

# PCIe Solutions on Xilinx FPGAs 初学者指南

## V1.0

hanson@comtech.com.cn

PCIe on Xilinx 初学者指南 .....	1
前言 .....	1
步步为营 .....	3
一，PCIe 层次结构 .....	3
事物协议 .....	3
头标 .....	4
BAR 空间 .....	6
中断 .....	7
二，器件选型 .....	9
三，仿真环境搭建 .....	9
四，调试 .....	14
五，驱动编写 .....	15
六，总结 .....	16
进阶 .....	22
DMA 模式 .....	22
DDR 缓冲 .....	23
经验、教训总结 .....	24

## 前言

北京奥运会那年接手了一个项目，需要将远程的 80MBps 的数据量传输到服务器，同时不能占用太多的处理器时间，因为服务器上同时运行着一套大型数据库软件。最后排除了 USB 方案、以太网方案，选用 Xilinx 的 PCIe 解决方案。经过漫长的摸索、学习过程，最后项目做成功的做完了，各种指标参数也都达到了。再后来做 FAE，见识了更多的 PCIe 应用，也有幸结识了更多的开发工程师。有的对 PCIe 的理解非常深入了，无论在性能和功能上都达到相当的深度；有的也和我原来一样，刚开始学习理解 PCIe 的应用，看书、看文档，有时候迷茫的找不到如何下手。对于前一种高手，这篇应用笔记可以略过了；对于后一种正在做产品、项目的工程师，希望能对您有所帮助。

## 什么时候需要用到 PCIe？

首先需要定位的是什么时候需要用到 PCIe 的问题。(PCIe 是什么这里就不做介绍了，不

然冗长的像写论文了。)

翻开电脑(台式机),主板上可以清晰的看到常用的接口,能和外面连接的插槽主要有:USB、Ethernet、PCIe、PCI、SATA/PATA、Audio、VGA/DVI/HDMI, UART/并口。其中 Audio、VGA/DVI/HDMI 制作专用数据输出, UART/并口的速率摆在那里,不适合高速数据传输,剩下的接口**最大吞吐率**如下:

接口	最大速率
USB	USB2.0 480Mbps, USB3.0 5Gbps(USB2.0 速率的 10 倍)
Ethernet	1Gbps
PCIe	X8 Gen1 双向各 16Gbps, X8 Gen2 双向各 32Gbps
PCI	2.112Gbps@66Mhz*32 位
SATA/PATA	SATA II 3.0Gbps, SATA III 6.0Gbps

排除协议开销,OS 开销,平常使用中 USB 能达到的速度 USB2.0 一般在 30-40MB; Ethernet 如果用硬件实现较低层次的协议在 70-80MBps; PCI 有 64 位的,但是普通 PC 或服务器一般是 32 位的,见过效率发挥的较好的能达到 120MB,但是 PCI 是共享总线的,如果总线有多个设备开销,这个速度就难保证了; SATA/PATA 在 PC 中一般用于存储,用于自定义设备的很少见。因此,如果数据量超过 100M,那 PCIe 最合适了。

速度是一方面考虑的原因,能否快速实现(有无成熟的设计方案),并且产品稳定、可靠是另外一个需要考虑的原因。

USB 有 Cypress 的 USB2.0 芯片,然后将总线接到 FPGA 上。网上有很多 USB 芯片的程序、驱动程序,甚至有公开的 GUI 程序,可以方便的完成 FPGA 到 PC 的通讯。

Ethernet 以太网也是比较成熟的接口。FPGA 外接 PHY 就可以和 PC 通讯了。以太网的协议非常复杂, FPGA 如果内部运行 CPU 加以太网协议栈,那可以支持高层的以太网协议,但是这样速度就折扣了,并且 FPGA 的开发难度也增加很多。如果只实现 MAC 层数据收发,难度相对简单一些。

#### 开发难度:

PCI PCIe 卡的方案也比较成熟了,既可以用 PLX 公司的桥片+FPGA 的方案,也可以直接用 FPGA 接 PCI IP 核的方式。PCI 经过长时间的积累,已经有相当多的上层和底层设计、调试软件,如果是购买的桥片,还有配套的开发工具简化设计流程。

SATA/PATA 通过 SATA/PATA 接口和 PC 进行数据通讯的方案比较少见,有 Intelliprop 一些列的 SATA Device IP Core,同时提供一些底层的软件支持。

PCIe PCIe 的方案种类延续了 PCI 的多样性。1,完全采用 FPGA 方案。Xilinx 早在 10 年前就把 PCIe 作为重要的支持方向,在高、中、低端 FPGA 内都集成有免费的 PCIe 硬核,同时提供了数种源码(包括驱动、软件)开放的参考设计,还有丰富的文档。无论是学习、还是产品设计,都能在众多的资料中找到相关的信息,减少设计、调试的时间(例如, Xilinx 的 PCIe 用户手册, xapp1052 DMA 参考设计, PIO 参考设计)。2,采用桥片方案,例如 PLX, IDT 等芯片,然后将转换的局部总线接入 FPGA。

研发产品需从多方面考虑,成本、可靠性、兼容性、性能等。单片 FPGA 方案越来越多的应用在各种产品的研发中。如果确定需要使用 FPGA 的 PCIe 方案,而此时对这一流程比较迷茫,那希望这篇指南能对您有所帮助。

# 步步为营

## PCIe 相关知识

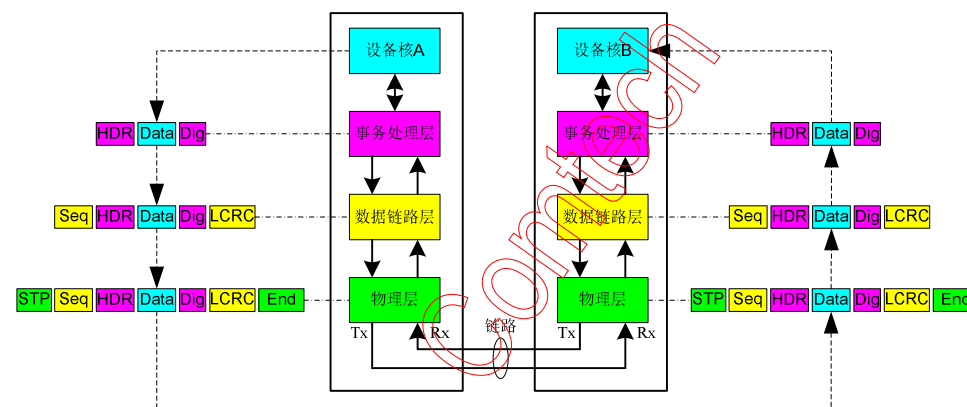
引用 Wiki <http://zh.wikipedia.org/zh-cn/PCI-E>, 简单介绍 PCIe 的基本特性, 其他不理解的没关系, 要么等一个流程下来自然而然明白了, 要么...就不需要知道 (Xilinx 封装好了)。

Wiki 上列出各版本 PCIe 的链路速度, 通道数。说白了, 链路速度高, 单位时间内传输的有效数据多; 通道数多, 同时传递的数据多。此外, 还有很给力的一点就是兼容, 板卡和主板自适应的匹配到最大的链路速度和通道数。也就是说, 如果这个板卡只支持 PCIe1.1 x8, 放在支持 PCIe2.0 x16 的槽上, 那两端会是应在 PCIe1.1 x8 的速率上; 如果有一天板卡升级了, 到 PCIe2.0 x8, 那主板会适应成 PCIe2.0 x8 上, 并且 PC 上的驱动和软件都不用变化。

