PCIE 学习笔记

文档主要包括四个部分：1）Magwizard 中例化模块的说明；2）内部结构；3）结合实际应用介绍应用层接口信号（我们主要帮客户解决这部分的问题，底层软件驱动部分由客户自己开发，Altera 不负责支持）；4）学习初期疑问及 AE 的解答。

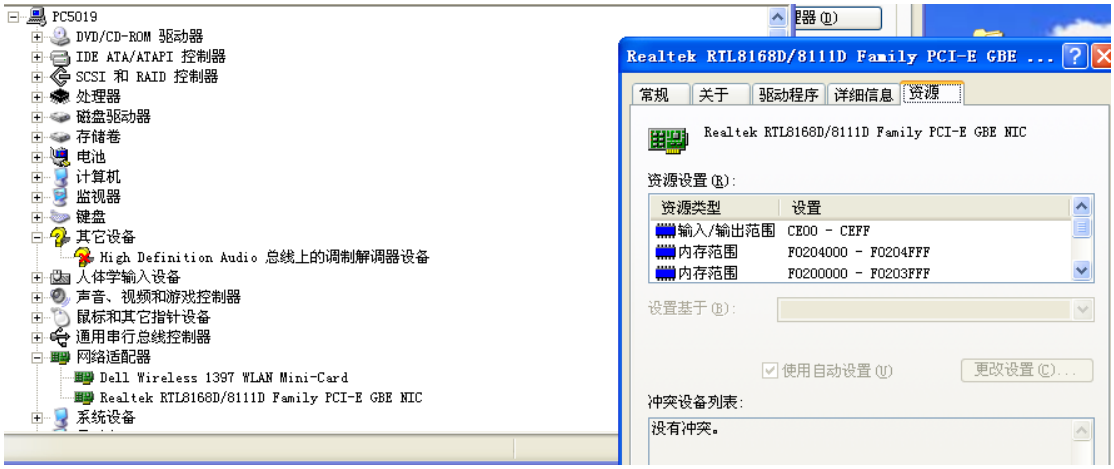
PCI Express Compiler 说明

一〉system setting:

1) Pcie core 的类型：软核、硬核。IVGX 和 2AGX 包含硬核
2) PHY type: 选择用不同的器件来实现，可以看到下面支持 lane 的数量的不同。
3) Port type: Native Endpoint 是比较新的类型，支持 MSI 中断消息（推荐类型）。Legacy Endpoint 不支持。Root point 是源端,endpoint 是目的端。
4) Xcvr ref_clk: 设置 reclk 的输入时钟，可以在手册中清楚看到，对于不同的器件，输入参考时钟的区别。
5) Application Interface: 用于指定 PCI Express 中传输层和应用层的接口，如果用 MegaWizard，建议采用 Avalon-ST.
6) Application clock: 指定应用的接口时钟，在选择硬核和软核时有区别。
7) Max rate: Gen1(2.5Gbps), Gen2(5.0Gbps)
8) Test out width: 设置 test_out 的宽度，对于不同的核和 lanes 有不同的设置。
9) PCIe reconfig: 重配置硬核只读配置寄存器。

二〉PCI register

1) BAR Type：主机以何种形式访问外部设备。BAR 的数量？
2) 参考设备管理器中/网络适配器/属性。可以对应这些 ID。 MSI 消息中断，windows 不支持，在 Vista 或 linux 中支持



三〉Capabilities Parameters

1) Tags supported 4-256

设置支持 non-posted 请求的 tags 数目。

Hard IP：32 or 64 tags for X1，X4 和 X8 模式。

Soft IP：4-256 for X1 和 X4 模式，4-32 for X8 模式。

2）Implement completion timeout disable

该选项只对 Gen2 的 root ports 和 endpoints 有效.

