILURE;
        }

        u8 Reg2ConfigCommandByte = 0x0E, Reg2OutputCommandByte = 0x06;
        u8 Reg2ConfigVal, Reg2OutputVal;

        //8th and 7th bit respectively are F_PCIE16_PERST_N and FN_PERST_N
        u8 Mask = 0b11 << 6;

        //Read state of config register
        Status = Gpio_Ext_Read_Reg(Reg2ConfigCommandByte, &Reg2ConfigVal);
        if (Status != XST_SUCCESS) {
                return XST_FAILURE;
        }
```

```
        //Write back setting 8th and 7th bit to output mode (LOW)
        Status = Gpio_Ext_Write_Reg(Reg2ConfigCommandByte, Reg2ConfigVal & (~Mask));
        if (Status != XST_SUCCESS) {
                return XST_FAILURE;
        }

        //Read state of output register
        Status = Gpio_Ext_Read_Reg(Reg2OutputCommandByte, &Reg2OutputVal);
        if (Status != XST_SUCCESS) {
                return XST_FAILURE;
        }

        //Write 0 to outputs (trigger reset)
        Status = Gpio_Ext_Write_Reg(Reg2OutputCommandByte, Reg2OutputVal & (~Mask));
        if (Status != XST_SUCCESS) {
                return XST_FAILURE;
        }

        //Write 1 to outputs (bring out of reset)
        Status = Gpio_Ext_Write_Reg(Reg2OutputCommandByte, Reg2OutputVal | Mask);
        if (Status != XST_SUCCESS) {
                return XST_FAILURE;
        }

        Status = Gpio_Ext_Stop();
        if (Status != XST_SUCCESS) {
                return XST_FAILURE;
        }

        return XST_SUCCESS;
}
```

Bibliography

[1] D. Cock, A. Ramdas, D. Schwyn, M. Giardino, A. Turowski, Z. He, N. Hossle, D. Korolija, M. Licciardello, K. Martsenko, R. Achermann, G. Alonso, and T. Roscoe, "Enzian: An open, general, cpu/fpga platform for systems software research," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 434–451. [Online]. Available: https://doi.org/10.1145/3503222.3507742

[2] "Virtex ultrascale+." [Online]. Available: https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus.html

[3] "Microblaze soft processor core." [Online]. Available: https://www.xilinx.com/products/design-tools/microblaze.html

[4] M. Jackson and R. Budruk, *PCI Express Technology: Comprehensive Guide to Generations 1.x, 2.x, 3.0*. MindShare, 2012.

[5] "Dma for pci express (pcie) subsystem." [Online]. Available: https://www.xilinx.com/products/intellectual-property/pcie-dma.html

[6] "Tca6424argjr." [Online]. Available: https://www.ti.com/product/TCA6424A/part-details/TCA6424ARGJR

[7] "Xilinx iic driver." [Online]. Available: https://xilinx.github.io/embeddedsw.github.io/iic/doc/html/api/xiic_8h.html

[8] "Tca6424a low-voltage 24-bit i2c and smbus i/o expander datasheet." [Online]. Available: https://www.ti.com/lit/ds/symlink/tca6424a.pdf

[9] "Virtual input/output (vio)." [Online]. Available: https://www.xilinx.com/products/intellectual-property/vio.html

[10] Xilinx, "Xdma pcie enumerate example," Apr 2020. [Online]. Available: https://github.com/Xilinx/embeddedsw/blob/master/XilinxProcessorIPLib/drivers/xdmapcie/examples/xdmapcie_rc_enumerate_example.c

[11] "The pci id repository." [Online]. Available: https://pci-ids.ucw.cz/read/PC/8086/2700

[12] "The pci id repository." [Online]. Available: https://pci-ids.ucw.cz/read/PC/10de/0640

[13] "Ibert for ultrascale/ultrascale+ gth transceivers." [Online]. Available: https://www.xilinx.com/products/intellectual-property/ibert_ultrascale_gth.html

[14] "Integrated logic analyzer (ila)." [Online]. Available: https://www.xilinx.com/products/intellectual-property/ila.html

[15] A. Ramdas, D. Cock, T. Roscoe, and G. Alonso, "The enzian coherent interconnect (eci): opening a coherence protocol to research and applications." Ithaca, NY: Capra Research Group, Computer Science and Electrical and Computer Engineering Department, Cornell University, 2021-04-15, Conference Paper, p. 18, workshop on Languages, Tools, and Techniques for Accelerator Design (LATTE '21); Conference Location: Online; Conference Date: April 15, 2021.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

NVMe on Enzian

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Anand	Shashank

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zurich, 06/02/2023	

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.