1 AXI4 协议介绍及波形分析

ZYNQ 作为一款 FPGA 与双核 ARM 结合的全可编程片上系统，独特的架构使其拥有着非常高的灵活性以及可拓展性，并十分适合用来做一些数据方面的处理，例如像视频监视、汽车先进驾驶辅助系统等等。这类应用场景通常都需要很高的数据实时传输与处理能力，但熟悉 ZYNQ 的朋友也都知道，ZYNQ 总体上是被明确分为了 PS 与 PL 两个部分，以 ZYNQ-7000 系列为例，如图 1-1：

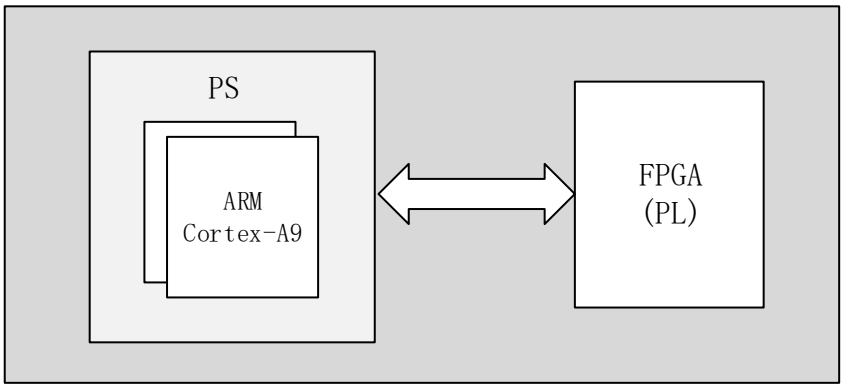


图 1-1 Zynq-7000 结构视图

其中 PS 部分就是以双核 ARM Cortex-A9 为核心的 SOC 处理系统，PL 就是我们常说的 FPGA。在上述的应用场景中，PL 侧通常都扮演着数据采集处理的角色，PS 则负责控制以及对数据做进一步的处理和输出。这类场景通常对数据的实时性有着很高的要求，为了让 PS 与 PL 间能够相互合作，充分发挥各自的特长，彼此间数据的实时交互也就成了重中之重。

ZYNQ 使用的是来自 ARM 公司的硬核 CPU，因此 PS 侧总线使用的也是 ARM 公司提出的 AMBA（Advanced Microcontroller Bus Architecture）协议，而 AXI（Advanced Extensible Interface，高级可扩展接口）则是 AMBA 协议中最重要的部分。AXI 作为一种面向高性能、高带宽、低延迟的片内总线协议，其目标是服务高性能和高时钟频率的系统设计。而这种特点也让其成为了解决 PS 与 PL 间数据交互的“不二人选”，因此 ZYNQ 的 PL 也使用的 AXI 协议进行数据传输。

AXI 最早的版本为 AXI3，协议中地址/控制和数据相位是分离的，支持不对齐的数据传输、Outstanding 传输访问和乱序访问。数据以突发（burst）的形式组织，只需要首地址，就能完成一次多数据的突发传输。

2010 年，ARM 公司发布了 AMBA 4.0 协议，AXI 协议也由 AXI3 升级为

AXI4。相较于 AXI3，AXI4 协议移除了一些不太实用的信号，比如移除了用于标志写指令 ID 的 WID 信号，因此 AXI4 不再支持乱序写；除此之外添加了一些新的信号，比如说用户信号和 Qos 信号（Quality of Service）；对一些功能也进行了修改，比较有代表的就是突发长度由原来的最高 16，变为了最高 256；除了这些之外，AXI4 还定义了一种新的协议——AXI4-Lite，这是一种简化版的 AXI4 协议，应用于一些总线性能要求较低的场景。

AXI3 和 AXI4 这两种总线协议在 ZYNQ-7000 系列器件中都有应用，比如，这系列器件的 PS 侧使用的就是 AXI3 协议来实现内部的互联通信，也就是说图 1-1 中蓝绿棕三种颜色的箭头代表的总线，使用的就是 AXI3 协议。再比如，搭建硬件逻辑系统时我们所用到的那些带 AXI 接口的 IP 核，通常使用的是 AXI4 接口，也有小部分使用 AXI3接口，以及少数能够进行协议转换的，同时带有 AXI4 和 AXI3 接口的 IP 核。

虽然 AXI3 和 AXI4 在 ZYNQ-7000 系列器件中都有应用，但是在我们平时的开发过程中，接触的更多的通常还是 AXI4 协议。并且这两个协议之间无论是通道功能、信号作用还是支持的模式都有很大一部分的相同，即使是在需要进行协议转换的场合，大多也能使用专门的 IP 自动完成转换。因此，从实用性角度来说，学习 AXI4 协议是要优于 AXI3 协议的，而 AXI4 协议又可以细分为 AXI4、AXI4-Lite、AXI4-Stream 三种总线协议。AXI4 接口协议可以说是 AXI4 协议的完整版，AXI4-Lite 是 AXI4 的简化版，AXI4-Stream 是一种特殊的只存在于 PL 的数据流范式协议，结构较为简单。因此，本节内容主要以带领大家学习 AXI4 接口协议为主，通过 AXI4 接口协议带大家了解 AXI4 协议的工作流程及原理，并结合具体的波形信号分析验证。

1.1 AXI4 协议

AXI4 协议共包含三种接口，也可以说是三种总线，分别为 AXI4（AXI-FULL）、AXI4-Lite、AXI4-Stream。

数据在这些总线中以 burst 的形式组织，称为 AXI Burst；传输 burst 所需的操作称为 AXI Transaction；每一个 burst 可以被拆分为多次进行传输，每个被拆分的待传输数据称为 Beat；每个 Beat 被传输的过程称为 AXI Transfer。因此 AXI Transaction = m*burst = m*n*beat = m*n*transfer（m，n >= 1）。

不同的总线，一次 burst 所支持的最大 beat 会有所不同： AXI4 中的限制是最多 256 个传输事务（AXI3 中为 16 个）；AXI4 Lite 每个 burst 则只允许 1 个传

输事务（Beat）；而 AXI4-Stream 由于比较特殊，支持无限制的传输。

三种总线在 ZYNQ 内部实现着 PS 与 PL 之间的交互，不同接口面向不同的应用场景，分别如下：

- AXI4：主要面向高性能地址映射（memory map）通信的需求，是面向地址映射的接口，在单地址传输的情况下最大允许 256 个时钟周期的数据突发长度。AXI4 总线允许符合 AXI4 的系统实现非常高的数据吞吐量，同时还支持数据大小调整、多个 outstanding 操作和乱序事务处理。在硬件级别，AXI4 允许每个 AXI 主从使用不同的时钟构建系统。 此外，AXI4 协议允许插入寄存器片以帮助时序收敛。
- AXI4-Lite：用于简单、低吞吐量的内存映射通信（例如，与控制寄存器和状态寄存器之间的通信）。是一个轻量级的地址映射单次传输接口，占用很少的逻辑单元。该接口是 AXI4 接口的简化版，突发长度从 256 被限制到 1，也就意味着无法进行突发传输，逻辑资源的减少，也就导致无法实现较为复杂功能。
- AXI4-Stream：主要面向高速流数据传输；与 AXI4 的区别是没有了地址接口，因此不涉及读写数据的概念，数据只是进行简单的接收与发送。这种方式减少了传输时的延时，允许无限制的数据突发传输规模。

无论是 AXI3、AXI4 还是 AXI4-Lite 总线都是以内存映射（Memory Map）协议为基础，所谓的内存映射是这些总线事务都涉及的，在系统内空间和数据中传输目标地址的概念。将内存中的一块区域与虚拟的缓冲区建立映射关系，当从缓冲区中取数据，就相当于读内存中的相应字节；而将数据存入缓冲区，就相当于写内存中的相应字节。通过该方式，便能够避免 I/O 操作，实现数据的快速读写。

而 AXI-Stream 总线使用的 AXI-Stream 协议，该协议用于以数据为中心和数据流范式的应用程序，其中不存在或不需要地址概念，就像串口发送数据的时候，只按一定的波特率往外发送一样，不关心接收者存放的细节，因而能够无限制的传输。

以上便是 AXI4 协议下三种总线协议的功能以及各自的应用场景，AXI4-Stream 总线由于特殊性，数据在总线中直接传输；而 AXI4 和 AXI4-Lite 总线为了实现上述的功能，内部将各种信号以通道为单位，共同组成了读写事务。

1.1.1 读写事务通道

AXI4 和 AXI4-Lite 接口由五个不同的通道组成，即：

1. 写地址通道（write address channel，AW）
2. 写数据通道（write data channel，W）
3. 写响应通道（write response channel，B）
4. 读地址通道（read address channel，AR）
5. 读数据通道（read data channel，R）

这五个通道一起组合成了读事务与写事务，用以在主设备与从设备间传输数据。读事务与写事务的结构如图 1-2 和图 1-3 所示：

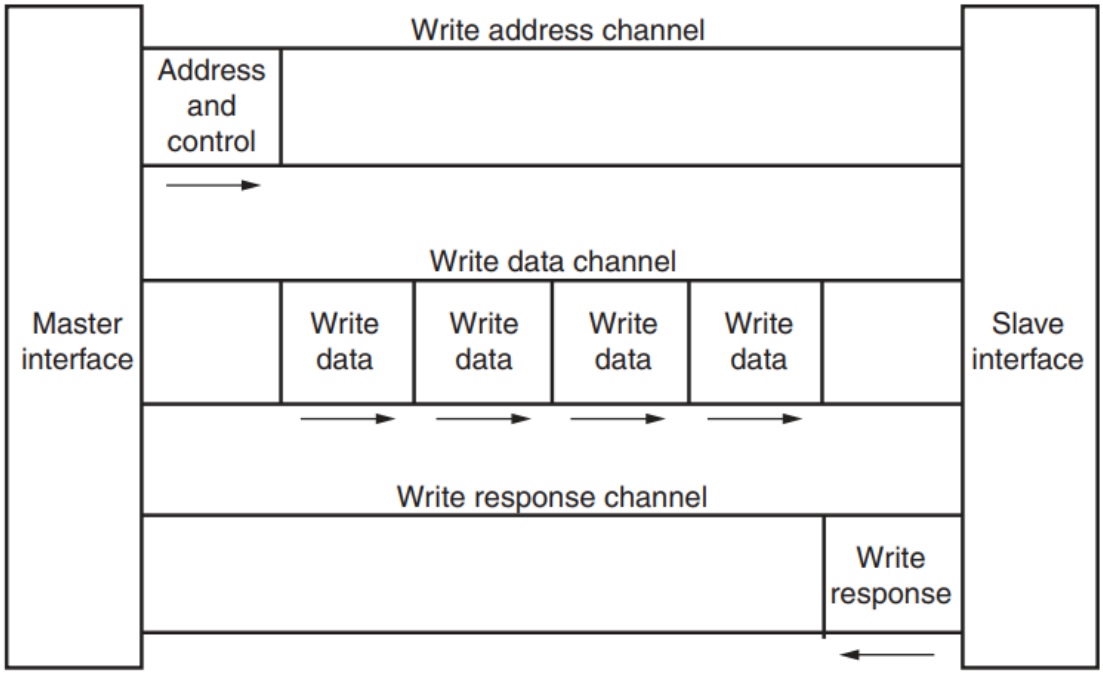


图 1-2 写事务结构

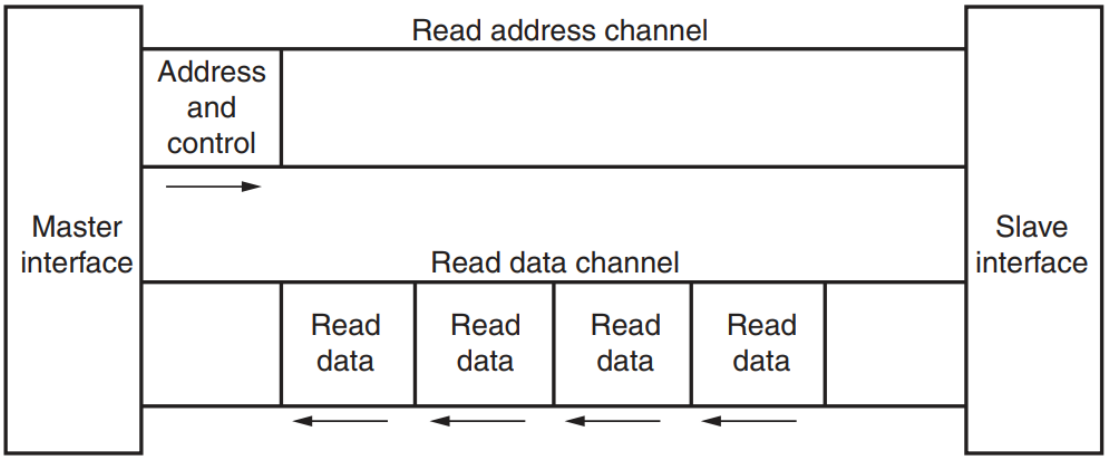


图 1-3 读事务结构

这两种事务包含以下特点：

①这 5 条独立的通道都包含一个双路的 VALID、READY 握手机制。信息源通过 VALID 信号来指示通道中的数据和控制信息什么时候有效。目地源用 READY 信号来表示何时准备好接收数据。传输地址信息和数据都是在 VALID 和 READY 同时为高时有效。

②读数据和写数据通道都包括一个 LAST 信号，用来指明一个事务传输的最后一个数据。

③读/写事务都有自己的地址通道，地址通道携带着传输事务所必须的地址和信息。

④读数据通道传送着从设备到主机的读数据和读响应信息。读响应信息指明读事务的完成状态。

⑤写数据通道传送着主机向设备的写数据和写控制信息（TLAST）。写响应通道提供了设备响应写事务的一种方式。在每一次突发式写会产生一个完成信号。

而每个 AXI4-Stream 都充当具有握手数据流的单个单向通道。因此，AXI4-Stream 接口没有以上五个通道，但是传输前需要先进行上述握手过程，在握手完成后，数据会被直接传输。

1.1.2 通道信号介绍

AXI4 接口的每个通道都包含了多个信号线，这些信号线的位宽大多都可以自定义，根据设计的需求，并不是所有的信号都需要被使用到，各个通道信号以及全局信号描述如下：

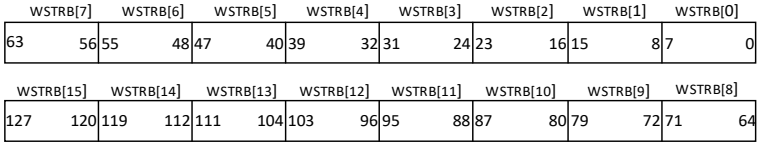
（1） 全局信号

信号	源	描述
ACLK	时钟源	公共时钟信号
ARESETn	复位源	公共复位信号

（2） 写地址通道信号

信号	源	描述
AWID	主机	写地址 ID，这个信号是写地址信号组的 ID tag。
AWADDR	主机	写地址，给出一次写突发传输的写地址
AWLEN	主机	突发式写的长度，突发式写所传输的数据的个数。数据个数 = AWLEN + 1
AWSIZE	主机	突发式写的大小，给出每个突发传输数据的字节数
AWBURST	主机	突发式写的类型。
AWLOCK	主机	锁类型。
AWCACHE	主机	Cache 类型。这信号指明事务的 bufferable、cacheable、write-through、write-back、allocate attributes 信息。
AWPROT	主机	保护类型，表明一次传输的特权级及安全等级。
AWQOS	主机	质量服务 QoS。
AWUSER	主机	用户自定义信号
AWVALID	主机	写地址有效。 1 = 地址和控制信息有效 0 = 地址和控制信息无效 这个信号会一直保持，直到 AWREADY 变为高。
AWREADY	从机	写地址准备好。这个信号用来指明设备已经准备好接受地址和控制信息了。 1 = 设备准备好 0 = 设备没准备好

（3） 写数据通道信号

信号	源	描述
WDATA	主机	写的数据。
WSTRB	主机	写数据有效的字节阀门，用来表明哪 8bits 数据是有效的。WSTRB[n]标示的区间为 WDATA[(8*n)+7:(8*n)] 
WLAST	主机	此次传输是最后一个数据传输。
WUSER	主机	用户自定义信号
WVALID	主机	写有效 1 = 写数据和阀门有效 0 = 写数据和阀门无效
WREADY	从机	写就绪。指明设备已经准备好接受数据了 1 = 设备就绪 0 = 设备未就绪

（4） 写响应通道信号

信号	源	描述

BID	从机	写响应 ID， 这个数值必须与 AWID 的数值匹配。
BRESP	从机	写响应。这个信号指明写事务的状态。 2'b00：OKAY，2'b01：EXOKAY，2'b10：SLVERR，2'b11：DECERR。
BUSER	从机	用户自定义信号
BVALID	从机	写响应有效。 1 = 写响应有效 0 = 写响应无效
BREADY	主机	接受响应就绪。该信号表示主机已经能够接受响应信息。 1 = 主机就绪 0 = 主机未就绪

（5） 读地址通道信号

信号	源	描述
ARID	主机	读地址 ID，用来标志一组读信号
ARADDR	主机	读地址，一次读突发传输的读地址
ARLEN	主机	突发式读长度，突发传输数据的个数，数据个数 = ARLEN + 1
ARSIZE	主机	突发式读大小，每个突发传输数据的字节数
ARBURST	主机	突发式读类型。
ARLOCK	主机	总线锁信号，可提供操作的原子性
ARCACHE	主机	Cache 类型。
ARPROT	主机	保护类型，一次传输的特权级及安全等级
ARQOS	主机	质量服务 QoS。
ARUSER	主机	用户自定义信号
ARVALID	主机	读地址有效。信号一直保持，直到 ARREADY 为高。 1 = 地址和控制信息有效 0 = 地址和控制信息无效
ARREADY	从机	读地址就绪。指明设备已经准备好接受数据了。 1 = 从机就绪 0 = 从机未就绪

（6） 读数据通道信号

信号	源	描述
RID	从机	读 ID tag。RID 的数值必须与 ARID 的数值匹配。
RDATA	从机	读数据。
RRESP	从机	读响应。这个信号指明读传输的状态：OKAY、EXOKAY、SLVERR、DECERR。
RLAST	从机	读事务传送的最后一个数据。
RUSER	从机	用户自定义信号
RVALID	从机	读数据有效。 1 = 读数据有效。 0 = 读数据无效。
RREADY	主机	读数据就绪。 1 = 主机就绪 0 = 主机未就绪

AXI4 中所有信号都在全局时钟 ACLK 的上升沿采样，所有输出信号都必须

在 ACLK 上升沿之后发生变化。从各个通道信号的名称可以看到，它们的命名是有规律的：读地址信号都是以 AR 开头（A：address；R：read）；写地址信号都是以 AW 开头（A：address；W：write）；读数据信号都是以 R 开头（R：read）；写数据信号都是以 W 开头（W：write）；应答信号都是以 B 开头（B：back（answer back））。

这些信号互相配合，进而实现各个通道的传输，下面以 AXI4 接口的典型时序为例，首先带大家简单了解 AXI4 如何实现读写事务的传输。

1.1.3 典型时序

下图中图 1-4 为 AXI4 突发写的典型时序，图 1-5 为 AXI4 突发读的典型时序，其中绿色信号为全局信号，红色信号为主机发送给从机的信号，蓝色信号为从机发送给主机的信号。

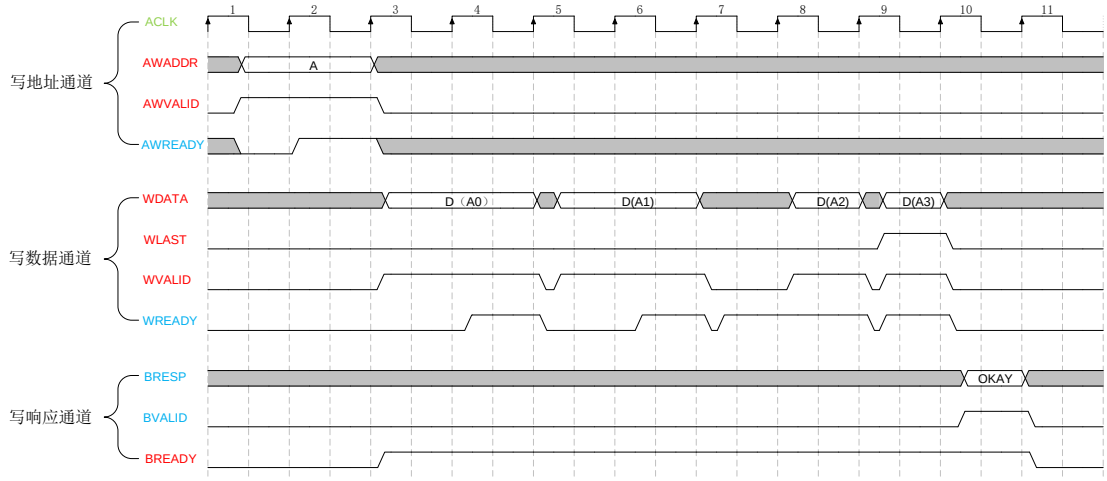


图 1-4 写事务典型时序图

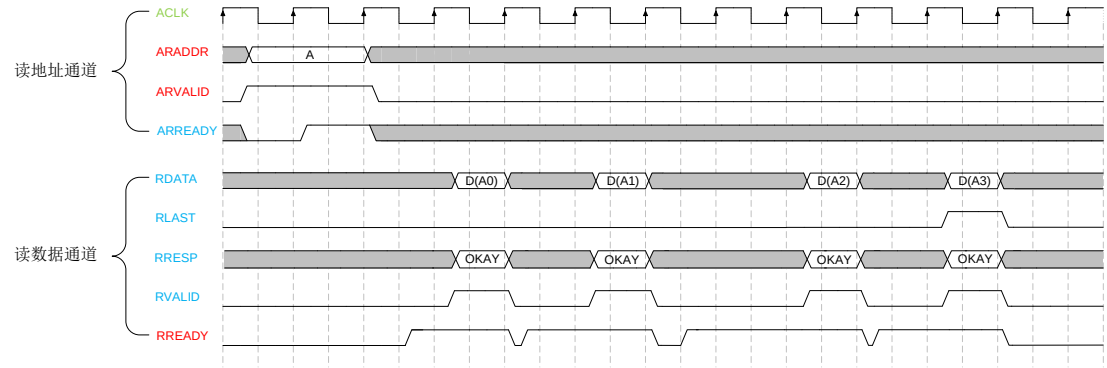


图 1-5 读事务典型时序图

在进行写事务时，主机首先会将待写入数据的地址以及控制信号放入写地址通道，随后进行握手，握手成功后，地址和控制信号被写入从机。这里的控

制信号包括突发长度、数据包大小、突发类型等等。接着主机通过写数据通道向从机传输数据，每传输一次数据就要进行一次握手，只有握手成功，数据才会被写入到从机中。当最后一组数据被放到写数据通道时，主机还会产生一个写控制信号 WLAST，用来告诉从机这是最后的数据。从机在接收了最后一 Beat 以及 WLAST 信号后，会通过写应答通道向主机发送写响应信号 BRESP，以告知主机，本次传输状态。