3）Completion Time out range

你可以选择 ABCD，分别对应不同的时间范围。

4）Error Reporting

就是你是否想显示这些错误信息。

5）MSI Capabilities

用来设置应用层请求数量，将此值设置给消息控制寄存器。SOPC 只支持 1 个 MSI。

6）link Capabilities

Link common clock：是否用系统提供的普通参考时钟给 PHY 来做参考时钟，建议选用。

Data link layer active reporting:只在 root port 有效

Link port number: 将只读端口数目设置到 link Capabilities 寄存器中。

7）Slot Capability

Table 3-3 中详细介绍了 Slot capability 寄存器中各个值对应的意义。

8）MSI-X capabilities

此中断只对 Hard IP 有效

MSI-X Table size：只读信号，系统软件读这个地方来确定 MSI-X Table Size。（主要+1 的关系）

MSI-X Table Offset: 指向 MSI-X table 的基地址。只读。

BAR Indicator: 用来将 MSI-Xtable 映射到 memory 空间，只读。

四）Buffer Setup

该页包括了接收和重试 buffer 的设置。

Maximum payload size: 设置最大的有效载荷大小，对于不同的器件有不同的上限值。

Number of virtual channels: 设置虚拟通道数。

Number of low-priority VCs:设置虚拟通道在低优先级仲裁组的数量。该值只能小于或等于虚拟通道数。

Retry buffer size:设置重试 Buffer 的深度，用来存储发射 PCI Express 包直到被确认。（Hard IP 时默认值）

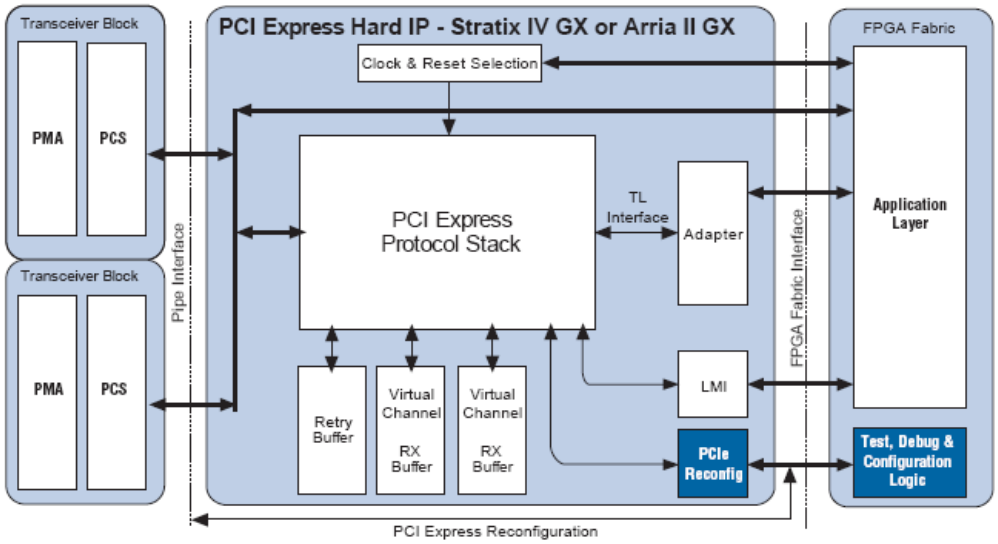
Maximum retry packets: 设置重试包的大小。（Hard IP 时默认值）

结构：

如果调用 Magwizard，完整的可以实现如下几层：

- Physical (PHY)
- Media Access Control (MAC)
- Physical Coding Sublayer (PCS)
- Physical Media Attachment (PMA)
- Data link
- Transaction

Figure 1–1. PCI Express Hard IP High-Level Block Diagram *(Note 1) (Note 2)*



一〉应用接口

有 Avalon-ST 接口、descriptor/data（soft IP only）、Avalon-MM 接口。Avalon 接口分成两种，一种是 Avalon-MM 接口，另一种是 Avalon-ST 接口。MM 接口是通过地址来读写数据，更多的用在控制逻辑上面。ST 接口是用于点到点的数据流接口，更多的用在高速通过率的模块中间。

RX Datapath

接收数据路径将传输层的数据发给 Avalon-ST 接口。有一个 FIFO 用来 buffer 传输层的数据直到流接口接受它。Avalon-ST RX datapath 有 3 到 6 个 pld_clk 周期的延迟。

TX Datapath

发射数据路径将 Avalon-ST 接口的数据发给传输层，有一个 FIFO 用来 buffer 传输层的数据传输层接受它

二〉处理层

处理层在应用层和数据链路层之间，它来产生和接收传输层的包。处理层包括：发射数据路径，配置空间和接收路径。你可以在 IVGX 中例化 1，2 个虚拟通道，在 2AGX 中只可以例化一个虚拟通道。

