

版本信息：

版本 REV2022

时间 10/15/2021

米联客2022版FPGA数据缓存方案(PSDDR-FDMA)

Milianke 2022 Xilinx ZYNQ 7000 Course

电子版自学资料

常州一二三电子科技有限公司

溧阳米联电子科技有限公司

版权所有

米联客(MILIANKE)04QQ 群: 516869816

米联客(MILIANKE)03QQ 群: 543731097

米联客(MILIANKE)02QQ 群: 86730608

米联客(MILIANKE)01QQ 群: 34215299

微信扫码注册米联客技术论坛www.uisrc.com免费享受更多资源



扫码关注微信公众平台“米联客(MILIANKE)”掌握更多信息动态



版本	时间	描述
Rev2016	2016-07-03	第一版
Rev2019	2019-05-09	第二版
Rev2020	2020-05-23	第三版
Rev2021	2021-04-20	第四版
Rev2022	2021-10-15	第五版
Rev2022	2022-09-30	<p>第五版更新说明：</p> <p>uiFDMA IP 从 3.0 升级到 3.1 增加 burst 最大长度可以用户设置，uifdma_dbuf 升级到 3.0 版本，支持 full 和 empty 信号控制。增加 AXI4 总线入门部分对 uiFDMA 和 uiDbu IP 的详细讲解。更加完善的教学 demo 方案。</p>

序 1:

FPGA 芯片是硬件技术而 FPGA 编程又称为硬件编程语言和流行的各类软件编程语言 C/C++、JAVA、python 等相比，掌握基础的硬件编程语言不是难事，难点是 FPGA 在每个专业领域的应用，只有充分理解了 FPGA，并且具有对自己所处行业专业背景认知，才能真正理解 FPGA 应该用在什么场合更加合适。

从业多年来，亲身经历了 FPGA 的发展历程，也深刻体会到未来 FPGA 应用领域可能发生的深刻变革，FPGA 从简单的逻辑门，发展到现在具备很多高速通信接口，而且最新 SOC 中也集成了 FPGA 单元，实现了 ARM 和 FPGA 单芯片。目前 XILINX 代表了业内领先的 FPGA 技术，已经可以把 FPGA\ARM\GPU\RFID 等集成到单芯片。FPGA 的逻辑资源也是达到了前所未有的密度以及超大容量。

很多人问我学习 FPGA 是否有前途，这个问题着实难以回答。我们可以一起来探讨以下几种情况，会是 FPGA 发挥作用的场合：

1) 数字 IC 设计工作

数字 IC 设计还是主要以硬件编程语言去设计数字 IC 芯片，通过硬件编程语言，软件可以把语言翻译成电路，自动布线工具可以完成布局布线，之后流片。由于流片费用非常贵，前期也可以用 FPGA 芯片模拟设计的数字 IC 芯片的功能。

2) 高速模数字信号采集分析

高速的 ADC,DAC 的数模信号处理的领域也是必然需要用到 FPGA，在无线通信、雷达信号处理等领域也都会用到。

3) 数字信号高速通信

FPGA 具备的高速接口也非常适合用于高速通信，比如 PCIE 通信、光通信、以太网通信

4) 视频图像

包括图像的拼接、缩放、高效的实时传输等， 4K 视频 、 8K 视频领域

5) 硬件加速算法

FPGA 的硬件加速领域也是目前的热门研究，也是 FPGA 未来最有前景的一个应用领域，已经有很多公司利用 FPGA 的硬件加速实现了很好的经济效应，但是目前 FPGA 的加速还没有做到普及，和传统的 GPU 相比，主要难度还是在开发难度上，一般小公司很难有实力取得突破。

6) 通用 CPU GPU 无法完成的工作

如果通用的 CPU 和 GPU 无法完成的工作任务，可以考虑下 FPGA 或者带 FPGA 的 SOC.

FPGA 到目前为止依然是一个小众的领域，如果专门为了学习 FPGA 而学习 FPGA 而不知道如何应用 FPGA，那么这是非常悲哀的一件事情，学习 FPGA 只是学习一门技能，而结合自己专业背景选择最合适的解决方案，解决问题才是最终的目的。

序 2:

米联客团队励志在 FPGA 领域可以贡献一份自己的力量，让 FPGA 从业者入门门槛降低、FPGA 应用难度减少，为中小型 FPGA 团队提供必要有价值的技术资料和硬件支持。我们希望可以和广大客户形成紧密的合作伙伴关系，一起创造共赢，各自实现自己的价值目标。

从 2019 年下半年开始我们米联客团队计划研发全新的 2020 版本教程。经过一年半时间的更新，2020 版本教程研发进度进入了关键攻坚时期。

2020 版本教程是适应当前 FPGA 技术发展，SOC 技术发展，新形势下，米联客做出的战略决策。2020 版本教程需要解决以下几个问题：

- 1)、FPGA 基础课程，需要解决长期以来没有认真解决好的课程内容，包括：FPGA 的构架、FPGA 常用 IP 使用、硬件编程经验、通信接口应用、代码规范、时序分析
- 2)、新版本 vitis 软件的使用技巧
- 3)、更多讲解高速通信的基础知识和应用解决方案
- 4)、ZYNQ-SOC 到 ZYNQ-MPSOC 教程统一部署，达到 90%以上 demo 使用方法一样。不管是学习 ZYNQ-SOC 还是 ZYNQ-MPSOC,已经学习的 demo 具有 2 个平台之间的互通性。
- 5)、ZYNQ 和 MPSOCLinux 课程做到适合初级入门，并提供丰富的应用 demo
- 6)、适应未来发展趋势，增加 FPGA 高层次加速算法编程领域的课程内容
- 7)、教程的构架能够支持 7 系列 FPGA、UltraScale 系列 FPGA、UltraScale+系列 FPGA 和 MPSOC

