**接口**

(*斜体是input*)

系统接口

|  |  |
| --- | --- |
| *sys\_rst\_n* |  |
| *sys\_clk* |  |

PCIE接口：多通道，差分信号线

发送

|  |  |
| --- | --- |
| pci\_exp\_txp(+) |  |
| pci\_exp\_txn(-) |  |

接收

|  |  |
| --- | --- |
| *pci\_exp\_rxp(+)* |  |
| *pci\_exp\_rxn(-)* |  |

**事务接口：公共接口、发送接口、接受接口**

公共接口

|  |  |
| --- | --- |
| user\_clk\_out | 1.事务等相关的处理都在这个信号的上升沿进行  2.sys\_rst\_n复位对该信号没有影响  3.上电后激活，当user\_reset\_out断言后稳定  4.其频率可以在工具中配置 |
| user\_reset\_out | 和事务相关的用户逻辑需要用该信号进行reset |
| user\_link\_up | 当核心和连接的上行链路伙伴端口准备好并能够交换数据包时被拉高 |
| fc\_ph[7:0] | 以下均为流控制协议相关内容  Posted Header Flow Control Credits |
| fc\_pd[11:0] | Posted Data Flow Control Credits |
| fc\_nph[7:0] | Non-Posted |
| fc\_npd[11:0] | Non-Posted |
| fc\_cplh[7:0] | Completion Header Flow Control Credits |
| fc\_cpld[11:0] | Completion Data Flow Control Credits |
| *fc\_sel[2:0]* | 流控制信息  • 000: Receive buffer available space  • 001: Receive credits granted to the link partner  • 010: Receive credits consumed  • 100: Transmit user credits available  • 101: Transmit credit limit  • 110: Transmit credits consumed |

发送接口

|  |  |
| --- | --- |
| *s\_axis\_tx\_tlast* | 1.传输帧结束  2.在s\_axis\_tx\_tvalid高时有效 |
| *s\_axis\_tx\_tdata[W-1:0]* | 要发送的数据 |
| *s\_axis\_tx\_tkeep[7:0]*  *(64-bit interface)*  *s\_axis\_tx\_tkeep[15:0]*  *(128-bit interface)* | 按字节选通tdata  当s\_axis\_tx\_tlast没有断言时，惟一的有效值是0xFF(64位)或0xFFFF(128位)。  断言s\_axis\_tx\_tlast时，  64位:只有0x0F和0xFF有效  128位:0x000F、0x00FF、0x0FFF和0xFFFF有效 |
| *s\_axis\_tx\_tvalid* | 发送源就绪:  表示用户应用程序在s\_axis\_tx\_tdata上呈现有效数据。 |
| s\_axis\_tx\_tready | 核心就绪接收数据 |
| *s\_axis\_tx\_tuser[3]* | 1.发送源中断数据发送  2.与s\_axis\_tx\_tlast同时断言 |
| tx\_buf\_av[5:0] | 传输缓冲区可用:表示在内核中可用的空闲传输缓冲区的数量。  每个自由传输缓冲区可以容纳一个TLP，最大负载大小(MPS)支持。传输缓冲区的最大数量由所支持的MPS和块RAM配置决定。 |
| tx\_err\_drop | 传输错误:表示内核因为长度冲突而丢弃了一个包，或者，当流传输时，数据没有以连续的时钟周期呈现。 |
| *s\_axis\_tx\_tuser[2]* | 传输流:表示一个包以连续的时钟周期呈现，在整个包写到核心之前，链路上的传输就可以开始了。通常称为传输直通模式。 |
| tx\_cfg\_req | 传输配置请求:在核心准备传输配置完成或其他内部生成的TLP时断言。 |
| *tx\_cfg\_gnt* |  |
| *s\_axis\_tx\_tuser[1]* | 转发错误  此输入将当前正在进行的包标记为错误中毒。  它可以在SOF和EOF之间的任何时间断言。如果断言(tx\_str)s\_axis\_tx\_tuser[2]，则不能断言该（tx\_err\_fwd）信号。 |
| *s\_axis\_tx\_tuser[0]* | 发送ECRC生成（tx\_ecrc\_gen）  附加端到端循环冗余检查(ECRC)摘要。这个输入必须在TLP开始时断言。 |

接收接口

|  |  |
| --- | --- |
| m\_axis\_rx\_tlast | 1.传输帧结束  2.在m\_axis\_rx\_tvalid高时有效 |
| m\_axis\_rx\_tdata[W-1:0] | 要接收的数据 |
| m\_axis\_rx\_tkeep[7:0]  (64-bit interface only) | 按字节选通tdata  当m\_axis\_rx\_tlast没有断言时，惟一的有效值是0xFF(64位)或0xFFFF(128位)。  断言m\_axis\_rx\_tlast时，  64位:只有0x0F和0xFF有效 |
| m\_axis\_rx\_tuser[14:10]  (128-bitinterface only) | 表明一个新的数据包开始（m\_axis\_rx\_tdata）  助记符rx\_is\_sof[4:0]  Bit 4：出现新的数据包时被断言  Bit 0-3：指定新的数据包开始的字节  • 5'b10000 = SOF at AXI byte 0 (DWORD 0)  m\_axis\_rx\_tdata[7:0]  • 5'b11000 = SOF at AXI byte 8 (DWORD 2)  m\_axis\_rx\_tdata[71:64]  • 5'b00000 = No SOF present |
| m\_axis\_rx\_tuser[21:17]  (128-bitinterface only) | 表明一个数据包的结束  助记符rx\_is\_eof[4:0]  有效值:  • 5'b10011 = EOF at AXI byte 3 (DWORD 0)  m\_axis\_rx\_tdata[31:24]  • 5'b10111 = EOF at AXI byte 7 (DWORD 1)  m\_axis\_rx\_tdata[63:56]  • 5'b11011 = EOF at AXI byte 11 (DWORD 2)  m\_axis\_rx\_tdata[95:88]  • 5'b11111 = EOF at AXI byte 15 (DWORD 3)  m\_axis\_rx\_tdata[127:120]  • 5'b01111 = No EOF present |
| m\_axis\_rx\_tuser[1] | 接受错误rx\_err\_fwd |
| m\_axis\_rx\_tuser[0] | 表示当前数据包有ECRC错误。在数据包EOF上断言。rx\_ecrc\_err |
| m\_axis\_rx\_tvalid | 接收源就绪:  表示核心在m\_axis\_rx\_tdata上呈现有效数据。 |
| *m\_axis\_rx\_tready* | 用户就绪接收数据 |
| *rx\_np\_ok* | 用户程序准备好接收Non-Posted TLPs时被断言 |
| *rx\_np\_req* | 接收Non-Posted TLPs的请求 |
| m\_axis\_rx\_tuser[9:2] | 指示当前事务的目标BAR(s)  • (rx\_bar\_hit[0])m\_axis\_rx\_tuser[2]: BAR0  • (rx\_bar\_hit[1])m\_axis\_rx\_tuser[3]: BAR1  • (rx\_bar\_hit[2])m\_axis\_rx\_tuser[4]: BAR2  • (rx\_bar\_hit[3])m\_axis\_rx\_tuser[5]: BAR3  • (rx\_bar\_hit[4])m\_axis\_rx\_tuser[6]: BAR4  • (rx\_bar\_hit[5])m\_axis\_rx\_tuser[7]: BAR5  • (rx\_bar\_hit[6])m\_axis\_rx\_tuser[8]: Expansion ROM  Address  If two BARs are configured into a single 64-bit address,  both corresponding rx\_bar\_hit bits are asserted.  •m\_axis\_rx\_tuser[8:4] are not applicable to Root  Port configurations.  • m\_axis\_rx\_tuser[9] is reserved for future use. |
| m\_axis\_rx\_tuser[16:15] | 保留 |