接下来有点晦涩了, 但是建议耐着性子看完吧。

## 一, PCIe 层次结构

PCIe 规范对于设备的设计采用分层的结构, 有事务层、数据链路层和物理层组成, 各层有都分为发送和接收两功能块。



在设备的发送部分, 首先根据来自设备核和应用程序的信息, 在事务层形成事务层包 (TLP), 储存在发送缓冲器里, 等待推向下层; 在数据链路层, 在 TLP 包上再串接一些附加信息, 这些信息是对方接收 TLP 包时进行错误检查要用到的; 在物理层, 对 TLP 包进行编码, 占用链路中的可用通道, 从查封发送器发送出去。<sup>1</sup>

事务层包 (TLP), 数据链路层包 (DLLP), 物理层 (PLP) 产生于各自所在层, 最后通过电或光等介质和另一方通讯。这其中数据链路层包 (DLLP), 物理层 (PLP) 的包平常不需要关心, 在 IP 核中封装好了。在 FPGA 上做 PCIe 的功能, 变成完成事务层包 (TLP) 的处理。

## 事物协议

上面提高过, 用 FPGA 实现需要的 PCIe 功能, 简单的说是在完成 TLP 的处理。TLP 有三部分组成, 帧头、数据、摘要 (或者称 ECRC)。TLP 头标长 3 或者 4 个 DW, 格式和内容随事物类型变化; 数据端为 TLP 帧头定义下的数据段, 如果该 TLP 不携带数据, 那该段为空。Digest 段 (Optional) 是基于头标、数据字段计算出来的 CRC, 成为 ECRC, 一般 Digest 段有 IP 核填充。所以, PCIe 的处理在用户层表现为处理 TLP 中头标和数据段。





在左下框，sof 代表帧的开始，eof 代表帧的结束，收到的帧数据为 0x0000\_0001\_01a0\_0a0f，0x0000\_0010\_de03\_7320，rem\_n 为高代表后面最后一个 DW 是无效的，因此，收到的完整的帧为 0x0000\_0001\_01a0\_0a0f\_0000\_0010。按照上述包格式分析，Fmt 是 2'b00，Type 是 5'b0000，接收的是 Non-Posted 3DW 存储器读请求包（MRd）（具体请求的是下文再讲）。既然是 Non-Posted 的请求包，就需要有完成包作为回复。

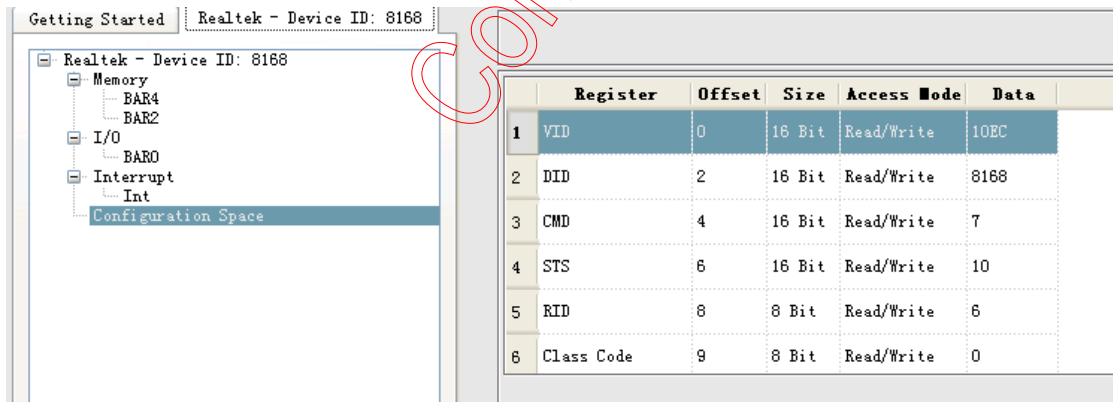
在右上框，发送的数据帧数据为 0x4a00\_0001\_01a0\_0004，0x01a0\_0a10\_0403\_0201。Fmt 是 2'b10，Type 是 5'b01010，判断为 3DW 带数据的完成包，0x4a00\_0001\_01a0\_0004\_01a0\_0a10 是头标，0x0403\_0201 是所带的数。

PCIe 设计很大一部分工作就是判断 RX 收到什么 TLPs，通过 TX 发送什么样的 TLPs。有时候发现异常，分析这些最基本的 TLP 包结构是非常有效的诊断方式。

## BAR 空间

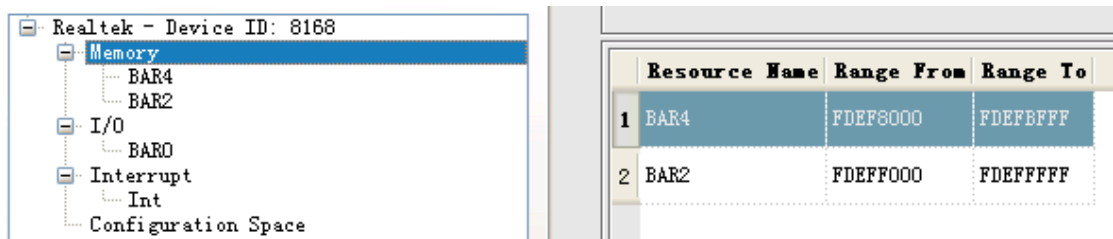
上面讲到 PCIe 通讯是靠发送 TLP 包，如果再进一步参考了 PCIe 更详细的书籍就能发现读写包里都有地址信息。如果板卡向 PC 发送 TLP 包，例如 MWr 包，那很好理解，地址信息就是 PC 的物理地址（注意这里说的是物理地址）；如果是 MRd 包，那 PC 收到后回复一个完成包，板卡从完成包分析出数据即得到 MRd 读取地址的数据。那 PC 如何读写板卡的数据呢？

简单的解释，PC 启动是，BIOS 探测所有的外设。对 PCIe（PCI）设备来说，BIOS 检测到板卡有多少个 BAR 空间，每个空间有多大，然后对应为这些 BAR 空间分配地址。对 PC 设备来说，它能“看”到 PCIe 板卡的空间只有 BAR 空间，也就只能访问这些 BAR 空间。也就是说，板卡可以发送合法的 PCIe TLP 包，并得到 PC 端的相应；但是 PC 端访问板卡被局限在 BAR 空间。例如，下图是 Realtek 网卡的 BAR 空间分配图，该网卡有 3 个 BAR 空间，BAR0、BAR2、BAR4，BAR0 空间配置为 I/O 空间，BAR2 和 BAR4 配置为 Memory 空间。



Register	Offset	Size	Access Mode	Data
1 VID	0	16 Bit	Read/Write	10EC
2 DID	2	16 Bit	Read/Write	8168
3 CMD	4	16 Bit	Read/Write	7
4 STS	6	16 Bit	Read/Write	10
5 RID	8	8 Bit	Read/Write	6
6 Class Code	9	8 Bit	Read/Write	0

图 3 Realtek 网卡的 BAR 空间



Resource Name	Range From	Range To
1 BAR4	FDEF8000	FDEFBFFF
2 BAR2	FDEF0000	FDEFFFFF