接收数据流程：

1） 处理层收到从数据链路层来的 TLP。

2） 配置空间用来确定处理层的包是否正确。

3） 在每个虚拟通道，处理层的包被存在接收 buffer 中一个特定的部分（由收发类型确定：posted,non-posted,completion）

4） 处理层 packet FIFO 块用来存储 buffer 传输层包的地址。

发射数据流程：

1） MegaCore 通过 tx_cred[21:0](for soft IP)或 tx_cred[35:0](for hard IP)来给应用层提供信息

2） 应用层会请求传输层给它包，此时应用层需要提供提供 PCI Express 传输字头在 tx_desc[127:0]中，已经数据在 tx_data[63:0]中。

3） Megacore 会确认 suffiicient flow control credits,并确定是相应还是延迟请求。

4） 处理层的数据被应用层转发。处理层仲裁各虚拟通道，然后选择优先级高的数据给

数据链路层（在 PCI Express Base Specification 1.0a，1.1 或 2.0 中，可以定义虚拟通道的优先级）。

三）数据链路层

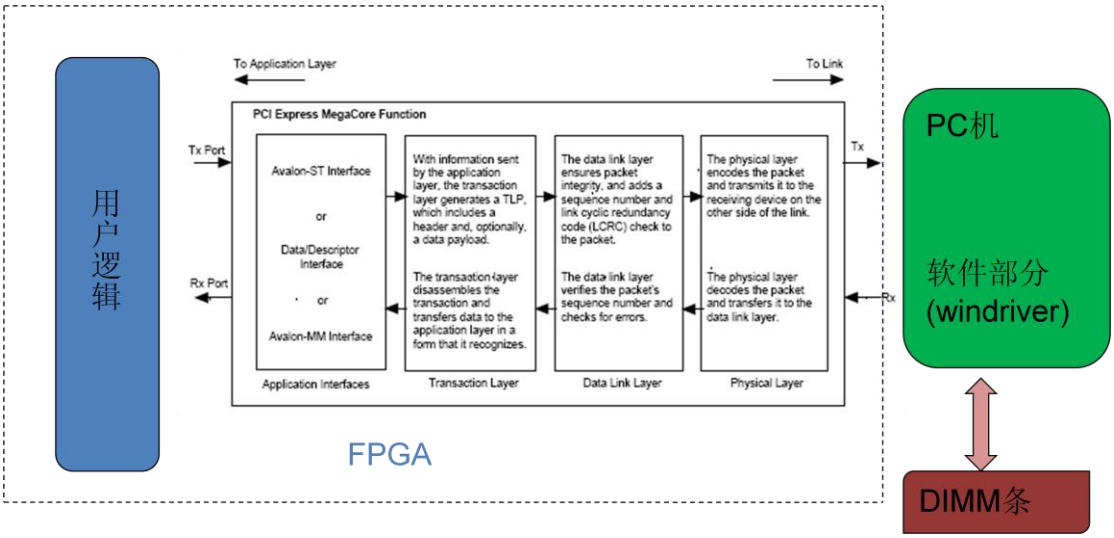
链路层在处理层和物理层之间，它主要负责包的信号完整性。Figure4-9 给出了数据链路层和与它相连的两层之间的接口信号。

四）物理层

物理层在 Megacore 的最底层，它通过高速 Serdes 和 link 相连。

实际应用（about chaining DMA）

下图是客户这边的一个实际应用结构简图。



我们主要需要帮工程师了解用户逻辑和 MegaCore 相连的的用法，软件部分主要由客户自己完成。

Altera 建议新的设计不要采用 Descriptor/Data 接口，而是采用 Avalon_ST。硬核和软核的主要功能区别就在于用 Avalon_ST 接口做配置和时钟管理。在 128-bits 模式，流接口时钟 pld_clk 是 core_clk 的一半；在 64bits 模式，两者相等。

Figure 5-1 和 Figure 5-2 分别给出了用 Avalon_ST 接口实现 Rootpoint 和 Endpoint 模式的信号。Table 5-2 和 Table 5-4 给出了 64/128 bits Avalon-ST 接收数据路径和发射数据路径中的信号。

