國立臺灣大學電機資訊學院電子工程學研究所

碩士論文

Graduate Institute of Electronic Engineering
College of Engineering and Computer Science
National Taiwan University

Master Thesis

支援 Xilinx AXI DMA 的 Linux UIO 驅動程式

Linux UIO driver for Xilinx AXI DMA

劉宇唐

Yu-Tang Liu

指導教授：鄭振牟 博士

Advisor: Chen-Mou Cheng, Ph.D.

中華民國 107 年 7 月

July 2018

摘要

近年來，由於 AI、VR 產業的崛起，FPGA 產業越來越受到重視。為了簡化 FPGA 的開發流程，使用嵌入式 Linux 會是一個不錯的方法。透過 Linux Kernel 提供的 UIO 驅動程式，我們可以把我們在硬體端設計出來的 IP 視為一個外部裝置，然後在 Linux 使用者空間裡的程式中，輕鬆地開發軟體端的應用。然而，有些硬體端的設計，卻無法透過同樣的方法，利用 UIO 驅動程式，建立裝置節點，而帶有直接記憶體存取 IP 的設計就是其中之一。由於 UIO 驅動程式並無法支援此種設計，我們必須擁有”root” 權限，才能使用我們的設計，但是提供”root” 給一般使用者並不是一個好方法。在此論文中，我們修改了 Linux 內建的 UIO 驅動程式，使得一般用戶也能在使用者空間中使用帶有 DMA 的硬體設計。

關鍵字: 賽靈思，直接記憶體存取，*AXI*，*Linux UIO* 驅動程式

Abstract

In recent year, increasing importance has been attached to FPGAs with the development of AI,VR. To simplify the development process on FPGAs, embedded Linux on FPGAs will be a good way. With UIO driver provided in Linux Kernel, we can mount our block design, that is, custom IP(Intellectual Property) core in Vivado as a device node, and program it in Linux userspace. However, there are some designs that UIO driver cannot recognize. The design with DMA(Direct Memory Access) is one of them. With this kind of design, because UIO driver is not working, we need ”root” to control our IP, and providing root privileges to users is never a good solution. In this thesis, we modify UIO driver so that users can easily use designs with DMA in user-space.

Keywords: *Xilinx, DMA, AXI, Linux UIO driver*

Contents

List of Figures

List of Tables

Chapter 1

Introduction

FPGA(Field Programmable Gate Array) is a special hardware device that allows people can design and verify their thoughts easily and quickly comparing to ASIC. To design hardware part in PL(Programmable Logic) side, we need to use development tools(e.g.Vivado) provided by FPGA's manufacturer. It is reasonable, because how to convert HDL to the designs that FPGA can recognize depends on rules of its own provider. So hardware design facilitation is basically in control of FPGA vendors. On the other hand, software designing should be much easier and freer, because we can use languages and libraries that we are familiar with. But in fact, we still rely on using SDK(Software Development Kits) tools to design our software system instead. In tradition, we build a "bare metal" program to control our system, and sometimes, the program may include some special libraries only provided in SDK. In software engineer's perspective, things should not be that complicated if we just want to do the simple things like, reading or writing data to registers in PL side. As a result, there come some solutions to make easier in software development.

1.1 Motivation

To simplify the development flow, introducing embedded system on FPGA becomes more popular in recent years. In an embedded system, if we choose Linux as our operating system, we can apply "UIO driver" to our custom IP in PL side and control it in user

space application, just like it is an external device. However there are still some issues, that make UIO driver can not work correctly, the design using AXI-Stream registers with DMA controller is one of them. In this situation, we can only control “DMA controller” to transfer data to or from our custom IP, and this needs “root” privileges. But giving root to a user that only want to control the custom IP is overkill. Hence, we need to find out a soluion to this problem.

1.2 Contribution

We propose a development flow to use UIO to control DMA controller to communicate with AXI-Stream IP, with a little modification to the UIO driver and specific format settings in device tree file. We rewrite **read()/write()** functions in UIO to send DMA transactions to DMA controller. The whole scenario is very simple and intuitive, and is not much different from the original flow. The data transfering efficiency is also good. We will introduce some bachground knowledge of our works in Chapter 2. Our main distribution is presented in Chapter 3. Chapter 4 shows the details of our working platform, and softwares that we use. The experimental result will be discussed in Chapter 5 “Analysis”. Chapter 6 will conclude all our works and mention some future works.

Chapter 2

Preliminaries

In this chapter, we introduce the background technology used in our work, including Embedded Linux, UIO Driver, Register Types in Vivado, and DMA.

2.1 Embedded Linux

Embedded Linux is a kernel and set of libraries and utilities designed to run on an embedded system(eg: router). In this section, instead of talking about the details of Linux kernel, we explain what happens when we port an OS(operation system) to a platform, in this thesis, we use FPGA as our platform.

Figure 2.1 shows the stages of booting Linux on the target platform. For example, when we turn on the FPGA, the board will boot ROM and find the boot mode setting, then load FSBL(First Stage Bootloader), which will load bitstream to initialize the PL side on FPGA. After, it will load the SSBL(Second Stage Bootloader), here we use u-boot for demonstration. The main purpose of u-boot is to load Linux Kernel, it loads the kernel image with ***devicetree file*** of the target platform. With the well-prepared file system, the Linux should run up successfully, in this thesis, we use uramdisk and linaro[1] as our root file system.

Figure 2.2 shows an example of how we boot Linux on FPGA[2]. In this example, we boot Linux on FPGA from SD card, so we need to format SD card into two partitions. The 1st partition is type FAT3, and this partition includes all required boot files,

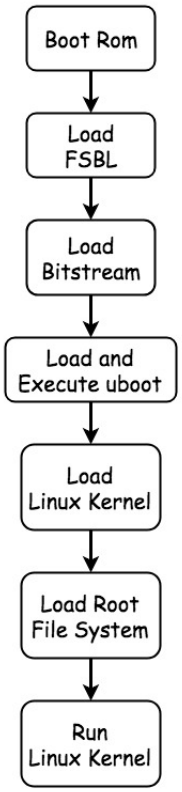

Figure 2.1: Linux Boot Stage on Target Platform

that is BOOT.bin, Linux Kernel(uImage), file system(uramdisk.image.gz), and device-tree file(devicetree.dtb). The 2nd partition is type ext4, and this partition is used as a root file system, so it includes the entire root file system, in our case, we use linaro.

2.1.1 Device Tree

Device Tree is a mechanism to describe all hardware and devices of a system. In early Linux kernel, hardware description is hardcoded in kernel files, so porting kernel to a different platform is painful. To solve this problem, Device Tree is introduced. Like x86 based system, we should consider Linux kernel image is a black box, and give the hardware information of system to the kernel.

In FPGA development flow, the whole system almost keeps the same, the only thing that might change is our design in PL(Programmable Logic) side. To boot Linux with different PL design, only a little modification of the devicetree file is needed.