图 4 Realtek 网卡 Memory 空间大小



图 5 Realtek 网卡 IO 空间大小

从上两幅图来看，BAR2 空间大小为 0x1000，在 PC 的其实地址为 0xFDEF\_F000；BAR4 空间大小为 0x4000，起始地址为 0xFDEF\_8000；BAR0 空间为 0x100，其实地址为 0x0000\_DE00。大小在接口芯片中已经定义好了，而起始地址在不同的 PC 上分配的地址是不同的，但是偏移地址相同，访问板卡的效果就是相同的。例如，假设需要访问该板卡 BAR2 空间的地址 0x80，在 PC 上访问 0xFDEF\_8080 就对应读写 BAR2 空间的 0x80 地址了。

在 FPGA 中，BAR 空间的设置根据用户逻辑设计的需求来定义大小。假设我们需要和上面网卡相同的设置，PCIe 核定制的时候就可以按照下列的选项：

**BAR 0 Options**  
☒ Bar0 Type: IO ☐ 64 bit ☐ Prefetchable  
 Size: 256 Bytes  
 Value: FFFFFFF01 (Hex)

**BAR 1 Options**  
☐ Bar1 Type: N/A ☐ 64 bit ☐ Prefetchable  
 Size: 2 Kilobytes  
 Value: 00000000 (Hex)

**BAR 2 Options**  
☒ Bar2 Type: Memory ☐ 64 bit ☐ Prefetchable  
 Size: 16 Kilobytes  
 Value: FFFFC000 (Hex)

**BAR 3 Options**  
☐ Bar3 Type: N/A ☐ 64 bit ☐ Prefetchable  
 Size: 2 Kilobytes  
 Value: 00000000 (Hex)

**BAR 4 Options**  
☒ Bar4 Type: Memory ☐ 64 bit ☐ Prefetchable  
 Size: 64 Kilobytes  
 Value: FFFF0000 (Hex)

**BAR 5 Options**  
☐ Bar5 Type: N/A ☐ Prefetchable  
 Size: 2 Kilobytes  
 Value: 00000000 (Hex)

图 6 PCIe 核 BAR 空间设置

## 中断

做过嵌入式和工程师应该对中断有较深的理解，中断处的调试相对于 BAR 空间也复杂一些。

PCIe 可以发出两种中断，一种是虚拟 INTx 信号线的，一种是 MSI 的。

过去 PCI 板卡发送中断通过拉低 INTx (INTA#, INTB#, INTC#, INTD#) 来申请中断，PC 检测到 INTx 的中断，就跳转执行 INTx 对应的中断驱动程序，驱动程序里需要操作板卡将 INTx 拉回去，不然就发生嵌套中断了。用下面两个时序图来解释 INTx 中断，收到的数据包内容为 0x3400\_0000\_0100\_0020, 0x0000\_0000\_0000\_0000，根据事物协议的章节分析该包，Fmt 为 2'b01, Type 为 5'b10100，是消息请求，Message Code 为 8'b0010\_0000，是中断 (INTx) 消息。根据下图可以看到 received\_assert\_inta 为 1，收到 INTA#中断了。



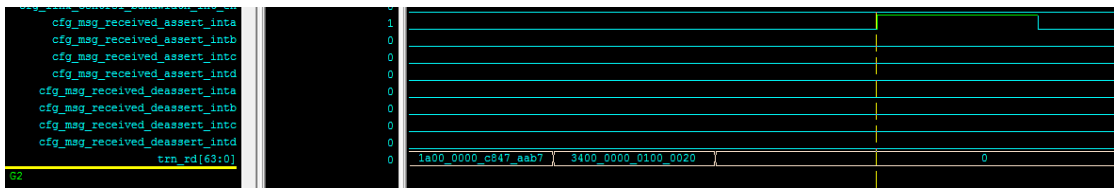


图 7 INTA# Assert

收到 INTA# 中断后的处理程序需要将 INTA# 拉高，如下图，收到的数据包内容为 0x3400\_0000\_0100\_0024, 0x0000\_0000\_0000\_0000, Fmt 为 2'b01, Type 为 5'b10100, 是消息请求, Message Code 为 8'b0010\_0100, 是中断 (INTx) 撤销消息。下图可以看到 received\_deassert\_inta 为 1, 收到 INTA# 中断撤销消息。

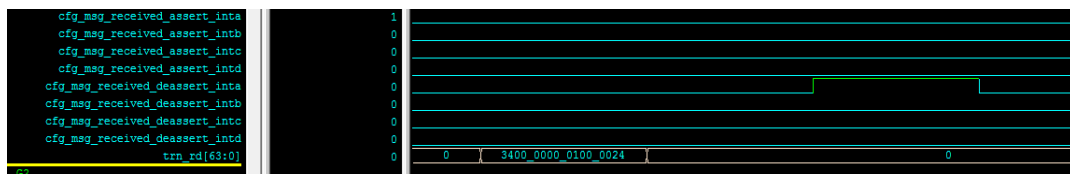
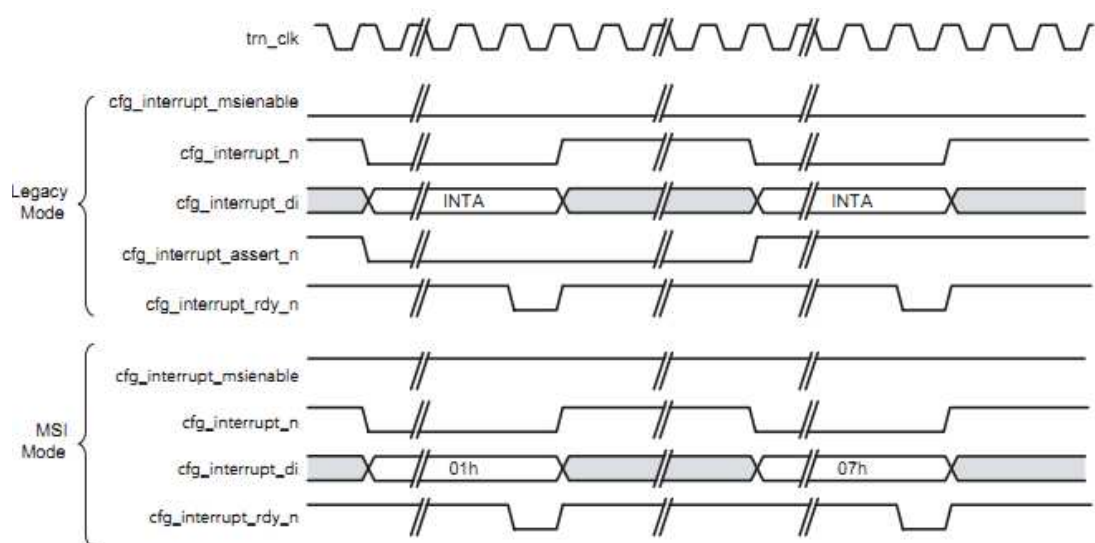


图 8 INTA# Deassert

MSI 是基于消息机制的, PC 启动后为 PCIe 板卡分配一个或多个消息地址, 板卡发送中断只需要向对应的地址内发送消息即可。消息内容中包含消息号, 每个消息号对应应在 PC 端的某一地址。

对比两种方式, INTx 是模拟 PCI 上中断线, MSI 是基于消息机制的。具体更细节的内容可以参考文献。需要注意的是, Windows XP 是不支持 MSI 的, 到了 WinNT 和 Win7 才支持。