由于客户实际应用采用硬核，所以这里主要介绍硬核。

1）64/128 bits Avalon_ST 接收端口

rx_st_ready: 用来指示应用层是否准备好接收数据。

Rx_st_valid: 和 rx_st_ready 之间的时序关系参考 Figure 5-13。

Rx_st_data: 接收数据总线。

Rx_st_sop: 包起始标志位

Rx_st_eop: 包结束标志位

Rx_st_empty：在 rx_st_data 数据低 64bits 结束传输时给出确认信号。该信号只在使用

硬核的 128bits 模式时有效。

Rx_st_err：当一个无法纠正的 ECC 错误被检测到，rx_st_err 会被拉高至少一个周期。（只在硬核中存在，并在 ECC 被使能时才有效）

Rx_st_mask: Application 使能这个信号来告诉 megacore 来停止发送 non-posted 请求。

Rx_st_bardec:

Rx_st_be: 参考 page137 页可以知道 rx_st_data 和 rx_st_be 之间的关系。

Root Port Mode Configuration Requests:

为了保证在 root port 模式时正常发送 CFG0 包，应用层需要等待，一直等到 CFG0 被发送到 Megacore 的配置空间，你可以在 CFG0 SOP 被发送到 Avalon-ST 后等至少 10 个周期，然后检查 tx_fifo_empty 是否为 0，再来确定是否发下一个包。

应用层可以执行 ECRC forwarding，但是对于 CFG0 包不需要执行 ECRC，此时 TLP 头中的 TDbits 会被置 0，因为这些包不需要传送到 PCI Express link 中

2）Clock Signals – Hard IP Implement

refclk : 模块的输入参考时钟，由 Parameter Settings 中确定。

Pld_clk: 应用层时钟，你必须用 core_clk_out 来驱动它。

Core_clk_out: 根据 link Width 和 Avalon-ST width 等来确定 Core_clk_out 的值，参考 Table 4-41。

P_clk: 只在仿真时采用

Clk250_out: 只在仿真时采用

Clk500_out: 只在仿真时采用

3） Reset Signal

Rstn: 配置空间的异步复位，只在 X8 的 soft IP 下使用。

Npor: 上电复位。

Srst: 同步数据路径复位，高有效。在 X1 和 X4 模式下有效。

Crst: 同步数据复位，高有效，用来复位 nonstickly 配置空间寄存器。

剩下一些复位输出状态信号不一一介绍了。

Figure5-23 和 Figure5-24 给出了各种复位信号之间的时序关系。

4）ECC 错误信号

Derr_cor_ext_rcv[1:0]: 显示接收 buffer 有一个可改正的错误

Derr_rpl: 显示重试 buffer 有一个不可更正的错误

Derr_cor_ext_rpl:显示重试 buffer 有一个可更正的错误

5）PCI express 中断（for endpoints）

当模块被配置成 endpoint 时，支持 legacy 中断、MSI 中断和 MSI-X 中断（该中断只在硬核有效）。

Legacy 中断：app_int_sts 输入信号来控制中断的产生，参考 figure5-30 和 figure 5-31，可以看到 app_inst_sts 和中断间的时序关系。

MSI 中断：app_msi_req 输入信号用来控制此中断的产生，当使能此信号后，它将会导致一个 MSI posted 写 TLP（基于 MSI 配置寄存器的值）。