不管有多麻烦，不管困难多大，面对挑战我们坚信胜利！

米联客团队
2021 年 4 月 26 日

目录

01AXI4-FULL-MASTER IP FDMA 介绍	4
1.1 概述	4
1.2AXI 总线协议介绍	4
1:AXI 总线概述	4
2:AXI-4 总线信号功能	5
3:数据有效的情况	7
4:突发式读写	7
5:AXI4 数据路由及缓存机制	9
1.3FDMA 源码分析	11
1:FDMA 的写时序	11
2:FDMA 的读时序	12
3:FDMA 的 AXI4-Master 写操作	12
4:FDMA 的 AXI4-Master 读操作	15
1.4FDMA IP 的封装	18
02AXI4-FULL-uiFDMA IP 仿真验证	27
2.1 概述	27
2.1saxi_full_mem IP 介绍	27
2.2 创建 FPGA 逻辑工程	27
2.3 添加 FDMA 接口控制代码	32
2.4 仿真文件	36
2.5 实验结果	37
03 使用 fdma 读写 axi-bram 测试	41
3.1 概述	41
3.2 系统框图	41
3.3 创建图形化逻辑工程	41
1:创建工程命名为 fpga_prj	41
2:创建 Block Design 并且命名为 system	44
3:添加图形化 FPGA IP	45
4:完成 IP 之间的信号自动	46
5:调整 IP 参数	47
6:引出 FPGA 接口信号	52
7:修改复位和信号名字	52
8:视图优化	53
9:地址分配	54
10:自动校验	54
11:校验图形逻辑工程	55
12:自动产生顶层文件	55
3.4 编写 FDMA 的 BRAM 测试代码	56
3.5 程序分析	61
1:总流程图	61
2:FDMA 的读写时序	61
3.6RTL 仿真	62
1:仿真 tb 文件	62
2:仿真测试	64
3.7 编译测试	65
04 使用 fdma 读写 DDR	68
4.1 概述	68
4.2 系统框图	68

4.3 基于图形化的逻辑设计	68
1:PS 复位设置	70
2:设置 PS HP Slave 接口	70
3:PL 输出时钟设置	71
4:地址空间分配	71
5:编写 FDMA 的 DDR 测试代码	71
6:程序分析	75
7:编译并导出平台文件	76
4.4 搭建 Vitis-sdk 工程	77
1:创建 SDK Platform 工程	77
2:创建 hello APP 工程	77
4.5 实验结果	78
05uifdma_dbuf 3.0 IP 介绍	80
5.1 概述	80
5.2uifdma_dbuf 的 ud 信号定义	80
5.3FDMA-DBUF IP 代码分析	81
1:FDMA-DBUF 写状态机	81
2:FDMA 的写时序波形图	82
3:FDMA-DBUF 写状态机	82
4:FDMA 的读时序波形图	82
5.4 源码分析	83
1:uidbuf.v	83
2:fs_cap.v	91
3:uidbufirq.v	93
06 uifdma_dbuf+fdma 实现数据流方案	94
6.1 概述	94
6.2 系统框图	94
6.3 基于图形化逻辑设计	94
1:uifdma_dbuf 设置	95
2:uifdma 设置	95
3 编写测试代码	96
4:地址空间分配	99
5:编译并导出平台文件	100
6.4 搭建 Vitis-sdk 工程	100
1:创建 SDK Platform 工程	100
2:创建 hello APP 工程	101
6.5 实验结果	101
07 基于 fdma 多路视频缓存数据构架方案	103
7.1 概述	103
7.2 系统框图	103
7.3 基于图形化的逻辑设计	103
1:帧同步设计	104
2:多路视频的同屏显示原理	104
3:uifdma_dbuf0 的参数设置	107
4:uifdma_dbuf1 的参数设置	107
5:uifdma_dbuf2 的参数设置	108
6:uifdma_dbuf3 的参数设置	108
7:uiFDMA 的参数设置	109
8:地址空间分配	109

9:编译并导出平台文件	109
7.4 搭建 Vitis-sdk 工程	110
1:创建 SDK Platform 工程	110
2:创建 hello APP 工程	110
7.5 实验结果	110
08fdma 数据通路加入 sobel 算法 IP 方案	112
8.1 概述	112
8.2 系统框图	112
8.3 基于图形化的逻辑设计	112
1:帧同步设计	113
2:多路视频的同屏显示原理	113
3:uifdma_dbuf_0 的配置	114
4:uifdma_dbuf_1 的配置	115
5:地址空间分配	115
6:编译并导出平台文件	115
8.4 搭建 Vitis-sdk 工程	116
1:创建 SDK Platform 工程	116
2:创建 hello APP 工程	116
8.5 硬件接线	117
8.6 实验结果	117
附录 1:更多关于米联客 FDMA IP 的方案介绍	118
附录 2: 常见问题	错误! 未定义书签。
1 联系方式	错误! 未定义书签。
2 售后	错误! 未定义书签。
3 销售	错误! 未定义书签。
4 在线视频	错误! 未定义书签。
5 软件下载	错误! 未定义书签。
6 经验分享	错误! 未定义书签。
7 官方博文	错误! 未定义书签。

01AXI4-FULL-MASTER IP FDMA 介绍

软件版本：vitis2021.1(vivado2021.1)

操作系统：WIN10 64bit

硬件平台：适用 XILINX A7/K7/Z7/ZU/KU 系列 FPGA

登录“米联客”FPGA 社区-www.uisrc.com 视频课程、答疑解惑！

1.1 概述

FDMA 是米联客的基于 AXI4 总线协议定制的一个 DMA 控制器。本文对 AXI4-FULL 总线接口进行了封装，同时定义了简单的 APP 接口提供用户调用 AXI4 总线实现数据交互。这个 IP 我们命名为 FDMA(Fast Direct Memory Access)。

有了这个 IP 我们可以统一实现用 FPGA 代码直接读写 PL 的 DDR 或者 ZYNQ/ZYNQMP SOC PS 的 DDR 或者 BRAM。FDMA IP CORE 已经广泛应用于 ZYNQ SOC/Artix7/Kintex7/ultrascale/ultrascale+系列 FPGA/SOC。

如果用过 ZYNQ/ZYNQMP SOC 的都知道，要直接操作 PS 的 DDR 通常是 DMA 或者 VDMA，然而用过 XILINX 的 DMA IP 和 VDMA IP，总有一种遗憾，那就是不够灵活，还需要对寄存器配置，真是麻烦。XILINX 的总线接口是 AXI4 总线，自定义 AXI4 IP 挂到总线上就能实现对内存地址空间的读写访问。因此，我们只要掌握 AXI4 协议就能完成不管是 PS 还是 PL DDR 的读写操作。

米联客封装的 AXI4 总线 IP 命名为 uiFDMA，自 2018 年第一版本发布后，就引起了很多 FPGA 工程师的兴趣，并且得到了广大 FPGA 工程师的好评，但是 FDMA1.0 版本还是有一些局限和 BUG，再实际的应用中被 FPGA 工程师发现，因此给了我们很多宝贵意见。

2020 和 2022 版本中 FDMA 版本从 1.0 升级到 3.0，Burst 默认长度为 256，并且自动计算剩余 burst 长度，相比 FDMA1.0 具有更好的可靠性，更高的效率，但是 3.0 发布后，当通常 4 个 FDMA 开始传输 1080P@60 帧的视频同时输出的时候，会导致某个通道总是处于饥饿状态，因为每次 AXI burst 256 长度太长了，所以我们下面升级到了 fdma3.1 版本。