Xilinx 的 PCIe 支持 INTx 和多达 256 条的 MSI 消息。有人担心了, 组织这些中断包发送, 那设计会不会很复杂? 在 Xilinx 的平台上, 中断和其他包是分开的, 中断发送是非常简单的, 并不需要用到 LocalLink 的 TX 接口, 只需要简单操作几条信号线就可以实现, 具体可参考对应 PCIe Core 的手册, 例如 V6 就可以参考 UG517。中断只需要操作 5 条信号线, PCIe 核就可以自己组织需要的中断包向外发送。





## 二，器件选型

Xilinx 在 Virtex 5 系列，Virtex 6 系列，Spartan 6 系列，还有刚刚发布的 7 系列 FPGA，Zynq-7000 系列都有 PCIe 的硬核 IP Core。各系列支持的 PCIe 硬核的速度分别为：

Product Name	FPGA Architecture	User Interface Width	Lane Widths Supported	Link Speeds Support	PCI Express Base Specification Compliance
8-lane	Virtex-5	64	x1, x2, x4, x8	2.5 Gb/s	V1.1
1-lane	Spartan-6	32	x1	2.5 Gb/s	V1.1
8-lane	Virtex-6	64, 128	x1, x2, x4, x8	2.5 Gb/s, 5.0 Gb/s	V2.0
4-lane	Artix-7	64	X1, x2, x4	2.5 Gb/s, 5.0 Gb/s	V2.0
8-lane	Kintex-7	64,128	x1, x2, x4, x8	2.5 Gb/s, 5.0 Gb/s	V2.0
8-lane	Virtex-7	64, 128, 256	x1, x2, x4, x8	2.5 Gb/s, 5.0 Gb/s, 8.0 Gb/s	V3.0

表 3 个系列支持的 PCIe 硬核规格

在决定设计的时候，就需要综合考虑成——选择芯片的系列、容量、速度等级；所需 PCIe 的速——选择 PCIe 最大的带宽；产品的无缝升级性——考虑位宽的变化对设计带来的影响等。

## 三，仿真环境搭建

有时候接到 PCIe 方面的电话，问题是：我那边一个 FIFO 产生数，通过 PCIe 发到电脑上，为啥最后（第一个）数据总是错误的？或是，我明明写 DMA 开始的寄存器了，怎么感觉板卡什么数据都没发呢？我用 Chipscope 抓数据，FPGA 太小了，抓的长度不够怎么办？这些问题我都只能回答一个方法，先把仿真环境搭建起来吧。

从我自己做 PCIe 的设计来说，大部分时间都在设计->仿真->修改->仿真中，即使遇到需要用 Chipscope 捕捉数据，那也是捕捉到仿真中没有考虑到的数据包，然后修改仿真激励，然后在仿真中复现刚才捕捉到的错误，然后再考虑修改设计。所以说做 PCIe 的设计，仿真是 too important to emphasis.

在 Xilinx 的平台，已经有完整的仿真平台框架了，在 PCIe 的用户手册中也有相当长的篇幅讲仿真。仿真平台的作用就是模拟一个设备，如果编写的是 PCIe 的 Device 设备，那就需要仿真出一个 Root Complex（可以理解为 PC 机）；如果编写的是 Root Complex（例如，FPGA 做主操作一个 PCIe RAID 卡），那就需要仿真出一个 Device 设备。这里简单介绍一下我用过的仿真。

### 1，Xilinx 的 DSPORT

用 Coregen 生成一个 PCIe 的工程，在 simulaion/dsport 中即是 DSPORT 的开源代码。该

仿真搭建起一个框架，已经完成了很多的发包、检测包等功能。例如，下面定义的 TASK 是 Type0 的配置读请求：

```
/******  
Task : TSK_TX_TYPE0_CONFIGURATION_READ  
Inputs : Tag, PCI/PCI-Express Reg Address, First ByteEn  
Outputs : Transaction Tx Interface Signaling  
Description : Generates a Type 0 Configuration Read TLP  
*****/
```

```
task TSK_TX_TYPE0_CONFIGURATION_READ;  
    input    [7:0]    tag_;  
    input    [11:0]   reg_addr_;  
    input    [3:0]    first_dw_be_;  
    begin  
        if (trn_lnk_up_n) begin  
  
            $display("[%t] : Trn interface is MIA", $realtime);  
            $finish(1);  
  
        end  
  
        TSK_TX_SYNCHRONIZE(0, 0);  
  
        trn_td          <= #(Tcq)    {  
                                1'b0,    // Researved Field  
                                2'b00,   // Fmt  
                                5'b00100,// Type  
                                1'b0,  
                                3'b000,  
                                4'b0000,  
                                1'b0,  
                                1'b0,  
                                2'b00,  
                                2'b00,  
                                10'b0000000001, // 32  
                                COMPLETER_ID_CFG,  
                                tag_,  
                                4'b0000,  
                                first_dw_be_    // 64  
                                };  
  
        trn_tsof_n      <= #(Tcq)    0;  
        trn_teof_n      <= #(Tcq)    1;  
        trn_trem_n      <= #(Tcq)    0;
```

```

trn_tsrc_rdy_n    <= #(Tcq)    0 ;

TSK_TX_SYNCHRONIZE(1, 0);

trn_td            <= #(Tcq)    {
                                COMPLETER_ID_CFG,
                                4'b0000,
                                reg_addr_[11:2],
                                2'b00,
                                32'b0
                                };

trn_tsof_n        <= #(Tcq)    1;
trn_teof_n        <= #(Tcq)    0;
trn_trem_n        <= #(Tcq)    8'h0F;
trn_tsrc_rdy_n    <= #(Tcq)    0 ;

TSK_TX_SYNCHRONIZE(1, 1);

trn_teof_n        <= #(Tcq)    1;
trn_trem_n        <= #(Tcq)    0;
trn_tsrc_rdy_n    <= #(Tcq)    1;

```

**end**

**endtask** // TSK\_TX\_TYPE0\_CONFIGURATION\_READ

对照着事物协议将的内容, 这个 TASK 组织了一个 Fmt=2'b00, Type=5'b00100 的包, TASK 中需要数据 Tag 和需要读取配置空间的地址 Addr。

在 pci\_exp\_expect\_tasks.v 中定义有接收包的 TASK, 例如下面就是等待接收 MRd 包的 TASK。

```

/*****
Task : TSK_EXPECT_MEMRD
Inputs : traffic_class, td, ep, attr, length, last_dw_be,
        first_dw_be, address
Outputs : status 0-Failed 1-Successful
Description : Expecting a memory read (32-bit address) TLP
              from Rx side with matching header fields
*****/
task TSK_EXPECT_MEMRD;

    input    [2:0] traffic_class;
    input          td;
    input          ep;
    input    [1:0] attr;
    input    [9:0] length;