各种中断模式具有互斥性。上电后，Megacore 自动在 INTx 模式，通过 msi_enable 可以调整到 MSI 中断模式或者 MSI-X 模式，

App_msi_req: 应用层 MSI 中断。

App_msi_tc[2:0]: 应用层 MSI traffic 类型。

App_msi_num[4:0]:应用层 MSI 偏远数量。

App_int_sts: 控制 legacy 中断。

6）PCI express 中断（for root ports）

在 root port 模式，Megacore 通过下面两种机制接收中断：

1〉 MSI—— root ports 通过 Avalon-ST 来接收 MSI 中断。这是一种 Memory 映射机制。

2〉 Legacy——该中断通过 int_status[3:0]引脚来进入应用层。

Aer_msi_num[4:0] : AER MSI 数目。该信号用来设置 MSI 和基地址之间的偏差。该信号只在 root port 模式有效。

Pex_msi_num[4:0]: 功耗管理 MSI 数目。该信号被 Power management 和热插拔用来控制消息中断数和基本消息中断数之间的偏差。

7）配置空间信号- Hard IP

配置状态寄存器通过一些控制信号来控制，它们和 core_clk 同步。Table5-15 列出了配置空间接口和热插拔的信号。

Tl_cfg_add: 那些被更新的寄存器地址。每 8 个 core_clk 更新一次。

Tl_cfg_ctl: 更新到寄存器的数据。每 8 个 core_clk 更新一次。

Tl_cfg_ctl_wr: 写信号，在 tl_cfg_ctl 被更新后，此信号会翻转。

Tl_cfg_sts: 配置状态位。也是每 8 个 core_clk 更新一次。Table5-15 详细列出了该信号种具体的对应关系。

Tl_cfg_sts_wr: 写信号，在 tl_cfg_sts 被更新后，此信号会翻转。没 8 个 core_clk 更新一次。

8) LMI 信号——Hard IP

LMI 信号可以用来读写 PCI Express 配置空间。这些接口和 pld_clk 同步。这些信号主要包括 lmi_dout、lmi_rden、lmi_wren、lmi_ack、lmi_addr、lmi_din，由于比较简单且 table5-19 给出了详细的介绍，这里就不介绍了。

9）power management 信号

这些信号在配置成 Avalon_ST 和 Descroptor/Data interface 时都有。

附录（学习 PCIE 时遇到的一些问题及 AE 的解答）

1 对于 PCI Express SOFT IP，Native EndPoint 中断是如何实现的？
A: MSI 或者 INTx.

2 传统的 PCI 中断 INTx 在 Native EndPoint 中是否支持，如何支持？
A: 支持。通过 INTx message 的方式来模拟传统的 INTx 方式。具体可参考 PCIe Spec (transaction layer spec).

3 对于 MSI 类型的中断，在用户手册上指出，上电后只有传统的 INTx 中断，而后软件通过向 MSI 配置寄存器中的 MSI 使能位写有效，
才能切换到 MSI 模式，传统的 INTx 中断和 MSI 中断相互排斥，这是怎么理解的？
A: 这是 IP 一个默认设定，MSI 使能在复位后是 disable 的，用户可以选择使用 INTx 中断或者是将 msi_enable 置高使用 MSI 中断。

4 常用的设计中采用传统的 INTx 类型中断还是 MSI 模式中断？
A: 推荐使用 MSI 中断。

5 写 MSI 中断时与配置空间寄存器值和 app_msi_tc 以及 app_msi_num 输入端口有何关系？
A: app_msi_tc 和 app_msi_num 在 MSI 中断产生时作为数据附在 Mwr TLP 包中。

6 在 Avalon-ST 形式的接口中为什么还掺杂了 Descriptor/Data 接口的内容，有哪些接口是 Avalon-ST 的，哪些接口是 Descriptor/Data 的？
A: Avalon-ST 和 Descriptor/Data 是两种不同的接口。推荐使用 Avalon-ST。两种接口不存在掺杂的说法。当然部分信号是相同或相近的。

7 在程序分层结构中有些模块使用了 ICM 方式，而有的没有，为什么采用这种方式，哪些模块使用 ICM？
A: ICM 只有在 Descriptor/Data 接口的 IP 时才会存在与 demo 中。Avalon-ST 接口的 demo 没有。

8 ALTERA 提供的 PCI Express Performance Demo 软件上的 DMA Write 和 DMA Read 数据流的方向是怎样的？是否是：DMA Write 时，
开发板作为 Requester 向主机内存 DMA 写数据。DMA Read 时，开发板作为 Requester 从主机内存中 DMA 读取数据。
A: 你的理解完全正确 J

9 以 Avalon-ST 接口形式生成的 PCI Express IP，其中 chaining DMA example 工程里 DMA 传输模块采用了增量式编译模块 ICM，
而 Dataheet 上讲 只有以 descriptor/data 方式产生的 IP 才会使用这一模块，这怎么理解？另外需要用户对 descriptor/data 接口的信号全面了解吗？
A: 是的，只有 descriptor/data 接口的 IP 才会使用这一模块。在 Avalon-ST 接口的 demo 中，这一模块实际上没起作用。

在做 demo 时兼顾了两种接口，所以在 Avalon-ST 接口的 demo 也有 ICM 的痕迹。
使用 Avalon-ST 接口时完全不需要了解 descriptor/data 接口的信号。