物理层接口

|  |  |
| --- | --- |
| pl\_initial\_link\_width[2:0] | • 000: Link not trained  • 001: 1-Lane link  • 010: 2-Lane link  • 011: 4-Lane link  • 100: 8-Lane link |
| pl\_phy\_lnk\_up | 物理层连接状态 |
| pl\_lane\_reversal\_mode[1:0] | Lane Reversal Mode: Indicates the current Lane Reversal mode.  • 00: No reversal  • 01: Lanes 1:0 reversed  • 10: Lanes 3:0 reversed  • 11: Lanes 7:0 reversed |
| pl\_link\_gen2\_cap | 表明PCIE是否是5.0 Gb/s |
| pl\_link\_partner\_gen2\_supported |  |
| pl\_link\_upcfg\_cap |  |
| pl\_sel\_lnk\_rate | • 0: 2.5 Gb/s  • 1: 5.0 Gb/s |
| pl\_sel\_lnk\_width[1:0] | • 00: 1-Lane link  • 01: 2-Lane link  • 10: 4-Lane link  • 11: 8-Lane link |
| pl\_ltssm\_state[5:0] | LTSSM State: Shows the current LTSSM state (hex).  • 0, 1: Detect Quiet  • 2, 3: Detect Active  • 4: Polling Active  • 5: Polling Configuration  • 6: Polling Compliance, Pre\_Send\_EIOS  • 7: Polling Compliance, Pre\_Timeout  • 8: Polling Compliance, Send\_Pattern  • 9: Polling Compliance, Post\_Send\_EIOS  • A: Polling Compliance, Post\_Timeout  • B: Configuration Linkwidth, State 0  • C: Configuration Linkwidth, State 1  • D: Configuration Linkwidth, Accept 0  • E: Configuration Linkwidth, Accept 1  • F: Configuration Lanenum Wait  • 10: Configuration Lanenum, Accept  • 11: Configuration Complete x1  • 12: Configuration Complete x2  • 13: Configuration Complete x4  • 14: Configuration Complete x8  • 15: Configuration Idle  • 16: L0  • 17: L1 Entry0  • 18: L1 Entry1  • 19: L1 Entry2 (also used for the L2/L3 Ready pseudo state)  • 1A: L1 Idle  • 1B: L1 Exit  • 1C: Recovery Rcvrlock  • 1D: Recovery Rcvrcfg  • 1E: Recovery Speed\_0  • 1F: Recovery Speed\_1  • 20: Recovery Idle  • 21: Hot Reset  • 22: Disabled Entry 0  • 23: Disabled Entry 1  • 24: Disabled Entry 2  • 25: Disabled Idle  • 26: Root Port, Configuration, Linkwidth State 0  • 27: Root Port, Configuration, Linkwidth State 1  • 28: Root Port, Configuration, Linkwidth State 2  • 29: Root Port, Configuration, Link Width Accept 0  • 2A: Root Port, Configuration, Link Width Accept 1  • 2B: Root Port, Configuration, Lanenum\_Wait  • 2C: Root Port, Configuration, Lanenum\_Accept  • 2D: Timeout To Detect  • 2E: Loopback Entry0  • 2F: Loopback Entry1  • 30: Loopback Active0  • 31: Loopback Exit0  • 32: Loopback Exit1  • 33: Loopback Master Entry0 |
| pl\_rx\_pm\_state[1:0] | RX Power Management State:  Indicates the RX Power  Management State:  • 00: RX Not in L0s  • 01: RX L0s Entry  • 10: RX L0s Idle  • 11: RX L0s FTS |
| pl\_tx\_pm\_state[2:0] | TX Power Management State: Indicates the TX Power Management  State:  • 000: TX not in L0s  • 001: TX L0s Entry0  • 010: TX L0s Entry1  • 011: TX L0s Entry2  • 100: TX L0s Idle  • 101: TX L0s FTS0  • 110: TX L0s FTS1  • 111: TX L0s FTS2 |
| *pl\_directed\_link\_auton* | 链路变化相关 |
| *pl\_directed\_link\_change[1:0]* |  |
| *pl\_directed\_link\_speed* |  |
| *pl\_directed\_link\_width[1:0]* | 传输流:表示一个包以连续的时钟周期呈现，在整个包写到核心之前，链路上的传输就可以开始了。通常称为传输直通模式。 |
| pl\_directed\_change\_done | 传输配置请求:在核心准备传输配置完成或其他内部生成的TLP时断言。 |
| *pl\_upstream\_prefer\_deemph* |  |

配置接口（只记录端点设备）

|  |  |
| --- | --- |
| cfg\_mgmt\_do[31:0] | 从配置空间读取到的数据 |
| cfg\_mgmt\_rd\_wr\_done | 配置读写完成 |
| *cfg\_mgmt\_di[31:0]* | 配置数据输入 |
| *cfg\_mgmt\_dwaddr[9:0]* | 配置DWORD地址 |
| *cfg\_mgmt\_byte\_en[3:0]* | 写字节使能 |
| *cfg\_mgmt\_wr\_en* |  |
| *cfg\_mgmt\_rd\_en* |  |
| *cfg\_mgmt\_wr\_readonly* | 允许写只读位 |
| cfg\_status[15:0] | 配置状态 |
| cfg\_command[15:0] | 配置命令 |
| cfg\_dcommand[15:0] | 设备控制寄存器 |
| cfg\_dcommand2[15:0] |  |
| cfg\_lstatus[15:0] | 链路状态 |
| cfg\_lcommand[15:0] |  |
| cfg\_aer\_ecrc\_gen\_en |  |
| cfg\_aer\_ecrc\_check\_en |  |
| cfg\_pcie\_link\_state[2:0] | PCI Express Link State: This encoded bus reports the PCI Express Link  State information.  • 000: L0  • 001: PPM L1  • 010: PPM L2/L3 Ready  • 011: PM\_PME  • 100: in or transitioning to/from ASPM L0s  • 101: transitioning to/from PPM L1  • 110: transition to PPM L2/L3 Ready  • 111: Reserved |
| *cfg\_trn\_pending* | 用户事务挂起:如果断言，设置设备状态寄存器中的事务挂起位。  注意:如果用户应用程序没有收到上游请求的完成，则必须断言此输入。 |
| *cfg\_dsn[63:0]* | 设备序列号 |
| cfg\_pmcsr\_pme\_en | //此处省略一些电源管理相关寄存器 |
| cfg\_received\_func\_lvl\_rst | 7s设备不支持该功能 |
| cfg\_vc\_tcvc\_map[6:0] | VC通道 |
| cfg\_msg\_received |  |
| cfg\_msg\_data[15:0] | Message Requester ID |

中断配置接口

|  |  |
| --- | --- |
| *cfg\_interrupt* | 中断请求，在cfg\_interrupt\_rdy高时 |
| cfg\_interrupt\_rdy |  |
| *cfg\_interrupt\_assert* | Configuration Legacy Interrupt Assert/Deassert Select: Selects  between Assert and Deassert messages for Legacy interrupts  when cfg\_interrupt is asserted. Not used for MSI interrupts.  Value Message Type  1 Assert  0 Deassert |
| *cfg\_interrupt\_di[7:0]* | 配置中断数据:对于MSIs，如果启用了多向量中断，端点必须驱动消息数据的一部分来指示MSI向量号。  cfg\_interrupt\_mmenable[2:0]表示的值确定端点提供的低阶消息数据的位数;没有使用cfg\_interrupt\_di[7:0]的其余上位。对于单矢量中断，不使用cfg\_interrupt\_di[7:0]。对于遗留中断消息(Assert\_INTx, Deassert\_INTx)，只支持INTA (00h)。 |
| cfg\_interrupt\_do[7:0] | 配置中断数据输出:端点MSI功能结构中消息数据字段的最低8位的值。此值用于提供信息和向后兼容性。 |
| cfg\_interrupt\_mmenable[2:0] | 配置中断多消息启用:这是多消息启用字段的值，并定义系统允许的多向量MSI的向量数量。数值范围从000b到101b。000b表示启用了单向量MSI，而其他值表示可以被cfg\_interrupt\_di覆盖的低阶位数[7:0]  • 000: 0 bits  • 001: 1 bit  • 010: 2 bits  • 011: 3 bits  • 100: 4 bits  • 101: 5 bits |
| cfg\_interrupt\_msienable | • 0: Only Legacy (INTX) interrupts or MSI-X Interrupts can be  sent.  • 1: Only MSI Interrupts should be sent. |
| cfg\_interrupt\_msixenable | • 0: Only Legacy (INTX) interrupts or MSI Interrupts can be sent.  • 1: Only MSI-X Interrupts should be sent. |
| cfg\_interrupt\_msixfm |  |
| *cfg\_pciecap\_interrupt\_msgnum[4:0]* |  |
| *cfg\_interrupt\_stat* |  |