```

```

input  [15:0] requester_id;
input  [7:0]  tag;
input  [3:0]  last_dw_be;
input  [3:0]  first_dw_be;
input  [29:0] address;

output          expect_status;

reg  [2:0]  traffic_class_;
reg          td_;
reg          ep_;
reg  [1:0]  attr_;
reg  [9:0]  length_;
reg  [15:0] requester_id_;
reg  [7:0]  tag_;
reg  [3:0]  last_dw_be_;
reg  [3:0]  first_dw_be_;
reg  [29:0] address_;

integer      i_;
reg          wait_for_next;

begin
    wait_for_next = 1'b1; //haven't found any matching tag yet
    while(wait_for_next)
    begin
        @ rcvd_memrd; //wait for a rcvd_memrd event
        traffic_class_ = frame_store_rx[1] >> 4;
        td_ = frame_store_rx[2] >> 7;
        ep_ = frame_store_rx[2] >> 6;
        attr_ = frame_store_rx[2] >> 4;
        length_ = frame_store_rx[2];
        length_ = (length_ << 8) | (frame_store_rx[3]);
        requester_id_ = {frame_store_rx[4], frame_store_rx[5]};
        tag_ = frame_store_rx[6];
        last_dw_be_ = frame_store_rx[7] >> 4;
        first_dw_be_ = frame_store_rx[7];
        address_[29:6] = {frame_store_rx[8], frame_store_rx[9],
frame_store_rx[10]};
        address_[5:0] = frame_store_rx[11] >> 2;

        $display("[%t] : Received MEMRD --- Tag 0x%h", $realtime, tag_);
        if(tag == tag_) //find matching tag
        begin

```

```

wait_for_next = 1'b0;
if((traffic_class == traffic_class_) &&
    (td == td_) && (ep == ep_) && (attr == attr_) &&
    (length == length_) && (requester_id == requester_id_) &&
    (last_dw_be == last_dw_be_) && (first_dw_be == first_dw_be_) &&
    (address == address_))
begin
    // header matches
    expect_status = 1'b1;
end
else // header mismatches, error out
begin
    $fdisplay(error_file_ptr, "[%t] : Found header mismatch in
received MEMRD - Tag 0x%h: \n", $time, tag_);
    $fdisplay(error_file_ptr, "Expected:");
    $fdisplay(error_file_ptr, "\t Traffic Class: 0x%h",
traffic_class);
    $fdisplay(error_file_ptr, "\t TD: %h", td);
    $fdisplay(error_file_ptr, "\t EP: %h", ep);
    $fdisplay(error_file_ptr, "\t Attributes: 0x%h", attr);
    $fdisplay(error_file_ptr, "\t Length: 0x%h", length);
    $fdisplay(error_file_ptr, "\t Requester ID: 0x%h",
requester_id);
    $fdisplay(error_file_ptr, "\t Tag: 0x%h", tag);
    $fdisplay(error_file_ptr, "\t Last DW byte-enable: 0x%h",
last_dw_be);
    $fdisplay(error_file_ptr, "\t First DW byte-enable: 0x%h",
first_dw_be);
    $fdisplay(error_file_ptr, "\t Address: 0x%h", address);
    $fdisplay(error_file_ptr, "Received:");
    $fdisplay(error_file_ptr, "\t Traffic Class: 0x%h",
traffic_class_);
    $fdisplay(error_file_ptr, "\t TD: %h", td_);
    $fdisplay(error_file_ptr, "\t EP: %h", ep_);
    $fdisplay(error_file_ptr, "\t Attributes: 0x%h", attr_);
    $fdisplay(error_file_ptr, "\t Length: 0x%h", length_);
    $fdisplay(error_file_ptr, "\t Requester ID: 0x%h",
requester_id_);
    $fdisplay(error_file_ptr, "\t Tag: 0x%h", tag_);
    $fdisplay(error_file_ptr, "\t Last DW byte-enable: 0x%h",
last_dw_be_);
    $fdisplay(error_file_ptr, "\t First DW byte-enable: 0x%h",
first_dw_be_);
    $fdisplay(error_file_ptr, "\t Address: 0x%h", address_);

```

```

        $fdisplay(error_file_ptr, "");
        expect_status = 1'b0;
    end
end
end
end
endtask

```

根据所需设计的 PCIe 功能的需要，可以在这些文件中添加所需要的 TASK。

## 2，PLDA 的 BFM

PLDA (<http://www.plda.com/index.php>) 在 ASIC 和 FPGA 的高速互联协议技术 IP Core 设计中处于领先地位.....(我不是托，不介绍多)。他们的 PCIe 仿真简单易用，例如

```

`BFM.xbfm_burst
(`XBFM_MRD,64'h2222222222220000,1024,databuf,3'b000,2'b00);完成了 Burst
MRd TLP。

```

```

`BFM.xbfm_wait_event(`XBFM_INTAA_RCVD);完成了等待中断的事件。

```

该 BFM 并不开放代码，但是 PLDA 提供了详细完整的说明文档以及用例，并且提供主流仿真工具的库。

平常见过的大部分工程师使用的仿真工具是 Modelsim。碰到 PCIe 工程仿真，一次流程下来时间是比较长的，如果加上 DDR 等仿真，一次可能需要 15 分钟。用 Modelsim 的工程师碰上的一个头疼的问题，没有加入 Wave 的波形看不到，无奈的 restart，再 run -all。在仿真 PCIe 的工程时这个问题尤为明显。解决这个问题，可以试试 Google “Modelsim Debussy” 或者 “Modelsim dump vcd”，前一种方法用 Debussy 保存所有到 fsdb 文件，后一种方法保存到 vcd 文件。如果想进一步提高仿真速度，可以试试 Google “VCS verdi”。

## 四，调试

编写好用户逻辑处理 TLP，仿真通过，程序正常编译通过，下载到 FPGA，重启机箱，Windows 弹出寻找驱动的框，那恭喜了，板卡被检测到了。对于这种人品好的情况，可以跳过下一段落。

有一部分没有检测到的情况，需要一步一步的排查问题了。PCIe 核用户手册中有详细讲如何一步一步诊断，官方网站也有各种案例和调试步骤等，

- 下面文章总结了一下不能识别的各种案例，你可以先参考一下。

<http://www.xilinx.com/support/answers/34777.htm>

- 下面有一些硬件调试步骤

<http://www.xilinx.com/support/answers/34151.htm>

- 不能识别的时候可以按照下面 Chipscope 的列表抓出下面这些信号看看

<http://www.xilinx.com/support/answers/39488.htm>

官方的文档覆盖了非常广的诊断，平常多数发现不了板卡的案例错误有：

- 1，时钟选择错误

PCIe 由主板给过来的时钟是 100Mhz，如果外部添加了 PLL 倍频到 125Mhz 或者 250Mhz，



那在生成 PCIe 核的时候，时钟需要对应的选择。这个问题在做仿真的时候，检查一下顶层给出的时钟是否和板卡上的匹配，一般在仿真阶段就可以排除掉。

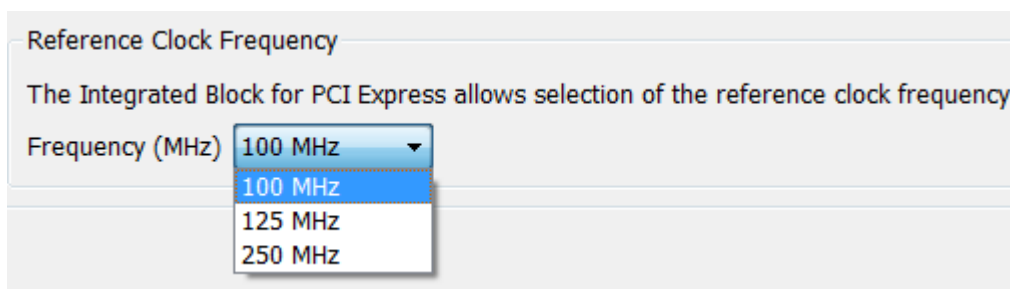


图 9 Ref Clock 选择

- 2, GTP (GTX) 未能正常工作  
平常碰到的有电源供电低于要求电压，或者纹波高于所能容忍的要求。或者 PCIe 经过的 PLL 输出的时钟不满足要求。或者 GTP(GTX)矫正电阻阻值不正确。
- 3, UCF 错误  
UCF 没有根据板卡设置，检查 GTP(GTX) 和时钟位置，以及复位的管脚约束。
- 4, 板卡某 Lane 上信号完整性错误  
某条 Lane 上有信号完整性问题，导致链路不能正确训练。可以试试用薄的绝缘胶带将 x2-x8 的 lane 都粘帖起来，只留下 x1 Lane。如果 x1 Lane 有问题，就没什么办法了。
- 5, PCIe 槽有特定的设置  
有些高级的服务器有管理功能，例如第一个 x16 的槽位只能安装显卡，如果检测不是显示卡，甚至 PC 不能启动。可以试试找一个普通的兼容 PC 机。如果有 Xilinx 的开发板，直接下载官方的程序，按照文档上的步骤一步一步执行，看 PC 能否发现板卡。