在进行读事务时，主机同样首先将待读取的地址以及控制信号放入读地址通道，随后进行握手，握手成功后，地址和控制信号被写入从机。接着从机将指定地址的读数据放入写数据通道，并产生读响应信号，当读数据通道握手成功时，数据与响应信号被发送给主机。当最后一 Beat 数据被放到读数据通道时，从机会拉高读控制信号 RLAST，以告知主机本次读传输结束。

我们可以看到，AXI4 的读事务只有两个通道，相对于写事务而言，没有了应答通道。这里的读地址通道同样被主机用来传输地址和控制信号，而读数据通道，除了用来传输待读取数据外，还用来传输读响应信号，指示读事务的完成状态，也因此，读事务不需要读响应通道。

至于写响应信号为什么要单独设立一个通道，这是因为 AXI 协议中存在握手机制， 每个通道在进行数据交互前，都需要先进行双向握手（VALID 和 READY 信号）。当两个信号都为高时，传输线上的数据才会有效，这种握手机制只允许一个方向的数据流，对于读事务，数据方向与响应方向一致，由从机流向主机；而对于写事务，数据方向与响应方向相反，所以就需要单独的响应通道。

1.1.4 握手机制

五个通道想要进行数据交互首先需要进行双向握手，握手时，传输源（发送方）会产生 VALID 信号来指明此时的数据或控制信号是否有效，目的源（接收方）会产生 READY 信号来告诉传输源，是否已经准备好接收数据或控制信号了。只有当这两个信号都为高时，才算握手成功，传输源会在握手成功时的时钟上升沿进行一次数据传输。

这种双向流控机制使得发送与接收双方都有能力控制传输速率，通过控制 VALID 和 READY 的高低电平来控制传输的时机以及速度。

既然是握手机制，自然也就跟我们平时握手时一样，会有个先后顺序，VALID 和 READY 在握手时共有三种关系：

1. VALID 信号先拉高， READY 信号后拉高

此时握手信号与数据以及时钟的关系如下：

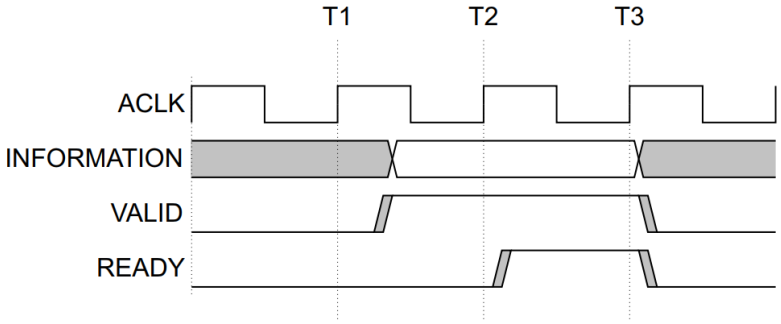


图 1-6 VALID 在 READY 前有效

这里的 ACLK 为 AXI4的全局时钟，INFORMATION 为待传输的内容，VALID/READY 为握手信号。从图中可以看到，VALID 信号在 T1 信号之后到来（拉高），与其一起来的还有数据、地址或者控制信号。而 READY 信号则是在 T2 之后被拉高，因为错过了上升沿，直到 T3 时刻才被检测到，此时握手成功，内容得以被传输。

AXI4 协议中规定，VALID 信号一旦拉高，在握手成功之前不能被拉低，因此，VALID 信号会一直等待，直到在上升沿时刻检测到 READY 为高后 VALID 才能被拉低。

同时，协议中还规定，发送方不能以 READY 信号作为 VALID 信号拉高的依据，而接收方可以但不是必须以 VALID 信号作为 READY 信号拉高的依据。因此，VALID 信号与 READY 信号都是完全独立自主的信号。这也正满足了前面所讲的“这种双向流控机制使得发送与接收双方都有能力控制传输速率”，即使发送方将 VALID 拉高，接收方也能通过 READY 信号控制传输时机与传输速率。

2. READY 信号先拉高，VALID 信号后拉高

此时握手信号与数据以及时钟的关系如下：

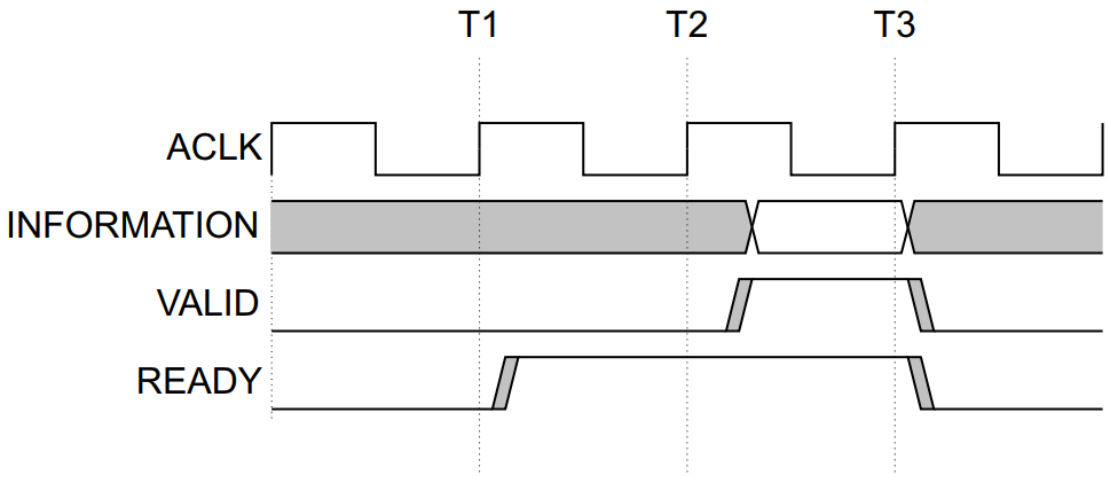


图 1-7 READY 信号在 VALID 信号前有效

可以看到，T1 时刻之后 READY 信号拉高，而 VALID 信号则是在 T2 时刻之后拉高，因为错过了时钟上升沿，所以在 T3 时刻才握手成功，此时，INFORMATION 中的信息被传输。

实际上，即使 READY 信号被拉高，只要 VALID 信号没有被拉高，接收方也可以拉低 READY 信号。例如，接收方置高 READY 后，发现自己还有传输需要完成，而此时发送方还没准备好数据（未置高 VALID），这时候接收方便能够拉低 READY 信号，转去处理其他传输，传输完成后再回来拉高 READY 等待接收数据。

3. VALID 信号和 READY 信号一起拉高

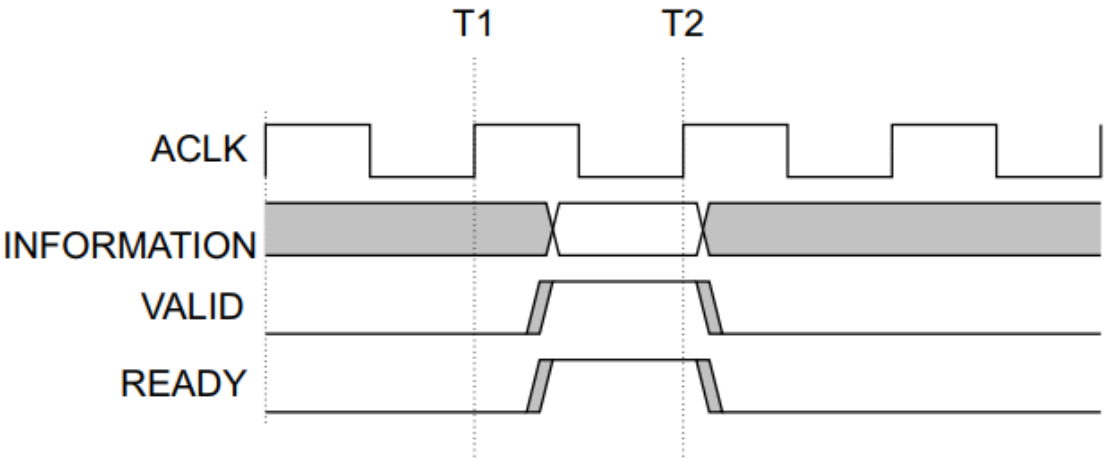


图 1-8 READY 和 VALID 信号同时有效

这种情况下就比较简单，READY 信号与 VALID 信号同时拉高，在下一个时钟上升沿也就是 T2 被检测到，此时握手成功，INFORMATION 得以传输。

五个通道都有自己的握手信号对，对应的名称如下：

表 1-1 事务通道握手信号对

传输通道	握手信号对	
	发送方	接收方
写地址通道	AWVALID	AWREADY
写数据通道	WVALID	WREADY
写应答通道	BVALID	BREADY
读地址通道	ARVALID	ARREADY
读数据通道	RVALID	RREADY

虽然前面我们说五个通道是独立工作的，但实际上信号间是有着一定的依赖性的。比如说，读数据时，必须是主机先将地址告诉从机了，从机才能够发送所需要的数据，毕竟只有知道了地址才能读取数据。同样的，握手对间也存在着一定的依赖性，其中写事务的握手对依赖关系如图 1-9 所示：

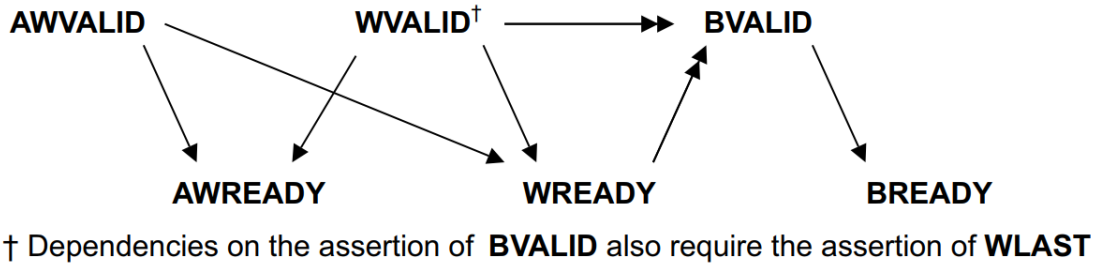


图 1-9 写事务握手对依赖关系

这里单向箭头所指的信号，可以在箭头开始前的信号被拉高之前或之后拉高（无依赖关系）。双向箭头所指的信号，必须在箭头开始前的信号拉高后才能被拉高（有依赖关系）。

从图中可以看到，在进行写事务时，握手对间仅 BVALID 对 WVALID 和 WREADY 有依赖关系，也就是只有当数据被发送（WVALID 和 WREADY 被拉高），并且是一次突发的最后一个数据被发送（TLAST 拉高）后， BVALID 才被允许拉高。

而图中写数据和写响应通道的握手信号不需要等待写地址通道的握手对信号先产生，也从侧面验证了，AXI4 的写传输是支持不对齐传输的，也就是数据可以在地址之后发送，也可以与地址一同发送，当然，如果接收方有足够的 FIFO，数据也可先于地址发送。

实际上，在图 1-9 中还存在着一个 bug，那就是如果数据先于地址发送，且数据已经发送完并拉高了 WLAST 信号， 而写地址通道此时还没有给出地址。这种情况下，按照上面的依赖关系，写响应通道是能够握手成功，发出响应信号的。但是对于写事务而言，此时我们的一次传输还没完成。

为此，AXI4 协议中专门增加了从机拉高 BVALID 信号所需的依赖（在 AXI3 中是不存在这个依赖的）：

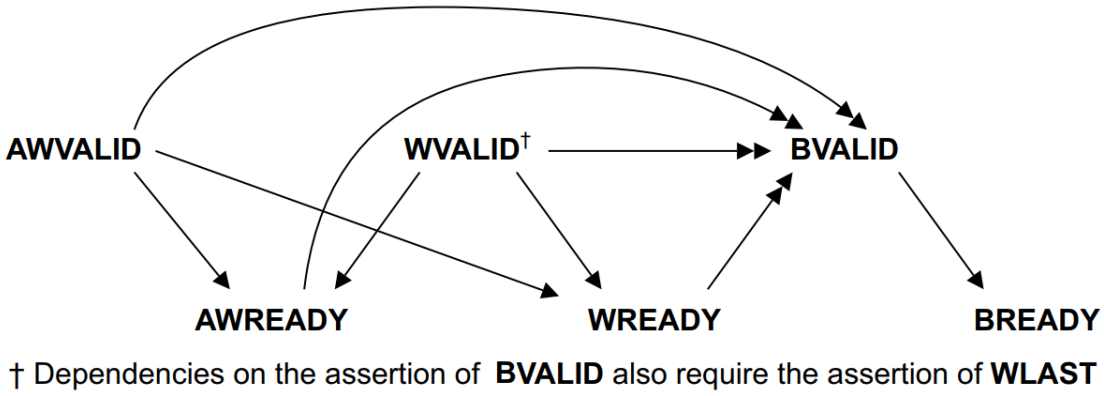


图 1-10 从机写响应握手对依赖关系

这样，写响应通道只有在主机给出地址并且一次突发完成后，才能够进行写响应。可以看到在这方面 AXI4 相对于 AXI3 而言，更为严谨。

然后是读事务握手对依赖关系：

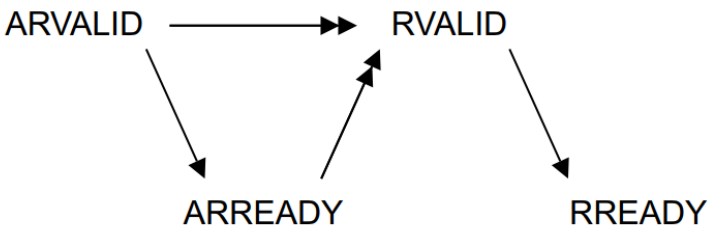


图 1-11 读事务握手对依赖关系

这里相对于写事务要简单不少，仅 RVALID 信号对 ARVALID 和 ARREADY 信号存在依赖关系，也就是前面我们说过的，读事务时，必须先写地址再读数据，毕竟从机不能未卜先知，在不知道地址的情况下向主机发送正确的数据。

1.1.5 传输结构

上述五个通道的信号对在握手成功之后，便可以开始传输了，根据各个通道的特性，AXI4 的传输可以分为三个部分，即地址传输结构、数据读写结构以及读写响应结构。

在开始讲 AXI4 的地址结构之前，我们首先需要知道一个在 AXI4 传输时一个重要的规定：单次 burst 传输中的数据，地址不能跨越 4K 边界。

这里的 4K 边界也就是指那些低 12 位全部为 0 的地址，例如：32'h00001000，32'h00002000，32'h00004000 等这些都为 4k 边界。由于系统中定义每一个 page 的大小为 4K，为了更好设定每个 slave 的访问 attribute，通常每一个 slave 会被

划分 4k 空间。

如果一次 burst 跨过了 4K 边界就会访问到两个 slave（假设两个 slave 分别为 A 和 B，且 A 的地址在 B 之前）。那么只有 A 会收到本次 burst 的地址和控制信号，因此只有 A 会产生响应而 B 不会有任何动作，进而导致传输无法完成。

所以为了避免出现这种情况，在设置 AXI4 传输地址时，尽量设置与 4K 边界对齐的地址。当无论如何都会跨越 4K 边界时，就需要对 burst 进行拆分，也就是分为多次传输。

1.1.5.1 地址结构

在 AXI4 的地址传输中，有着三项对通信起着至关决定性的信号，分别是 AxLEN、AxSIZE、AxBURST，这三组信号控制着传输的突发长度（burst length）、突发大小（burst size）、突发类型三个参数：

（1）突发长度（burst length）

突发长度用来控制一次 burst 共需要多少次传输，也就是有多少个 transfer，该项由 AxLEN 信号控制。在 AXI4 中，不同突发类型，一次 burst 所能支持的最大传输次数也会有所不同，在 INCR 类型下支持最大长度为 256，其他类型的最大长度为 16。关于各个突发类型，我们稍后再讲。

（2）突发传输大小（burst size）

突发传输大小用来控制每次 transfer 的数据总线位宽，该项由 AxSIZE 信号控制，具体计算公式如下：burst size＝2^AxSIZE，单位为 Byte。AxSIZE 信号的位宽为 3，因此突发传输大小最高为 128Bytes，如表 1-2 表：

表 1-2 突发传输宽度编码

AxSIZE	传输字节数	AxSIZE	传输字节数
3’b000	1	3’b100	16
3’b001	2	3’b101	32
3’b010	4	3’b110	64
3’b011	8	3’b111	128

（3）突发传输类型（AxBURST）

AXI4 中无论读写都支持三种突发传输类型，分别是 FIXED、INCR、WRAP。突发传输类型由 AxBURST 信号控制，具体如下：

表 1-3 突发类型编码

AxBURST[1:0]	Burst type
0b00	FIXED（固定）
0b01	INCR（递增）
0b10	WRAP（回环）
0b11	保留

这三种突发类型下传输数据的方式分别如下：

1. FIXED：每次 burst 使用的都是同一个地址，该突发类型用于对同一地址的重复访问，例如加载和清空 FIFO。
2. INCR：该类型是最常用的类型，地址会跟据突发数据的大小自动累加，例如大小为 4 字节的 INCR 突发传输，每个 transfer 的地址是前一个地址加上 4。该突发类型适用于对 RAM 等通过地址映射的存储介质进行读写操作。
3. WRAP：该传输较为复杂，首先传输起始地址要求必须与每次传输大小对齐，然后突发传输的长度只能为 2,4,8,16 中的一个。

在进行 WRAP 类型传输时，会根据起始地址计算下边界地址（wrap boundary）和上边界地址，地址会依照 INCR 的方式进行累加，当达到上边界地址时，会回到下边界地址继续累加，如此循环往复。因此该类型的突发适用于对 cache 的访问。

有关边界地址计算公式如下，首先是下边界地址（这里的 INT 用来进行向下取整）：

Wrap_Boundary = (INT(Start_Address / (Number_Bytes × Burst_Length)))
× (Number_Bytes × Burst_Length).

接下来是上边界地址：

Address_N = Wrap_Boundary + (Number_Bytes × Burst_Length)

根据这两个公式我们就能计算出上下边界的值，进而计算出每次传输的地址。例如：在进行一次突发写时，AWADDR = 0x34, AWLEN = 0x07, AWSIZE = 0x02, AWBURST = 0x03

那么根据这些信号我们可以知道，该突发写为 WRAP 类型，起始地址为 0x34，每次传输数据大小为 2^2=4 字节，一次 burst 需要分为 7+1=8 次传输。

因此我们可以计算出：

- 下边界地址=(INT(52/(4x8)))x(4x8)=32=0x20

- 上边界地址=32+(4x8)=64= 0x40

最终求出传输地址依次为 0x34、0x38、0x3C、0x20（从上边界 0x40 回到下边界）、0x24、0x28、0x2C、0x30。

1.1.5.2 数据读写结构

AXI4 协议在进行数据突发传输时，除了典型时序中的传输方式外，还可能面临窄传输、不对齐传输以及大小字节端共存的情况，在这几种情况下 AXI4 数据的传输格式如下：

(一) 窄传输（Narrow Transfer）

在 AXI4 传输中，突发传输位宽不一定要与通道位宽一致，当突发传输位宽小于通道位宽时，我们称其为窄传输（Narrow Transfer）。在不同的传输类型下，通道的有效位会有所不同，协议中规定，在 INCR 和 SWAP 传输类型时，每个 Beat 都会使用不同的字节位置（byte line），而在 FIXED 传输类型时，每个 Beat 都使用相同的字节位置。

在进行窄传输的写操作时，因为写数据通道存在写选通信号 WSTRB，所以主机会使用该信号来指定写通道中哪些位置上的字节是有效的，WSTRB[n]标示的区间为 WDATA[(8*n)+7:(8*n)]。而在进行窄传输的读操作时，由于读数据通道没有类似的选通信号，所以就需要主机根据突发传输位宽与总线宽度，计算出当前总线中有效数据所在字节位置，然后截取数据。

这里以窄传输的写操作为例，图 1-12 是一个 INCR 类型的写突发传输，起始地址为 0，一次突发有 5 个 transfer，每个 Beat 的位宽为 8 位，总线位宽为 32 位，白色为我们要发送的数据，灰色为无效数据（填充数据）：

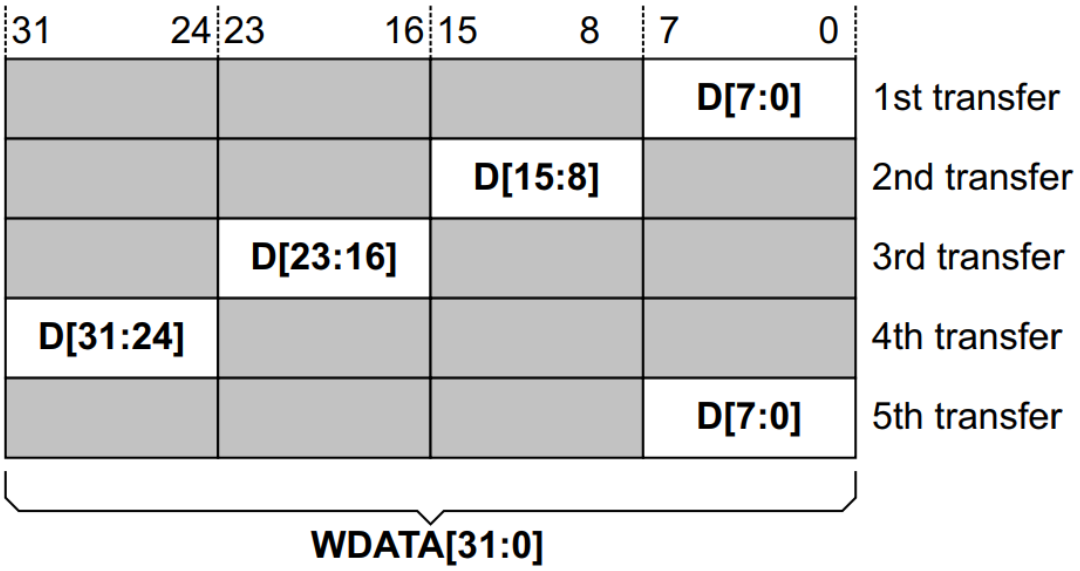


图 1-12 INCR 窄传输

可以看到，每次 transfer 时有效数据使用的数据总线字节位置不同，在传输第一个 Beat 时使用的 D[7:0]，传输第二个 Beat 时使用的 D[15:8]....按顺序使用不同的数据总线字节位置。这种写入方式，可以通过这种方式，位宽较小的数据可以不进行位宽转换，直接写入到 DDR 这类存储介质中，然后再通过 WSTRB 信号屏蔽掉无效位即可。

(二)非对齐传输（Unaligned transfers）

通常来说，在 AXI4 的突发传输中，每个数据的地址都会对齐到传输大小，例如，一个 32-bit 宽的传输通常会对齐到 4-byte 边界。但是在一些场景中，我们可能会希望在一个与传输大小不对齐的地址上开始突发传输，这种我们称之为非对齐传输。

非对齐传输会对每次 burst 的第一个 transfer 进行干预，以确保后续的 transfer 为对齐传输。对于写事务而言，在进行第一次 transfer 时数据会按照总线地址边界对齐的方式进行传输，主机再通过写选通信号 WSTRB 屏蔽掉对应的位，以保证有效数据与传输地址的对齐，这样，后续的 transfer 中，数据与地址便变成了对齐传输。而读事务时，主机则需要根据传输地址，在接收数据后将有效部分提取出来。