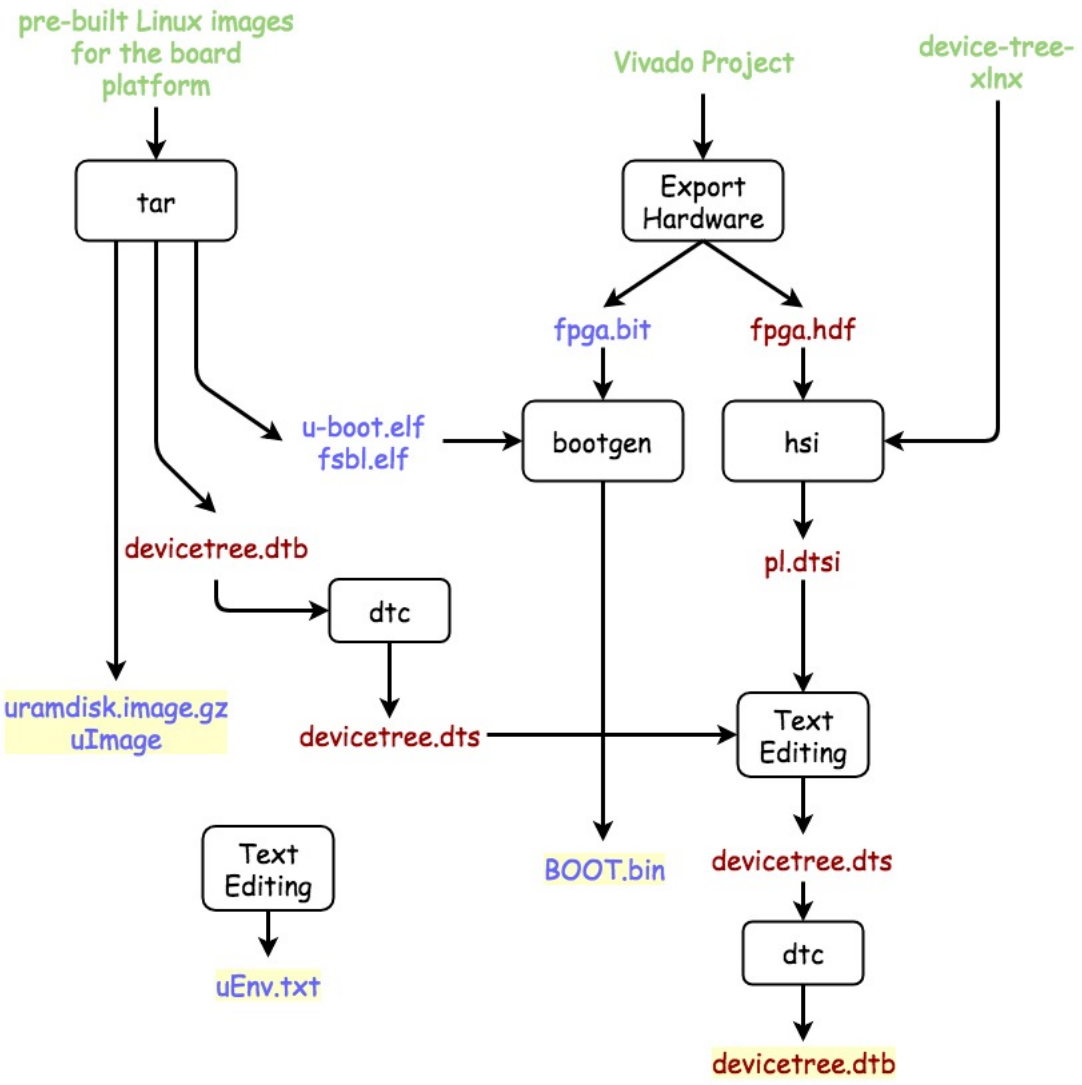

Figure 2.2: Embedded Linux on FPGA

2.1.2 Linux Device Driver

The role of a device driver is a bridge between hardware and software, it communicates with the device and provides functions to applications, like Figure 2.3 shows. A device driver is basically in Linux kernel space, which means if the driver is somehow not working correctly, like getting stuck in an infinite for-loop, the entire system will hang. Hence, a device driver needs to be very robust.

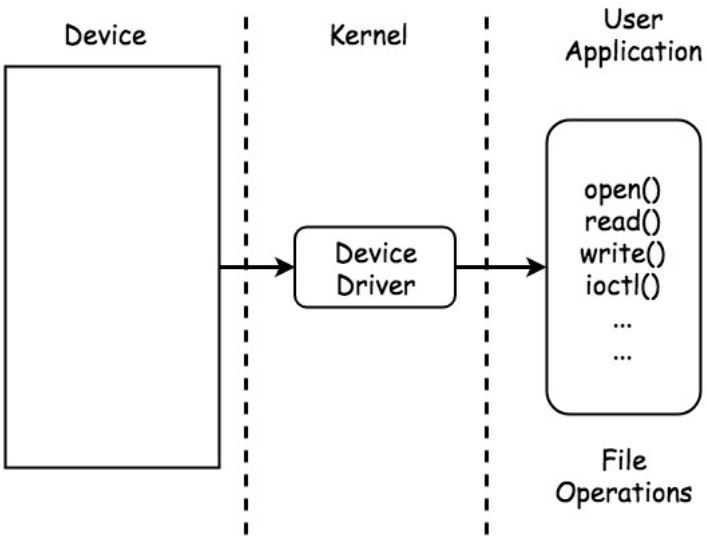

Figure 2.3: Linux Device Driver

2.2 UIO

For many types of devices, creating a Linux kernel device driver is overkill. All that is really needed is some way to handle an interrupt and provide access to the memory space of the device. To address this situation, the userspace I/O system (UIO) was designed. Hardware that is ideally suited for an UIO driver fulfills all of the following:

- The device has memory that can be mapped.
- The device can be controlled completely by writing to this memory.
- The device usually generates interrupts.
- The device does not fit into one of the standard kernel subsystems.

Figure 2.4 shows how the UIO system works, in software-side of FPGA development, we only care about the value in the hardware register and when we can get the correct value, so memory-mapping to a userspace application and interrupt handler is really enough in our design flow[3].

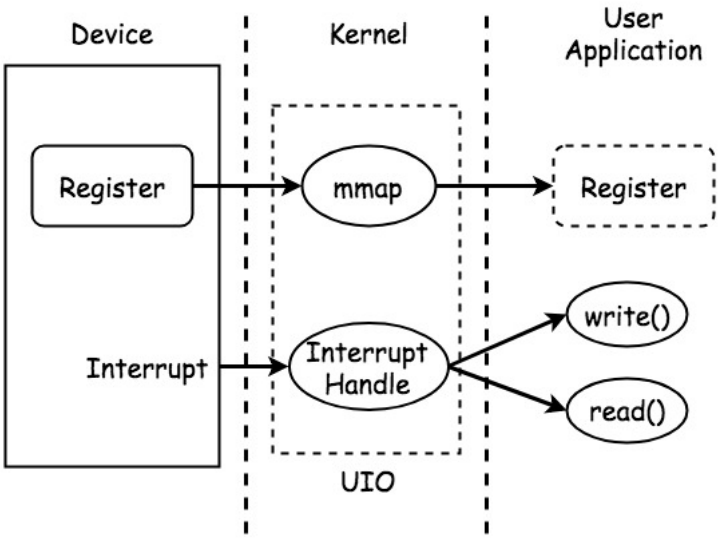

Figure 2.4: The UIO way.