上面 5 点仍然没有解决，那就需要按照给出的链接，检查 FPGA 的内部信号以判断。

## 五，驱动编写

驱动是用户程序和硬件中间的一层封装，一个好的驱动对提高系统的性能和稳定性至关重要。上面提高过的包，PC 会主动发起 BAR 空间的读写包，并相应板卡发送的完成包、消息包等；板卡端也可以向 PC 发送 MWrr, MRd, Msg 等 TLP 包。在 BAR 空间章节讲到，PCIe 板卡只有 BAR 空间能够被 PC “看到”，BAR 空间内的操作就可以用一些工具直接读写访问。如下图是用 Windriver 读写 BAR2 空间。

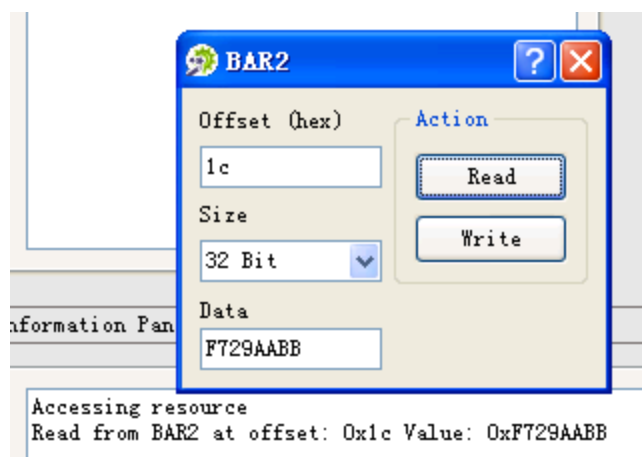


图 10 BAR 空间读写

而 BAR 空间之外的操作，例如板卡的中断，板卡发送的 MWr TLP 包就需要通过写驱动来“观察”到。

常见的在 Windows 下编写驱动的方法有：

1, Windriver

Windriver (<http://www.jungo.com/st/index.php>) 是一个跨平台的驱动编写软件，它在操作系统的底层放置了一个驱动，然后用户编写的驱动和 Windriver 的驱动交互。这样可以最大的保证用户编写的代码可移植性和兼容性。对于不太熟悉驱动编写的工程师来说，用 Windriver 是一个非常不错的选择，其丰富详尽的文档以及众多的例程可以快速完成驱动的编写。唯一不足之处是它不是免费的。

2, DriverStudio

说到 DriverStudio, 就不得不提到早期 WDM (Windows 驱动开发框架) 的复杂, DriverStudio 将 Windows 的 WDM 封装起来, 提供用户简单的接口。但是不足之处也很多, 一是这款软件早已经停止开发了, 二是不能在 Win7 上使用。

3, DDK

对于 Windows 的驱动开发, DDK (Driver Development Kit) 是 MS 提供的一整套开发框架。初学者或许要花费一些时间精力才能真我 DDK 的开发。

在 Linux 下开发驱动, 可以购买一份 Windriver Linux 版, 或者用 Linux 的驱动编写方法吧。Xilinx 在 XAPP1052 参考设计中提供了匹配该 DMA 设计的 Windows DDK 和 Linux 驱动的代码, 作为学习可以参考使用。

## 六，总结

最后, 用 Xilinx XAPP1052 的 DMA 设计将上面的内容串联起来, 也为下面 DMA 模式做一个铺垫。XAPP1052 是一个简单的板卡作为 Master 的 DMA, 用于验证 Xilinx PCIe 硬核接口所能达到的速度。其原理框图如下:

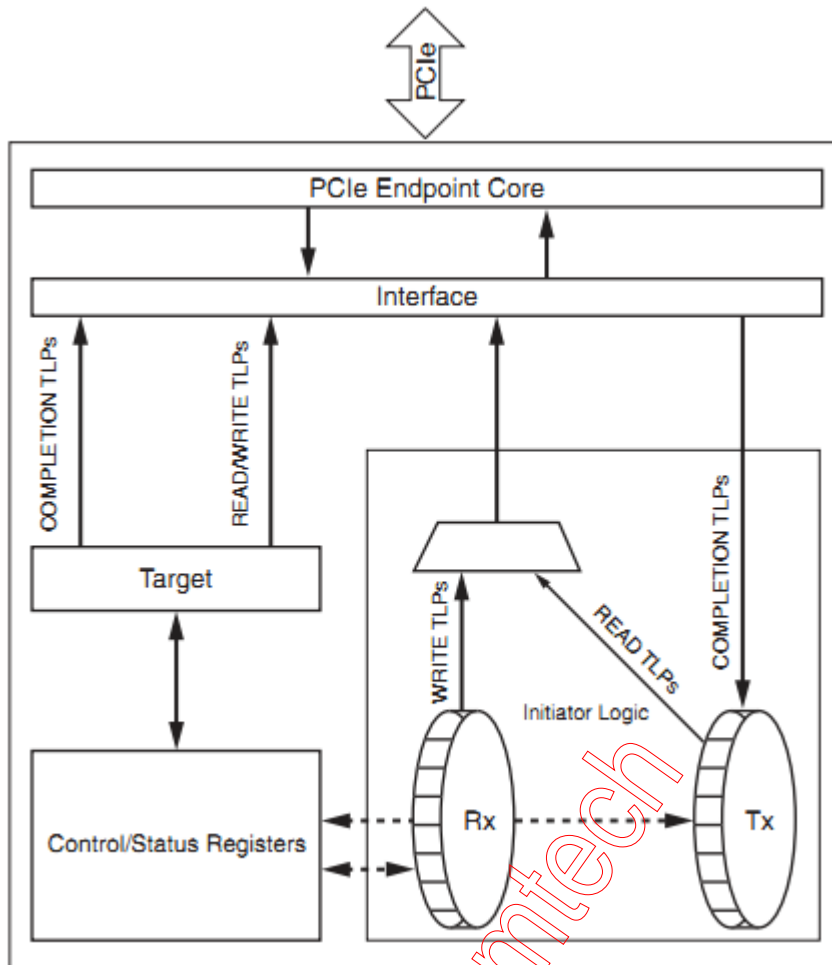


图 11 XAPP1052 参考设计框图

**Target logic** 用于捕捉 PC 发出的单次 MWrr 和 MRd。如果是 MWrr TLP，那根据写入的地址和内容来对应改变 Control/Status Registers 的内容；如果是 MRd TLP，该段逻辑读取 Control/Status Registers 的内容，然后根据 MRd 的地址回复相应的 Cpld TLP。