以图 1-13 为例，这是 32 位位宽的总线上，两个不同写事务对应的数据传输格式。灰色方格为无效数据，白色方格为有效数据，每个方格内的数字代表数据所对应的地址，每个写事务左侧为传输参数：

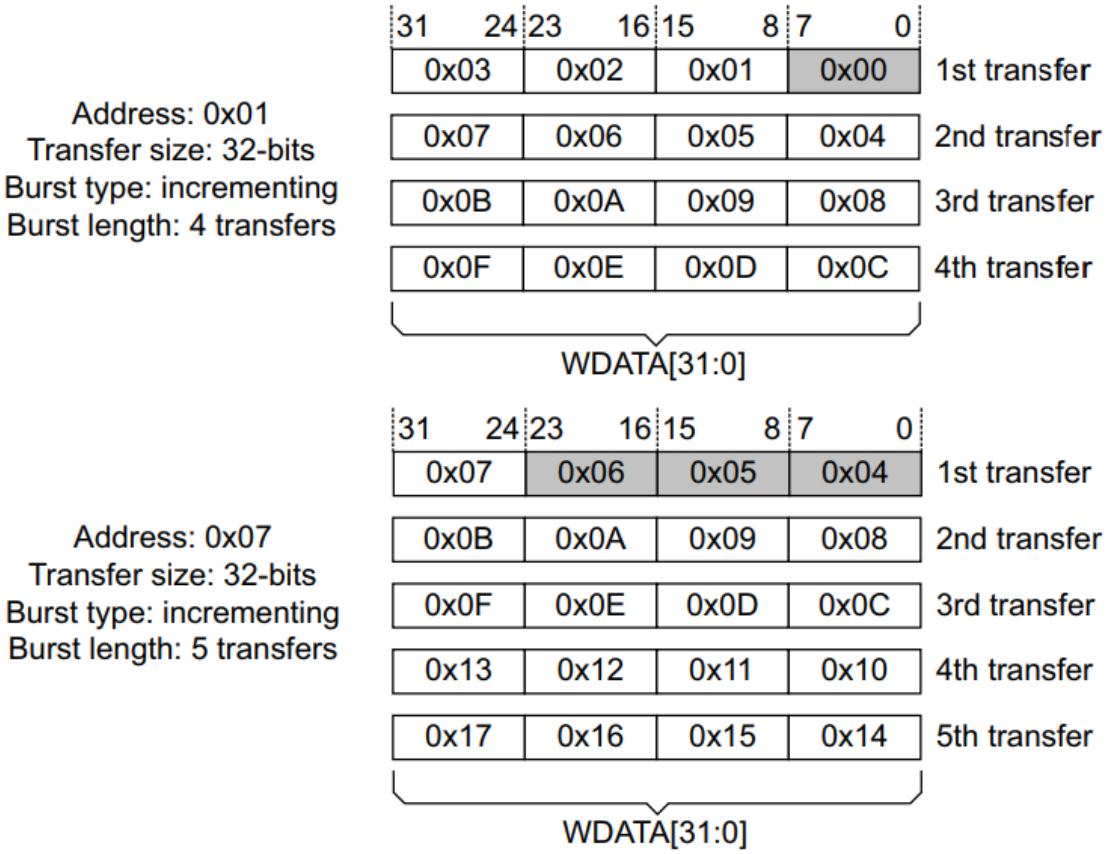


图 1-13 非对齐传输

可以看到，两个写事务的数据与总线地址是对齐的，第一个写事务的起始地址为 0x01，因此在进行第一个 transfer 主机屏蔽了 0x01 地址之前的数据，此时的传输为非对齐传输，在后续的 3 个传输中，数据与地址一一对应，为对齐传输。在第二个写事务中，起始地址为 0x07，因此主机屏蔽掉了前几个地址来确保传输对齐，同样的，第一次 transfer 为非对齐传输，后续四个 transfers 都为对齐传输。

当然了，这两个写事务中传输大小与总线大小是一样的，但在实际应用场景中也有传输大小小于总线大小的情况，也就是我们在前面讲到的窄传输。在窄传输模式下，总线也是可以使用非对齐传输的，如图 1-14 所示：

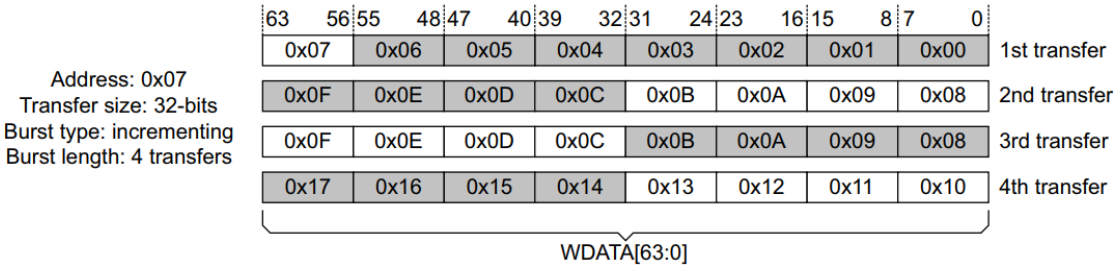


图 1-14 窄传输模式下的非对齐传输

可以看到这是一个写事务，传输类型为 INCR，传输大小为 32 位，总线位宽为 64 位，那么根据前面窄传输的知识，在第一次 transfer 时，数据应该使用低 32 位的总线数据位，但是由于起始地址为 0x07，所以实际使用的为高 32 位的总线数据位。同时，主机通过 WSTRB 信号将传输中 0x07 地址之前的数据屏蔽，以确保后续 transfer 为对齐传输，因此在后续的三个 transfers 中，数据都是按照正常的窄传输方式传输。

(三)字节不变（Byte invariance）

我们都知道，在计算机的内部，字节有两种排列方式，即低位字节存放在高地址端，高位字节存放在低地址端的大端字节序，以及低位字节存放在低地址端，高位字节存放在高地址端的小端字节序。

AXI 协议为了方便访问保存在同一个内存空间中的混合端数据结构，定义了字节不变端序，对于存储中包括多个字节的数据结构：

1. 无论大端还是小端字节序，每个数据结构存储空间的分配方式相同
2. 该数据结构按照其大端或小端模式决定字节存储的地址顺序
3. 在传输过程中不考虑数据结构的大小端，按照字节原先存储的顺序，原样传输并存放至对端

也就是说字节不变端序在传输时，不会对数据结构的大小端进行解析转换，而是严格按照字节的存储顺序进行传输并转存，防止大小端共存产生数据覆盖。至于数据到底是大小端的处理，则由接收方在内部进行。

图 1-15 为需要进行字节不变访问的数据结构示例，其中标头信息使用的是小端字节序，有效负载使用的大端字节序：

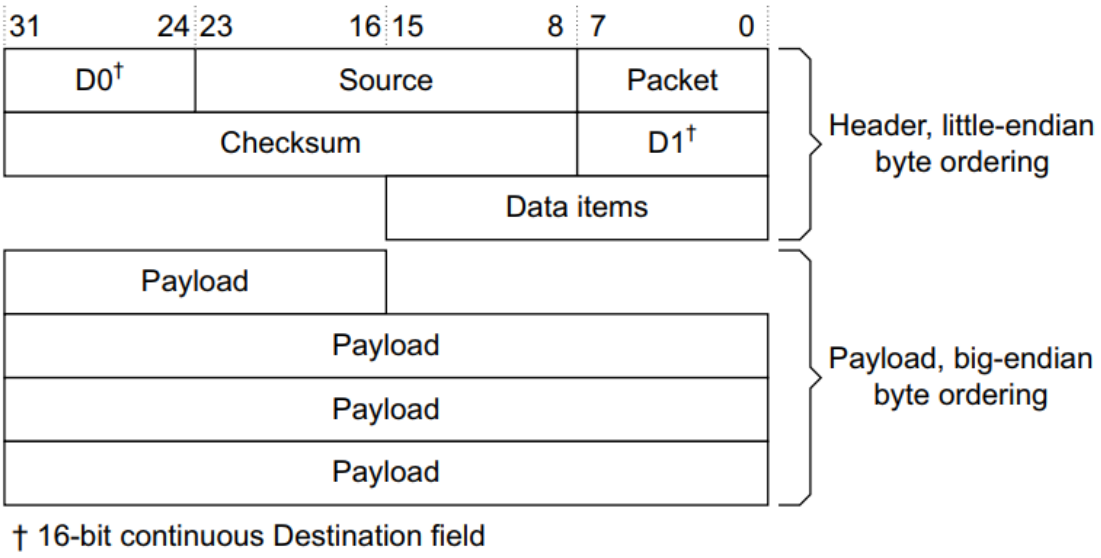


图 1-15 混合端数据结构

可以看到，在这种情况下，使用字节不变端序能够确保对部分标头信息的小端访问不会破坏结构中的其他大端数据。

1.1.5.3 读响应结构

AXI4 的读写事务中都有各自对应的响应信号，其中写事务中的响应信号为 BRESP，存在于写响应通道；读事务的响应信号为 RRSEP，存在于读数据通道。这两个信号的位宽都为 2bit，从机通过响应信号发送不同的值给主机来反馈传输的状态。响应信号的值及所代表的意义如下：

xRESP[1:0]	响应	说明
0b00	OKAY	正常访问成功/独占访问失败
0b01	EXOKAY	独占访问成功
0b10	SLVERR	从机错误
0b11	DECERR	解码错误，通常由互联组件产生，指示事务地址处没有从设备

这里的独占访问即主机通过 AxLOCK 信号对某个内存地址的数据，在某个时间段内享有独有的访问。

SLVERR 响应通常是从机接收到了传输指令，但是由于一些原因，从设备需要向主机返回错误状态时产生。至于出错的原因，包括但是不仅限于以下几种：

- FIFO 或者缓冲区溢出或不足
- 不支持该传输位宽
- 尝试向读保护的地址写入数据

- 超时

需要注意的是，写响应只有在一次 burst 结束时，才会给出响应信号，而读响应则会响应 burst 的每次 transfer。同时，协议还规定就算在传输的过程中出现了错误，仍要执行完所需数据量的传输。例如，在进行一次 16 个 transfer 的突发读过程中，从机在进行了 6 个 transfer 后出现了错误，并给出了相应的响应信号，余下的 10 个 transfer 从机任然需要传输，并对每个 transfer 做出响应。在这种情况下，主机可以直接丢弃掉读取的数据。

1.1.6 Outstanding 传输

Outstanding 传输也就是未完成传输，该传输下不需要等待前一个 burst 传输完成就可以发送下 N-1 个操作。如果总线不支持 outstanding 或者 outstanding 能力为 1，那么主机和从机间便按照典型时序图中的模式传输。

当 Outstanding 能力 N>1 时，其读写操作类似于图 1-16 和图 1-17：

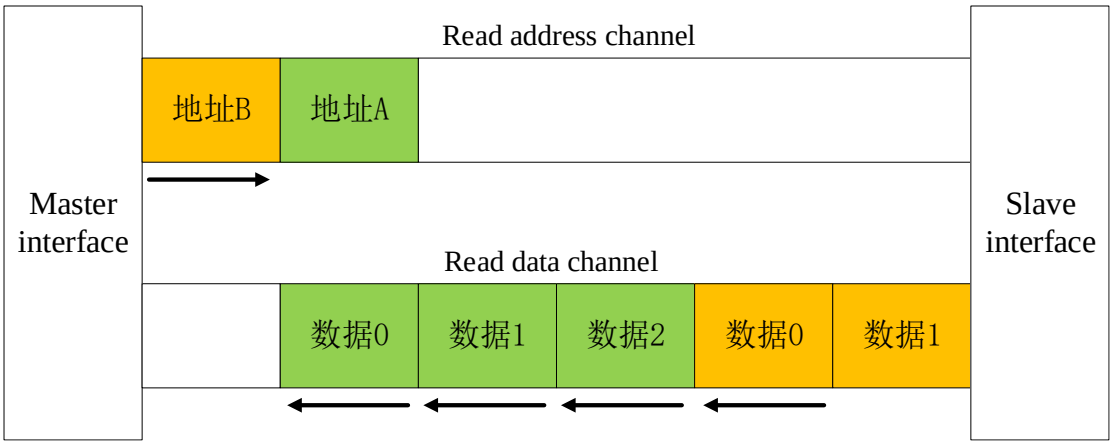


图 1-16 outstanding 传输（读）

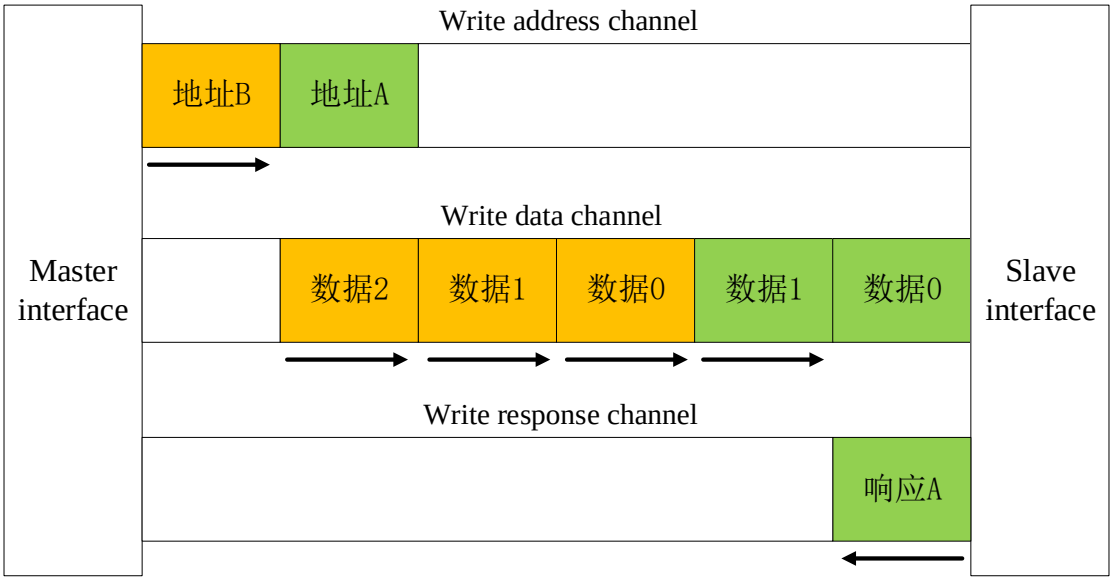


图 1-17 outstanding 传输（写）

主机可以连续发送 N 个读写地址命令，这期间如果读数据没有返回，则需要等待读数据返回，如果有读数据返回，则返回了几个，那么仍然可以接着发几个，也就是可以有 N 个未完成的读指令或数据。

以图 1-18 为例，图中是一个 outstanding 的读事务时序：

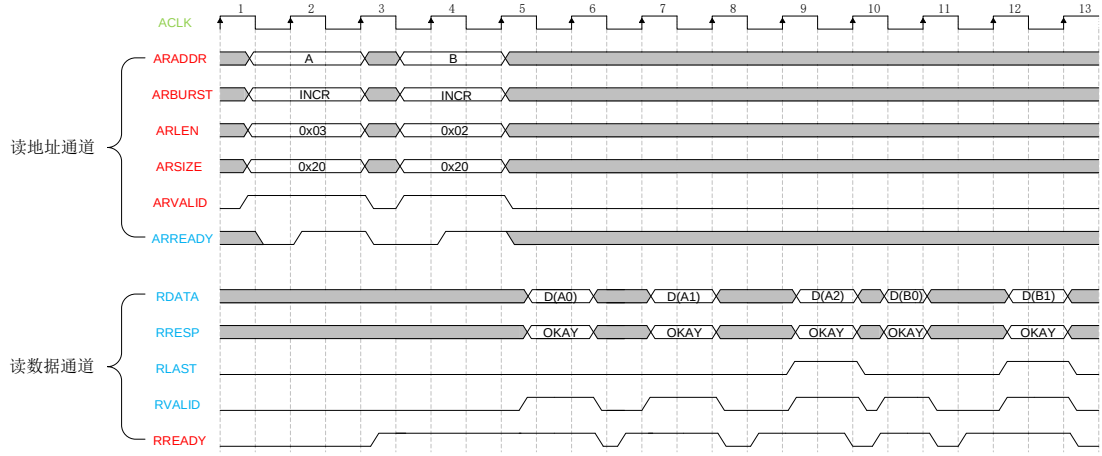


图 1-18 outstanding 读传输

从图中可以看到，主机首先向从机发送了两个读地址指令。根据信号的值，我们可以知道，这两条指令的含义分别是：使用 INCR 传输类型，从起始地址 A 开始，读取 3 个位宽为 32 的数据；使用 INCR 传输类型，从起始地址 B 开始，读取 2 个位宽为 32 的数据。

在主机发送第二个指令得到同时，从机开始根据第一个指令传输从地址 A 开始的 3 个连续的 32 位数据 D(A0)~D(A2)，并在每次传输时，给出响应信号，

反馈当前传输状态。D(A2)为第一次 burst 的最后一个数据，因此在传输 D(A2)的同时，从机会拉高 RLAST。

如果按照典型时序图中的操作，从机在执行完第一条指令后，需要等待主机再次握手成功后发送的第二条指令。而 outstanding 传输由于能够提前发送指令，从机在执行完第一条指令后，不需要再去等待主机握手，能够很快的开始执行第二条指令，也就是图 1-18 中向主机传输 2 个 32 位数据 D（B0）和 D（B1）的过程。在传输的过程中，从机同样需要对每次传输做出响应，并在 burst 的最后一个 transfer 时，拉高 RLAST 以通知主机当前为最后一个数据。

因此，我们可以发现，Outstanding 传输能够减小多个指令间的等待时间，提升多个 burst 传输时的效率。理论上只要总线 outstanding 的能力越高，总线的效率就越高，当然也不能无限制的发，不然就容易造成总线拥塞。

1.2 AXI 互联

严格来说，AXI 协议就是一个点对点的主从接口协议，每个 AXI 设备都有自己的主从接口。通常在一个 AXI 系统中都会存在有多个主机和多个从机，主机可能会与其中一个或多个从机存在着联系，因此彼此间存在着一主一从、多主一从、一主多从和多主多从的情况。下图图 1-19 便是典型的 AXI 接口互联结构：

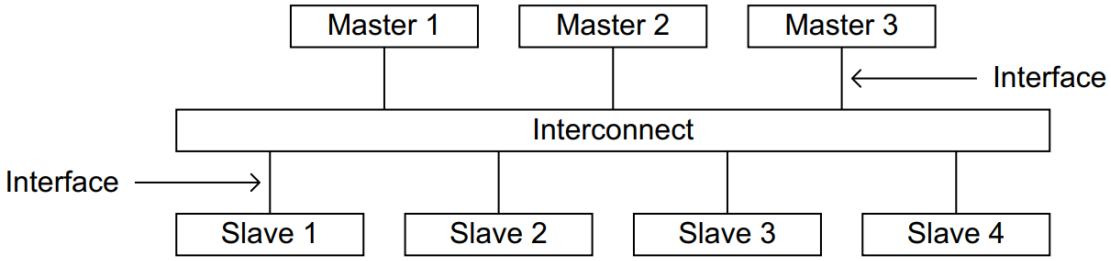


图 1-19 接口与互联

对于总线接口相同的 AXI 主从设备之间是可以直接相连的，但是通常来说，我们更习惯用特定的 Interconnect 核来连接各个 AXI 主从设备（包括不同总线接口的 AXI 主从设备）。正如前面所说，通常一个 AXI 系统中存在多个主机和从机，如果我们将相同接口的总线直连，就意味着每个总线都使用单独的总线通信，这对于总线资源而言，无疑是一种巨大的浪费。况且这些主从之间的传输并不需要一直占用总线，总线的利用率也非常的低。而如果在设计中加入 Interconnect 核，使用这类核作为主从接口之间的桥，这类核会对设计中各个

AXI 通道总线资源进行统一的管理、分配和调度，大大提高总线利用率节省总线资源。

除了这些之外，这类核通常还具有协议转换等功能，以满足不同总线接口之间的传输需求。以 AXI3、AXI4 和 AXI4-Lite 接口为例，这些接口都包含地址和数据总线，我们通常使用的是 AXI Interconnect 核或者 AXI SmartConnect 核来连接主从接口。也就是我们在进行硬件逻辑系统设计时最常看到的，系统自动为我们添加的两个基础 IP。

这两个 IP 核在功能上有很大的相似度，除了对总线统一管理调配外，还支持 AXI4、AXI3、AXI4-Lite 协议的互转、突发拆分等功能，这是因为这两个 IP 核在内部都包含了 AXI Data Width Converter、AXI Protocol Converter 软核 IP 实例。这两个实例 IP 在 Interconnect 核配置期间被配置和连接，每个 IP 都可以在 Vivado 的 IP 库中直接搜索找到。

AXI Data Width Converter 负责数据位宽的转换，以及在发现数据传输长度超过了最大突发长度时，对突发事务进行拆分。

AXI Protocol Converter 则是负责协议之间的转换，支持 AXI4 或 AXI3 到 AXI4-Lite 协议的转换以及 AXI4 到 AXI3 的协议转换。在进行协议转换时，该 IP 会被缺失的信号补充默认值，对多余的信号进行屏蔽，对不同定义的同一信号进行转换。如果需要，该 IP 在进行协议转换时，同样也会对传输事务进行拆分，以满足协议突发长度。

当然了，既然是两个不同的 IP，也就说明 AXI Interconnect 和 AXI Smartconnect 的应用场景是不同的。虽然这两个 IP 核都能将一个或多个 AXI 内存映射主设备连接到一个或多个内存映射从设备。但是，AXI SmartConnect 更紧密地集成到 Vivado 设计环境中，以自动配置和适应连接的 AXI 主从 IP，受用户干预最少；AXI Interconnect 则可用于所有内存映射设计。AXI SmartConnect IP 通过合成针对重要接口进行优化的低面积定制互连，能够以很低的延迟为系统提供最大吞吐量，因此，在某些高带宽应用情况下，使用 AXI SmartConnect 可以提供更好的优化。

对于新的中高性能设计，建议使用 AXI SmartConnect IP，因为它在面积和时序方面提供了更好的向上扩展。而 AXI Interconnect 面向一些低性能 (AXI4-Lite) 或中小型复杂度设计更具效率性。

由于 AXI4-Stream 接口没有地址通道，所以不需要使用到 AXI Interconnect 或 AXI SmartConnect，通常可以直接相互连接。而在一些需要转换数据类型的

情况下可以连接到 DMA IP 核或 AXI4-Stream Interconnect(axis_interconnect) IP 核后再连接到目标接口。

1.3 物理接口

AXI 总线存在的意义是为了实现 PS 与 PL 之间的交互，而为了实现这种交互，在 PS 与 PL 之间，存在有三种物理接口，即 AXI_ACP、AXI_HP、AXI_GP。这些接口我们可以通过 ZYNQ IP 核的配置预览界面看到，如图 1-20 中红框部分所示：