接口中最关键的其实是事务相关的接口，前期的话打算围绕事务接口展开，进行仿真和实际板子上的测试。

PIO demo

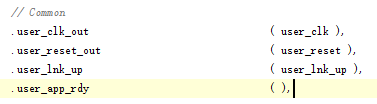
|-- pcie\_support

|-- pipe\_clock:这个不用关注 主要和 Clocking Sharing Interface有关

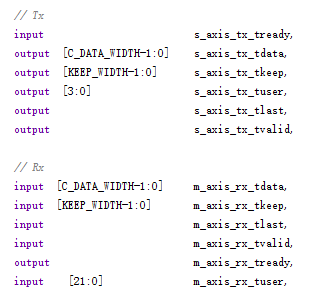
|-- pcie:具体接口可以看pg054，前期只关注有用的接口

|-- app:具体的处理逻辑

app时钟、复位、核心及链路是否准备好了、app\_rdy没有使用



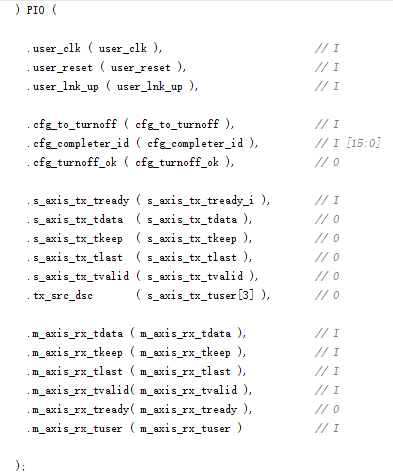
收发接口



剩下的配置接口先不关注

最核心的是

PIO



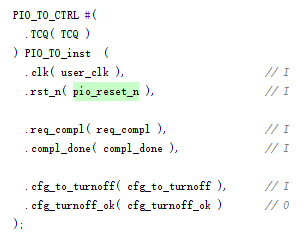
cfg\_turnoff\_ok:通知关闭电源是安全的



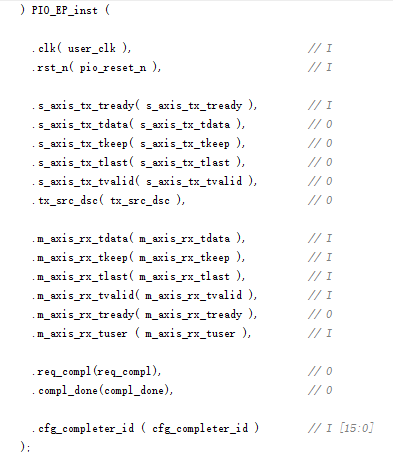
PIO包含控制和收发引擎

主要是控制cfg

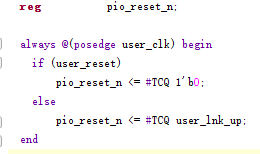
控制



收发



注意复位信号变了，user\_reset 🡪 pio\_resert\_n



即处理了user\_reset又处理了user\_link\_up

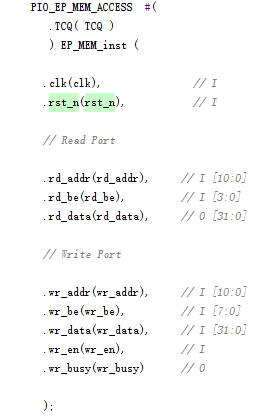
两个模块之间的联系（这两个信号都是从收发引擎到控制模块）



先看简单的控制模块，做的就是

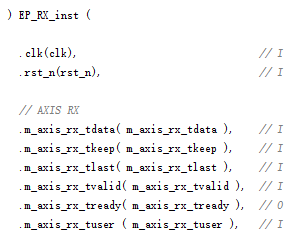
Turn-off OK if requested and no transaction is pending

PIO\_RX PIO\_TX PIO\_MEM\_ACCESS

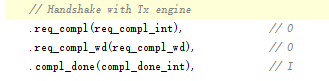


Memory读写

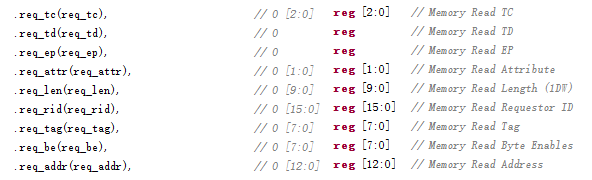
RX引擎



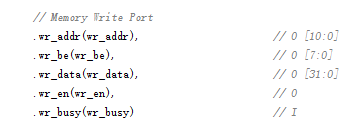
时钟，复位，以及从核心读取数据



和TX的握手信号

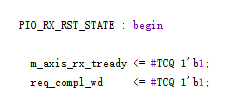


TLP包相关

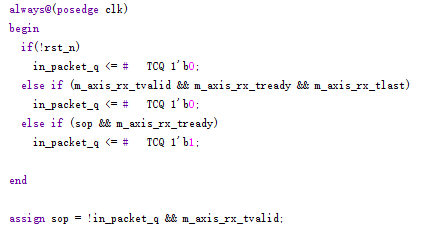


内存写端口

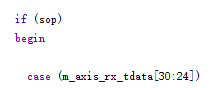
以64位分析



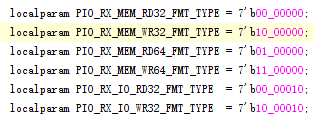
这个状态是reset之后的状态，等待sop信号



sop信号是数据开始传输的标志



tdata[30:24]是TLP包的Fmt\_Type字段



存储器读写 带数据（4DW）和无数据(3DW)

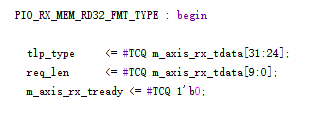
对于存储器读的时候 4DW其实是64位寻址

IO读 无数据3DW

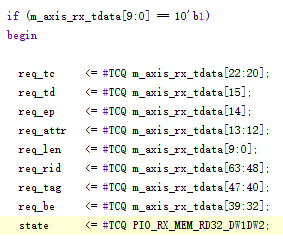
IO写 带数据 3DW

PIO中只处理这几种消息

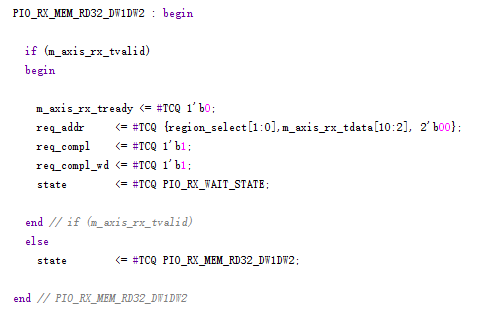
存储器读 无数据3DW



第一个64位会填充TLP基础字段

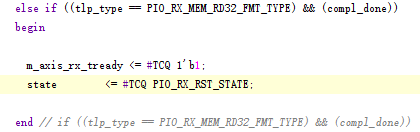


然后转移到下一个状态，去处理地址，这里地址只有13位



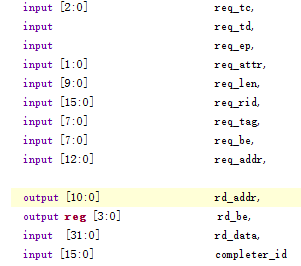
region\_select是选择区域（应该是demo自定义的），并且断言req\_compl