2.3 DMA

DMA(Direct Memory Access) is a feature that allows hardware subsystems to access main system memory independent of the CPU. For example, when CPU wants to submit a DMA transaction, it needs to give the DMA controller where the data is(memory address), and size of the data(data length). After submit, the CPU can go back to work for other tasks. Once the transaction is done, the CPU will receive an interrupt from DMA controller and run the callback function.

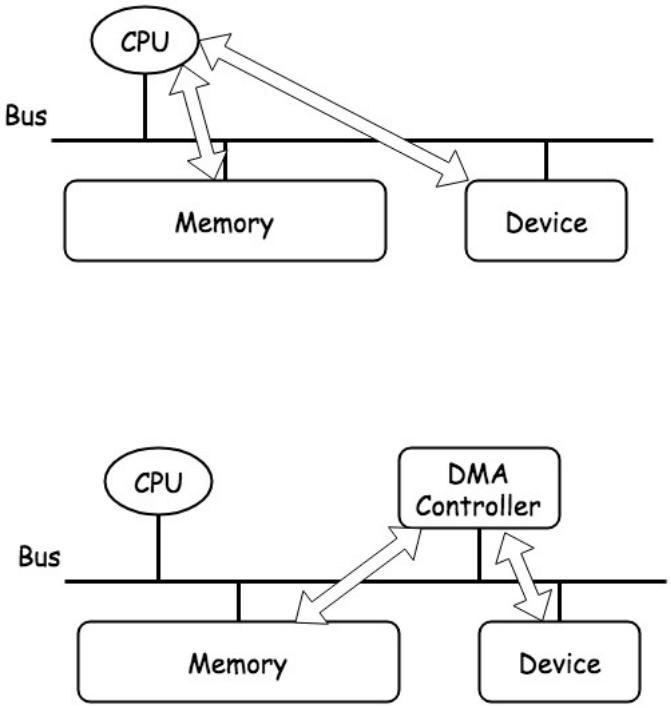

Figure 2.5: Direct Memory Access

2.3.1 DMA Engine

In real world, we use DMA controller to handle DMA works, by setting value to registers to submit the transactions. But this is in hardware perspective, if we want to use DMA more smartly, we need to abstract this concept to software-level, then there is the "DMA Engine". By using the DMA Engine, we can use DMA easily by following the steps below:

1. Request Slave Channel
2. Set parameters
3. Get a descriptor for transaction
4. Submit the transaction
5. Issue pending requests and wait for callback notification

Chapter 3

Proposed solution

There are three types register in Vivado design, Lite,Full and Stream,

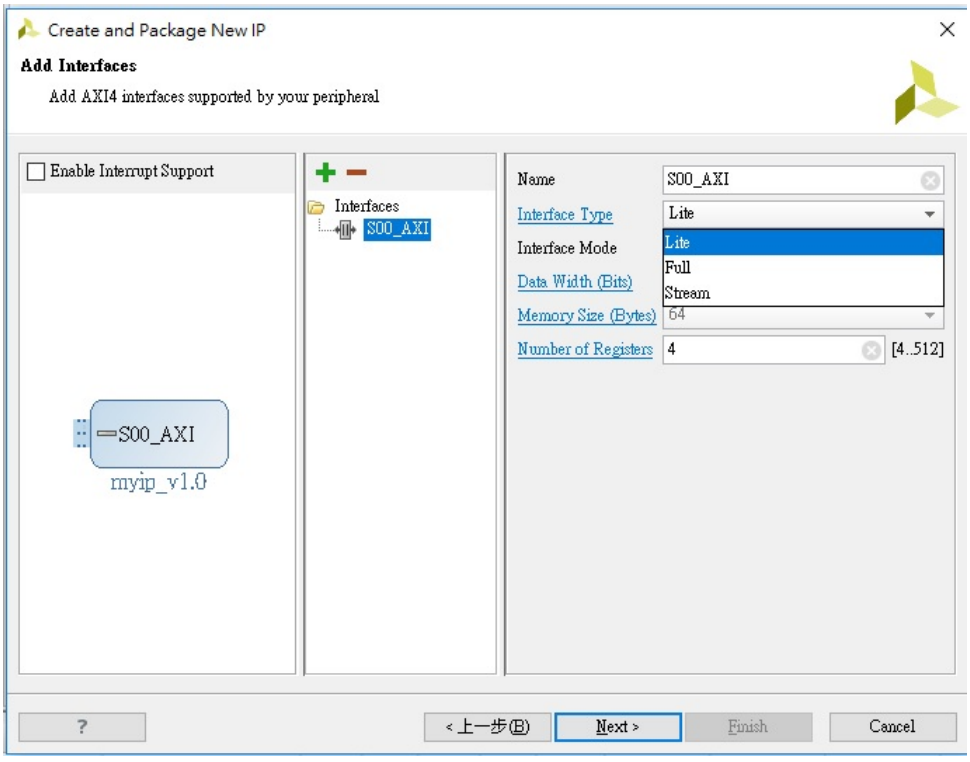

Figure 3.1: Register types in Vivado

These three types follow the protocol defined in AXI bus interface, and in our observation, UIO driver is working if we choose first two types, the thing goes wrong when we choose the third type “Stream” type. In **AXI reference guide** [4], we find that lite and full register follow the AXI4-Lite and AXI4 protocol, these protocols need to send the memory address when we transfer data. We can consider AXI4-Lite protocol is an

easy-setting but function-restricted version of AXI4 protocol. While the stream register which follows the AXI4-Stream protocol is totally different, it does not need to specify the memory address and is a unidirectional channel. That is, in practical, if we want to read and write operation, we need at least two channels. This kind protocol is obvious that it can't be recognized in operation system as system memory.

Now we can finally conclude why UIO driver doesn't work on some IPs with DMA, because AXI4-Stream is a special bus protocol which is not compatible with the AXI4 bus protocol. Figure 3.2 and Figure 3.3 show the difference of two designs. So, if we want to use UIO driver to control our custom AXI4-Stream IP, we need to adapt UIO driver to control the DMA controller so that we can use it to submit DMA transaction to our IP.

We have modeled the high-level problem and proposed a possible solution, but there still has some concerns. How do we submit DMA transaction through UIO to DMA controller? Is there any problem when doing the DMA transaction?