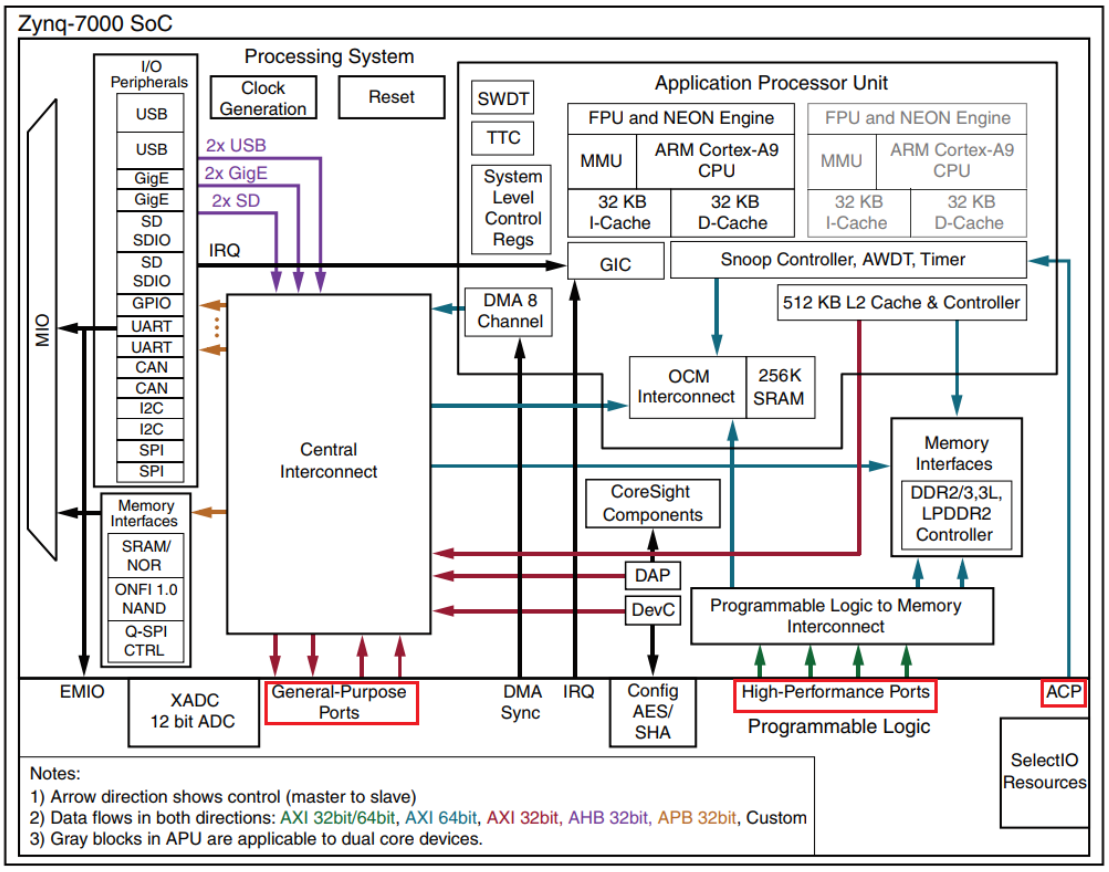


图 1-20 AXI 接口

图中用三种颜色的箭头标出了这三种接口在 PS 侧内部的通路，虽然位于 PS 与 PL 之间，但是这三个接口属于 PS 侧，因此所使用的都是 AXI3 协议。每个接口的功能如下：

- AXI_ACP（CPUs and accelerator coherency port）：CPU 和加速器一致性端口，是 ARM 多核架构下定义的一种接口，用来管理 DMA 之类的不带缓存的 AXI 外设，PS 端是 Slave 接口。
- AXI_HP（High performance ports）：高性能/带宽的标准接口，共有四个

接口，PL 模块作为主设备连接。

- AXI_GP（General purpose ports）：通用的 AXI 接口，包括两个 32 位主设备接口和两个 32 位从设备接口，该接口直接连接到的是中央互联（central interconnect），然后由中央互联再连接到片上互联（OCM interconnect）和存储器接口（Memory Interface）上。

这三种 AXI 接口在使用时，PS 与 PL 所扮演的角色如表 1-4 所示：

表 1-4 AXI 接口

接口名称	接口描述	主机	从机
M_AXI_GP0	通用型（AXI_GP）	PS	PL
M_AXI_GP1		PS	PL
S_AXI_GP0	通用型（AXI_GP）	PL	PS
S_AXI_GP1		PL	PS
S_AXI_ACP	加速器一致性端口，缓存一致性事务（ACP）	PL	PS
S_AXI_HP0	高性能端口（AXI_HP），带有读/写 FIFO 和 DDR 控制器上的两个专用内存端口，以及通向 OCM 的路径。AXI_HP 接口也称为 AFI。	PL	PS
S_AXI_HP1		PL	PS
S_AXI_HP2		PL	PS
S_AXI_HP3		PL	PS

ACP 接口常常用来作为专用指令加速模块接口，PL 端可以直接从 PS 部分的 Cache 中拿到 CPU 的计算结果，同时也可以第一时间将逻辑加速运算的结果送至 Cache 中。

HP 接口本身带有读写缓冲 FIFO，并且连接到 DDR 与 OCM，因此常常被用来向 DDR 等存储介质读写数据，常见的有 AXI4 协议经过 Interconnect 核转换协议后通过 HP 接口将数据写入 DDR，或者 AXI4-Stream 通过 VDMA 之类的数据搬运核，经由 HP 接口将数据以内存映射的方式写入 DDR。

GP 接口本身不具有 HP 接口的高性能和高吞吐，且连接的是中央互联，因此常被用来控制外设配置或对外设进行小量数据的读写，最常见的应用就是 PS 通过 GP 接口经由 Interconnect 核连接到目标 IP 的 AXI4-Lite 接口，对 IP 核初始化配置。

以上便是 AXI4 协议、AXI 互联结构以及 PS 与 PL 间实现互联的接口，对于 ZYNQ-7000 系列器件来说，这些仅仅只是理论上的东西，只有实践才是检验真理的唯一标准。所以接下来我们就需要动手，构建一个使用 AXI4 与 PS 通信的 AXI 系统，通过 ila 抓取信号波形进行分析，来验证上述内容是否正确。

1.4 设计实例

虽然 Xilinx 为我们提供了许多使用 AXI4 协议接口的 IP，但是这些 IP 通常除了 AXI4 外还会有一些其他的功能和相关信号，并不适合用来做验证设计。并且这些 IP 代码通常并不是开源的，我们无法直接查看代码实现过程。

因此，从学习角度考虑，这里我们使用自定义 IP 核模板来创建一个自定义 AXI4 主接口的 IP 进行验证。用户不用担心 AXI4 协议的代码实现，在创建时，模板会自动为我们添加一段代码，该代码会向 DDR 中指定地址写入一串从 0 开始自加的数据，并读回比较，反馈比较结果。

1.4.1 创建带 AXI4 主接口的自定义 IP 核

自定义 IP 的创建较为简单，这里只给出用户需要修改的几个操作项，如图 1-21 和图 1-22 所示。详细的自定义 IP 的步骤流程可以参考自定义 IP 核章节。

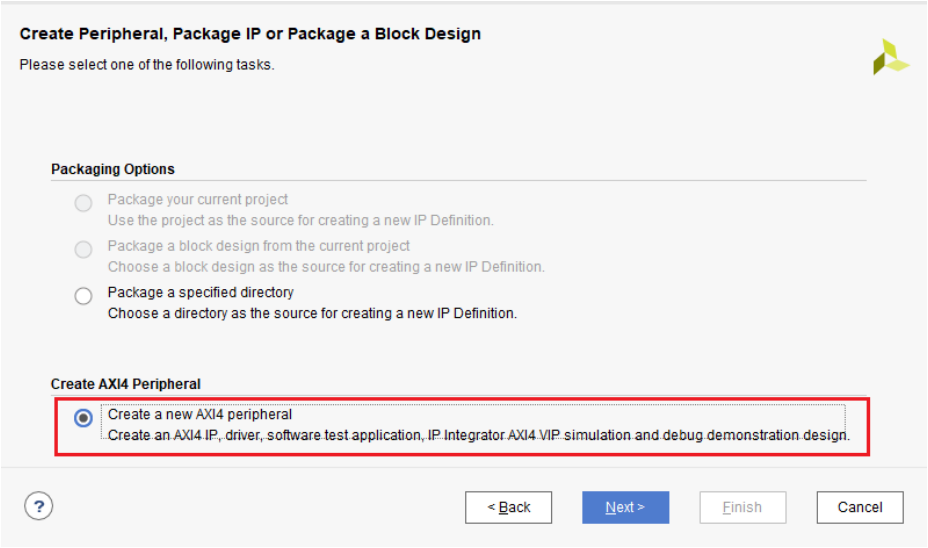


图 1-21 创建带 AXI4 接口 IP

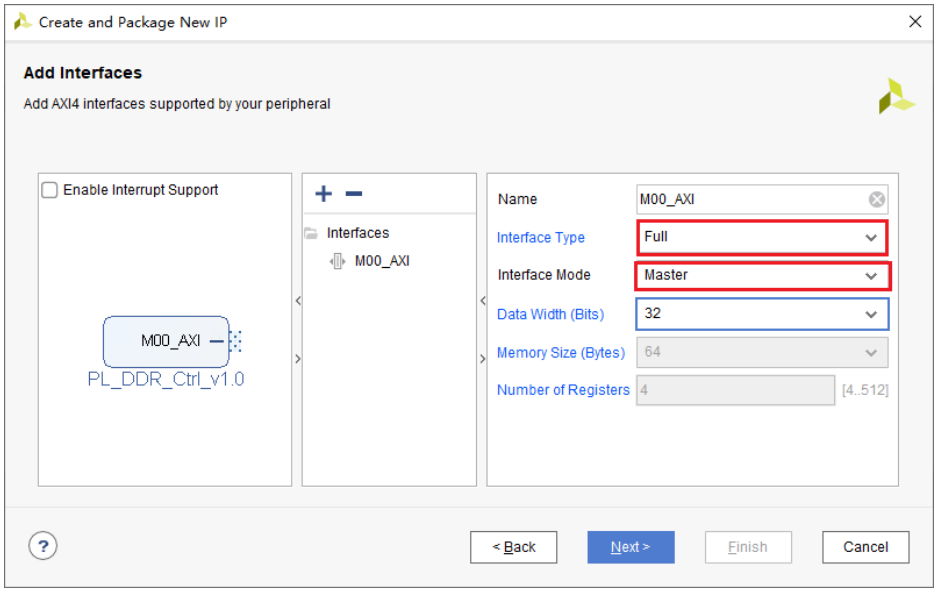


图 1-22　选择 AXI4-FULL 类型的主机接口

创建完成后的 IP 核会被存在 User Repository 下的 AXI Peripheral 文件夹中，双击 IP 核就可以看到 IP 核的 GUI 配置界面，如图 1-23 所示：

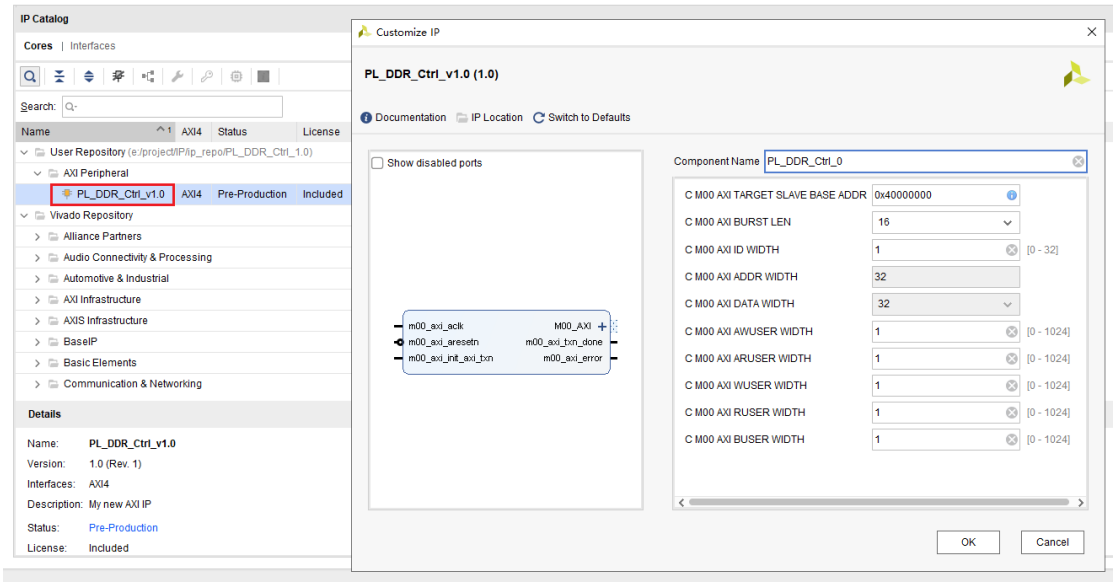


图 1-23　自定义 IP 核 GUI 配置界面

GUI 配置项中，自上而下，前五个参数分别是目的地址、突发长度、ID 信号位宽、地址总线位宽、数据总线位宽。这里的 ID 信号，用于在乱序传输时，将数据与地址以及响应信号相对应，由于 AXI4 中删除了 WID，所以只能够进行乱序读。至于后五个参数用来设置五个通道的用户自定义信号位宽，自定义信号的作用可以由用户通过逻辑编程自己设定。

接下来我们可以通过右键 IP 核选择在 IP 封装器中编辑来简单查看一下 IP

核的源码，如图 1-24 所示：

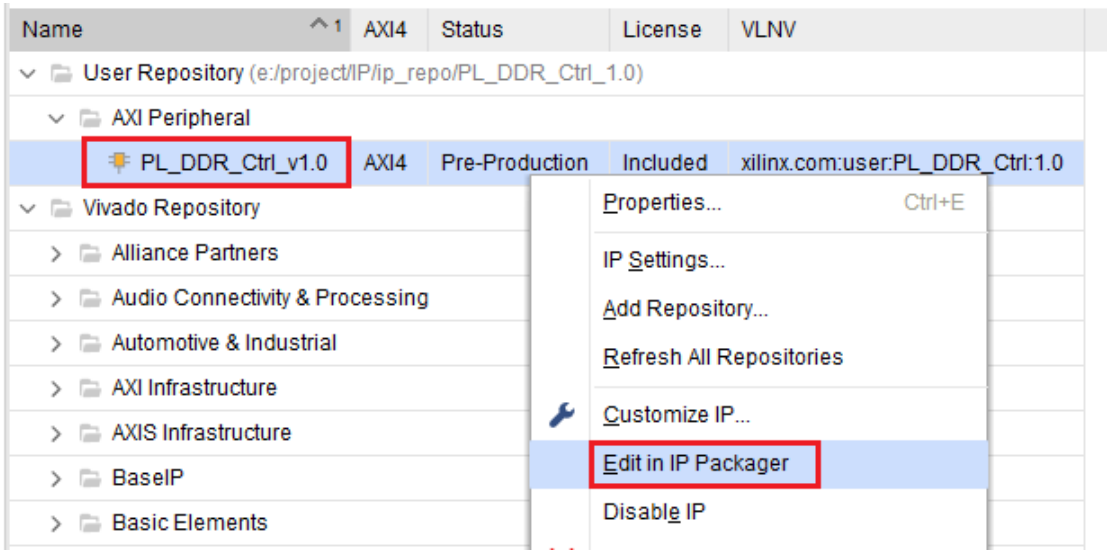


图 1-24 编辑 IP

在 Source 栏中打开 xxxx_v1_0_M00_AXI.v 文件，该文件中存放的就是自定义 IP 核的源码。首先是第 11~30 行，这里是从顶层传参来的 GUI 配置界面中的 10 个参数，如图 1-25 所示：

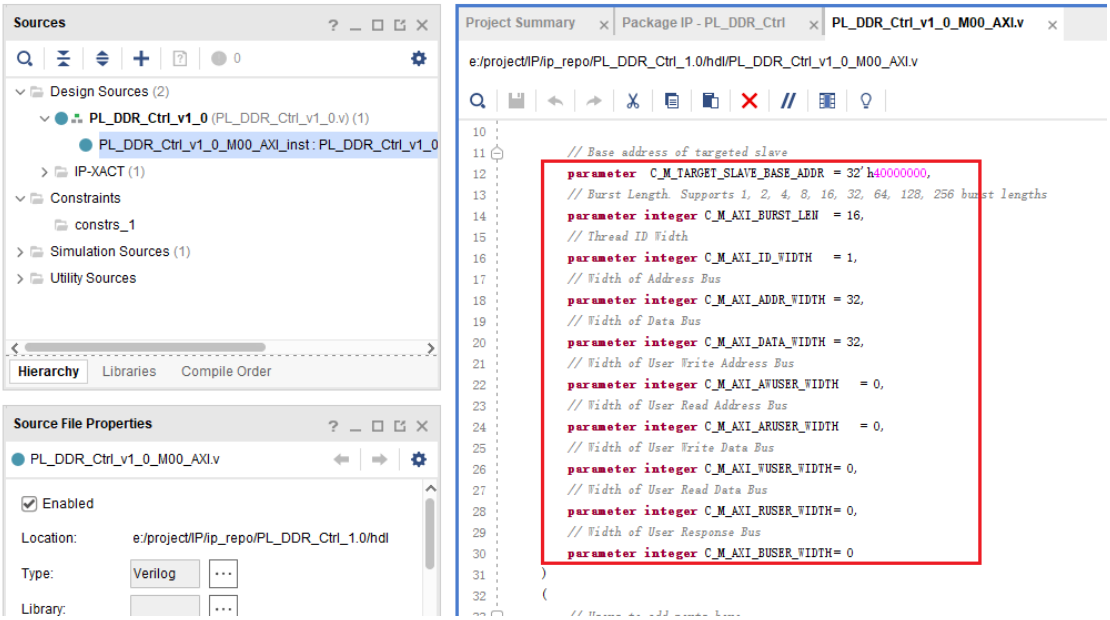


图 1-25 配置参数

随后是 38 行~155 行的接口信号，每个信号都有注释说明，这里我们需要注意的是 INIT_AXI_TXN、TXN_DONE 和 ERROR 信号，如图 1-26 所示：

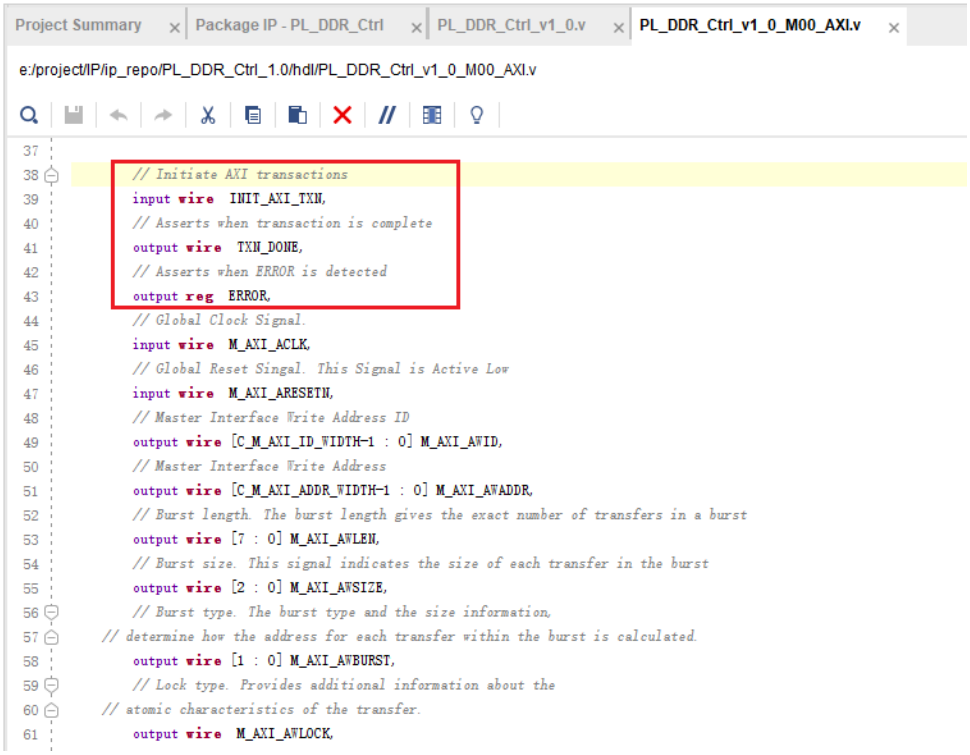


图 1-26 启动传输信号

INIT_AXI_TXN 信号用来启动 AXI 传输，高电平有效；TXN_DONE 信号用来反馈比较完成（以及写数据成功）；ERROR 信号用来反馈数据读写错误。

在代码的 241~288 行中对一些信号进行了定义，如图 1-27 所示：

```
248    //INCR burst type is usually used, except for keyhole bursts
249    assign M_AXI_AWBURST    = 2'b01;
250    assign M_AXI_AWLOCK = 1'b0;
251    //Update value to 4'b0011 if coherent accesses to be used via the Zynq ACP port. Not Allocated, Modifiable, not Bufferable. Not Bufferable sin
252    assign M_AXI_AWCACHE    = 4'b0010;
253    assign M_AXI_AWPROT = 3'h0;
254    assign M_AXI_AWQOS  = 4'h0;
255    assign M_AXI_AWUSER = 'b1;
256    assign M_AXI_AWVALID    = axi_awvalid;
257    //Write Data(W)
258    assign M_AXI_WDATA  = axi_wdata;
259    //All bursts are complete and aligned in this example
260    assign M_AXI_WSTRB  = {(C_M_AXI_DATA_WIDTH/8){1'b1}};
261    assign M_AXI_WLAST  = axi_wlast;
262    assign M_AXI_WUSER  = 'b0;
263    assign M_AXI_WVALID = axi_wvalid;
264    //Write Response (B)
265    assign M_AXI_BREADY = axi_bready;
266    //Read Address (AR)
267    assign M_AXI_ARID   = 'b0;
268    assign M_AXI_ARADDR = C_M_TARGET_SLAVE_BASE_ADDR + axi_araddr;
269    //Burst LENgth is number of transaction beats, minus 1
270    assign M_AXI_ARLEN  = C_M_AXI_BURST_LEN - 1;
271    //Size should be C_M_AXI_DATA_WIDTH, in 2^n bytes, otherwise narrow bursts are used
272    assign M_AXI_ARSIZE = clogb2((C_M_AXI_DATA_WIDTH/8)-1);
273    //INCR burst type is usually used, except for keyhole bursts
274    assign M_AXI_ARBURST    = 2'b01;
```

图 1-27 信号绑定

我们可以看到 WRBURST 和 ARBURST 信号的值都为 2’b01，也就是说读

写通道都是使用的 INCR 传输类型。

代码中通过注释为我们划分出了各个通道的相关事务实现代码，这里列举出部分。写入 DDR 的数据通过逻辑生成，如图 1-28 所示：

```
452     /* Write Data Generator
453      Data pattern is only a simple incrementing count from 0 for each burst  */
454       always @(posedge M_AXI_ACLK)
455       begin
456         if (M_AXI_ARESETN == 0 || init_txn_pulse == 1'b1)
457           axi_wdata <= 'b1;
458         //else if (wnext && axi_wlast)
459         //  axi_wdata <= 'b0;
460         else if (wnext)
461           axi_wdata <= axi_wdata + 1;
462         else
463           axi_wdata <= axi_wdata;
464         end
```