**Control and status registers** 包含控制 DMA 控制器的寄存器，这些寄存器均映射在 BAR 空间中。PC 通过 BAR 空间的读写，产生 MWrr 和 MRd 包，经过上面的 Target Logic 相应的写入或读取寄存器，从而控制 DMA 的操作。

**Initiator Logic** 用于产生 MWrr 和 MRd 以及分析 PC 相应的 Cpld 包，是完成 DMA 读写的主要部分。

板卡将板内的数据发送到 PC，并由 PC 的应用程序能处理的一个完整流程为：

#### 1, 复位 Initiator

```
case IOCTL_INITIATOR_RESET:
    // Perform a reset of the Initiator device by setting the low bit
    of the Device
    // Control Register (DCR) high.
    KdPrint(("s3_1000.sys: IOCTL_INITIATOR_RESET\n"));
    address = (ULONG) deviceExtension->MemoryStart[FPGA] +
    DCR_OFFSET;
    length = 1;
    WRITE_REGISTER_ULONG ((PULONG) address, 0x1);
```

```

        // A reset happens prior to setting up any DMA transfer, so this
is a good time
        // to also initialize the transfer length values in our device
extension.

deviceExtension->ReadLength = 0;
deviceExtension->WriteLength = 0;

// Reset the interrupt monitor variables.
deviceExtension->ReadDone = FALSE;
deviceExtension->WriteDone = FALSE;

break;

```

其中，address 计算的是 BAR 空间的物理地址，deviceExtension->MemoryStart[FPGA]得到的是板卡 BAR 空间的起始地址（BAR 空间章节内叙述过），DCR\_OFFSET 的值为 0x0（见 ioctl.h 中定义）。

WRITE\_REGISTER\_ULONG ((PULONG) address, 0x1); 这行命令是在 address 中写入一个 32 位（ULONG）的数据。于是，在板卡端就会接收到一个 MWr 包，由于写的是 BAR 空间，Xilinx 会有一位 bar\_hit[6:0] 表示写入的是那一个 BAR 空间，同时 MWr 包中可以提取出需要写入的地址，在这个命令中就是 DCR\_OFFSET。

```

`BMD_64_RX_MEM_WR32_QW1 : begin

    if ((!trn_reof_n) &&
        (!trn_rsrc_rdy_n) &&
        (!trn_rdst_rdy_n)) begin

        addr_o      <= trn_rd[44:34];
        wr_data_o    <= trn_rd[31:00];
        wr_en_o      <= 1'b1;
        trn_rdst_rdy_n <= 1'b1;
        bmd_64_rx_state <= `BMD_64_RX_MEM_WR32_WT;

    end else

        bmd_64_rx_state <= `BMD_64_RX_MEM_WR32_QW1;

    end

case (a_i[6:0])

    // 00-03H : Reg # 0
    // Byte0[0]: Initiator Reset (RW) 0= no reset 1=reset.
    // Byte2[19:16]: Data Path Width
    // Byte3[31:24]: FPGA Family

```

```

7'b0000000: begin

    if (wr_en_i)
        init_rst_o <= wr_d_i[0];

        rd_d_o <= {fpga_family, {4'b0}, interface_type,
version_number, {7'b0}, init_rst_o};

    if (init_rst_o) begin

        mwr_start_o <= 1'b0;
        mrd_start_o <= 1'b0;

    end

end

```

上面一段代码显示，当接收到 MWr 包时，TLP 中的地址和数据均被记录下来，同时 wr\_en 被置高一次。下面一段代码，检测到 wr\_en 后，判断如果地址是 7'b0000000，就将写入数据的最低位赋值到 init\_rst\_o 中。软件中写入的 BAR 空间地址为 0，数据为 1，因此 init\_rst\_o 就被写入 1，从而达到了复位 Initiator 的结果。

下面其他写入寄存器的流程同复位 Initiator 的流程相同，只有偏移地址和写入的 32 位数据不同，下面再提及 BAR 空间写入数据的分析过程同上面一样了。

## 2, 清除 Initiator 的复位

```

case IOCTL_CLEAR_INITIATOR_RESET:
    // Clears a reset of the Initiator device by setting the low bit
of the Device
    // Control Register (DCR) low, then setting it low again.
    KdPrint(("s3_1000.sys: IOCTL_CLEAR_INITIATOR_RESET\n"));
    address = (ULONG) deviceExtension->MemoryStart[FPGA] +
DCR_OFFSET;
    length = 1;
    WRITE_REGISTER_ULONG ((PULONG) address, 0x0);
    break;

```

## 3, 填写相关 DMA 信息

例如，TLP 大小，DMA 发送数据的数据 (PATTERN)，DMA 的目的起始地址，DMA 传输的数据长度等，操作对应的驱动和 RTL 代码都可以参考第 1 步和第 2 步。这里有一点需要注意的是，在 PC 上需要申请一个物理地址连续的缓冲区，因为硬件发送的数据地址是顺序递增的。

## 4, 启动 DMA

在设置好一些列寄存器后，启动 DMA，然后等待传输结束。

```

case IOCTL_DMA_START:
    // Initiate any DMA transfers that have been set up in the Read/Write
registers.
    status = STATUS_PENDING;
    LogMsg(DeviceObject, L"Control: PCIe BMDMA Transfer Starting");

```

```

        address = (ULONG) deviceExtension->MemoryStart[FPGA] +
DCSR_OFFSET;
        length = 0; // Nothing
returned
        regValue = READ_REGISTER_ULONG ((PULONG) address); // Read the
entire register to preserve bits
        regValue |= *pBuffer; // Set the new
register bits passed in
        KdPrint(("s3_1000.sys: IOCTL_DMA_START"));
        KdPrint((" dma start: pBuffer = 0x%x, New DCSR = 0x%x\n",
*(pBuffer), regValue));
        WRITE_REGISTER_ULONG ((PULONG) address, regValue); // Write
the new register value
        break;

```

#### 5, 处理 DMA 完成

如果开启了 DMA 完成后发送中断，那么可以在中断服务程序里或者中断延迟过程调用中处理（DPC）；如果没有开启，那么可以通过通过查询 DMA 寄存器的方式判断 DMA 是否已经完成。

在传统的 INTx 型中断中，所有挂在于该中断线的设备都可能产生中断，换句话说，如果这条中断线上有中断，操作系统会将这些设备的中断服务程序都执行一边。所以，下面的代码是读取板卡的寄存器状态，然后判断是否为对应板卡发生的中断。

```

        address = (ULONG) pDevExt->MemoryStart[FPGA] + DCSR_OFFSET;
        regValue = READ_REGISTER_ULONG ((PULONG) address); // Read the
register

// The following should be combined
if (~pDevExt->ReadDone && (regValue & 0x1000000)) // Read Complete
{
    pDevExt->busy = FALSE; // prevents race with CheckTimer
    pDevExt->timer = -1;
    IoRequestDpc(pDevObj, pDevExt->ReadIrp, (PVOID)pDevExt);
    pDevExt->ReadDone = TRUE; // One read operation
only per run
    myInt = TRUE;
}

```

上面 regValue = READ\_REGISTER\_ULONG ((PULONG) address); // Read the register 从板卡 BAR 空间中读 32 位（ULONG）寄存器的操作，address 为 BAR 空间的起始地址加上 0x4，对应的 HDL 代码为：

```

7'b0000001: begin

    if (wr_en_i) begin
        mwr_start_o <= wr_d_i[0];
        mwr_relaxed_order_o <= wr_d_i[5];
    end
end