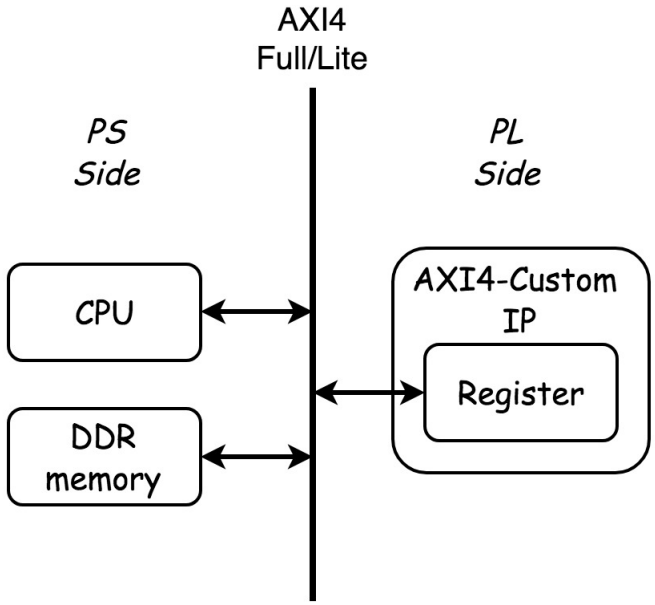

Figure 3.2: Custom IP with AXI4-Lite/Full Register.

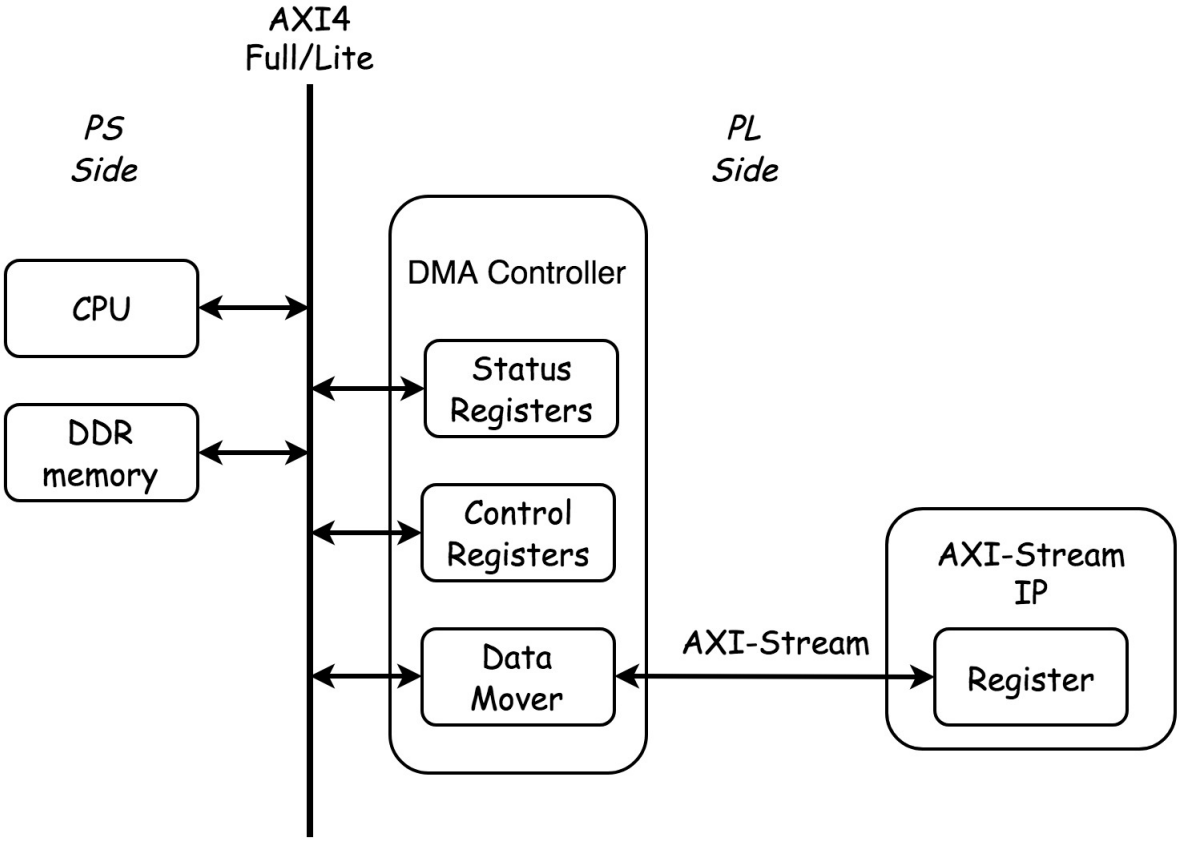

Figure 3.3: Custom IP with DMA and AXI-Stream Register.

3.1 Problems

In this section, we will talk about File Operations, Continuous Memory For DMA, and Cache Coherency. We will explain these problems and show how we solve these problems in the following subsections.

3.1.1 File Operations.

Let's see a simple example of controlling custom AXI4-Full/Lite IP with UIO,

```
int main(void)
{
  int fd = open("/dev/uio0", O_RDWR);
  void *ptr = mmap(0, 0x10000, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0)
    ;
  volatile uint32_t *ctrl = (uint32_t *)ptr;
  *ctrl = 0x00000000;
  ...
}
```

Typically, we open the device node to get device register pointer and memory-map to user memory, then we assign the value to those memory address to control the registers. Please note that, we need file operations to communicate with DMA in UIO driver, but in origianl scenario, we only use **mmap()** and the following value assignment has nothing to do with UIO driver. We need to find some file operations as a function entry.

3.1.2 Continuous Memory For DMA

In traditional DMA transaction, it can only accept a contiguous (nonsegmented) block of physical memory, so, if we want to use DMA in userspace, and we can not get a contiguous memory space(like CMA),

3.1.3 Cache Coherency

While using DMA to do the data transfer, it may lead cache coherency problems. If we want to receive data to the buffer through the DMA, but the buffer is in cache now, to apply transaction, we give controller the buffer address and length. Once the transaction is done, we read the buffer and the value is the same as old value. CPU think the value in memory is not changed because whole data transfer is through DMA controller, so CPU

keeps the old buffer data in the cache, that makes the difference between cache data and real data. Figure 3.4 shows the cache coherency problem, both read and write may lead to this problem, so if we want to transfer the correct data, we must solve this problem.