图 1-28 生成写数据

写数据对应的地址生成如图 1-29 所示：

```
344       // Next address after AWREADY indicates previous address acceptance
345         always @(posedge M_AXI_ACLK)
346         begin
347           if (M_AXI_ARESETN == 0 || init_txn_pulse == 1'b1)
348             begin
349               axi_awaddr <= 'b0;
350             end
351           else if (M_AXI_AWREADY && axi_awvalid)
352             begin
353               axi_awaddr <= axi_awaddr + burst_size_bytes;
354             end
355           else
356             axi_awaddr <= axi_awaddr;
357           end
```

图 1-29 生成写地址

这里的 burst_size_bytes 是每次 burst 所需的地址，可以看到每进行一次写 burst，写地址就增加一次。但是仅仅这样还不能知道写 burst 时使用的典型时序还是突发传输。这时候就需要看主机的 awvalid 信号了，如图 1-30：

```
328         always @(posedge M_AXI_ACLK)
329         begin
330
331           if (M_AXI_ARESETN == 0 || init_txn_pulse == 1'b1 )
332             begin
333               axi_awvalid <= 1'b0;
334             end
335           // If previously not valid , start next transaction
336           else if (~axi_awvalid && start_single_burst_write)
337             begin
338               axi_awvalid <= 1'b1;
339             end
340           /* Once asserted, VALIDs cannot be deasserted, so axi_awvalid
341           must wait until transaction is accepted */
342           else if (M_AXI_AWREADY && axi_awvalid)
343             begin
344               axi_awvalid <= 1'b0;
345             end
346           else
347             axi_awvalid <= axi_awvalid;
348           end
```

图 1-30 产生 awvalid 信号

这里的 init_txn_pulse 信号用来判断启动传输开始的 INIT_AXI_TXN 信号的边沿，init_txn_pulse=1，代表上升沿。start_single_burst_write 信号用来表示单次写 burst 的开始。因此我们可以知道，在复位以及激励后，awvalid 信号会被拉低，此时如果单次 burst 传输开始，awvalid 信号就会被拉高，然后等待与从机的 awready 信号握手，一旦握手成功后，awvaild 再次被拉低，直到下次 burst 开始 awvalid 才能被拉高。因此，IP 核在进行写事物时，一次 burst 只会传输一个地址，只有当这次写 burst 完成后，才会写下一个地址。

每次传输所包含的 burst 个数由对应 burst 计数器控制，如图 1-31 所示：

```
213   wire [C_TRANSACTIONS_NUM+2 : 0]    burst_size_bytes;
214   //The burst counters are used to track the number of burst transfers of C_M_AXI_BURST_LEN burst length needed to transfer 2^C_MASTER_LENGTH bytes of data.
215   reg [C_NO_BURSTS_REQ : 0]  write_burst_counter;
216   reg [C_NO_BURSTS_REQ : 0]  read_burst_counter;
217   reg    start_single_burst_write;
218   reg    start_single_burst_read;
219   reg    writes_done;
```

图 1-31 transfer 计数器

这两个计数器的计数区间由C_NO_BURSTS_REQ信号控制，该信号的计算公式如图 1-32：

```
162       // function called clogb2 that returns an integer which has the
163       // value of the ceiling of the log base 2.
164       function integer clogb2 (input integer bit_depth);
165       begin
166         for(clogb2=0; bit_depth>0; clogb2=clogb2+1)
167           bit_depth = bit_depth >> 1;
168         end
169       endfunction
170
171     // C_TRANSACTIONS_NUM is the width of the index counter for
172     // number of write or read transaction.
173      localparam integer C_TRANSACTIONS_NUM = clogb2(C_M_AXI_BURST_LEN-1);
174
175     // Burst length for transactions, in C_M_AXI_DATA_WIDTHs.
176     // Non-2^n lengths will eventually cause bursts across 4K address boundaries.
177      localparam integer C_MASTER_LENGTH    = 12;
178     // total number of burst transfers is master length divided by burst length and burst size
179      localparam integer C_NO_BURSTS_REQ = C_MASTER_LENGTH-clogb2((C_M_AXI_BURST_LEN*C_M_AXI_DATA_WIDTH/8)-1);
180     // Example State machine to initialize counter, initialize write transactions,
```

图 1-32 C_NO_BURSTS_REQ 信号计算公式

这里的 clogb2()函数用来计算位宽，C_MASTER_LENGTH 为定值 12，因此我们可以得出，C_NO_BURSTS_REQ = 12-clogb2((突发长度*传输宽度/8)-1)。每次传输包含的 burst 个数为$2^{C_NO_BURSTS_REQ}$，结合突发长度与突发数据宽度就可以计算出一次传输总的数据量。

读数据的地址计算如图 1-33 所示：

```
543       // Next address after ARREADY indicates previous address acceptance
544         always @(posedge M_AXI_ACLK)
545         begin
546           if (M_AXI_ARESETN == 0 || init_txn_pulse == 1'b1)
547             begin
548               axi_araddr <= 'b0;
549             end
550           else if (M_AXI_ARREADY && axi_arvalid)
551             begin
552               axi_araddr <= axi_araddr + burst_size_bytes;
553             end
554           else
555             axi_araddr <= axi_araddr;
556         end
```

图 1-33 读地址计算

这里同样是写一个地址进行一次读 burst，只有上一个 burst 完成了才会写下一个地址，推导的过程同理，这里不再为大家赘述，arvalid 信号的产生过程如图 1-34 所示：

```
529         always @(posedge M_AXI_ACLK)
530         begin
531
532           if (M_AXI_ARESETN == 0 || init_txn_pulse == 1'b1 )
533             begin
534               axi_arvalid <= 1'b0;
535             end
536           // If previously not valid , start next transaction
537           else if (~axi_arvalid && start_single_burst_read)
538             begin
539               axi_arvalid <= 1'b1;
540             end
541           else if (M_AXI_ARREADY && axi_arvalid)
542             begin
543               axi_arvalid <= 1'b0;
544             end
545           else
546             axi_arvalid <= axi_arvalid;
547         end
```

图 1-34 生成 arvalid 信号

读写数据比较逻辑如图 1-35 所示：

```
653     //Register and hold any data mismatches, or read/write interface errors
654
655       always @(posedge M_AXI_ACLK)
656       begin
657         if (M_AXI_ARESETN == 0 || init_txn_pulse == 1'b1)
658           begin
659             error_reg <= 1'b0;
660           end
661         else if (read_mismatch || write_resp_error || read_resp_error)
662           begin
663             error_reg <= 1'b1;
664           end
665         else
666           error_reg <= error_reg;
667       end
```

图 1-35 读写数据比较

这里 read_mismatch、write_resp_error、read_resp_error 信号为 1 时分别表示读写数据不匹配、写响应错误、读响应错误。当这些信号有一个为 1 时，都会让 error_reg 信号为 1。而我们前面介绍过的 ERROR 信号通过 assign 与 error_reg 信号绑定，来反馈数据读写是否错误。

简单了解完 IP 源码后，将 IP 重新封装，关闭 IP 管理工程，准备开始搭建硬件逻辑系统。

1.4.2 硬件逻辑系统设计

首先创建一个新的 Vivado 工程，将刚刚创建的 IP 核导入到工程中，随后新建一个模块设计，开始搭建硬件逻辑系统。

1.4.2.1 IP 添加与配置

搭建本次设计硬件逻辑系统所需的 IP 核如图 1-36 所示：

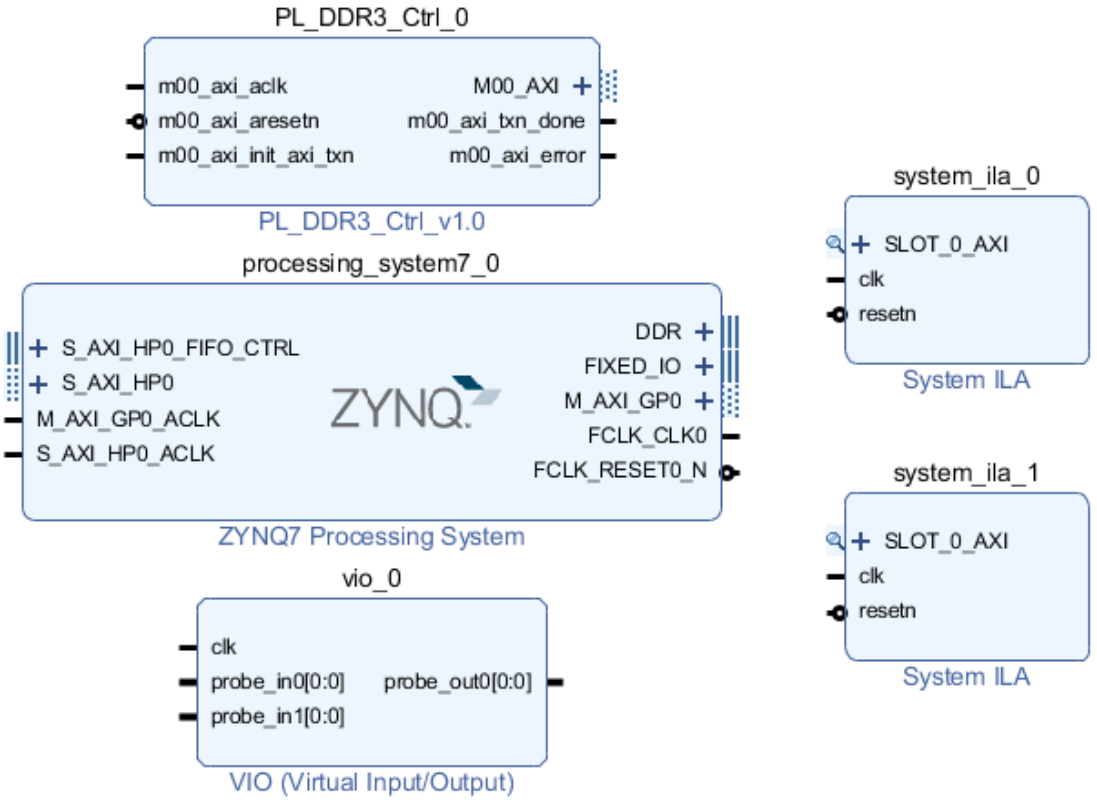


图 1-36 所需添加 IP

ZYNQ 核以及自定义 IP 核自不必多说，是必须添加的 IP。要想让自定义 IP 核工作，我们需要输出一个高电平信号给其 m00_axi_init_axi_txn 引脚(该引脚连接的内部信号 INIT_AXI_TXN)。因此设计里使用 VIO 核，在提供输出信号的同时还能接收自定义 IP 核输出的两个反馈信号 。

system ila 在功能上等同于 ila，但是在相较于 ila，它能够自动对部分总线协议进行分析并分组，例如本次设计所涉及的 AXI 总线协议，system 能够帮我们分析出各个操作，并按照五个通道将信号进行分组归纳。

由于自定义 IP 核用的 AXI4 协议接口，而 HP 接口为 AXI3 协议接口，因此在后续软件的自动连接时，软件会自动为我们添加 Interconnect 核进行协议转换。为此，这里我们需要添加两个 system ila，其中一个用于监测自定义 IP 的 AXI4 接口时序，另一个用来检测 HP 接口时序，通过对比两个接口的时序就可以得出 Interconnect 核在互联时的作用。

添加完成后对每个 IP 核逐一配置：

(一)ZYNQ 核

设计中自定义 IP 核会向 DDR 读写一定量的数据，根据物理接口中的内容，PL 对 DDR 的访问通常是通过 HP 接口，且 PL 作为从机。因此，我们需要在 PS-PL Configuration 中使能 HP 从接口，可以看到 HP 接口默认为 64 位数据位宽，如图 1-37 所示：

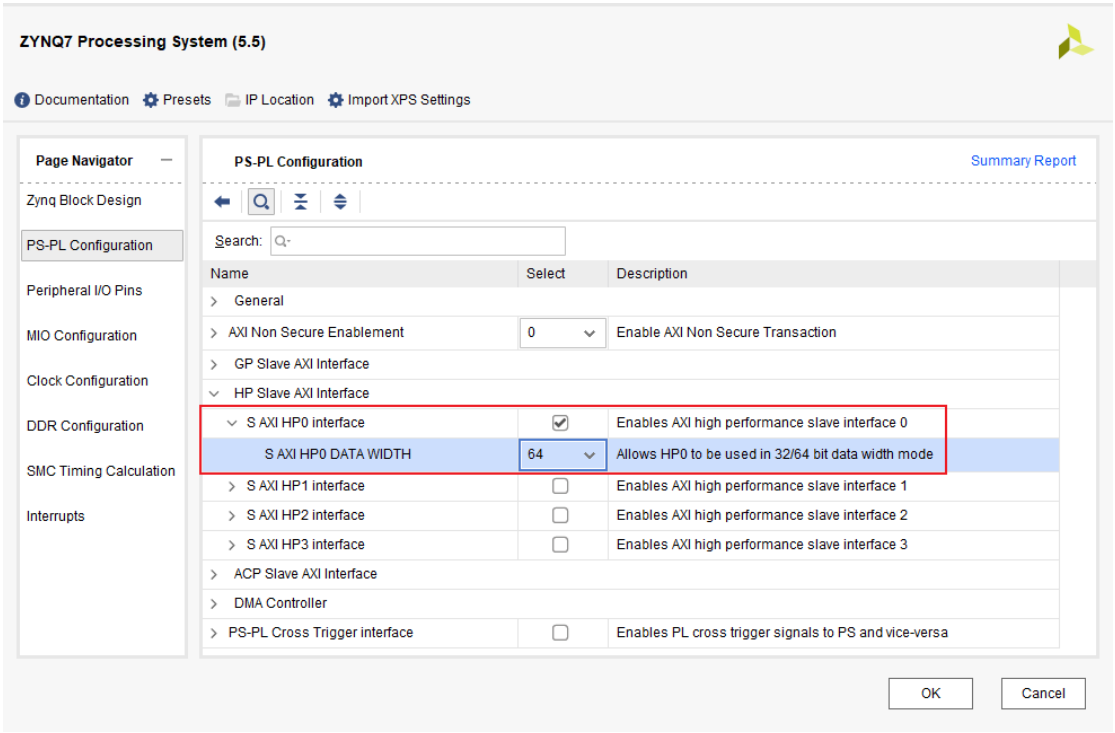


图 1-37　使能 HP 从接口

随后再将 DDR 的型号设置为 MT41K128M16 JT-125 便完成了 DDR 的配置。

(二)配置自定义 IP

自定义 AXI4 主接口 IP 核的配置如图 1-38 所示：

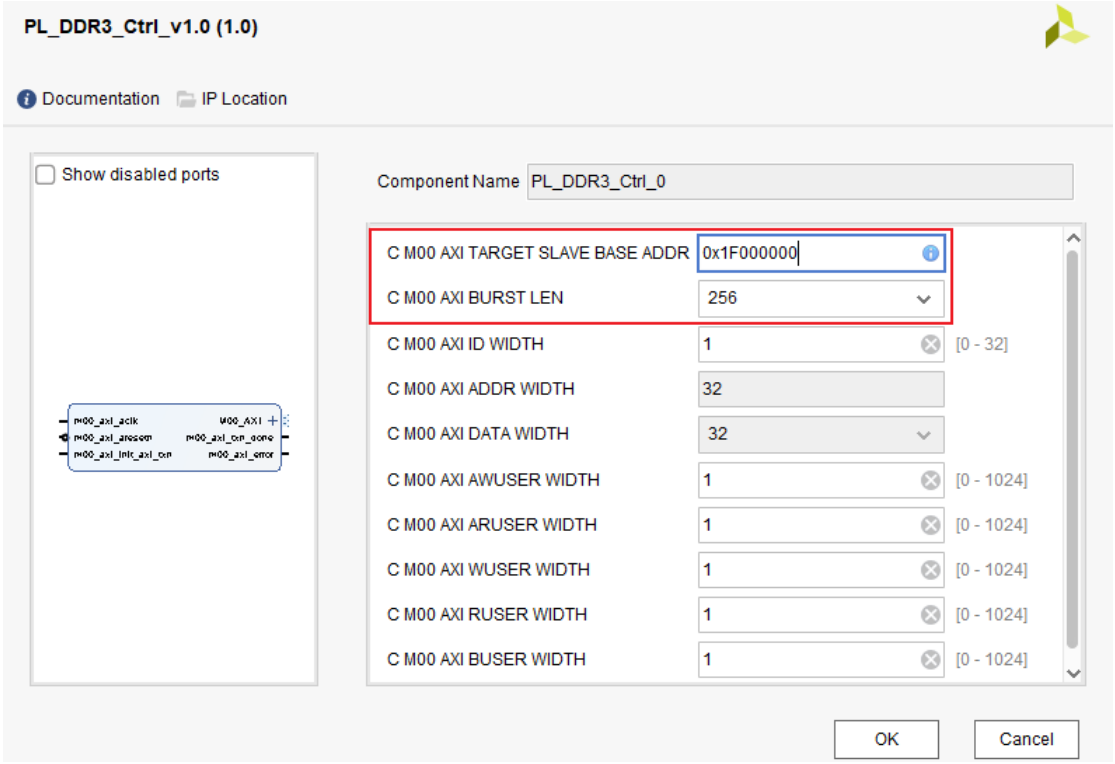


图 1-38 自定义 AXI4 主接口 IP 配置

首先是数据写入 DDR 的目标地址，该项默认为 0x40000000。但是 ACZ702 开发板上使用的是两片 128M 的 16 位宽的 DDR，通过计算我们可以知道，DDR 的最高地址为 128*2*16*1024*1024/8=536870912=0x20000000。为了不产生地址溢出以及地址跨 4K 边界的现象，这里我们可以设置一个 4K 对齐的地址 0x1F000000。

AXI4 接口在 INCR 类型的突发下，支持最高 256 长度的突发，因此这里我们设置为 256。结合总线数据位宽为 32，以及在分析 IP 核源码时所得到的传输次数计算公式我们可以计算出 C_NO_BURSTS_REQ = 12- clogb2((256*32/8)-1)=12-10=2，因此一次传输包含2^2=4 个 burst。进而我们可以计算出发送数据的总量为 256*32*4/8=4KByte，这些数据刚好占用 4K 地址，由于我们地址是 4K 对齐，因此不会造成跨 4K。同时我们可以计算出，在该配置下，AXI4 接口的地址增量为 256*32/8 = 1024 = 0x400，因此，传输的四个地址依次为 0x1f000000、0x1f000400、0x1f000800、0x1f000c00。

(三)VIO 核

VIO 用来接收自定义 IP 核的的反馈信号 TXN_DONE 和 ERROR，产生自定义 IP 核的激励信号，所以需要两个输入端口和一个输出端口，相关配置如图

1-39 所示，其余项保持默认即可。

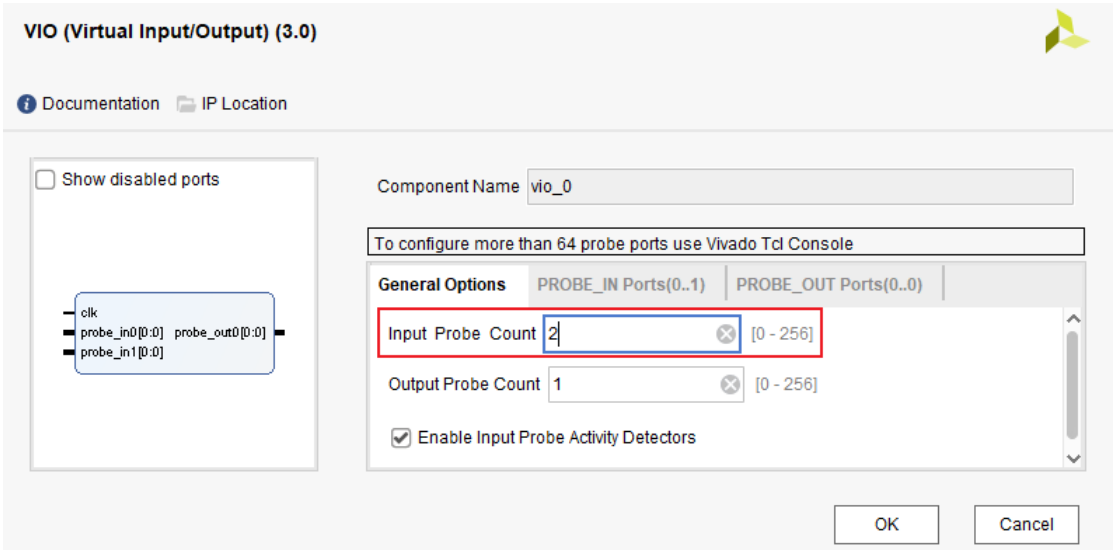


图 1-39 配置 VIO

(四)System ILA

首先是 System ila 的常规设置项，前面我们已经计算过，设计传输的是 1024 个总线宽度的数据。数据是先写入再读出，且从写入到读出之间必然会有时间间隔，因此，System ILA 的深度要大于 2048，为了保险起见，这里我们将两个的深度都设置为 8192，如图 1-40 所示：

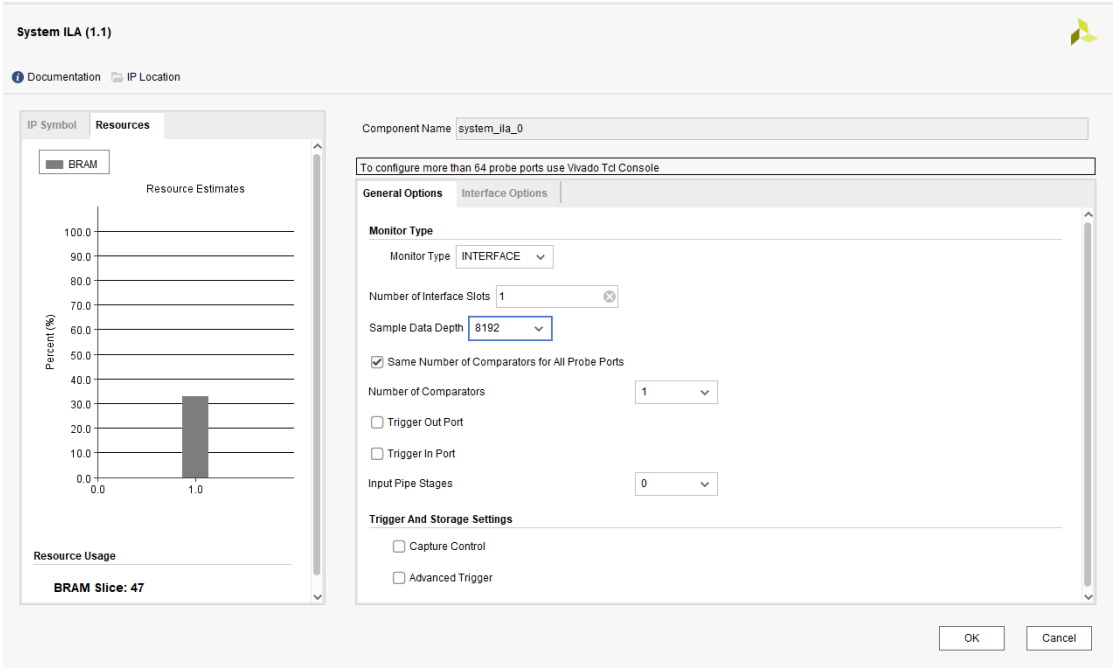


图 1-40 常规项设置

接下来是接口配置项，在这项中我们需要修改的是读写事物的 outstanding

能力。在前面分析自定义 AXI4 主接口的 IP 核时，我们已经知道了其不使用 outstanding 传输，因此用于监测 AXI4 主接口的 system ila 不需要配置该项。