借此新一期 2022 版本 FDMA 专辑视频资料发布之际，我们对本专辑的 FDMA3.0 做小幅度修改，升级到 FDMA3.1 版本。相比 3.0 版本默认 256 burst 长度，在多个 FDMA 同时使用的时候会导致 AXI4 总线上某一个通路大量占用总线带宽，3.1 版本可以手动设置 AXI4 的最大 burst 长度，可以在多个 FDMA 同时使用的时候，通过设置合理的 burst 长度，来优化总线上某个通路同一时刻独占 AXI4 总线的时间。

uiFDMA3.1 新增特性：

- 1: 支持多个 FDMA IP 同时挂到 AXI-interconnect 总线，同时工作
- 2: 支持自动计算 AXI-Burst 长度，使用起来非常简单，只需要给出 FDMA burst 需要 burst 的总长度。
- 3: 支持 AXI-Burst 最大长度的人工设置

从本文开始，我们从多个应用方案来演示 FDMA 的用途。

本文实验目的：

1: 分析 FDMA 源码，掌握基于 FDMA 的 APP 接口实现 AXI4-FULL 总线接口的访问。

2: 掌握自定义总线接口封装方法

1.2 AXI 总线协议介绍

关于 AXI4 总线的更多内容可以学习“米联客 2022 版 AXI4 总线专题篇”相关课程内容，以下我们继续给出 AXI 总线相关的描述。

1: AXI 总线概述

在 XILINX FPGA 的软件工具 vivado 以及相关 IP 中有支持三种 AXI 总线，拥有三种 AXI 接口，当然用的都是 AXI 协议。其中三种 AXI 总线分别为：

AXI4: (For high-performance memory-mapped requirements.) 主要面向高性能地址映射通信的需求，是面向地址映射的接口，允许最大 256 轮的数据突发传输；

AXI4-Lite: (For simple, low-throughput memory-mapped communication) 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。

AXI4-Stream: (For high-speed streaming data.) 面向高速流数据传输；去掉了地址项，允许无限制的数据突发传输规模。

由于 AXI4 和 AXI4-Lite 信号大部分一样，以下只介绍 AXI4 信号。另外对于 AXI4-Stream 协议不再本文中介绍，后面有单独介绍的文章。

2:AXI-4 总线信号功能

2-1:时钟和复位

信号	方向	描述
ACLK	时钟源	全局时钟信号
ARESETn	复位源	全局复位信号，低有效

2-2:写地址通道信号：

信号	方向	描述
AWID	主机 to 从机	写地址 ID，用来标志一组写信号
AWADDR	主机 to 从机	写地址，给出一次写突发传输的写地址
AWLEN	主机 to 从机	AWLEN[7:0]决定写传输的突发长度。AXI3 只支持 1~16 次的突发传输 (Burst_length=AxLEN[3:0]+1)，AXI4 扩展突发长度支持 INCR 突发类型为 1~256 次传输，对于其他的传输类型依然保持 1~16 次突发传输 (Burst_Length=AxLEN[7:0]+1)。 burst 传输具有如下规则： wrapping burst ,burst 长度必须是 2,4,8,16 burst 不能跨 4KB 边界 不支持提前终止 burst 传输
AWSIZE	主机 to 从机	写突发大小，给出每次突发传输的字节数支持 1、2、4、8、16、32、64、128
AWBURST	主机 to 从机	突发类型： 2'b00 FIXED：突发传输过程中地址固定，用于 FIFO 访问 2'b01 INCR：增量突发，传输过程中，地址递增。增加量取决于 AwSIZE 的值。 2'b10 WRAP：回环突发，和增量突发类似，但会在特定高地址的边界处回到低地址处。回环突发的长度只能是 2,4,8,16 次传输，传输首地址和每次传输的大小对齐。最低的地址整个传输的数据大小对齐。回环边界等于 (AwSIZE*AwLEN) 2'b11 Reserved
AWLOCK	主机 to 从机	总线锁信号，可提供操作的原子性
AWCACHE	主机 to 从机	内存类型，表明一次传输是怎样通过系统的
AWPROT	主机 to 从机	保护类型，表明一次传输的特权级及安全等级
AWQOS	主机 to 从机	质量服务 QoS
AWREGION	主机 to 从机	区域标志，能实现单一物理接口对应的多个逻辑接口
AWUSER	主机 to 从机	用户自定义信号
AWVALID	主机 to 从机	有效信号，表明此通道的地址控制信号有效
AWREADY	从机 to 主机	表明“从”可以接收地址和对应的控制信号

2-3:写数据通道信号：

信号名	方向	描述
WID	主机 to 从机	一次写传输的 ID tag
WDATA	主机 to 从机	写数据
WSTRB	主机 to 从机	WSTRB[n:0]对应于对应的写字节，WSTRB[n]对应 WDATA[8n+7:8n]。WVALID 为低时，WSTRB 可以为任意值，WVALID 为高时，WSTRB 为高的字节线必须指示有效的数据。

WLAST	主机 to 从机	表明此次传输是最后一个突发传输
WUSER	主机 to 从机	用户自定义信号
WVALID	主机 to 从机	写有效, 表明此次写有效
WREADY	从机 to 主机	表明从机可以接收写数据

写响应信号:

信号名	方向	描述
BID	从机 to 主机	写响应 ID tag
BRESP	从机 to 主机	写响应, 表明写传输的状态
BUSER	从机 to 主机	用户自定义
BVALID	从机 to 主机	写响应有效
BREADY	主机 to 从机	表明主机能够接收写响应

2-4: 读地址通道信号:

信号	方向	描述
ARID	主机 to 从机	读地址 ID, 用来标志一组写信号
ARADDR	主机 to 从机	读地址, 给出一次读突发传输的读地址
ARLEN	主机 to 从机	ARLEN[7:0]决定读传输的突发长度。AXI3 只支持 1~16 次的突发传输 (Burst_length=AxLEN[3:0]+1), AXI4 扩展突发长度支持 INCR 突发类型为 1~256 次传输, 对于其他的传输类型依然保持 1~16 次突发传输 (Burst_Length=AxLEN[7:0]+1)。 burst 传输具有如下规则: wrapping burst, burst 长度必须是 2,4,8,16 burst 不能跨 4KB 边界 不支持提前终止 burst 传输
ARSIZE	主机 to 从机	读突发大小, 给出每次突发传输的字节数支持 1、2、4、8、16、32、64、128
ARBURST	主机 to 从机	突发类型: 2'b00 FIXED: 突发传输过程中地址固定, 用于 FIFO 访问 2'b01 INCR : 增量突发, 传输过程中, 地址递增。增加量取决 AxSIZE 的值。 2'b10 WRAP: 回环突发, 和增量突发类似, 但会在特定高地址的边界处回到低地址处。回环突发的长度只能是 2,4,8,16 次传输, 传输首地址和每次传输的大小对齐。最低的地址整个传输的数据大小对齐。回环边界等于 (AxSIZE*AxLEN) 2'b11 Reserved
ARLOCK	主机 to 从机	总线锁信号, 可提供操作的原子性
ARCACHE	主机 to 从机	内存类型, 表明一次传输是怎样通过系统的
ARPROT	主机 to 从机	保护类型, 表明一次传输的特权级及安全等级
ARQOS	主机 to 从机	质量服务 QoS
ARREGION	主机 to 从机	区域标志, 能实现单一物理接口对应的多个逻辑接口
ARUSER	主机 to 从机	用户自定义信号
ARVALID	主机 to 从机	有效信号, 表明此通道的地址控制信号有效
ARREADY	从机 to 主机	表明“从”可以接收地址和对应的控制信号

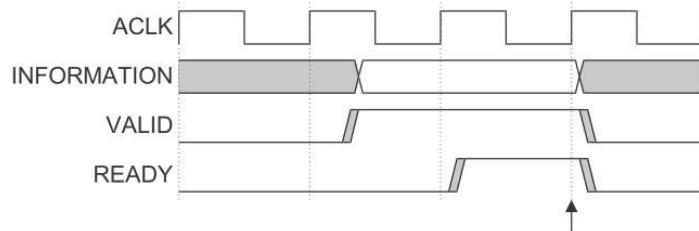
2-5: 读数据通道信号:

信号名	方向	描述
RID	从机 to 主机	一次读传输的 ID tag
RDATA	从机 to 主机	读数据
RRESP	从机 to 主机	读响应, 表明读传输的状态
RLAST	从机 to 主机	表明此次传输是最后一个突发传输
RUSER	从机 to 主机	用户自定义信号
RVALID	从机 to 主机	读有效, 表明数据总线上数据有效
RREADY	主机 to 从机	表明主机准备好可以接收数据

3:数据有效的情况

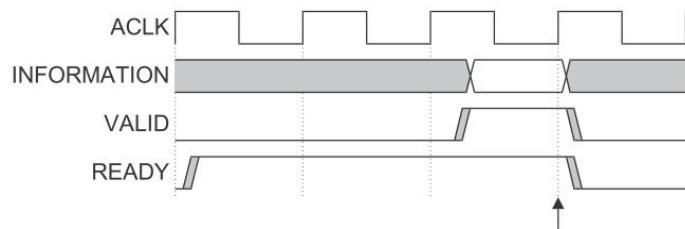
AXI4 所采用的是一种 READY, VALID 握手通信机制, 简单来说主从双方进行数据通信前, 有一个握手的过程。传输源产生 VLAID 信号来指明何时数据或控制信息有效。而目地源产生 READY 信号来指明已经准备好接受数据或控制信息。传输发生在 VALID 和 READY 信号同时为高的时候。VALID 和 READY 信号的出现有三种关系。

(1) VALID 先变高 READY 后变高。时序图如下:



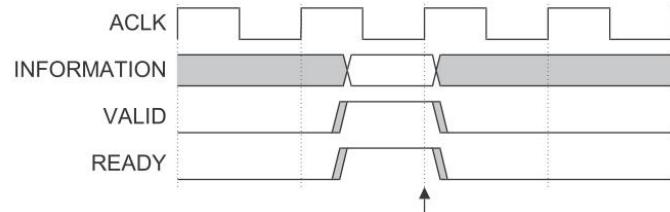
在箭头处信息传输发生。

(2) READY 先变高 VALID 后变高。时序图如下:



同样在箭头处信息传输发生。

(3) VALID 和 READY 信号同时变高。时序图如下:



在这种情况下, 信息传输立马发生, 如图箭头处指明信息传输发生。

4:突发式读写

4-1:突发式写时序图

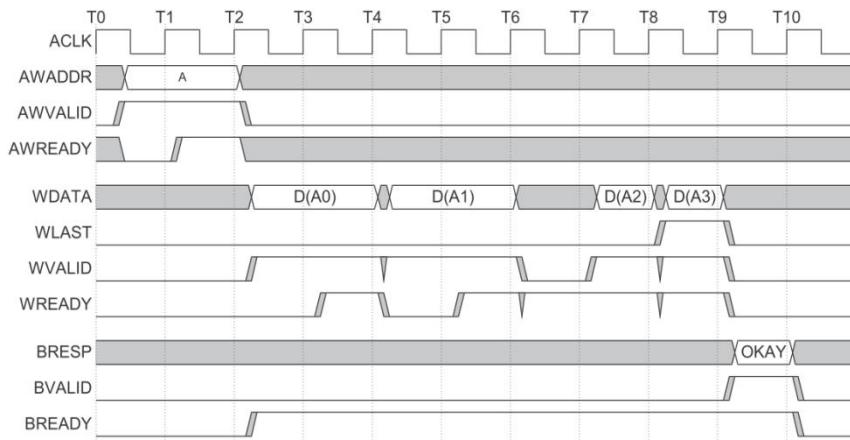
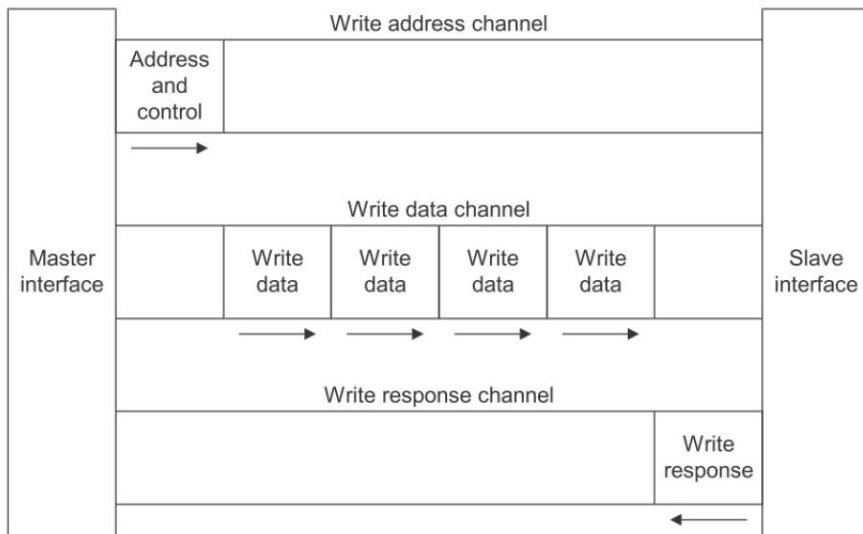
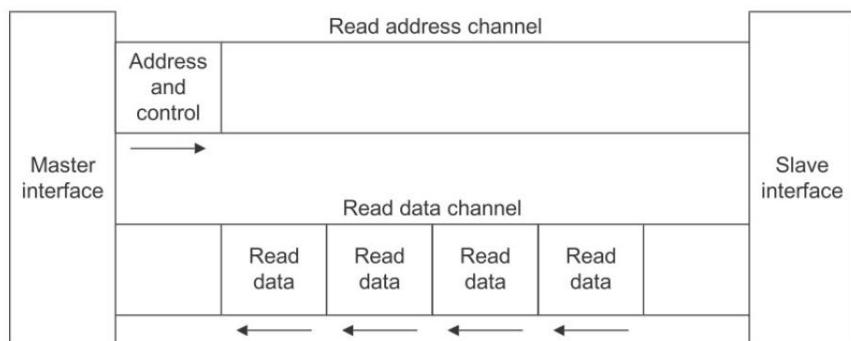