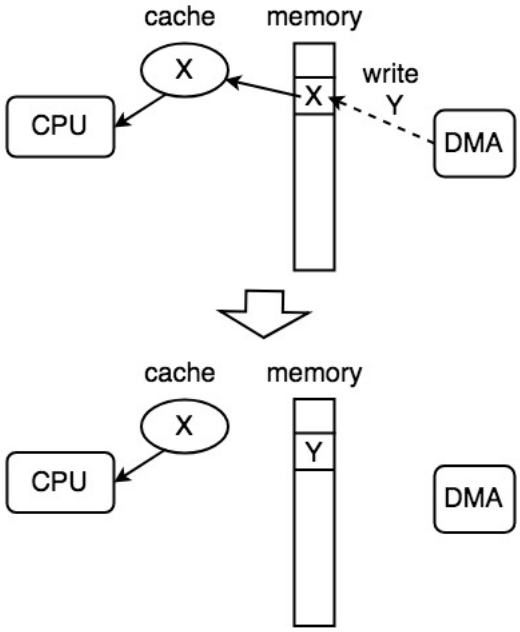

Figure 3.4: Cache Coherency Problems.

3.2 Linux UIO driver for AXI DMA

This sections will demonstrate our main work and explain how we solve the problems we have mentioned in previous section.

Let's take a look at two file operations **write(), read()** in UIO. Figure 3.5 shows what these two functions doing, basically, these two functions in UIO driver is the handle about interrupt control. However, in our design, the UIO node is actually a virtual device node, so interrupt control and memory mapping are no more needed. That means we can use **write(), read()** to do the ***real*** read/write, that is, Figure 3.6.

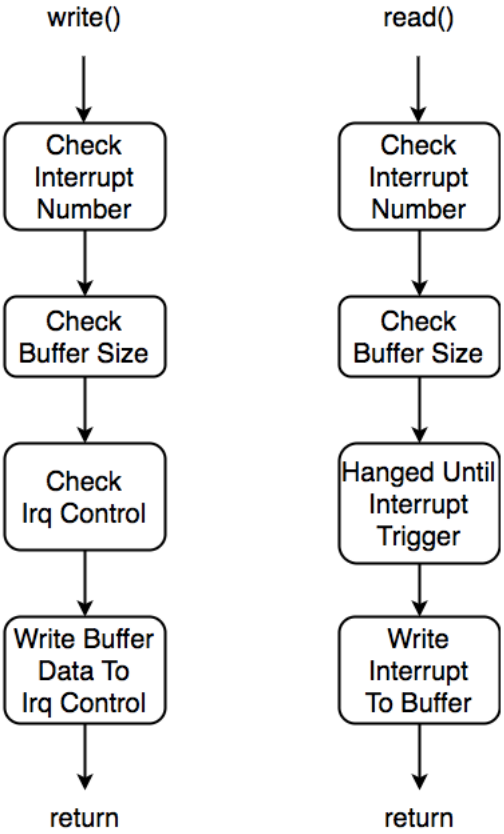

Figure 3.5: UIO write/read functions

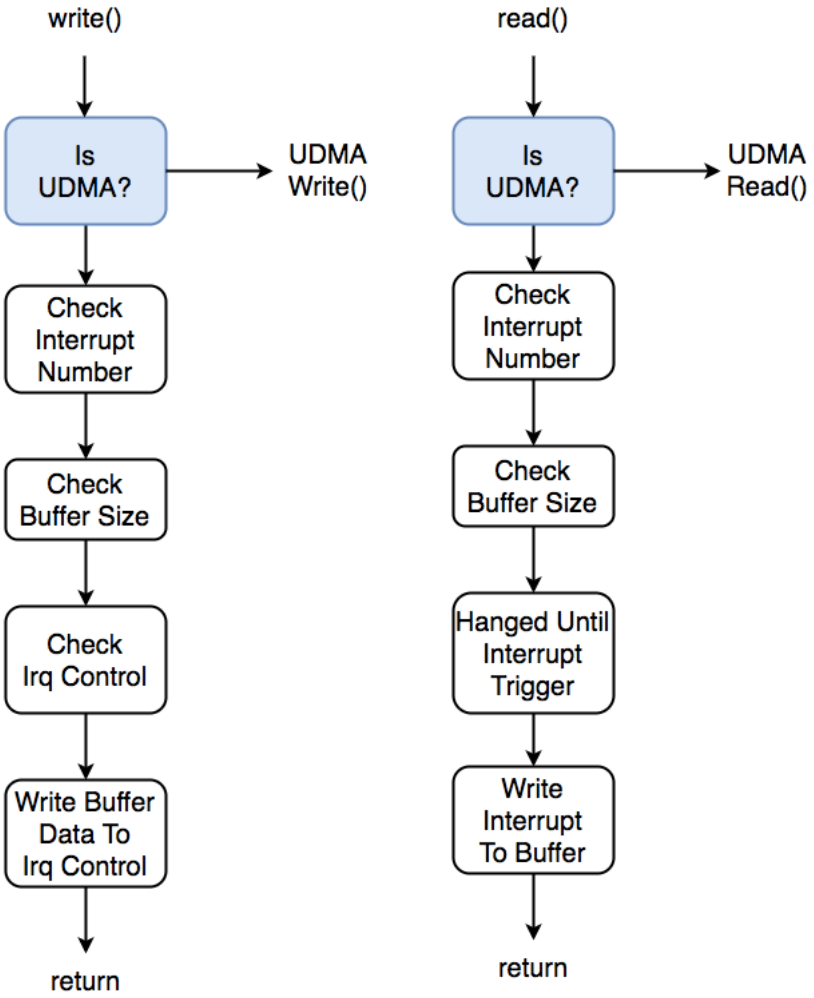

Figure 3.6: UIO write/read functions(modified)

To solve the continuous memory issues, we apply DMA Scatter Gather mode. This mode allows non-contiguous (nonsegmented) block of physical memory and this mode need to turn on in Vivado design first. In this mode, DMA controller automatically gives the start address of the segmented of memory after the previous transaction of segmented memory is completed. To apply this mode, we need to construct a special data structure, Scatterlist, which collects start address and lengths of the segmented block of user buffer memory. DMA engine will do the transaction according to this list.

To avoid cache coherency problem, if transaction direction is memory-to-device, we need to flush cache to memory before submitting a transaction, if the direction is device-to-memory, we need to invalidate the cache after the transfer and before the CPU accesses memory. These two problems will be considered in “udma prepare dma” shows in Fig-

ure 3.7 In “udma prepare dma” function, we first **construct the scatterlist**, and call dma_-map_sg() API provided by DMA engine to deal with the cache coherency problem in **Map the Scatterlist** stage. After all these settings, we can submit our DMA transaction to DMA controller, the following process is just like we have discussed in Section 2.3. Figure 3.8 illustrates the stages in UDMA_prepare_DMA() function.