AXI4 接口的数据经过 Interconnect 核转换后会传输给 HP 接口，最终写入到 DDR，Interconnect 核是支持突发传输的，HP 接口自带用于读写缓冲的 FIFO，因此也支持突发传输。所以数据在从 Interconnect 核接口传输到 HP 接口的过程中，很可能会使用突发传输。AXI4 接口只有在上一个 burst 传输完成后才开始下一个 burst，因此对于 Interconnect 来说，要做的就是将这一个 burst 的传输拆分并 outstanding 给 HP 接口，因此我们可以计算出所需突发能力为：

256*32/(16*64)=8

所以，监测 HP 接口的 system ila（这里笔者用的 system ila1），需要将其读写 outstanding 能力设置为 8，如图 1-41 所示：

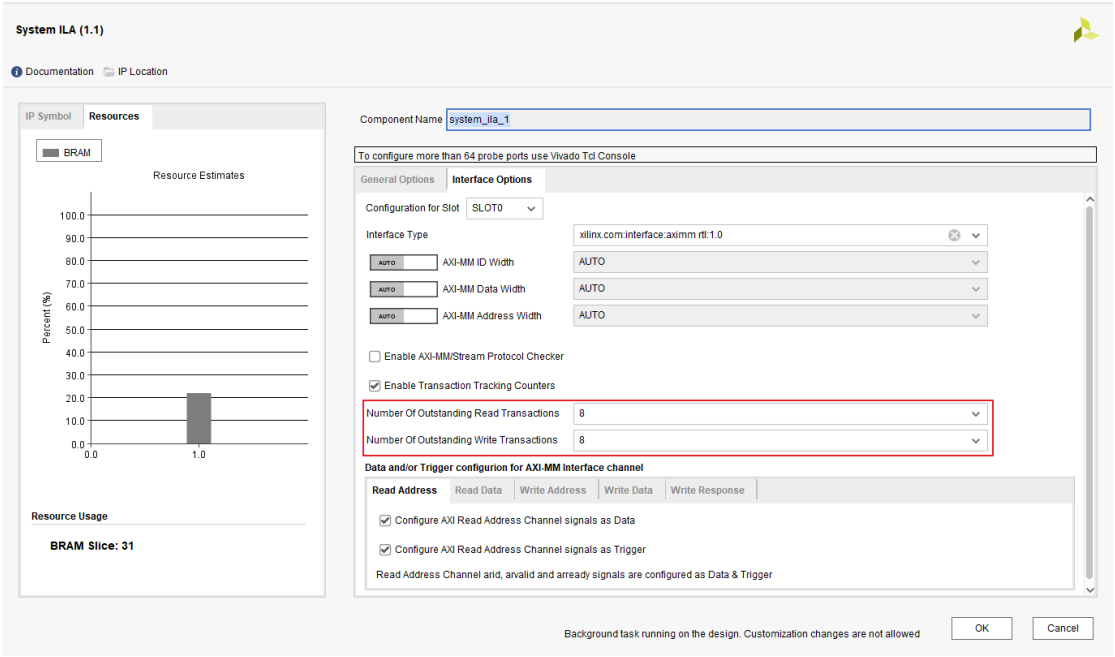


图 1-41　设置 outstanding 能力

这里的 Interface Type 用来设置接口类型，默认为 aximm.rtl 也就是 AXI 总线接口，我们保持默认即可。

1.4.2.2 端口连接

ZYNQ 核默认是开启了 GP 主接口的，虽然设计中我们不会用到，但是为了不让系统报错，我们需要为其提供时钟，因此手动连接 M_AXI_GP0_ACLK 接口和 FCLK_CLK0。随后依次点击“Run Block Automation”“Run Connection Automation”，软件会自动为我们连接好绝大部分接口，如图 1-42 所示：

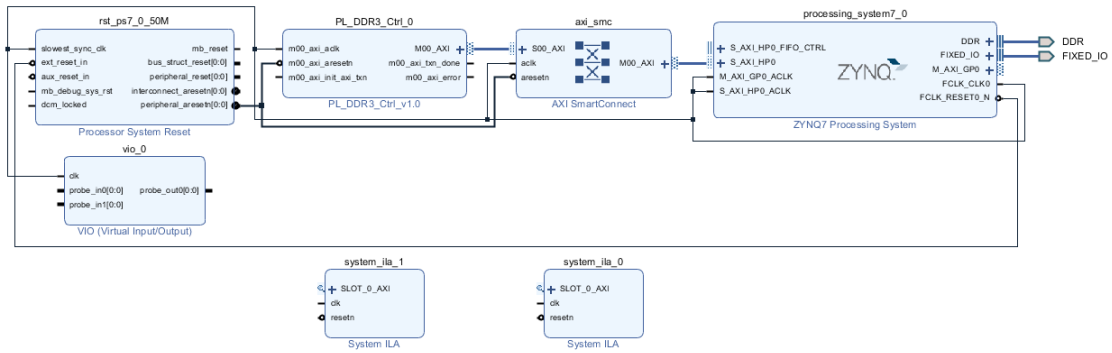


图 1-42 运行模块自动化与自动化连接

可以看到，软件自动为我们添加了一个 AXI SmartConnect 核来连接自定义 IP 核的 AXI4 主接口和 HP 接口。

接下来手动将 VIO 的输入输出接口连接到自定义 IP 核，以方便我们产生自定义 IP 核的激励信号并观测其反馈信号的变化。如图 1-43 所示：

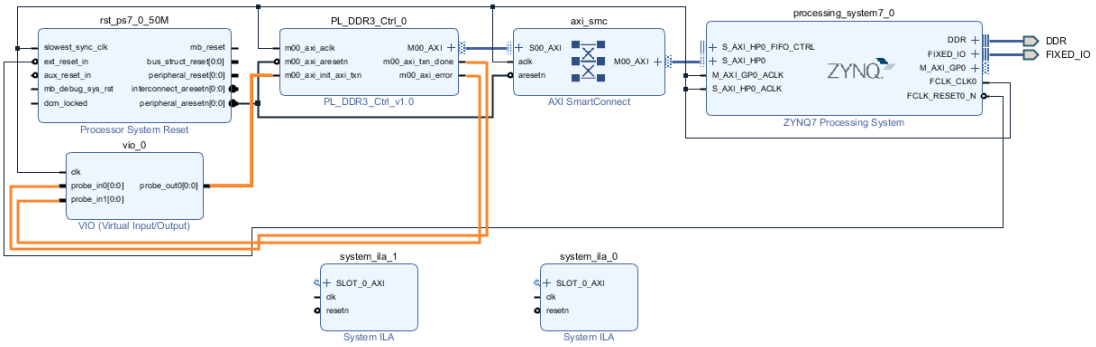


图 1-43 连接 VIO

随后连接 system ila 的探针，将读写 outstanding 能力设置为 8 的 system ila 连接到 HP 接口（这里笔者的为 system ila1），另一个连接到自定义 IP 核的 AXI4 主接口，也就是 M00_AXI 接口，如图 1-44 所示：

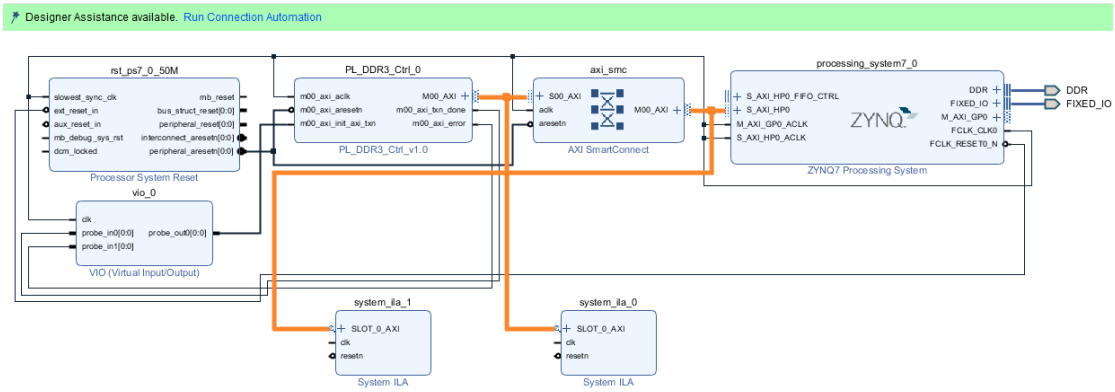


图 1-44 连接探针

随后点击弹出的“Run Connection Automation”，软件会自动为我们连接好

两个 system ila 的时钟与复位。至此，硬件系统搭建完成，如图 1-45 所示：

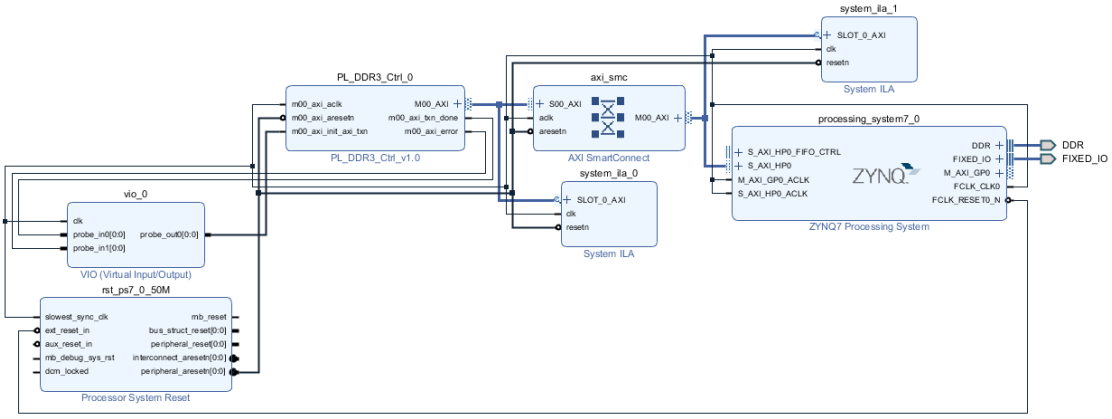


图 1-45　硬件逻辑系统

由于设计使用的纯 PS 资源，所以不需要进行引脚约束步骤，将设计生成输出并封装后，点击 bit 生成比特流，将比特流与硬件描述文件一起导出到 SDK 就可以开始 CPU 软件程序设计了。

1.4.3 CPU 软件程序设计

软件程序设计的第一步当然还是创建一个空模板的 SDK 工程，完成后，在 src 文件夹下新建一个 main.c 函数，在其中添加以下代码：

```
#include "xil_cache.h"
#include "string.h"

int main(void)
{
    memset(0x1F000000,0xFF,1024*4);
    Xil_DCacheDisable();
    while(1);
    return 0;
}
```

这里代码相较简单，首先使用 memset 将接下来会被我们读写的地址内的数据全部改为 0xFF，以方便调试时观察变化；随后禁用掉 cache，让数据直接写入 DDR；最后使用 while（1）让软件进行死循环，以方便我们随时能使用 VIO 激活传输，然后通过调试查看寄存器数值。

关于 cache 这里有一点需要说明的是，在 AXI 协议中，有专门用来控制数据是存储在 cache 还是 buffer 或 DDR 中的信号，也就是 AxCACHE 信号。但是，对于 ZYNQ-7000 系列器件的 DDRI（DDR 接口）而言，这个信号是无意义的，这一点在 UG585 的 10.2.3 章节中就有明确提出，如图 1-46 所示：

10.2.3 AXI Feature Support and Limitations

This list shows supported and unsupported features for the AXI ports into the DDRI:

- Fixed burst type is not supported. Note that the behavior is unknown if this transfer type is received at one of the AXI ports.
- Byte, half-word and word sub-width commands are supported.
- EXCL accesses are only supported on a single DDR port, ie., there is no support for EXCL accesses across DDR ports.
- AWPROT/ARPROT[1] bit is used for trust zone support, AWPROT/ARPROT[0], and AWPROT/ARPROT[2] bits are ignored and do not have any effect.
- ARCACHE[3:0]/AWCACHE[3:0] (cache support) are ignored, and do not have any effect.
- Sparse AXI write transfers (random strobes asserted/de-asserted for any data beat) are supported.
- Unaligned transfers are supported.

图 1-46 DDRI 中 AXI 接口的支持与限制

可以看到，除了 AxCACHE 信号在 DDRI 中不支持外，FIXED 传输类型也是不支持的，也就是说，数据不能以 FIXED 的传输类型写入 DDR。

添加完代码后保存设计，确认无误后便可以烧录验证了。

1.4.4 板级验证

1.4.4.1 系统所需硬件

1. ACZ702 开发板一个
2. 电源线一根（可选）
3. Type-c 下载线一根

1.4.4.2 硬件连接

本次系统设计硬件连接如图 1-47 所示：

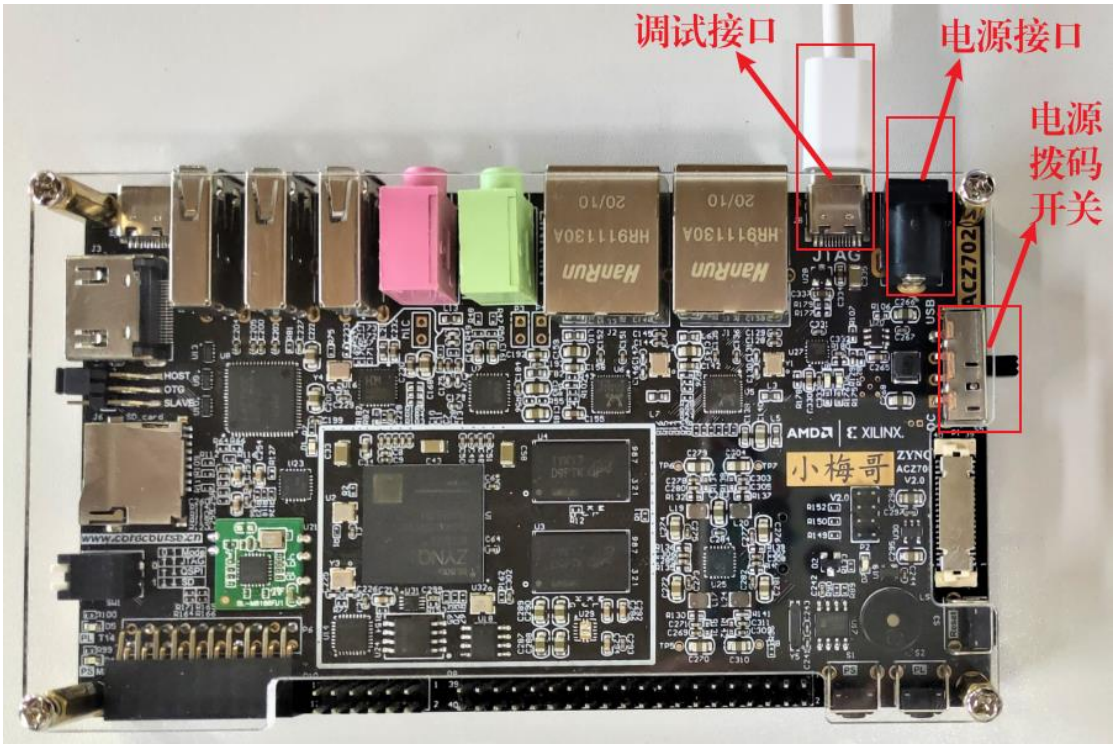


图 1-47 硬件连接

设计对供电要求较低，因此可以仅使用 Type-c 供电，连接好硬件后，将电源拨码开关拨到对应供电侧就可以开始烧录调试了。

1.4.4.3 调试验证

首先新建一个 Debug Configurations，由于我们需要查看寄存器值，所以需要创建一个 System Debugger。并确保图 1-48 中的配置项：

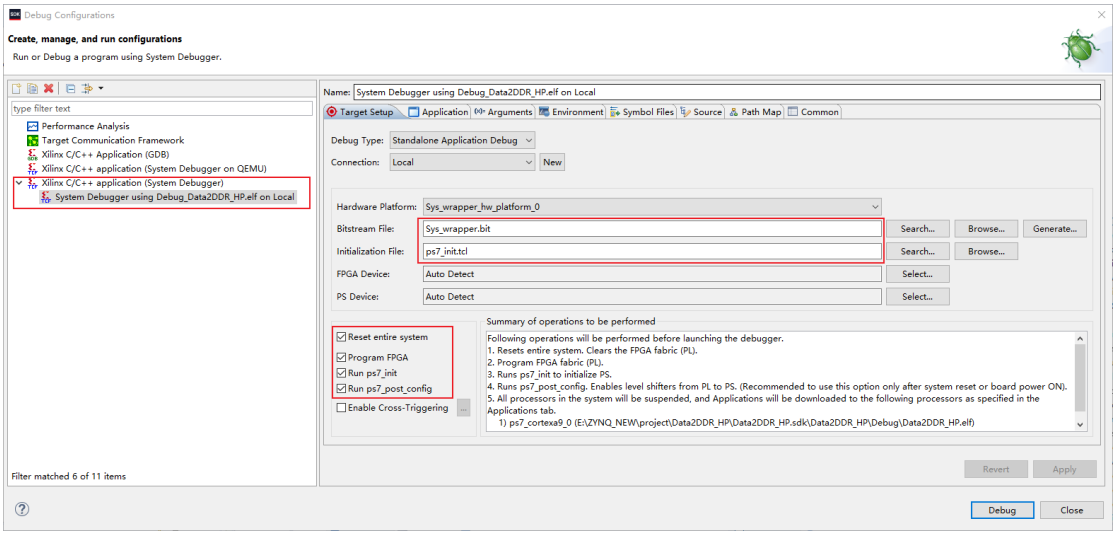


图 1-48 创建 Debug Configurations

配置完成后，点击 Debug 等待程序烧录进开发板中，在烧录过程中软件会提示我们是否打开 Debug 界面，点击 Yes 即可。如果想要返回原来界面可以点击右上角的表格+C 的图标，如图 1-49 所示：