读取到地址就要等待 从memory读，转移到等待状态。



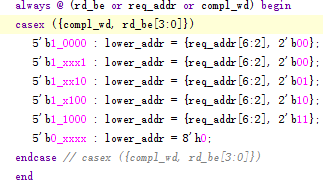
等待compl\_done,这个信号是TX引擎控制的

对于TX引擎

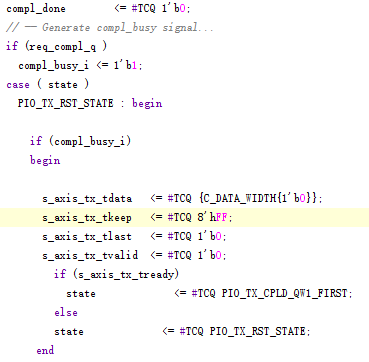


很多信号都是从RX引擎收到的

在TX前面，主要做了字节选择

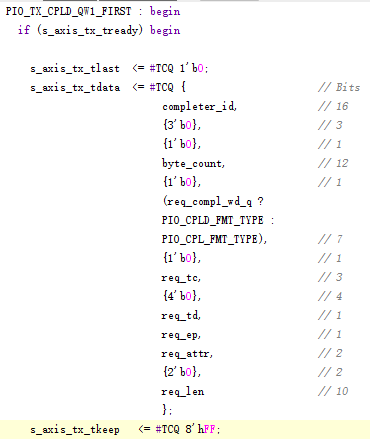


进入TX的reset状态

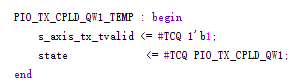


等待核心准备好接收数据的情况下

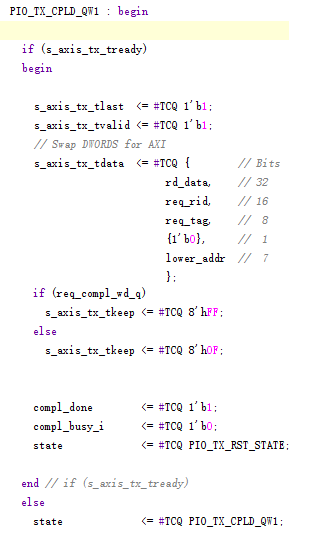
进入FIRST状态



拼接第一个64位数据！req\_compl\_wd就是看是否带数据，然后转移到



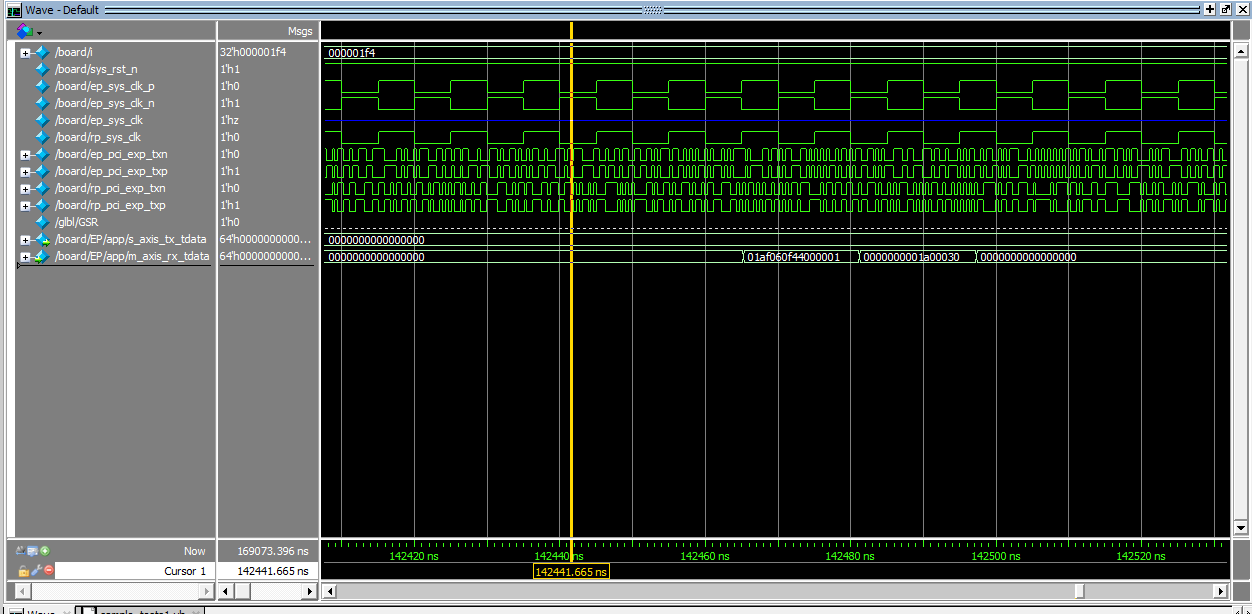
告诉核心总线上的数据是有效的，然后进入



接着就是第二个数据，通知RX引擎,发送结束，进入RST状态

关注rd\_data, 这个数据在MEM\_ACCESS中产生的，这个数据的产生我也没有搞懂

仿真分析





01af060f 44000001

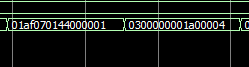
00000000 01a00030

01000100000000000000000000000001‬

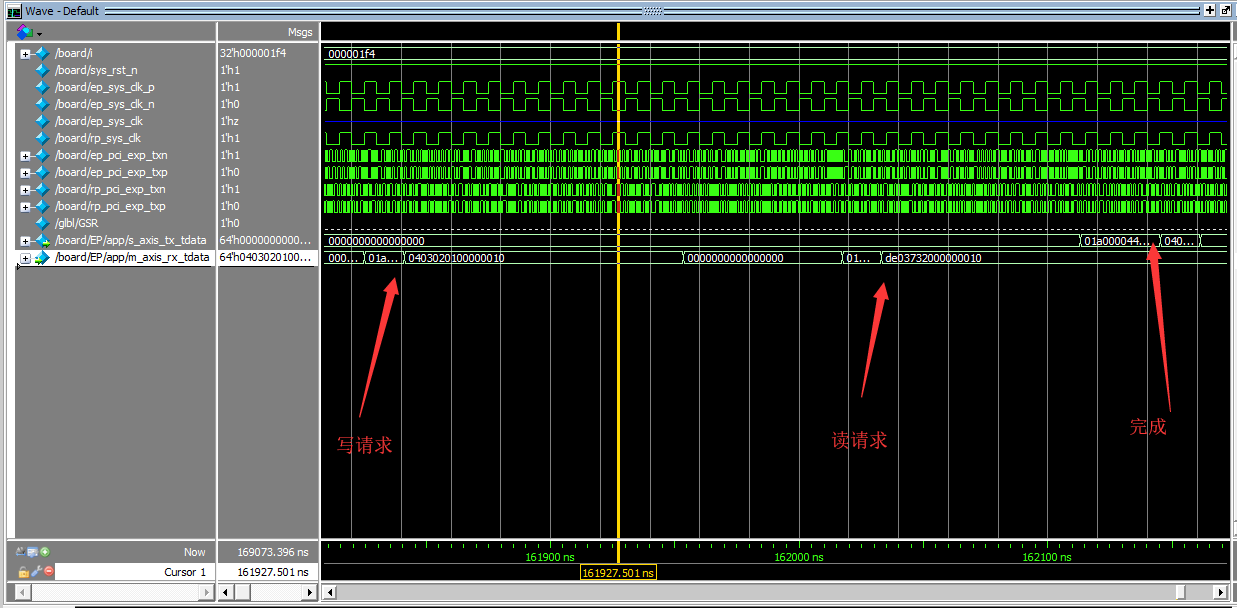
配置写请求

Request ID

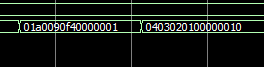
Bus num



这两个包都是配置写请求 pio中对于这两个是不响应的



写请求



01a0090f40000001

01000000000000000000000000000001‬

存储器写请求

00000001101000000000100100001111

00000000000000000000000000010000‬

Address

00000100000000110000001000000001

Data

读请求



01a00a0f00000001

de03732000000010

00000000000000000000000000000001‬

存储器读3dw

00000001101000000000101000001111

00000000000000000000000000010000‬

读地址对应之前的写地址

完成包



01a000044a000001

0403020101a00a10

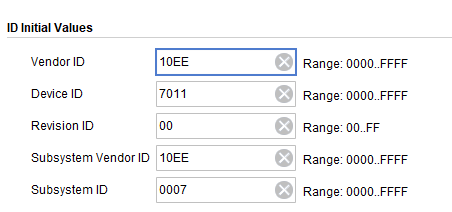
01001010000000000000000000000001‬

带数据完成

对应请求id

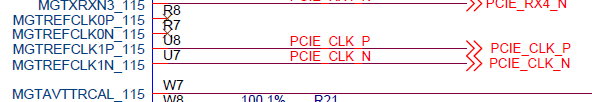
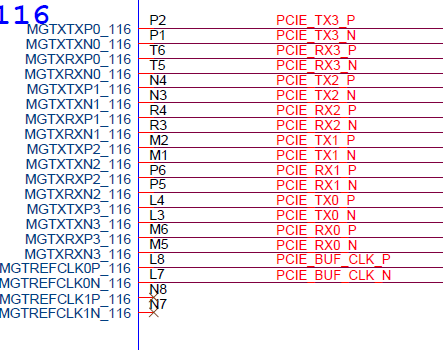
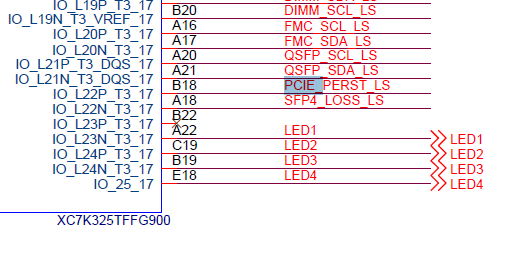
数据对应之前的写数据