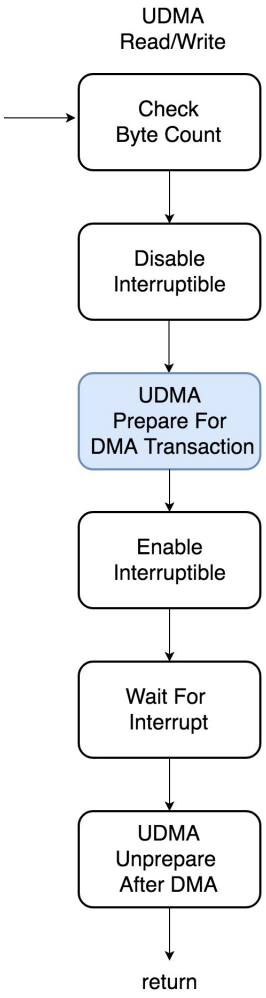

Figure 3.7: UDMA write/read functions

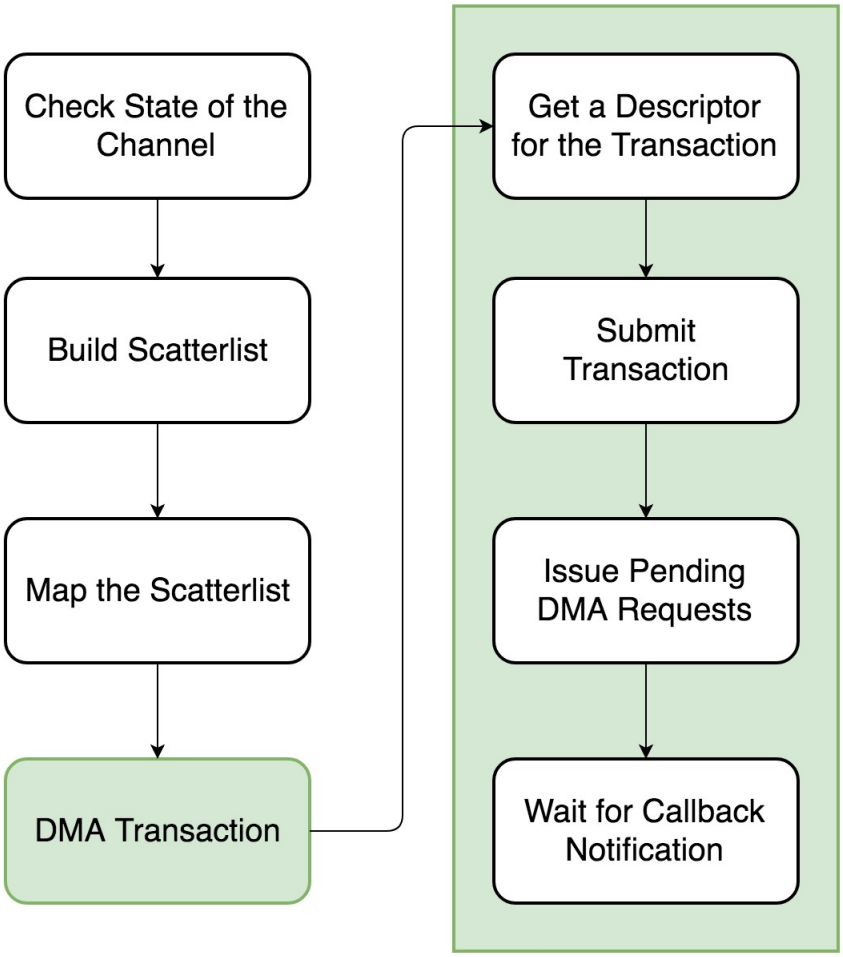

Figure 3.8: UDMA prepare for dma

3.3 Implementation

In this section, we will give a simple tutorial on how to use our new UIO driver and some settings. The entire design environment will be shown in Chapter 4. The full details of usage and code will be uploaded to GitHub repository[5].

3.3.1 Device Tree File Format

First, we need to set our virtual device in our device tree file, in AXI4 IP, we have device block looks like:

```
my_customIP@43c00000 {
  compatible = "generic-uio";
  reg = <0x43c00000 0x10000>;
  interrupts = <0 29 1>;
  interrupt-parent = <0x3>;
  xlnx,s00-axi-addr-width = <0x6>;
  xlnx,s00-axi-data-width = <0x20>;
}
```

It contains IP name, IP register address, register length, interrupt control...etc. These are essential properties if you want to apply a driver to control the device. But in our design, we have no real device, so the device node will look like:

```
udma0 {
    compatible = "generic-uio";
    dmas = <dma-channel1 dma-channel2>;
    dma-names = "loop_tx", "loop_rx";
    ezdma,dirs = <2 1>;
};
```

Where dmas property refers to the DMA channel under “axidma” in device tree, for example, if “axidma” looks like:

```
loopback_dma: axidma@40410000 {
    #dma-cells = <1>;
    compatible = "xlnx,axi-dma";
    reg = < 0x40410000 0x10000 >;
    xlnx,include-sg;
    loopback_dma_mm2s_chan: dma-channel@40410000 {
        compatible = "xlnx,axi-dma-mm2s-channel";
        interrupt-parent = <&gic>;
        interrupts = <0 31 4>;
        xlnx,datawidth = <0x20>;
        xlnx,sg-length-width = <14>;
        xlnx,device-id = <0x1>;
    };
  loopback_dma_s2mm_chan: dma-channel@40410030 {
        compatible = "xlnx,axi-dma-s2mm-channel";
        interrupt-parent = <&gic>;
        interrupts = <0 32 4>;
        xlnx,datawidth = <0x20>;
        xlnx,sg-length-width = <14>;
        xlnx,device-id = <0x1>;
    };
};
```

Then “dmas” should be “<&loopback_dma 0 &loopback_dma >”, dma-names is fixed in the driver, please make sure the names are the same as the setting. “dirs” tells the driver the direction of DMA channel. If the direction is not the same as the declaration in “axidma”, it will fail when UIO is probing. “mm2s” means “memory map to stream”, represents for the tx channel. “s2mm” means “stream to memory map” represents for the rx channel. After all these settings, our UIO driver should probe the device correctly and create a device node under /dev, like /dev/uio0.

3.3.2 Compile New Kernel

Because we modified the UIO driver and add some new library in Linux kernel, we need to compile a new kernel. First, replace the old “uio.c” and “uio_pdrv_genirq.c” with new files. Then put “udma.c” under “drivers/uio” folder, and “udma.h” under “include/linux” folder. We need to add “obj-y += udma.o” in Makefile under “drivers/uio” . After, we can compile our new kernel with new UIO driver which can support DMA functions.

3.3.3 Environment Variables Settings in U-boot

Same as the we mentioned in Chapter 2, we boot Linux on FPGA from SD card with two partitions. The boot files in first partition have much changed in our scenario, “devicetree.dtb” and “uImage” we have discussed in former subsection. The modification in “uEnv.txt” is quite easy, this file provides additional environment variables for the bootloader, u-boot. It will look like:

```
bootargs=console=ttyPS0,115200 root=/dev/mmcblk0p2 rootwait rw
    earlyprintk uio_pdrv_genirq.of_id=genric-uio
...
```

To combine driver and device, please make sure that the string behind “uio_pdrv_genirq.of_id=”(in this case, is “generic-uio”) is same as the property “compatible” of UIO node in device tree file.