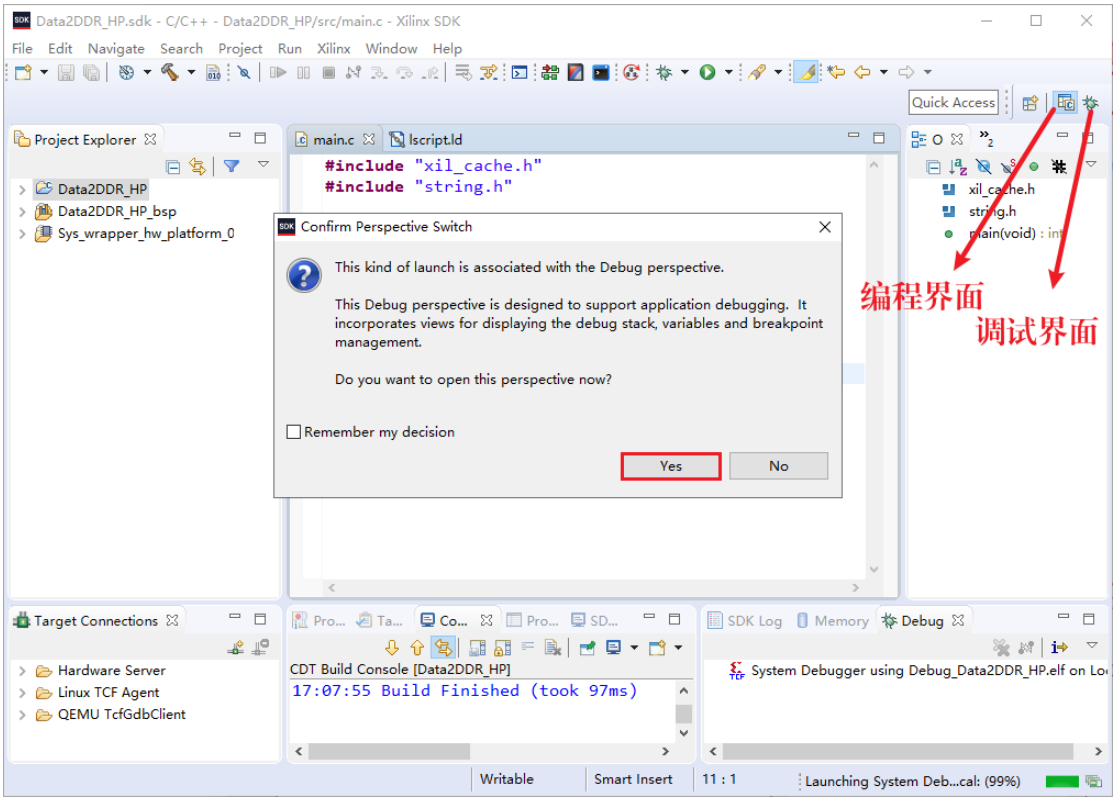


图 1-49 打开调试界面

程序成功烧写进开发板后，我们首先在 Debug 页面中找到 Memory 窗口，在其中添加我们要查看的寄存器地址 0x1F000000，如图 1-50 所示：

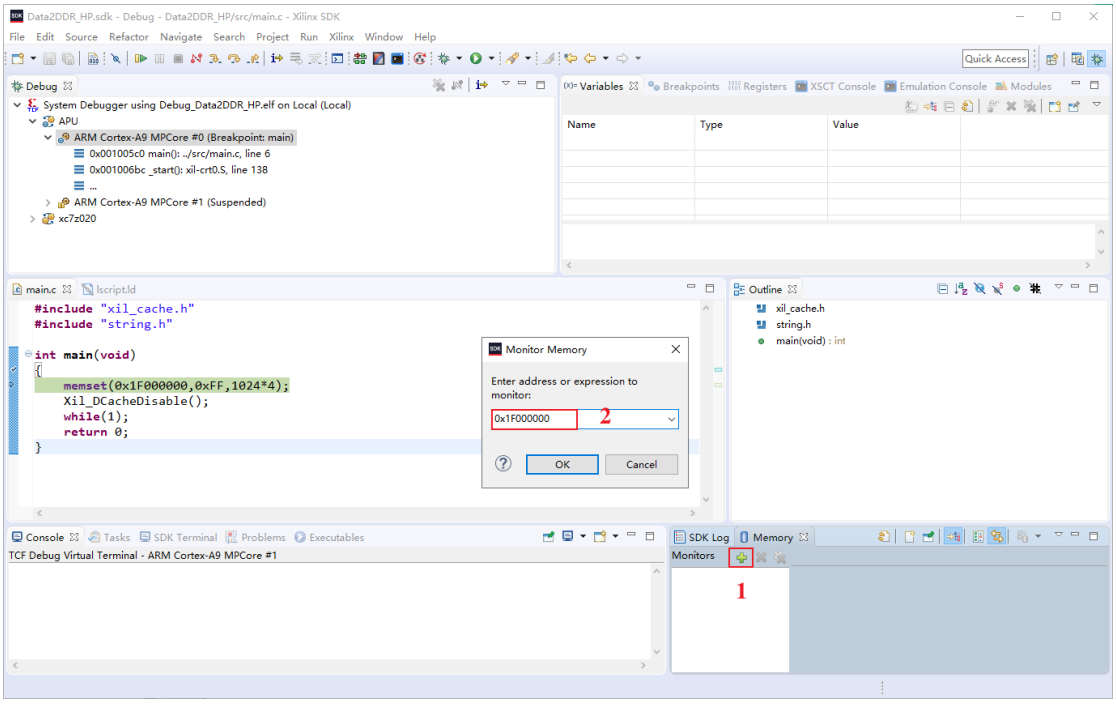


图 1-50 添加待观测寄存器地址

此时右下角就会出现寄存器对应的值，如图 1-51 所示：

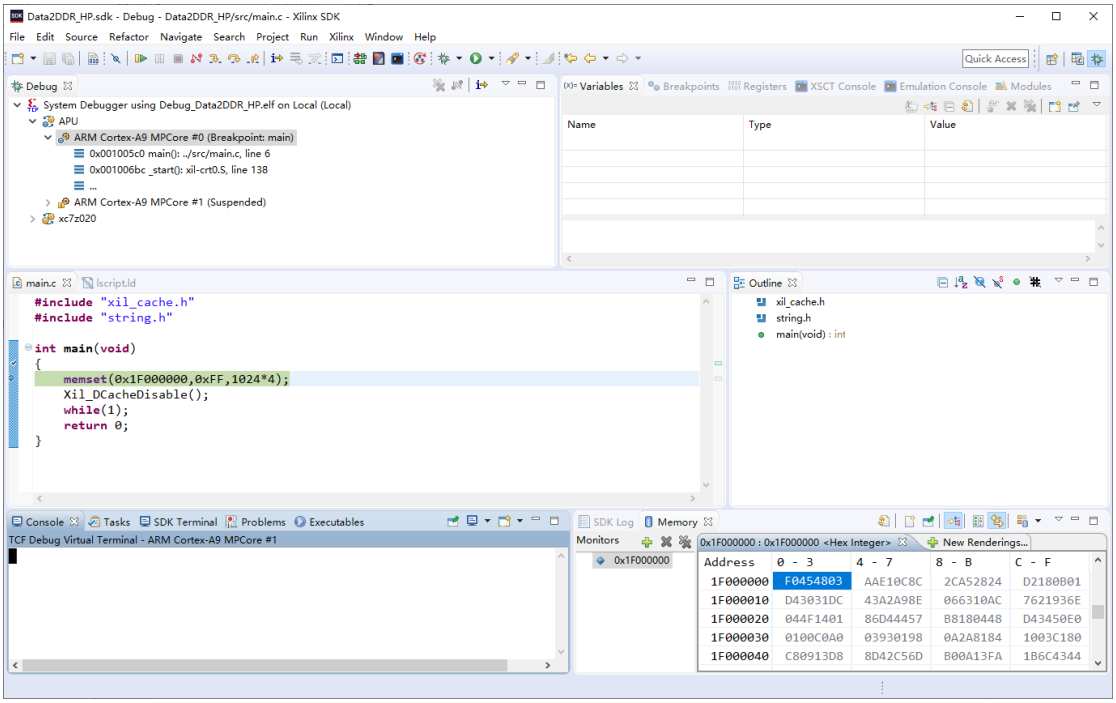


图 1-51 使用 Memory 功能查看寄存器值

接下来我们点击两次单步执行，将从 0x1F000000 地址开始的 4K 个寄存器的值全变为 0xFF，并禁用 cache，如图 1-52 所示：

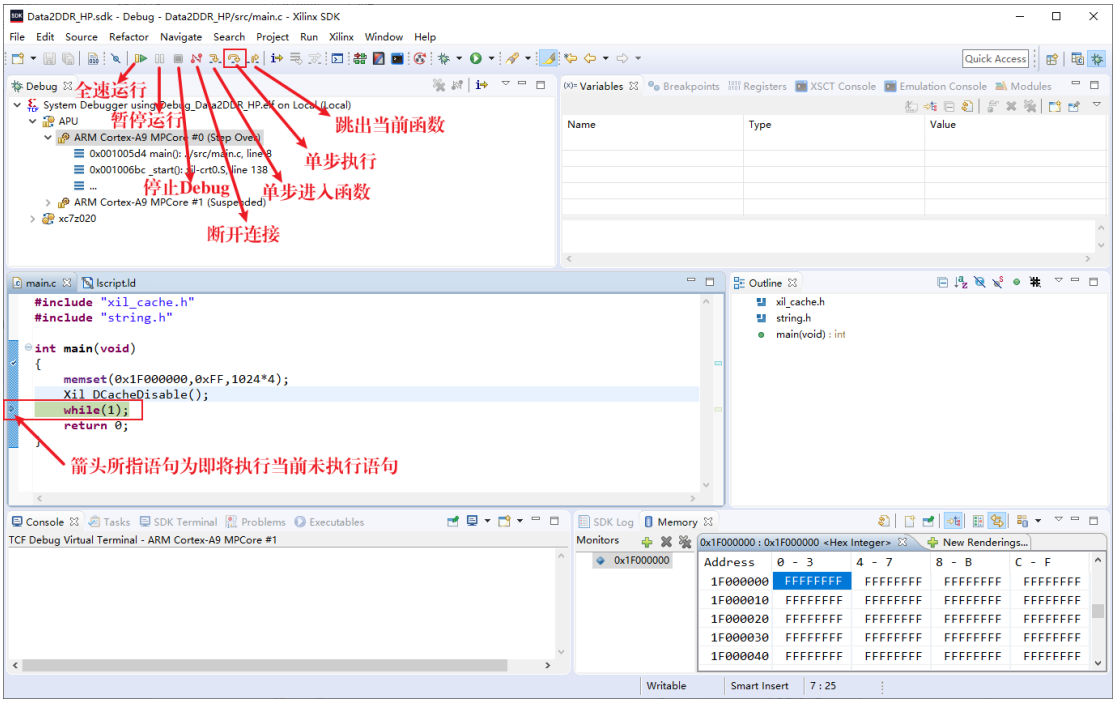


图 1-52 单步执行

接下来返回 Vivado 窗口，打开硬件管理，连接上开发板，此时软件会检测到我们添加的 VIO 和 system ila，并为我们打开相关窗口。

首先是两个 system ila，为了方便检测波形，我们可以设置它们触发条件为 awvalid 信号的上升沿，然后点击三角符号等待触发，如图 1-53 所示：

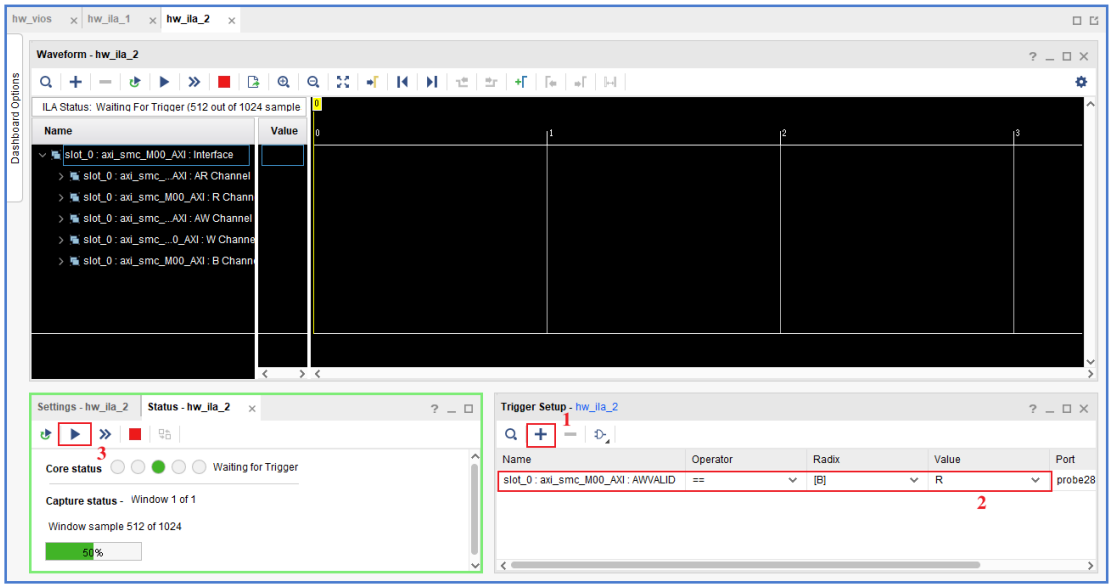


图 1-53 添加触发条件并开始等待触发

然后打开 VIO 界面，探针需要我们手动添加，如图 1-54 所示：

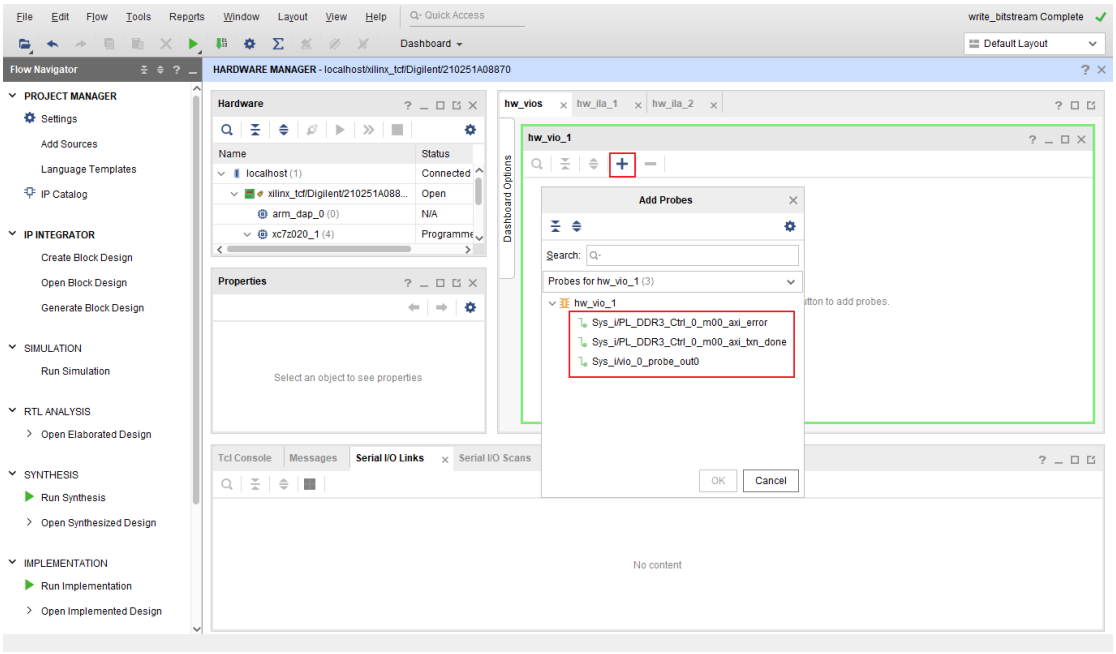


图 1-54　添加 VIO 探针

为了方便输出信号的控制，我们可以将 VIO 的输出设置为高电平输出按钮，这样只要我们点击就会输出 1，松开就会输出 0，如图 1-55 所示：

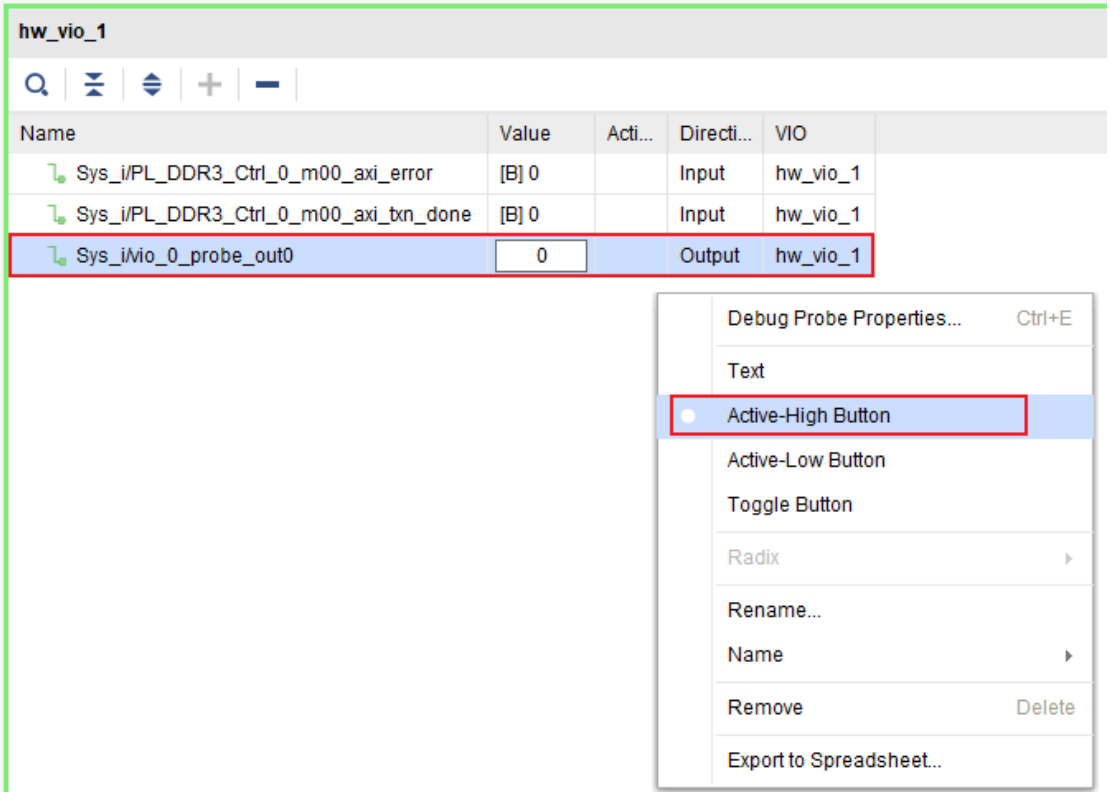


图 1-55　设置探针形式

接着单击按钮，自定义 IP 核开始工作，AXI 总线开始传输数据，等待片刻，

VIO 中的输入接口便会发生变化，如图 1-56 所示：

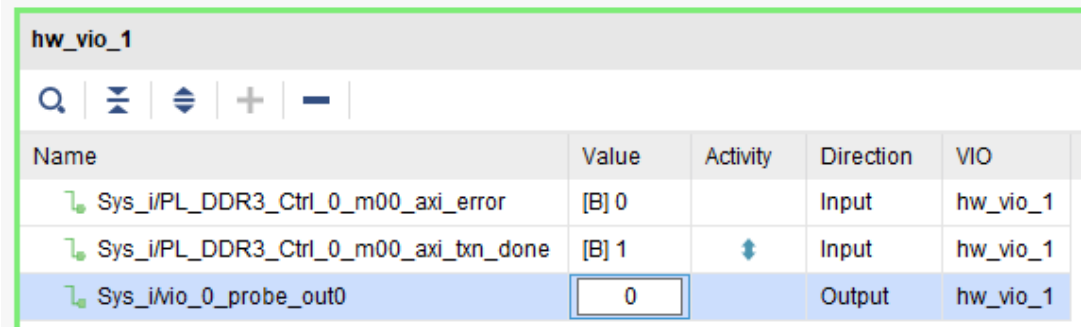


图 1-56 检测自定义 IP 反馈信号

txn_done 信号值为 1 说明写事物完成，error 信号为 0 说明读写数据一致，且传输没有出错。此时我们返回 SDK 的 Debug 界面，再点击暂停 Debug，可以看到，从 0x1f000000 地址开始一直到 0x1f000fff，寄存器内的值从 0x00000001 一直累加到 0x00000400，如图 1-57 所示：

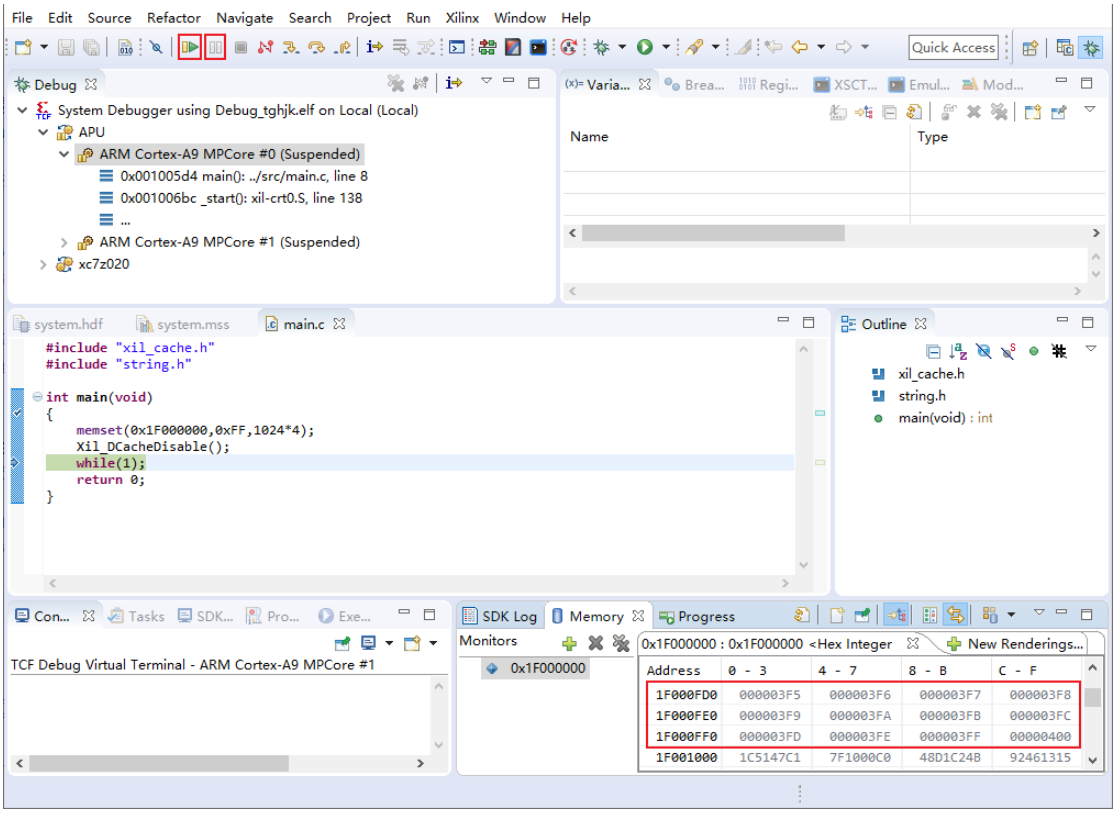


图 1-57 相关寄存器值

至此说明突发读写成功，接下来返回到 Vivado 的 ila 界面，此时两个 ila 已经触发成功。

1.4.4.3.1 AXI4 接口时序

笔者在配置时，ila_1 抓取的是自定 IP 核的 AXI4 主接口时序，ila_2 抓取的是 HP 接口时序，这里我们首先观察 AXI 接口时序，如图 1-58 所示：

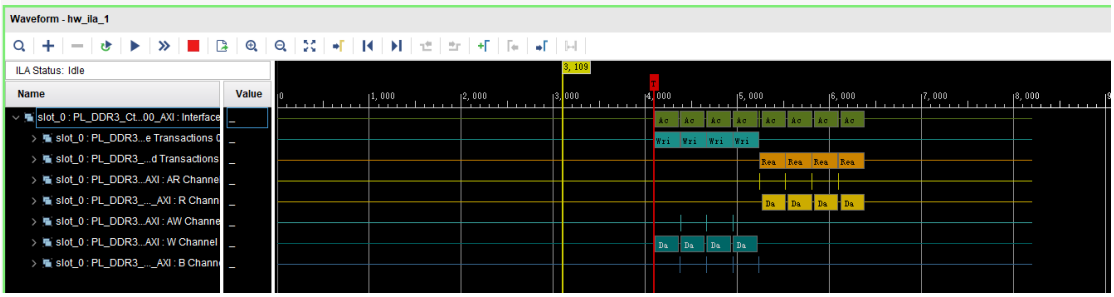


图 1-58 AXI4 接口时序

可以看到 AXI4 主接口正如我们分析的，共有四次写突发和四次读突发。接下来我们展开各个通道的信号，验证它们的关系，首先是 AXI4 主接口的写地址通道，其时序如图 1-59：

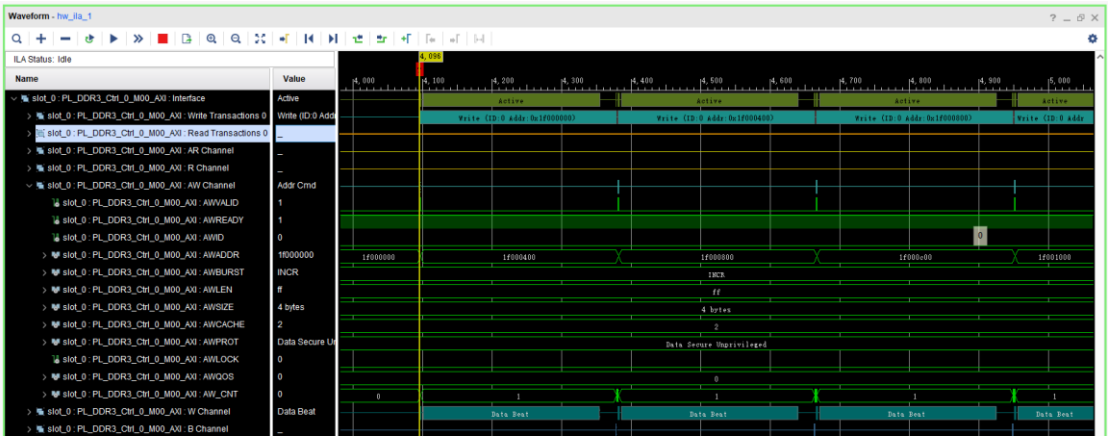


图 1-59 AXI4 主接口写地址通道时序

可以看到，AWLEN 信号为 0xff，也就是突发长度为 256；AWSIZE 为 4Byte，也就是突发大小为 32bit 与总线一致；AWBURST 为 INCR，也就是传输类型为 INCR，因此地址每次的增量为 256*4 = 1024 = 0x400；当 AWREADY 信号与 AWVALID 信号握手成功时，总线传输一次地址，一共传输四次，依次为 0x1f000000、0x1f000400、0x1f000800、0x1f000c00。可以看到，这些数据与我们前面的推论全部一致。

接下来是写数据通道，其时序如图 1-60 所示：

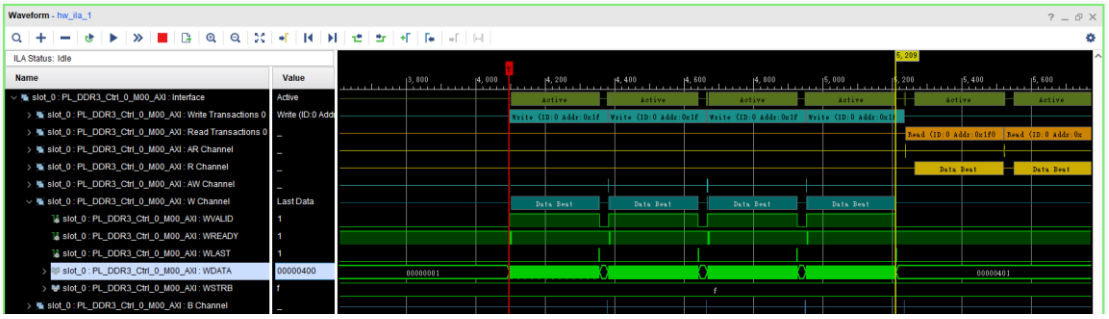


图 1-60　写数据通道时序

可以看到 WRSTB 为 0xf，代表数据低 32 位有效，也就是都有效。在总线时钟的每个上升沿，如果 WVALID 信号和 WREADY 信号握手成功便发送一个新数据（这里 ila 并没有抓取全局时钟信号），数据从 0x00000001 开始一直加到 0x00000400。也就是在发送每次 burst 的最后一个数据的期间，WLAST 信号也会被拉高。