把PIO程序烧写到fpga，插入pcie



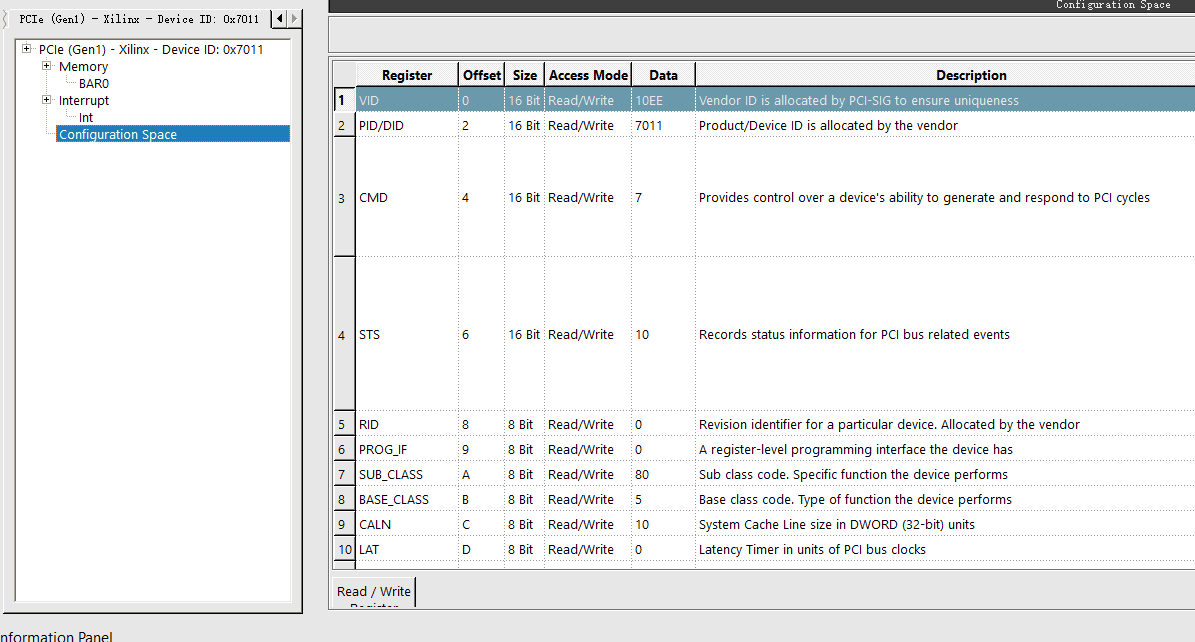
上电之后识别不到板卡

问题出在管脚约束上面，查原理图rst、clk、收发，配置成对应的

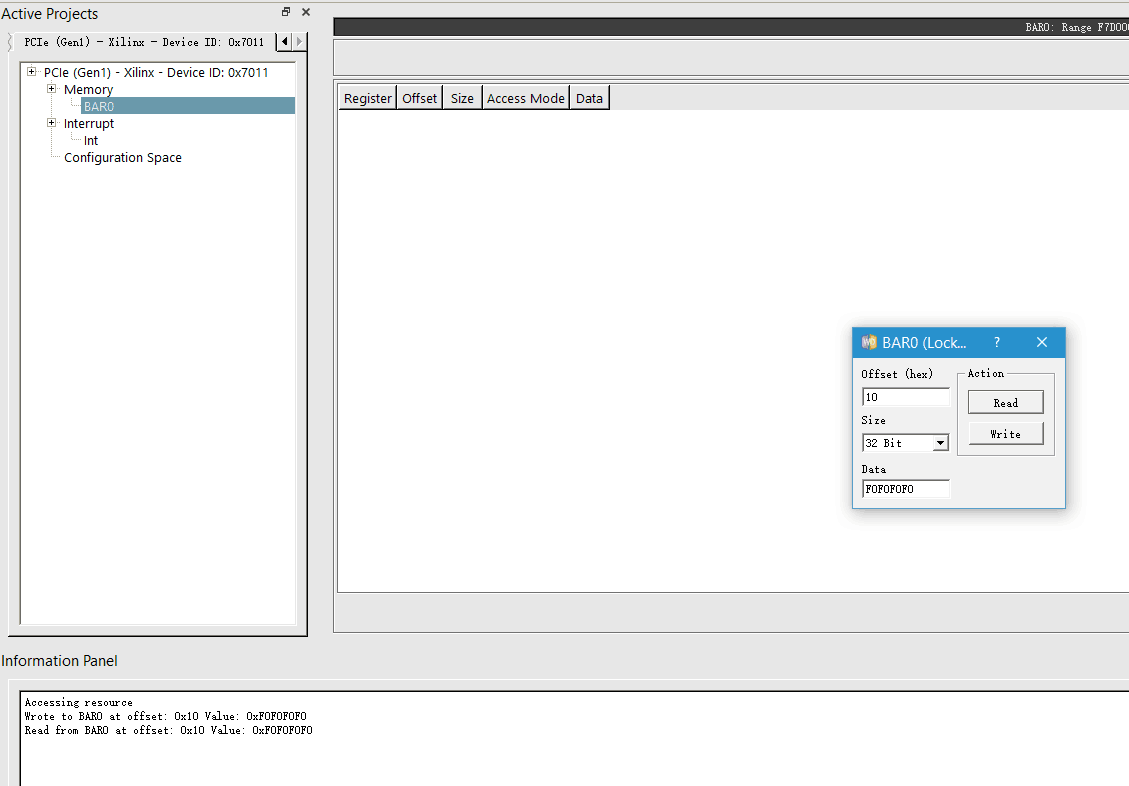


并将物理层的link\_up引到led上观察

使用windriver



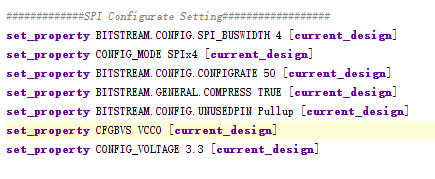
发现了板卡，并能够进行读写



向偏移量10的地方写0xF0F0F0F0，然后正确读出

在windows下的机器上是完成了上述步骤，在ubuntu下link\_up没有被拉高

后面发现了一个问题，上电非常慢，加上约束之后



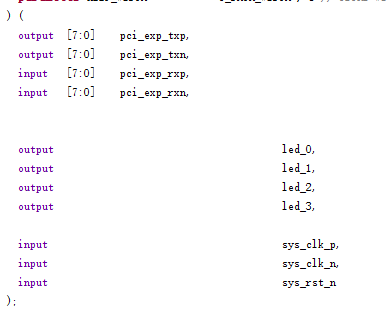
加速从Ubuntu也可以读取到板子

后面的话有两个需要学习的地方

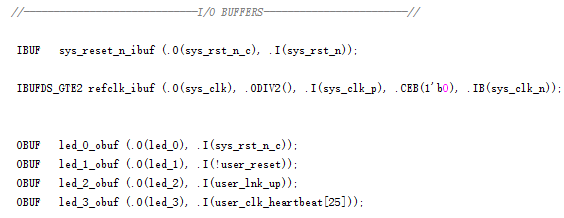
1. DMA
2. 驱动

个人感觉先学习驱动方面会比较好，这样能够加深PIO的理解，DMA的配置过程中也是会用到驱动去读写PCIe板卡，但由于我时间有限，只能先学习DMA了

DMA顶层接口



和PIO是没有什么区别的



然后是xilinx的I/O Buffers

大型设计一般推荐使用同步时序电路。同步时序电路基于时钟触发沿设计，对时钟的周期、占空比、延时和抖动提出了更高的要求。为了满足同步时序设计的要求，一般在FPGA设计中采用全局时钟资源驱动设计的主时钟，以达到最低的时钟抖动和延迟。 FPGA 全局时钟资源一般使用全铜层工艺实现，并设计了专用时钟缓冲与驱动结构，从而使全局时钟到达芯片内部的所有可配置单元(CLB)、I/O单元(IOB)和选择性块RAM(Block Select RAM)的时延和抖动都为最小。为了适应复杂设计的需要，Xilinx的FPGA中集成的专用时钟资源与数字延迟锁相环(DLL)的数目不断增加，最新的 Virtex II器件最多可以提供16个全局时钟输入端口和8个数字时钟管理模块(DCM)。与全局时钟资源相关的原语常用的与全局时钟资源相关的Xilinx器件原语包括：IBUFG、IBUFGDS、BUFG、BUFGP、BUFGCE、BUFGMUX、BUFGDLL和DCM等

1. IBUFG即输入全局缓冲，是与专用全局时钟输入管脚相连接的首级全局缓冲。所有从全局时钟管脚输入的信号必须经过IBUF元，否则在布局布线时会报错。 IBUFG支持AGP、CTT、GTL、GTLP、HSTL、LVCMOS、LVDCI、LVDS、LVPECL、LVTTL、PCI、PCIX和 SSTL等多种格式的IO标准。

2. IBUFGDS是IBUFG的差分形式，当信号从一对差分全局时钟管脚输入时，必须使用IBUFGDS作为全局时钟输入缓冲。IBUFG支持BLVDS、LDT、LVDSEXT、LVDS、LVPECL和ULVDS等多种格式的IO标准。

3. BUFG是全局缓冲，它的输入是IBUFG的输出，BUFG的输出到达FPGA内部的IOB、CLB、选择性块RAM的时钟延迟和抖动最小。

4. BUFGCE是带有时钟使能端的全局缓冲。它有一个输入I、一个使能端CE和一个输出端O。只有当BUFGCE的使能端CE有效(高电平)时，BUFGCE才有输出。

5. BUFGMUX是全局时钟选择缓冲，它有I0和I1两个输入，一个控制端S，一个输出端O。当S为低电平时输出时钟为I0，反之为I1。需要指出的是BUFGMUX的应用十分灵活，I0和I1两个输入时钟甚至可以为异步关系。

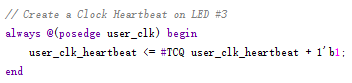
6. BUFGP相当于IBUG加上BUFG。

7. BUFGDLL是全局缓冲延迟锁相环，相当于BUFG与DLL的结合。BUFGDLL在早期设计中经常使用，用以完成全局时钟的同步和驱动等功能。随着数字时钟管理单元(DCM)的日益完善，目前BUFGDLL的应用已经逐渐被DCM所取代。

8. DCM即数字时钟管理单元，主要完成时钟的同步、移相、分频、倍频和去抖动等。DCM与全局时钟有着密不可分的联系，为了达到最小的延迟和抖动，几乎所有的DCM应用都要使用全局缓冲资源。DCM可以用Xilinx ISE软件中的Architecture