Figure 3.9 gives a simple illustration of our scenario. Unlike having lots of changes in 1st partition of SD card, we keep 2nd partition as usual, this partition provides the root file system when we boot up our Linux, just keeps it as the same.

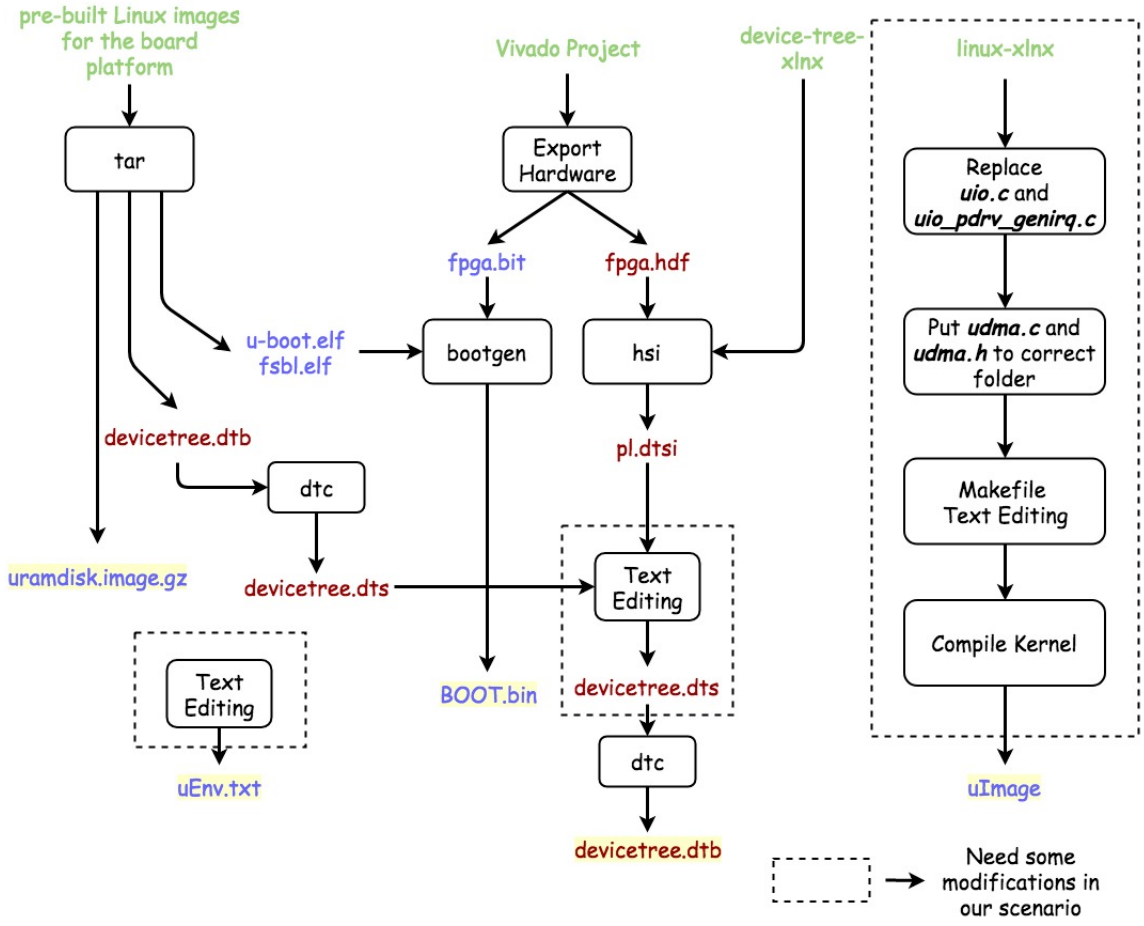

Figure 3.9: Embedded Linux on FPGA(UDMA ver.)

3.3.4 Example

We will demonstrate how to use our driver in C.

```
Example1:

#define PACKET_SIZE (2048)

uint8_t tx_buf[PACKET_SIZE];
uint8_t rx_buf[PACKET_SIZE];

int main(void)
{
  int fd = open("/dev/uio0", O_RDWR);
  rv1 = write(fd, tx_buf, PACKET_SIZE);
  rv2 =  read(fd, rx_buf, PACKET_SIZE);

}
```

First, we open the device node to get the device pointer, and if we want to write the data in tx_buf to the device, we call write() function. If we want to read the data in the device to the rx_buf, we call read() function. All three parameters are easy to understand in the above example, the first parameter is the device pointer, the second one is the buffer pointer

and the third one is the data size. Please note that the data size is the number of bytes, so if you declare buffer as type **uint32_t** which is a 32-bit array then your PACKET_SIZE in function need to be multiplied by 4.

```
Example2:

#define PACKET_SIZE (2048)

uint32_t tx_buf[PACKET_SIZE];
uint32_t rx_buf[PACKET_SIZE];

int main(void)
{
  int fd = open("/dev/uio0", O_RDWR);
  rv1 = write(fd, tx_buf, PACKET_SIZE *4);
  rv2 =  read(fd, rx_buf, PACKET_SIZE *4);

}
```

The return value is the number of bytes, basically, it is equal to the value of the third argument you put in the function. In example1, rv1 and rv2 are 2048, in example2, rv1 and rv2 are 8192.

Chapter 4

Environment Framework

In this chapter, we introduce the device, environment and software that we used in our experiment.

(i) We choose Xilinx ZedBoard as our target FPGA platform.

  (a) Dual ARM® Cortex™-A9 MPCore™

  (b) 512 MB DDR3 memory (1066 Mbps)

(ii) Linux Kernel is compiled from the GitHub repository provided by Xilinx.[6]

  (a) Kernel version is 4.9.0

  (b) GCC version is 5.4.0

(iii) Linaro[7] is put in the 2nd partition of SD card as our root file system.

(iv) All our custom stream IPs are designed(and provided) in Vivado 2016.04.

(v) Devicetree files are generated by SDK 2016.04 and github repository provided by Xilinx[8]

Although we use some relatively old version of Xilinx development software, our contribution is basically out of the version problem. If the version compatible is well handled by Xilinx, our flow and code should work well directly, and if not, only little modification is needed.

4.1 Some Issues about Devicetree File

This section we talk about some issues about a devicetree file. We have mentioned devicetree file in former chapter 3.3.1, but here we'll discuss some problems we met when we do the experiment, and it might be some bugs of the devicetree compiler or devicetree generator of Xilinx SDK. Because this is not about how we solve the problem, so we separate this part here.