Figure 1-6 Write burst

这一过程的开始时，主机发送地址和控制信息到写地址通道中，然后主机发送每一个写数据到写数据通道中。当主机发送最后一个数据时，WLAST 信号就变为高。当设备接收完所有数据之后他将一个写响应发送回主机来表明写事务完成。

2:突发式读的时序图



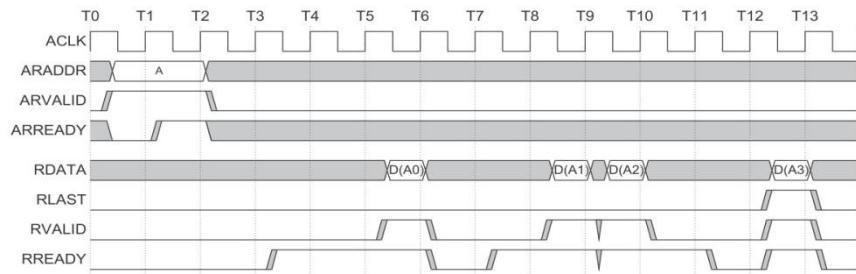
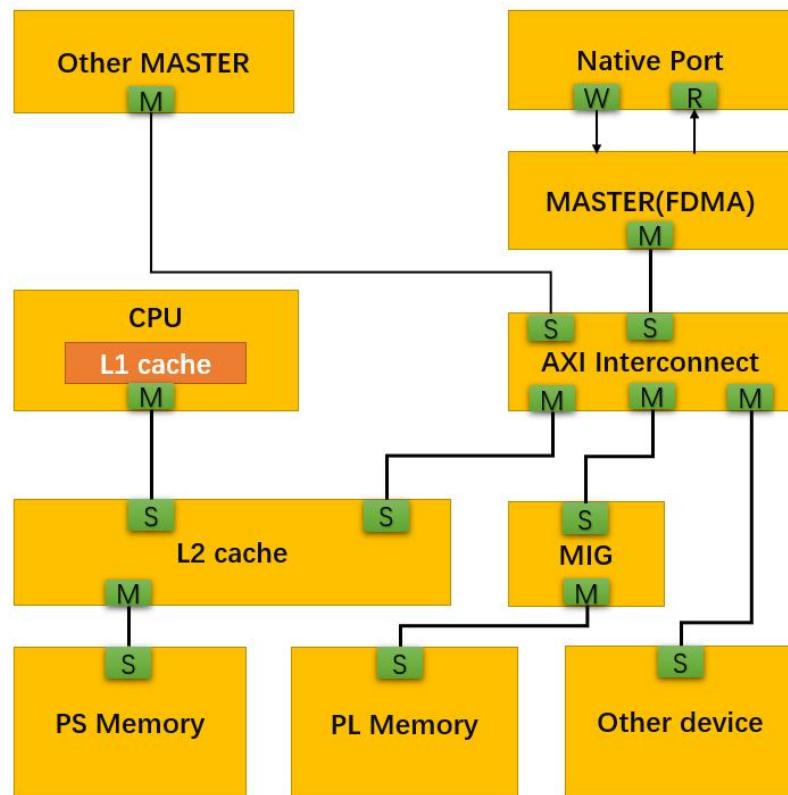


Figure 1-4 Read burst

当地址出现在地址总线后，传输的数据将出现在读数据通道上。设备保持 VALID 为低直到读数据有效。为了表明一次突发式读写的完成，设备用 RLAST 信号来表示最后一个被传输的数据。

5:AXI4 数据路由及缓存机制



在上图中，我们描述了典型的 AXI 总线互联方式，经过 AXI-interconnect 可以完成多个 MASTER 和多个 SLAVE 之间的互联。比如对于 XILINX 的 SOC 来说，内存有 PS 的 DDR 也有 PL 的 DDR，还有基于 AXI4 总线的其他 IP，比如 PCIE 的 XDMA IP 等，都可以通过 AXI-interconnect 实现高效的互联，这样各个外设之间的数据就能非常方便的基于 AXI4 总线实现共享访问。

这里我们有必要了解下 AXI4 总线中的缓存机制，主要是 ARCACHE[3:0]和 AWCAHE[3:0]的设置。由于关于这两个参数我们并没有查阅到非常详尽的应用说明，这里米联客的教程以我们查阅的资料和我们自己的理解介绍这两个参数。

5-1:Modifiable 和 Non-modifiable transaction

当设置 AxCACHE[1] = 0，则是 Non-modifiable transaction，Non-modifiable transaction 不能被拆分成多个 transaction 传输，也不能合并 transaction 传输。

在 Non-modifiable transaction 中以下信号不能修改。

AXADDR	传输起始地址
AXSIZE	传输长度
AXLEN	突然长度
AXBURST	突发类型
AXLOCK	锁类型
AXPORT	保护类型

当设置 AxCACHE[1] = 1，则是 modifiable transaction， modifiable transaction 能被拆分成多个 transaction 传输，也能合并 transaction 传输，因此 AxADDR、AxSIZE、AxLEN、AxBURST 可以被改变。