Wizard直接生成。



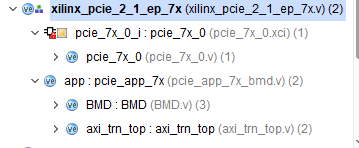


LED3是对核心的输出user\_clk的计数

对应的十进制

（可能是我算错了，后面实际情况的时候再来看）

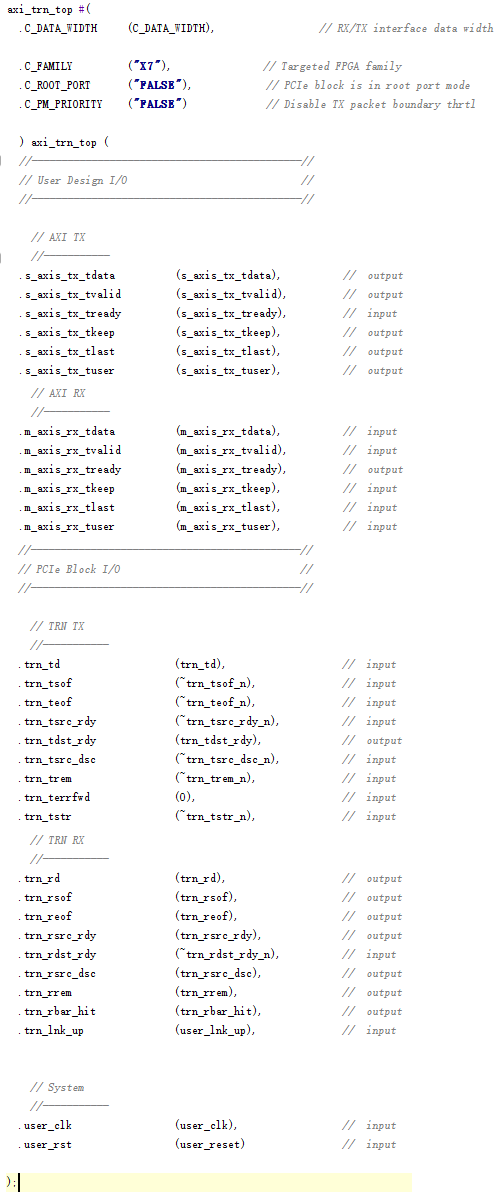
这是主要的结构，和PIE还是类似的，分为PCIe核心和APP



具体的分析还是围绕APP展开

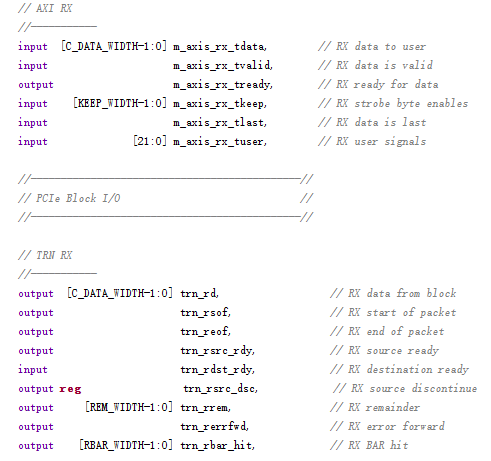
先看axi\_trn\_top

前面的AXI接口和核心相连，后面的的接口主要是和BMD模块连接



其中分为RX和TX

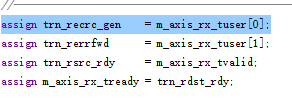
RX



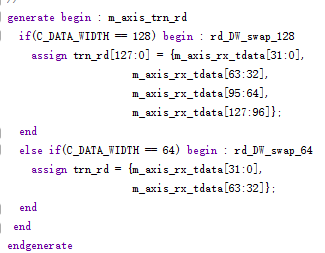
Bar的选中

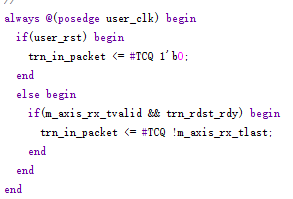


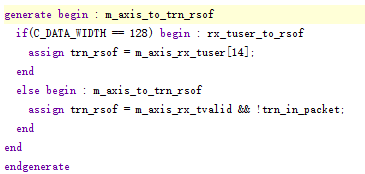
和几个关键的连线



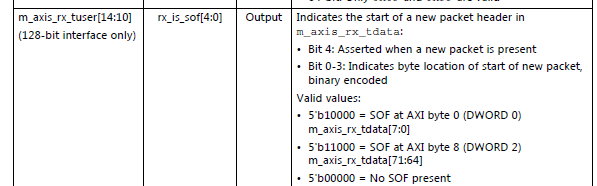
数据DW排序





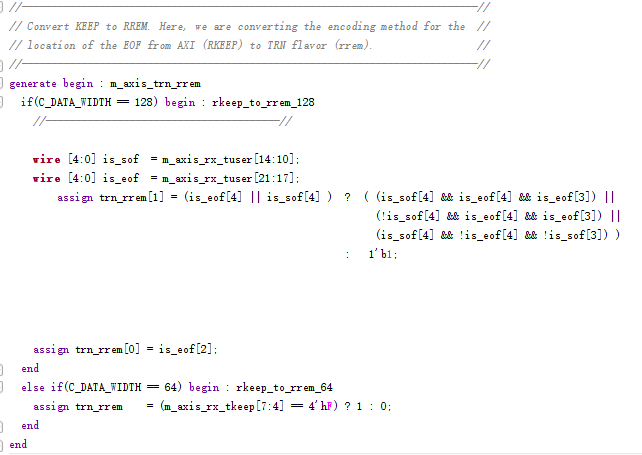


对于128位来说

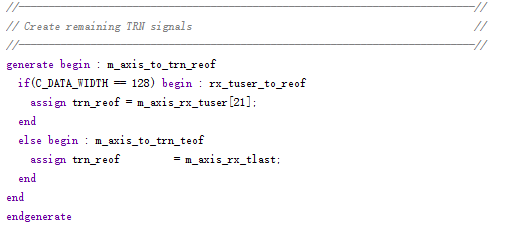


直接可以标示一个包的开始

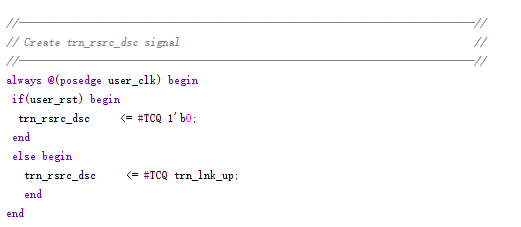
对于64位则需要靠axi的信号进行判断



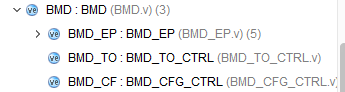
不太懂，后面再看



包结束