10 PCI Express Performance Demo 中 DMA write 和 DMA read 所指的数据流的方向是怎样的？
A：DMA write 是指由板卡的 DMA 控制器向内存写入数据操作，此时板卡的 TX 数据有效，
DMA read 是指板卡的 DMA 控制器从内存中读出数据操作，此时板卡的 RX 数据有效

9 关于 TLP 中字段中长度的理解，在协议中 TLP Header 中 Length 字段在不同类型的 TLP 中的含义是否相同？
比如 Memory Read Request(MRd)，该字段什么含义？而对于 Completion with Data(CplD)，该字段什么含义？
对于 Memory Write Request(MWr)，该字段什么含义？
如果均为对应的 TLP 中 PayLoad 的长度，那么 MRd 类型的 TLP 中 PayLoad 包含什么内容？
A：对于 Mrd 的 TLP，length 是指要读的数据长度，Mwr 的 TLP 指的是 TLP 中的数据长度。单位都是 DW (4 bytes)。

10 模块 altpcierd_cpld_rx_buffer 具体要实现的功能是什么？
模块将发送某一个 TAG 的 MRd TLP 的长度写入到 TAG RAM 中，而后对应某一各 TAG 的 Cpld 又从 TAG RAM 中读出是长度值是做何用途？
A：这个过程是为了比较 tag 值是否一样，从而确定是否为该 Mrd 返回的 Cpld。386 行是一个判断条件，在读请求的所有数据返回前进行该操作。

11 chaining DMA 的配置空间，对于 extended capabilities 的 MSI capability，其地址和数据值为什么是 0，MSI 中断是怎么实现的？
在 MegaWizard 时 MSI 设置为 4，这个在配置空间中怎么体现，另外，通过查看系统资源发现给开发板仅分配了一个中断号为 4 的中断资源，
而传统中断加 4 个 MSI 中断应该有 5 个中断，这个怎么理解？
A：MSI 是通过地址来发中断消息的，在不同的地址上写中断信息来使软件触发中断。
由于 win XP 操作系统并不支持 MSI 中断，所以，其实在客户的应用中，MSI 的中断并没有真正的打开，所以 MSI 的地址和数值都是 0，
并且只有一个中断资源。如果客户需要用到 MSI 的中断支持，可能需要自己在其他的操作系统中来编制软件，配置 MSI 中断地址，从而打开这个功能。

12 配置空间的管理，对于用于管理配置空间的 TLP 是否是在 IP 核中实现，而这些 TLP 是提供给应用的，用户无法在应用层监测到配置 TLP？
A: 配置空间的管理在 PCIe IP 内部实现，不需要用户逻辑参与。

13 关于 altpcierd_cdma_app_icm 模块中以下几个信号的含义，请解释？
RX 信号：

```
output          rx_mask0
output          rx_ws0
input           rx_dv0
input           rx_dfr0
input [15:0] rx_ecrc_bad_cnt
```

TX 信号：

```
output                          tx_dv0
output                          tx_dfr0
input                           tx_ws0
input                           tx_mask0
input                           cpld_rx_buffer_ready
input [TXCRED_WIDTH-1:0] tx_cred0
input [15:0]                    rx_buffer_cpl_max_dw
```

A: 其中 rx/tx 信号就是 descriptor/data 接口的信号，请参考 PCIe 手册上关于 descriptor/data 接口信号的描述。另外一些是内部信号：
input [15:0] rx_ecrc_bad_cnt: CRC 校验时错误计数器
input cpld_rx_buffer_ready : 内部 buffer 的 ready 信号
input [15:0] rx_buffer_cpl_max_dw: Mrd 时 RX Buffer 允许的最大 payload szie，以 dw 为单位.