5-2:AXI4 存储类型参数描述

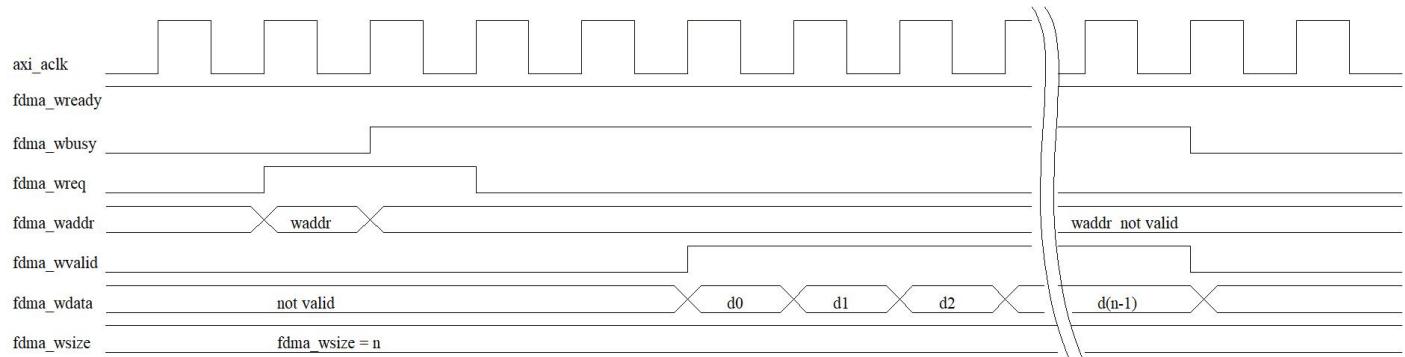
ARCACHE[3:0]	AWCACHE[3:0]	Memory type
0000	0000	<p>Device Non-bufferable</p> <p>1- write response 必须从 final destination 响应</p> <p>2- Read data 必须从 final destination 响应</p> <p>3- Transactions are non-modifiable(cacheable)</p> <p>4-读不能被 prefetched, 写不能被 merged</p> <p>5-来自于同一 ID 并到同一个 Slave 的所有 Non-modifiable 读写 transactions 必须保持顺序</p>
0001	0001	<p>Device Bufferable</p> <p>1- write response 可以从一个中间节点响应</p> <p>2-write response 必须及时到达 final destination, 而不能永远存储在 buffer 中</p> <p>3- Read data 必须从 final destination 响应</p> <p>4-Transactions are non-modifiable</p> <p>5-读不能被 prefetched, 写不能被 merged</p> <p>6-同一 ID 并到同一个 Slave 的所有 Non-modifiable 读写操作必须保持顺序</p>
0010	0010	<p>Normal Non-cacheable Non-bufferable</p> <p>1-write response 必须从 finaldestination 响应</p> <p>2-Read data 必须从 final destination 响应</p> <p>3-Transactions are modifiable</p> <p>4-写可以 merge</p> <p>5-到有重叠目的地址的来自于同一 ID 的读写 transactions 必须保持顺序</p>
0011	0011	<p>Normal Non-cableable Bufferable</p> <p>1-write response 可以从一个中间节点响应</p> <p>2-Write transaction 必须对 finaldestination 及时可见</p> <p>3-Read data 要么从 final destination 得到，要么从一个正在到它的 final destination 的 write transaction 响应（“它”指 write transaction）</p> <p>4-如果 read data 从一个 write transaction 得到，它必须是该 write 的最近版本，并且，这个 data 不能被缓存下来。</p> <p>5-到有重叠目的地址的来自于同一 ID 的读写 transactions 必须保持顺序</p>
1010	0110	<p>Write-through No-allocate</p> <p>1-write response 可以从一个中间节点响应</p> <p>2-Write transaction 必须对 finaldestination 及时可见</p> <p>3-Read data 可以从中间的 cache 响应</p> <p>4-Transactions are modifiable</p> <p>5-Reads 可以被 prefetched</p> <p>6-Writes 可以被 merged</p> <p>7-Cache lookup is required</p> <p>8-到有重叠目的地址的来自于同一 ID 的读写 transactions 必须保持顺序</p>

		8-建议不要对 read and write transactions 进行 allocation 操作
1110(0110)	0110	Write-through Read-allocate 同 Write-through No-allocate 功能一样
1010	1110(1010)	Write-through Write-allocate 同 Write-through No-allocate 功能一样
1110	1110	Write-through Read and Write-allocate 同 Write-through No-allocate 功能一样
1011	0111	Write-back No-allocate 1-write response 可以从一个中间节点得到 2-Write transaction 可以对 finaldestination 不及时可见 3-Read data 可以从中间的 cache 得到 4-Transactions are modifiable 5-Reads 可以被 prefetched 6-Writes 可以被 merged 7-Cache lookup is required 8-到有重叠目的地址的来自于同一 ID 的读写 transactions 必须保持顺序 9-建议不要对 read and write transactions 进行 allocation 操作
1111(0111)	0111	Write-back Read-allocate 同 Write-back no allocate 功能一样
1011	1111(1011)	Write-back Write-allocate 同 Write-back no allocate 功能一样
1111	1111	Write-back Read and Write-allocate 同 Write-back no allocate 功能一样

1.3 FDMA 源码分析

由于 AXI4 总线协议直接操作起来相对复杂一些，容易出错，因此我们封装一个简单的用户接口，间接操作 AXI4 总线会带来很多方便性。先看下我们计划设计一个怎么样的用户接口。

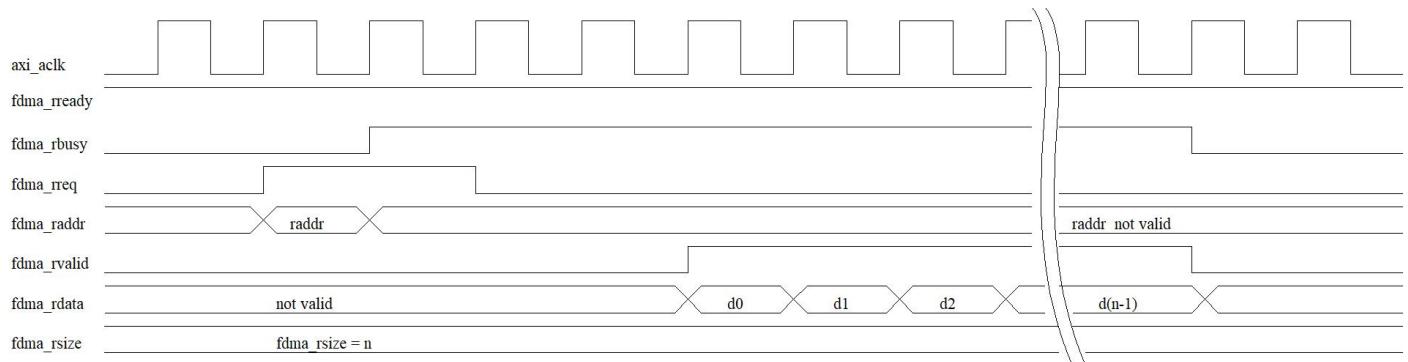
1:FDMA 的写时序



fdma_wready 设置为 1，当 fdma_wbusy=0 的时候代表 FDMA 的总线非忙，可以进行一次新的 FDMA 传输，这个时候可以设置 fdma_wreq=1，同时设置 fdma burst 的起始地址和 fdma_wszie 本次需要传输的数据大小(以 bytes 为单位)。当 fdma_wvalid=1 的时候需要给出有效的数据，写入 AXI 总线。当最后一个数写完后，fdma_wvalid 和 fdma_wbusy 变为 0。