然后是写应答通道，其时序如图 1-61 所示：

图 1-61　写响应通道时序

可以看到，这里 BRESP 全程都为 OKAY，好像全程都在响应 OKAY，但是在信号意义上并不是这样。BRESP 的默认值为 2'b00，当 BVALID 和 BREADY 信号握手成功时，从机会给出本次 burst 的结果，只有这时候的 BRESP 才算是真正意义上的响应，此时主机才会根据 BRESP 信号的值判断传输状态。

写地址、写数据、写响应一起组成了写事务通道，各个通道间信号的时序关系如图 1-62 所示：

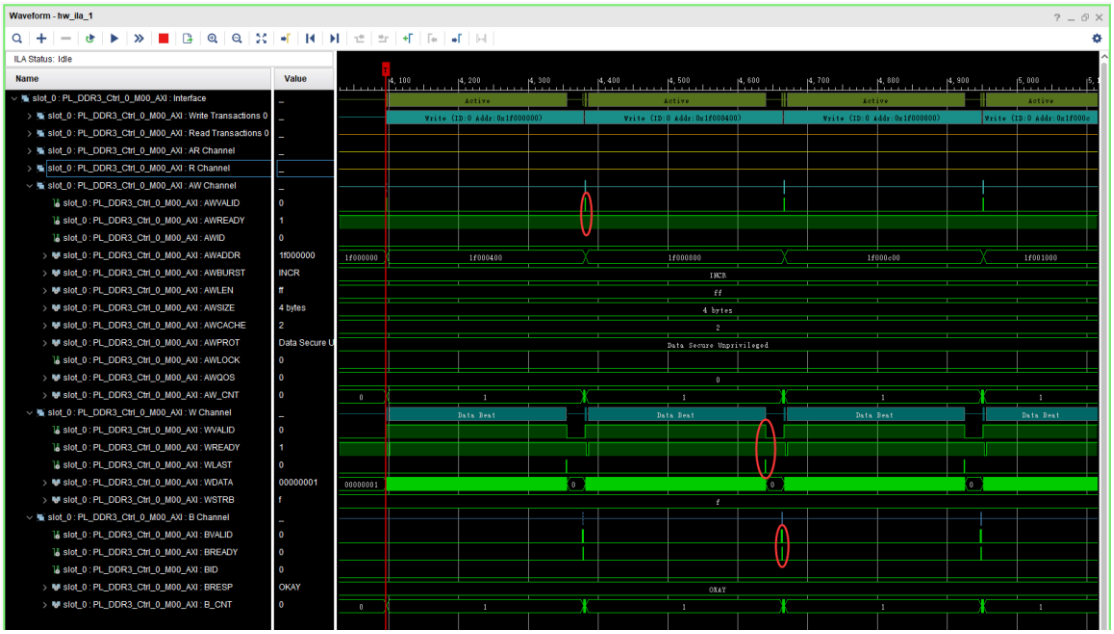


图 1-62 AXI4 主接口写事物时序

可以看到在这个时序中，写地址与写数据同时被写入，符合我们在介绍 AXI4 接口时所提到的，写事务中写地址与写数据在时序上没有先后关系。除此之外，三个通道间的握手对信号也满足写事务握手对依赖关系，即：写通道握手必须在写地址通道、写数据通道握手成功且 WLAST 拉高之后才能拉高。

接着是读地址通道，其时序如图 1-63 所示：

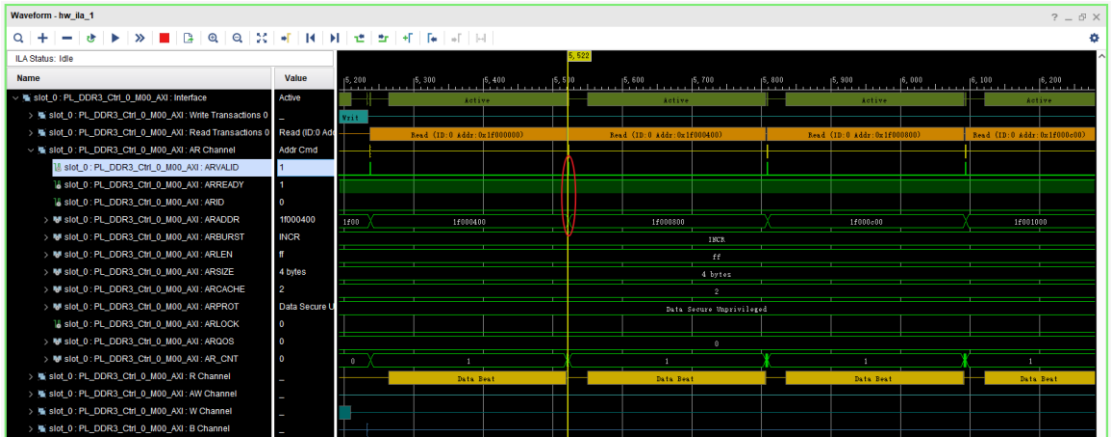


图 1-63 读地址通道时序

可以看到，读通道传输类型为 INCR，传输长度为 256，传输大小为 32 位，每当通道握手成功时，写入即将突发的地址，依次为 0x1f000000、0x1f000400、0x1f000800、0x1f000c00，与写入的地址一致。

最后是读数据通道，其时序如图 1-64 和图 1-65 所示：

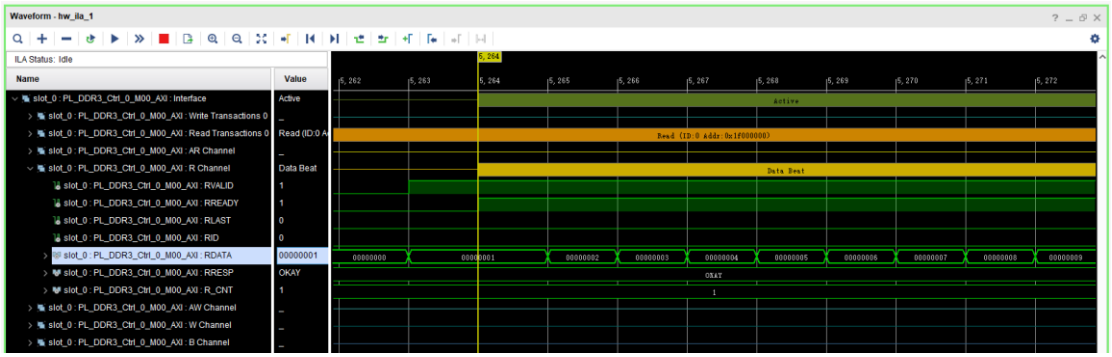


图 1-64　读数据通道（传输开始）

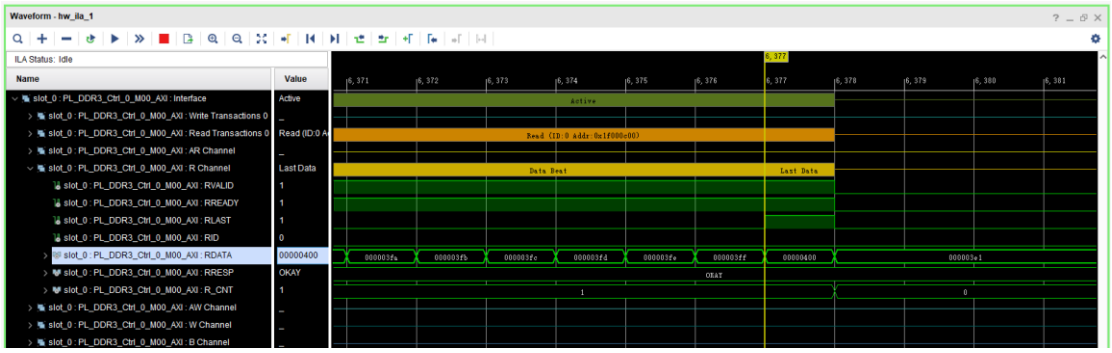


图 1-65　读数据通道（传输结束）

当通道握手成功时，开始读取数据，数据从 0x00000001 开始一直读到 0x00000400 结束。传输一共分为四次 burst，因此在读每次 burst 的最后一个数据时，会拉高 RLAST。RRESP 信号虽然与写事物时 WRESP 一样，全程为 OKAY（2'b00），但是意义却不同。读通道每读一个数据都会响应一次，因此，每次读数据通道握手成功读取数据时，RRESP 信号所代表的才是响应。

读地址和读数据通道组成的读事务时序如图 1-66 所示：

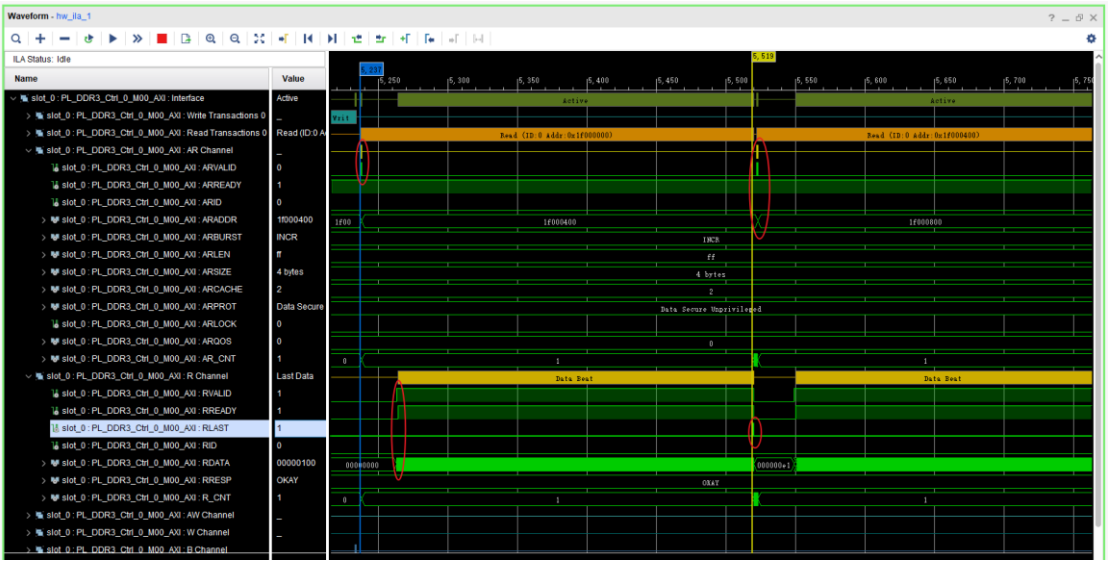


图 1-66 读事务时序

不同于写事务时地址与数据同时发送，读事务时，主机先发送地址和控制指令，从机必须在接收到指令后才能发送读取的数据。在传输每个 burst 的最后一个数据期间，从机拉高 RLAST，随后等待主机发送下一个地址。

至此，我们可以看到，AXI4 接口的整体时序与我们原理部分一致，无论是单个通道还是读写事务，其内部的信号关系以及依赖关系都与我们所讲的一致。

1.4.4.3.2 HP（AXI3）接口时序

接下来我们再来简单看看 HP（AXI3）接口，其时序如图 1-67 所示：

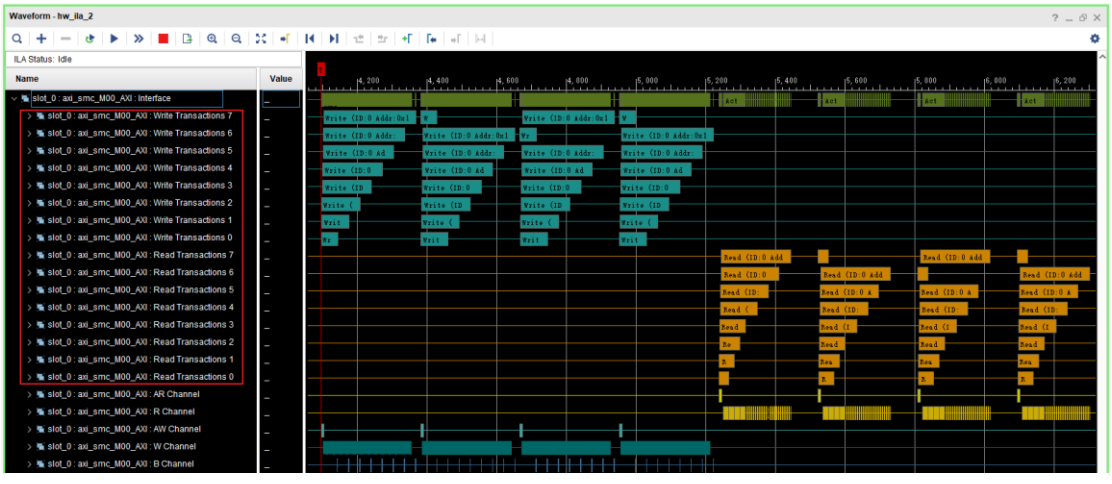


图 1-67 HP 接口时序

可以看到，原来在 AXI4 接口中只有读写各 1 个 transactions，现在却是读写各 8 个 transactions。也就是将原来的每个 burst 拆分为了 8 个更小的 burst，这么做是因为 AXI4 接口中设置的突发长度为 256，突发大小为 32，而 HP 接口

（AXI3 协议）中突发长度最高为 16，数据位宽我们保持的默认值 64bit，因此每个 burst 必须拆分为 8 次小 burst 才能适配 HP 接口。

接下来，我们展开信号，简单观察下 AXI3 总线读写事务的时序。首先是写事务，图 1-68 为前 8 个写 burst 的部分时序：

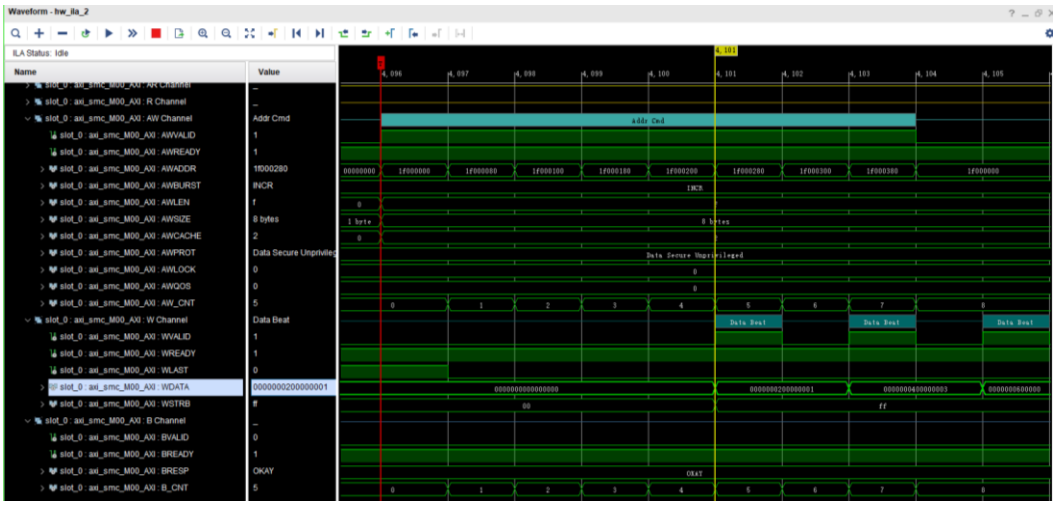


图 1-68 HP 接口写事务部分时序

可以看到，写事务传输类型为 INCR，传输长度为 0xf，也就是 16，传输大小为 8Bytes。主机首先连续传输 8 个待写入的地址，随后在传输地址的过程中开始传输数据。因此，我们可以判断出此时为 outstanding 传输，传输地址个数为 8，因此 outstanding 能力为 8。AXI3 总线宽度为 64，从 AXI4 接口接收的每两个连续数据会被拼接为一个 64 位的数据输出，也就是图中的 0x0000000200000001 等。WSTRB 信号为 0xff，因此写入的数据全部有效。

在传输每个 burst 的最后一个数据期间，WLAST 信号同样会被拉高，此时 BRESP 信号的值为响应信号，如图 1-69 所示：

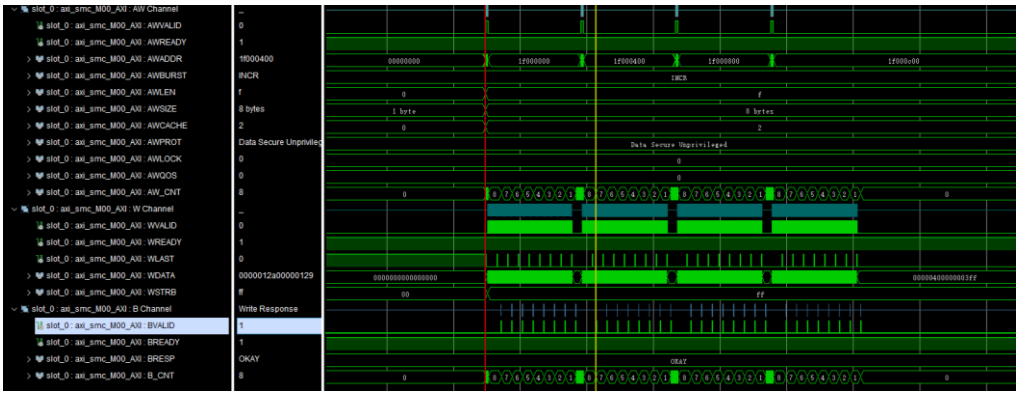


图 1-69 HP 接口写事务时序

然后是读事务，图 1-70 为前 8 个读 burst 的部分时序：

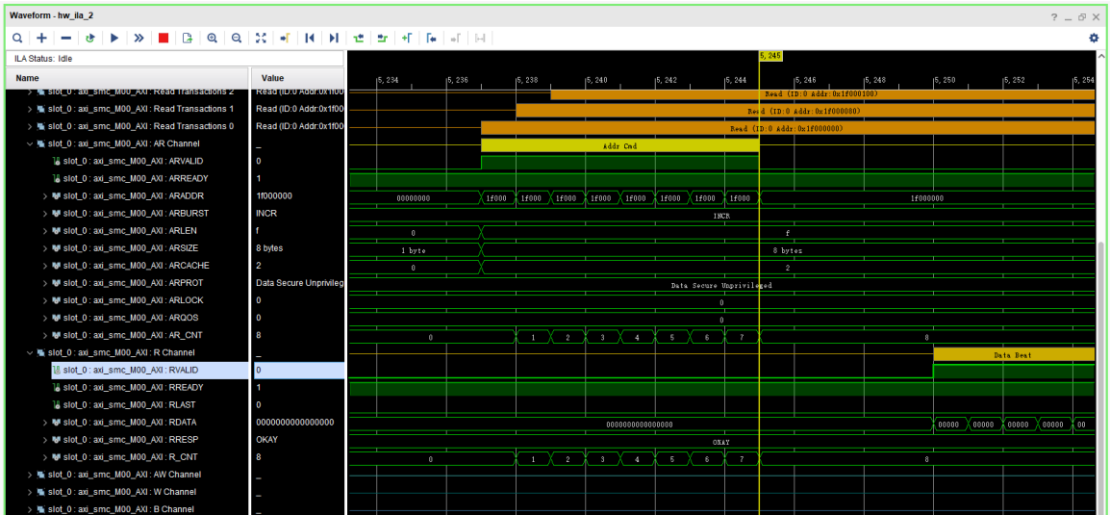


图 1-70 HP 接口读事务部分时序

图中由于缩放原因，地址显示不全，实际依次为：0x1f000000、0x1f000080、0x1f000100、0x1f000180、0x1f000200、0x1f000280、0x1f000300、0x1f000380。

可以看到，读事务同样为 outstanding 传输，主机首先一次性写入接下来要突发的地址以及控制信号，随后从机接收到指令后便开始传输数据。数据传输的过程中 RRESP 信号反馈当前传输状态，在传输每个 burst 最后一个数据期间，RLAST 信号被拉高，如图 1-71 所示：

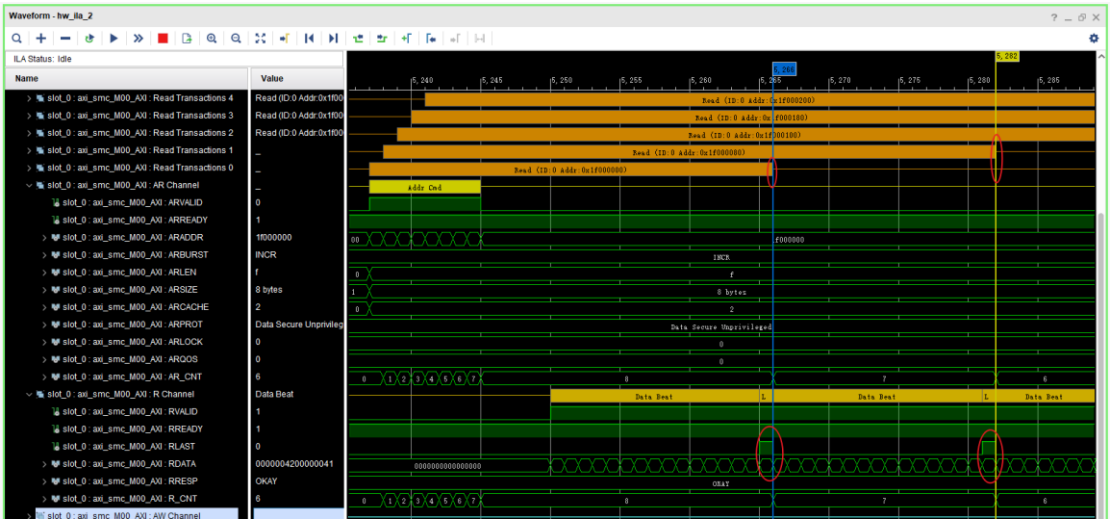


图 1-71 HP 接口读事务时序

通过对以上 AXI4 主接口和 HP 接口时序的分析，我们已经验证了前面原理部分内容的准确性，同时也可以看到，在需要使用 Interconnect 核作为桥连接不同 AXI 接口时，即使主机不支持 outstanding 传输，但只要从机支持 outstanding

传输，那么数据在从 Interconnect 核传输给从机时，仍可能会使用 outstanding 传输。

1.5 总结

本章我们学习了 AXI4 总线协议的相关内容，了解了 AXI4 总线的通道组成、读写事物时序、相关传输类型以及特定的传输模式，并结合 Xilinx 的自定义 AXI4 模板 IP 核对总线协议中各个通道内的信号、通道间握手对的依赖关系、读写事务的传输时序以及 AXI 系统中 Interconnect 核的功能作用进行了验证。

本章只是对 AXI4 接口协议中，较常用的部分进行了简单的介绍，AXI4 协议整体是比较复杂的，大家可以参考ARM 官方手册 IHI0022E。为了方便大家学习使用，我们已经提前将手册下载放在了 ACZ702 开发板的资料包中，路径如下：*小梅哥 ACZ702 型 Zynq 开发板资料\盘 A_ACZ702 开发板标准配套资料\10_Xilinx 原厂文档*。