```



```

mwr_nosnoop_o <= wr_d_i[6];
mwr_int_dis_o <= wr_d_i[7];
mrd_start_o <= wr_d_i[16];
mrd_relaxed_order_o <= wr_d_i[21];
mrd_nosnoop_o <= wr_d_i[22];
mrd_int_dis_o <= wr_d_i[23];
end

rd_d_o <= {cpld_data_err_i, 6'b0, mrd_done_o,
           mrd_int_dis_o, 4'b0, mrd_nosnoop_o,
mrd_relaxed_order_o, mrd_start_o,
           7'b0, mwr_done_i,
           mwr_int_dis_o, 4'b0, mwr_nosnoop_o,
mwr_relaxed_order_o, mwr_start_o};

```

当地址为 0x4（这里最低的两位去掉了）时，rd\_d\_o 赋值了一些 DMA 状态，然后 FPGA 组织了一个 Cpld 包回应，READ\_REGISTER\_ULONG 于是返回 rd\_d\_o 的值。

一般在中断 ISR 中仅执行基本的中断清楚等操作，剩下的事情放在延时过程调用中。

整个 DMA 的流程是通过软件、驱动、HDL 代码无缝的配合才能完成的。

Comtech

## 进阶

### DMA 模式

#### Block DMA

上述 xapp1052 就是一个典型 Block DMA，PC 申请一小片物理空间，设置好对应寄存器后启动 DMA。由于 OS 上不容易申请一大片连续的物理空间，如果一次 DMA 传输的数据比较大，比如说 100MB，那就需要分很多次传输。对于这 100MB 数据，可以在 PC 上申请 512K 的，然后重复 DMA 200 次即可。显而易见，每次中断 OS 会有开销，而且每次传输 DMA 完成后，需要将这 512K 拷贝到其他内存区域，否则下次 DMA 后数据就被覆盖了。中断处理和数据拷贝是 Block DMA 性能进一步提高的障碍。

#### Scatter Gather DMA

Scatter Gather DMA（SG DMA）就方便的解决 Block DMA 中的问题，他通过建立链表式的描述符，每个描述符包含 PC 这端的物理地址、传输长度、是否是这个链表的最后一个、下一个链表的物理地址等信息，也就是说包含了一次 DMA 所需要的全部信息。板卡顺次执行链表的 DMA 指定的信息，在完成最后一个描述符后就通过一次中断或查询作为操作的结束。这种方式是如何提高 DMA 传输的效率呢？举一个视频处理的例子，视频的处理往往以一幅（帧）图像为单位，一幅图像的大小有的可达数十 MB。在 PC 上申请几十 MB 连续的物理内存，很可能返回分配失败。那可以在 PC 上申请内存中的页表（Windows 上非分页内存的一个页表为 4K，申请几十 MB 这种内存是比较轻松的），然后将页表组织成描述符链表并启动 SG DMA。板卡按照链表的信息一次性将这写页表填满数据。由于非分页内存存在各级别均可以使用，用户的程序可以直接处理这些图像而不需要像 Block DMA 一样拷贝到用户程序才能处理。对比 Block DMA，几百次的中断处理剩下一一次，几百次的内存拷贝也没有了，从而效率大大的提高。

#### Minimum Latency DMA

有的应用需求的数据量并不是很大，但是对数据从进入板卡到 PC 能处理的延时非常关注，例如每到板卡 32B 后需要最短的时间按内能被应用程序处理。

如果采用 Block DMA，没收到 32B 就产生中断并利用 DMA 传输至内存，首先是中断的处理已经是 uS 级别，其次是总体来说的数据效率非常低，PC 的负载也很高。

采用 SG DMA 效果也差不多，延时和速度不可兼得。

对于这种情况，可以借鉴 SG DMA，在 PC 内生成一个环状链表，链表内的信息比 SG DMA 多一个 Stick Bit。板卡每收到 32B/64B 就组织一个 MWr 包填充一个描述符指定的空间，同时再发送一个 MWr 修改这个 Stick Bit。软件顺序判断哪些 Stick Bit 被修改了，就可以去区 Stick

Bit 所属描述符的空间了。由于是环状链表，板卡运行起来可以循环运行，用户那边处理的速度需要快过板卡接收的速度。

这种方式不经过中断，也不去查询板卡的信息，仅查询 PC 本地内存的数据，所以系统负载以及时延都可以有效的控制。

## Multi-DMA

当有多路独立的数据需要通过 DMA 传输到 PC，对目前的设计可以通过寄存器控制 DMA 接收的数据源。当然，也可以有多个独立 DMA 核，每个 DMA 负责其中一个或多个收发通道，这种方式可以简化 HDL 的结构以及软件和驱动的结构。

Xilinx 目前的参考设计不支持 SG 功能和 Multi-DMA 功能，而一般成熟的商业 DMA 都包含这些功能，例如 PLDA，NWlogic。

## DDR 缓冲

无论是 Block DMA 还是 SG DMA，如果数据速率较高，例如对 PCIe 1.1 x8lane 下需要接收 400MBps 的数据，或者是需要形成完整的帧 PC 才能读取，这些情况需要 FPGA 外先缓存接收/发送的数据。DDR，QDR，SRAM 都可以作为这些缓冲，总体来说，DDR 作为容量大、成本低、速度高的一种存储器，非常适合做这种缓冲。

## Virtual FIFO

对于顺序的、没有帧结构的数据流，将 DDR 虚拟成一个多路 Virtual FIFO 可以有效配合 DMA。数据输入只需要送入一个 FIFO Write 接口，而 DMA 从 FIFO Read 接口读取数据即可。

Xilinx 提供了一个开源的参考设计，将 DDR3 封装成多路 Virtual FIFO，效率可以达到 DDR3 最大带宽的 85%，因此，对速率高、或者有多种数据通路的设计可以方便的集成到 DMA 接口。

## Virtual RAM

对于有帧结构，并且帧长度变化的，例如变速率的视频帧。对于带帧结构的应用，用户程序往往系统一个 DMA 传输将整帧传输到接收区中，并且 0 地址对应这帧头。如果将 DDR 封装为缓存，在帧长度变化的时候，应用程序通过 DMA 读取的数据往往丢失了帧头。因此对这样应用，将 DDR 封装为 Virtual RAM 方便用户程序的处理。

Xilinx 的 MIG 生成 DDR 接口包含命令通道、地址通道、写入数据通道、读出数据通道，已经可以作为一个 Virtual RAM 了。在 Xilinx 的这个接口上完成仲裁、并记录读数据的顺序就可以方便的扩展为多路的 Virtual RAM。

## 经验、教训总结

通过上面的讲解，希望对刚开始接触、并不太熟悉的工程师来说能够抛砖引玉。

PCIe DMA 是一个系统功能，如果需要从基础的参考设计开始做产品，那对工程师来说是一个非常不错的锻炼机会。

如果是产品设计，采用成熟的合作第三方也是一种不错的选择。第三方的设计有良好的组织结构，完整的指导文档，以及配合的驱动程序和测试程序。

---

<sup>1</sup> 马鸣锦，朱剑冰，何红旗，杜威. PCI、PCI-X 和 PCI Express 的原理及体系结构. 清华大学出版社，2007-4.

<sup>2</sup> MindShare, Inc , Ravi Budruk, Don Anderson, Tom Shanley. PCI Express System Architecture. Addison Wesley

Comtech