AXI4 总线最大的 burst length 是 256，而经过封装后，用户接口的 fdma_size 可以任意大小的，fdma ip 内部代码控制每次 AXI4 总线的 Burst 长度，这样极大简化了 AXI4 总线协议的使用。

2:FDMA 的读时序



`fdma_rready` 设置为 1，当 `fdma_rbusy=0` 的时候代表 FDMA 的总线非忙，可以进行一次新的 FDMA 传输，这个时候可以设置 `fdma_rreq=1`，同时设置 fdma burst 的起始地址和 `fdma_rsize` 本次需要传输的数据大小(以 bytes 为单位)。当 `fdma_rvalid=1` 的时候需要给出有效的数据，写入 AXI 总线。当最后一个数写完后，`fdma_rvalid` 和 `fdma_rbusy` 变为 0。

同样对于 AXI4 总线的读操作，AXI4 总线最大的 burst length 是 256，而经过封装后，用户接口的 `fdma_size` 可以任意大小的，fdma ip 内部代码控制每次 AXI4 总线的 Burst 长度，这样极大简化了 AXI4 总线协议的使用。

3:FDMA 的 AXI4-Master 写操作

以下代码中我们给出 axi4-master 写操作的代码分析注释

```
//fdma_axi_write-----
reg      [M_AXI_ADDR_WIDTH-1 : 0]      axi_awaddr  =0; //AXI4 写地址
reg                  axi_awvalid = 1'b0; //AXI4 写地有效
wire     [M_AXI_DATA_WIDTH-1 : 0]      axi_wdata   ; //AXI4 写数据
wire                  axi_wlast   ; //AXI4 写 LAST 信号
reg                  axi_wvalid  = 1'b0; //AXI4 写数据有效
wire
      w_next=(M_AXI_WVALID & M_AXI_WREADY); //当 valid ready 信号都有效，代表 AXI4 数据传输有效
reg [8 :0]
      wburst_len = 1 ; //写传输的 axi burst 长度，代码会自动计算每次 axi 传输的 burst 长度
reg [8 :0]
      wburst_cnt = 0 ; //每次 axi bust 的计数器
reg [15:0]
      wfdma_cnt = 0 ; //fdma 的写数据计数器
reg
      axi_wstart_locked = 0; //axi 传输进行中，lock 住，用于时序控制
wire [15:0] axi_wburst_size = wburst_len * AXI_BYTES; //axi 传输的地址长度计算

assign M_AXI_AWID      = M_AXI_ID; //写地址 ID，用来标志一组写信号，M_AXI_ID 是通过参数接口定义
assign M_AXI_AWADDR    = axi_awaddr;
assign M_AXI_AWLEN     = wburst_len - 1; //AXI4 burst 的长度
assign M_AXI_AWSIZE    = clogb2(AXI_BYTES-1);
assign M_AXI_AWBURST   = 2'b01; //AXI4 的 busr 类型 INCR 模式，地址递增
assign M_AXI_AWLOCK    = 1'b0;
assign M_AXI_AWCACHE   = 4'b0010; //不使用 cache，不使用 buffer
assign M_AXI_AWPROT    = 3'h0;
assign M_AXI_AWQOS     = 4'h0;
assign M_AXI_AWVALID   = axi_awvalid;
assign M_AXI_WDATA     = axi_wdata;
assign M_AXI_WSTRB     = {(AXI_BYTES){1'b1}}; //设置所有的 WSTRB 为 1 代表传输的所有数据有效
assign M_AXI_WLAST     = axi_wlast;
```

```

assign M_AXI_WVALID      = axi_wvalid & fdma_wready; //写数据有效, 这里必须设置 fdma_wready 有效
assign M_AXI_BREADY      = 1'b1;

//-----
//AXI4 FULL Write

assign axi_wdata        = fdma_wdata;
assign fdma_wvalid       = w_next;
reg    fdma_wstart_locked = 1'b0;
wire   fdma_wend;
wire   fdma_wstart;
assign  fdma_wbusy = fdma_wstart_locked ;

//在整个写过程中 fdma_wstart_locked 将保持有效, 直到本次 FDMA 写结束

always @(posedge M_AXI_ACLK)
  if(M_AXI_RESETN == 1'b0 || fdma_wend == 1'b1 )
    fdma_wstart_locked <= 1'b0;
  else if(fdma_wstart)
    fdma_wstart_locked <= 1'b1;

//产生 fdma_wstart 信号, 整个信号保持 1 个 M_AXI_ACLK 时钟周期

assign fdma_wstart = (fdma_wstart_locked == 1'b0 && fdma_wareq == 1'b1);

//AXI4 write burst lenth busrt addr -----
//当 fdma_wstart 信号有效, 代表一次新的 FDMA 传输, 首先把地址本次 fdma 的 burst 地址寄存到 axi_awaddr 作为第一次 axi burst 的地址。如果 fdma 的数据长度大于 256, 那么当 axi_wlast 有效的时候, 自动计算下次 axi 的 burst 地址

always @(posedge M_AXI_ACLK)
  if(fdma_wstart)
    axi_awaddr <= fdma_waddr;
  else if(axi_wlast == 1'b1)
    axi_awaddr <= axi_awaddr + axi_wburst_size ;

//AXI4 write cycle -----
axi_wstart_locked_r1, axi_wstart_locked_r2 信号是用于时序同步

reg axi_wstart_locked_r1 = 1'b0, axi_wstart_locked_r2 = 1'b0;
always @(posedge M_AXI_ACLK)begin
  axi_wstart_locked_r1 <= axi_wstart_locked;
  axi_wstart_locked_r2 <= axi_wstart_locked_r1;
end

// axi_wstart_locked 的作用代表一次 axi 写 burst 操作正在进行中。

always @(posedge M_AXI_ACLK)
  if((fdma_wstart_locked == 1'b1) && axi_wstart_locked == 1'b0)
    axi_wstart_locked <= 1'b1;
  else if(axi_wlast == 1'b1 || fdma_wstart == 1'b1)
    axi_wstart_locked <= 1'b0;

//AXI4 addr valid and write addr-----
always @(posedge M_AXI_ACLK)
  if((axi_wstart_locked_r1 == 1'b1) && axi_wstart_locked_r2 == 1'b0)
    axi_awvalid <= 1'b1;
  else if((axi_wstart_locked == 1'b1 && M_AXI_AWREADY == 1'b1)|| axi_wstart_locked == 1'b0)
    axi_awvalid <= 1'b0;

//AXI4 write data-----
always @(posedge M_AXI_ACLK)