14 在 CplD TLP 中有一个 Byte Count Field，对于以单个 CplD 来应答 MRd 时，PCIE 协议上讲就是 MRd 的 Length*4，而对于由多个 Completion 应答 MRd 时，该字段为还遗留的 Byte 数目，但是我看了 Chaining DMA 实例中 DMA Read 时，该字段不是很好理解，该字段究竟是怎样计算的？计算公式是什么？
A: 当一个 Mrd 需要多个 Cpld 来响应时，该字段是指剩余的数据长度(包含该 Cpld 带的数据)。举个例子，Mrd 读 4K bytes，分 4 个 1K bytes 的 Cpld 来响应，那么第二个 Cpld 包的该字段为 3K (H’400)。

15 在 RC Slave 模块，对于 RC 的 MRd，该模块以一个 CplD 来返回（因为我看到 CplD 的 Byte Count 字段就是 length*4），但一个 CplD TLP 的 Payload 毕竟有限，对于 length 较大的 MRd，RC Slave 模块怎么处理？怎么分成多个 CplD？
A: 在我们的 chaing-DMA 设计中，Max read request size 被设置为和 max payload size 相同。所以不存在一个 Mrd 需要多个 Cpld 来响应的情况。
一般情况下，Max read request size 远大于 max payload size，这种情况下会出现多个 Cpld 响应一个 Mrd 的情况。一般的以 max payload size 大小将 Completion 分割为多个小包。

16 在 Chaining DMA 实例中定义了一个地址宽度为 12，数据宽度为 128 的本地 Avalon 存储器，该存储器作何用途？DMA Read，DMA Write，RC Slave 的 MRd、MWr 是否都对该存储空间操作，存储空间如何分配的，TX 和 RX 分别如何仲裁？

17 上次讲到 Chaining DMA 实例 DMA 控制器在本地实现，那么该 DMA 主要独占哪部分的总线？利用 signaltap 观察，我理解应该是 RC 到 EP 之间的 PCIE 总线，而 RC 到内存的总线没有独占，而是通过北桥仲裁实现 RC 与内存的通信，这样的理解正确吗？如果正确，能否开启北桥到内存的 DMA，使得 EP 到内存均为独占式，提高 DMA 效率？

18 通过在 SignalTap 中查看 altpcierd_cdma_app_icm 模块的 app_msi_req 和 app_int_sts，发现在 DMA 读写过程中这两个信号均没有被拉高，
也就是说 DMA 过程没有使用中断的方式，那么 DMA 过程是如何实现的？DMA 整个过

程都不使用 DMA 吗？

19 在 DMA 控制寄存器中有一个“EPLAST_ENA”寄存器，我理解该寄存器的作用：

如果 EPLAST_ENA 使能有效，则每一个描述符对应的 DMA 执行结束后，

就向主机的描述符表的“EPLAST”表项写入被执行描述符表的索引号，

当一个描述符表中的所有描述符被 DMA 执行完毕，则主机的描述符表的“EPLAST”表项填入的就是最后一个被 DMA 的表项，

如果“EPLAST”与 DMA 控制寄存器 DW3 对应的“RCLAST－Idx of last descriptor to process”相同，则应用程序和 Endpoint 应用层都

会知道该描述符表被执行完毕。也就是说描述符表 DMA 结束是可以采用查询的方式进行的。

20 在进行 DMA 操作时，是否需要把 DMA 起始地址对齐到 QWORD？

A: 没有必要 QWORD (8 bytes)对齐，允许从任意地址开始。但要注意 PCIe RX/TX port rx_st_data/tx_st_data 数据总线中包头和数据的分布关系不同，请参考 PCIe 9.1 手册 145/146 页 Figure 5－14 到 5--16.

21 在 DMA 操作时，在 Window 下申请的地址空间在逻辑上连续而在物理上不连续，如何利用进行 Scatter/Gather 的 DMA 操作？

A: 从 chaining-DMA 设计的角度来看，它使用的是逻辑地址，所以不用关心这个问题。逻辑地址到物理地址的映射由 driver 完成。Windriver 应该有相应的函数。

22 在 Jungo Windriver 中 Chaining DMA 如果采用中断方式，如何在核模式下编写中断函数？

A: Windriver 文档中有提到如何编写用户中断乘虚。请参考所附文档以下章节：

11.6.5 Handling Interrupts in the Kernel PlugIn

23 在 PCI Express IP 核中有两个中断应答信号：app_msi_ac（MSI 中断应答）和 app_int_ack（传统中断应答）。在应用层产生中断请求后 IP 根据什么条件来产生有效的中断应答信号？这与主机的中断处理有关系吗？

A：当应用层发出中断请求信号后(app_int_sts 或 app_msi_req)，PCIe IP 将产生一个 Memory write 的包来传递中断，当 PCIe IP 判断可以传送中断 Memory write 包后，就用 app_int_ack 或 app_msi_ack 来响应中断。