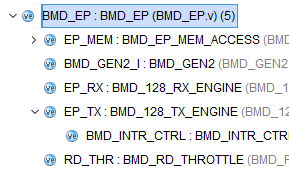


TX是从trn到AXI的转换，后面如果需要再详细分析

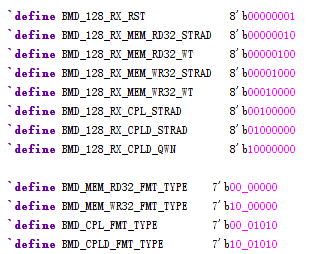


其中BMD\_TO\_CTRL和PIO中的类似

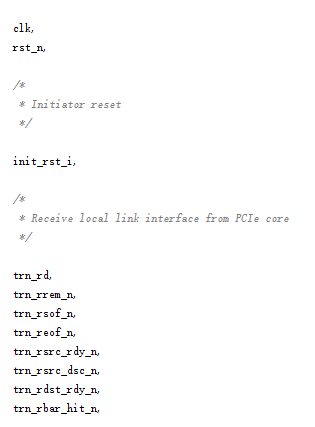
BMD\_CFG\_CTRL中是对配置的控制，用到的时候再分析

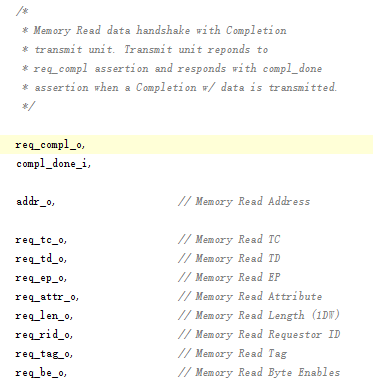


还是从RX开始分析

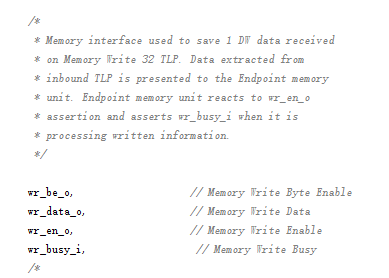


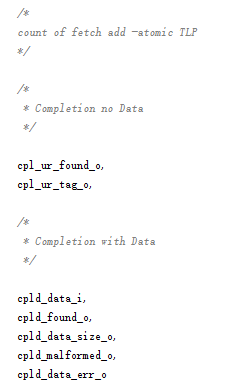
这里面只处理了32位地址的读和写 TLP都是3DW





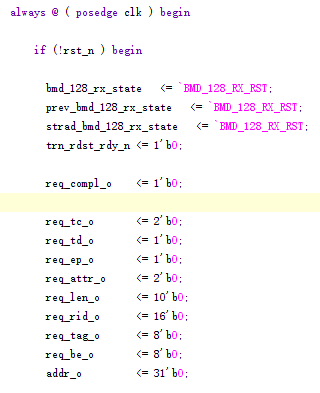
这部分和PIO也很像

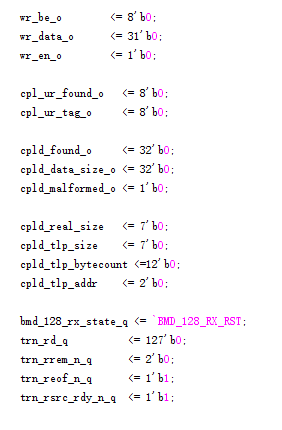




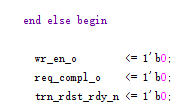
以上就是RX引擎的接口

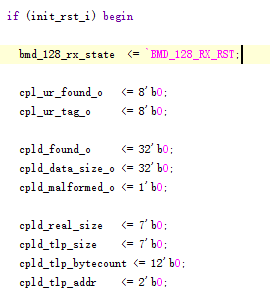
复位

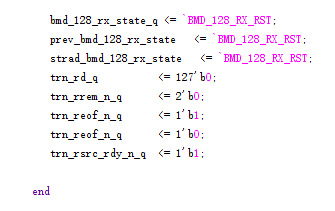




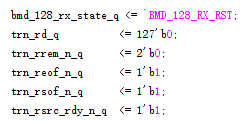
————————————————————————————————————————————————————————————————————————————————————————————————————————————————————