There may have some wrong settings if you directly use the devicetree file generated by Xilinx SDK. For example, the interrupt settings of DMA controller or our custom IP in the deivcetree file may not follow the Vivado design. The interrupt number and interrupt type may not be the same as we expect. See the following example:

```
dma@40400000 {
  #dma-cells = <0x1>;
  clock-names = "s_axi_lite_aclk", "m_axi_sg_aclk", "m_axi_mm2s_aclk",
   "m_axi_s2mm_aclk";
  clocks = <&clk 0xf &clk 0xf &clk 0xf &clk 0xf>;
  compatible = "xlnx,axi-dma-1.00.a";
  interrupt-parent = <0x3>;
  interrupts = <0x0 0x1d 0x4 0x0 0x1e 0x4>;
  reg = <0x40400000 0x10000>;
  xlnx,addrwidth = <0x20>;
```

In "interrupt" property, "interrupts = <0x0 0x1d 0x4 0x0 0x1e 0x4>;" means we have two interrupts, <0x0 0x1d 0x4> and <0x0 0x1e 0x4>, because we have two DMA channels in our design. The second property(0x1d and 0x1e) is the interrupt number and should be determined when we connect the DMA controller to the CPU in Vivado design. But in fact, the generated devicetree file sometimes may set a wrong number to this property, and cause the driver can not probe the device correctly. The third property(in our example, both are 0x4) is the interrupt type, this property is related to how we treat the interrupt signal, because the devicetree compiler can't know how we design, it just put a default value to this property. So make sure that the settings in the devicetree file are corresponding with the design. The interrupt type definition is in [9]

Some label problem may happen also. In devicetree, we can define labels for convenience, let's take a look at the above example again, note that, in "clocks" property, it uses label "&clk", to get the correct clock, so there must have some settings like:

```
clk: clkc@100 {
  #clock-cells = <0x1>;
  compatible = "xlnx,ps7-clkc";
  fclk-enable = <0xf>;
  clock-output-names = "armpll", "ddrpll", "iopll", "cpu_6or4x", "
    cpu_3or2x", "cpu_2x", "cpu_1x", "ddr2x", "ddr3x", "dci", "lqspi", "
    smc", "pcap", "gem0", "gem1", "fclk0", "fclk1", "fclk2", "fclk3", "
    can0", "can1", "sdio0", "sdio1", "uart0", "uart1", "spi0", "spi1",
    "dma", "usb0_aper", "usb1_aper", "gem0_aper", "gem1_aper", "
    sdio0_aper", "sdio1_aper", "spi0_aper", "spi1_aper", "can0_aper", "
    can1_aper", "i2c0_aper", "i2c1_aper", "uart0_aper", "uart1_aper", "
    gpio_aper", "lqspi_aper", "smc_aper", "swdt", "dbg_trc", "dbg_apb";
  reg = <0x100 0x100>;
  ps-clk-frequency = <0x1fca055>;
  linux,phandle = <0x1>;
  phandle = <0x1>;
};
```

But in the devicetree file generated by SDK, it sometimes uses the label without presetting the label, using the above example, it uses “&clk” in “dma” node, but it does not set “clk” label to “clkc” node. As a result, if there are some error messages about the reference when generating the devicetree file, make sure the above problems are solved.

Chapter 5

Analysis

To analyze our works, we find a similar driver called “EZDMA”[10] on GitHub repository as a comparison. The difference between our works and EZDMA is that we modify the UIO driver, and EZDMA is a whole new kernel driver to do the same thing. In our experiment, we use three different AXI4-Stream IPs to test our, one is **AXI4-Stream FIFO** provided in Vivado, the others are custom AXI4-Stream IPs. We rewrite the **tinyAES**[11] from OpenCores[12] to a version with DMA, and we write a **ECDSA IP(with DMA)** with curve secp256k1 which is used in the Bitcoin system.

1

5.1 Comparison

The performance of our driver can be seen in the table 5.1, the **Frequency** is the fastest frequency that our IP can work correctly, this is determined when we designed our IP in Vivado. In this experiment, we send a transmit package to the DMA controller and send receive package to the DMA controller, that means we send input to the custom IP and we read the output from the custom IP, and we repeat 1,000,000 times to get the throughput. The transmit package size we sent is shown in **Input** column, and receive package size is shown in **Output** column.

IP	Frequency(Mhz)	Input(Byte)	Output(Byte)	UDMA	EZDMA
FIFO	100	2048	2048	16.907 MB/s	16.492 MB/s
tinyAES	250	32	16	0.331 MB/s	0.334 MB/s
ECDSA	10	128	128	0.016 MB/s	0.015 MB/s

Table 5.1: The test result of 3 different IPs.

The result shows that our driver has almost the same throughput as the EZDMA has. Because our driver is a modified version of UIO, and EZDMA is an additional kernel driver, we consider the result is totally reasonable and acceptable.

Chapter 6

Conclusion and Future Works

In this thesis, we design a flow and modify the UIO driver so that we can control our custom AXI4-Stream IP in Linux on FPGA easily. The driver efficiency is satisfactory, and our modification is also little, we even provide a script file to run all things, if set properly. The final goal of our works is that our modified UIO driver can be merged into Linux kernel. Rather than writing a whole new driver, we think that most FPGA developers use UIO driver to communicate with their own IP, so make UIO driver "better" is a "better" way to solve the problem that we have no useful driver to help us control the AXI4-Stream IP. In conclusion, we think our work is useful for people who want to develop an application on FPGA

In future, there are still some works can be done, for example, Xilinx has other DMA components provided in Vivado, that is, AXI VDMA and AXI CDMA. VDMA is a special DMA used for video purpose, and CDMA is a memory-mapped to memory-mapped DMA, similar to AXI DMA with Stream FIFO. We are not very sure whether the usage of these components is as same as AXI DMA, this needs more test and research.

References

[1] “Linaro,” https://www.linaro.org/developers/, accessed: 2018-07-04.

[2] “Embedded linux on zedboard repository,” https://github.com/concise/zedboard-hwswcodesign-example, accessed: 2018-07-04.

[3] “The userspace i/o howto,” https://www.kernel.org/doc/html/v4.11/driver-api/uio-howto.html, accessed: 2018-07-04.

[4] “Xilinx axi reference guide,” https://www.xilinx.com/support/documentation/ip_-documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf, accessed: 2018-07-04.

[5] “Linux uio for axi dma,” https://github.com/yllibliu/dma, accessed: 2018-07-04.

[6] “Xilinx linux kernel repository,” https://github.com/Xilinx/linux-xlnx, accessed: 2018-07-04.

[7] “Xilinx axi reference guide,” https://releases.linaro.org/ubuntu/images/developer/15.12/linaro-vivid-developer-20151215-714.tar.gz, accessed: 2018-07-04.

[8] “Xilinx device tree repository,” https://github.com/Xilinx/device-tree-xlnx, accessed: 2018-07-04.

[9] “Interrupt wiki,” https://en.wikipedia.org/wiki/Interrupt, accessed: 2018-07-04.

[10] “Ezdma project on github,” https://github.com/jeremytrimble/ezdma, accessed: 2018-07-04.

[11] “Aes project on opencores,” https://opencores.org/project/tiny_aes, accessed: 2018-07-04.

[12] “Opencores,” https://opencores.org/, accessed: 2018-07-04.