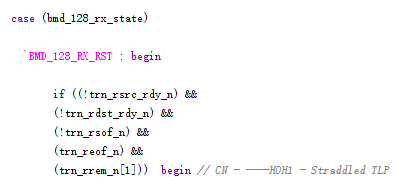


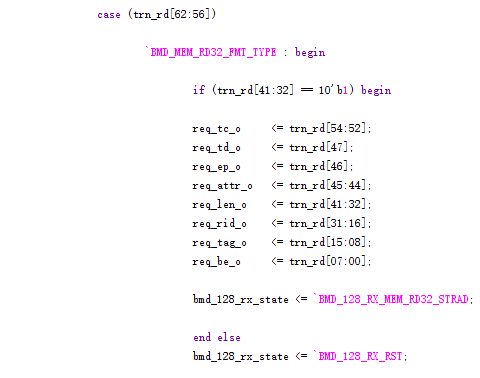
判断是否是初始化复位init\_rst是从MEM\_ACCESS来的

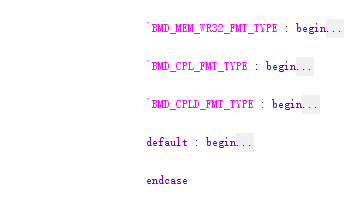


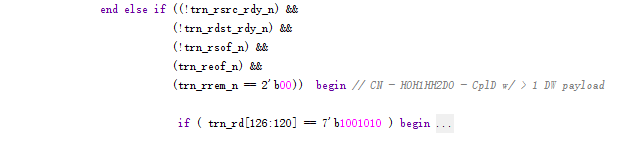
在每个时钟周期开始时都会执行的操作

后面是状态转换机

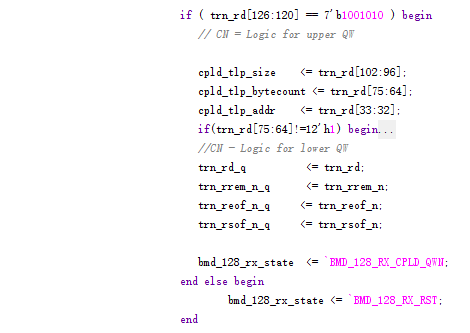






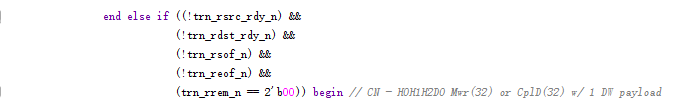


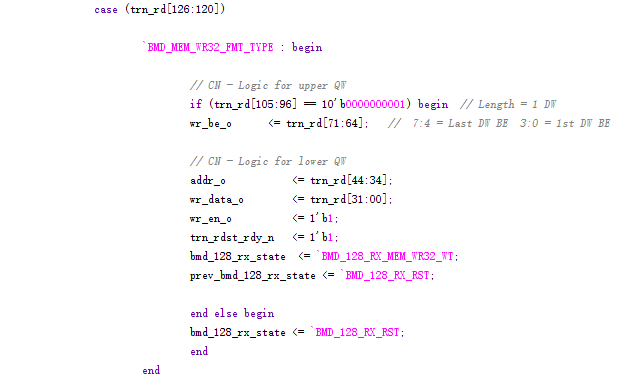
这个应该是讲CplD包

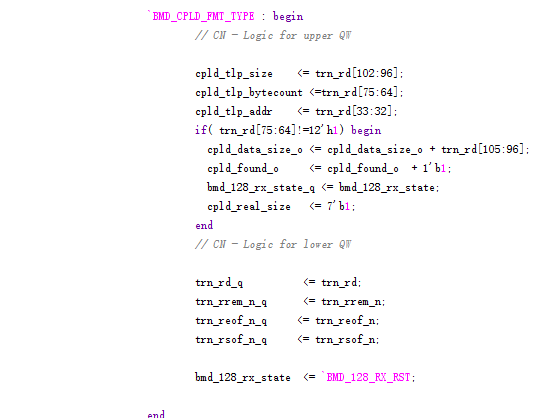


并且数据大于1DW的情况

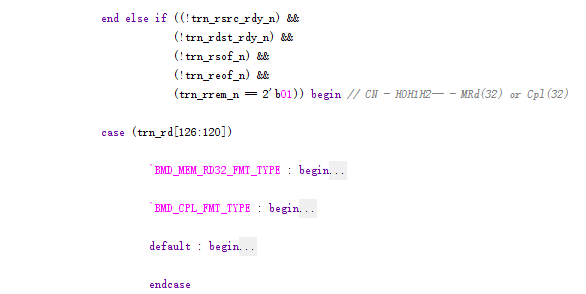
接下来是处理数据1DW的情况，包括了CplD和写请求

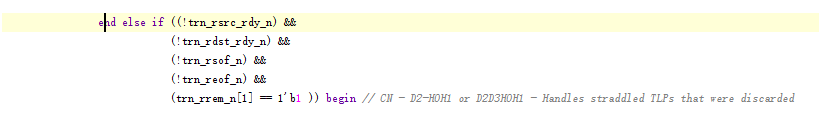






然后是读和Cpl的包





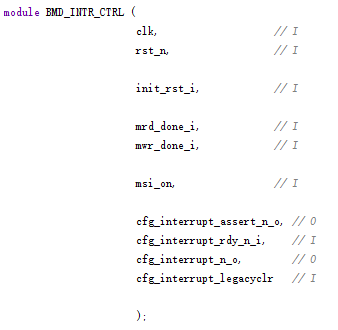
这一部分就是为了处理数据和TLP位置

后面是复杂的包处理

然后看发送引擎

实在是看不下去了。。。。。

关注对中断的控制



DMA大致流程

这里先简单介绍一下DMA读过程：

1、驱动程序向操作系统申请一片物理连续的内存;

2、主机向该地址写入数据；

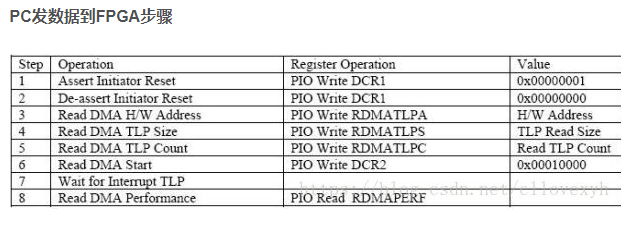
3、主机将这个内存的物理地址告诉FPGA；

4、FPGA向主机发起读TLP请求—连续发出多个读请求；

5、主机向FPGA返回CPLD包—连续返回多个CPLD；

6、FPGA取出CPLD包中的有效数据；

7、FPGA发送完数据后通过中断等形式通知主机DMA完成；



DMA写过程如下：

1、驱动程序向操作系统申请一片物理连续的内存；

2、主机将这个内存的物理地址告诉FPGA；

3、FPGA向主机发起写TLP请求，并将数据放入TLP包中—连续发出多个写请求；

4、FPGA发送完数据后通过中断等形式通知主机DMA完成；

5、主机从内存中获取数据；