```

```

if((axi_wstart_locked_r1 == 1'b1) && axi_wstart_locked_r2 == 1'b0)
    axi_wvalid <= 1'b1;
else if(axi_wlast == 1'b1 || axi_wstart_locked == 1'b0)
    axi_wvalid <= 1'b0;//

//AXI4 write data burst len counter-----
always @(posedge M_AXI_ACLK)
    if(axi_wstart_locked == 1'b0)
        wburst_cnt <= 'd0;
    else if(w_next)
        wburst_cnt <= wburst_cnt + 1'b1;

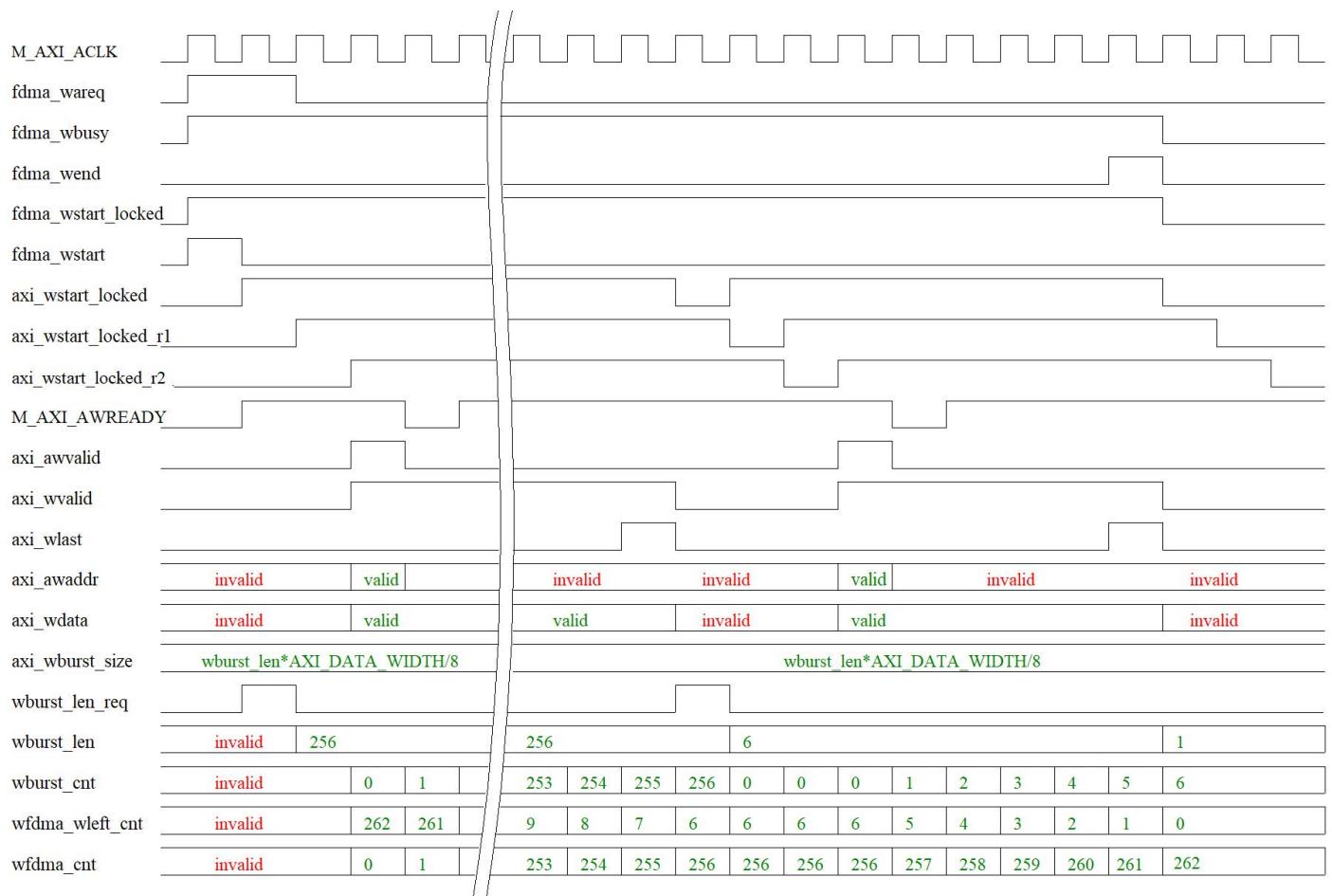
assign axi_wlast = (w_next == 1'b1) && (wburst_cnt == M_AXI_AWLEN);
//fdma write data burst len counter-----
reg wburst_len_req = 1'b0;
reg [15:0] fdma_wleft_cnt =16'd0;

// wburst_len_req 信号是自动管理每次 axi 需要 burst 的长度
always @(posedge M_AXI_ACLK)
    wburst_len_req <= fdma_wstart|axi_wlast;

// fdma_wleft_cnt 用于记录一次 FDMA 剩余需要传输的数据数量
always @(posedge M_AXI_ACLK)
    if( fdma_wstart )begin
        wfdma_cnt <= 1'd0;
        fdma_wleft_cnt <= fdma_wsize;
    end
    else if(w_next)begin
        wfdma_cnt <= wfdma_cnt + 1'b1;
        fdma_wleft_cnt <= (fdma_wsize - 1'b1) - wfdma_cnt;
    end
//当最后一个数据的时候，产生 fdma_wend 信号代表本次 fdma 传输结束
assign fdma_wend = w_next && (fdma_wleft_cnt == 1 );
//一次 axi 最大传输的长度是 256 因此当大于 256，自动拆分多次传输
always @(posedge M_AXI_ACLK)begin
    if(M_AXI_ARESETN == 1'b0)begin
        wburst_len <= 1;
    end
    else if(wburst_len_req)begin
        if(fdma_wleft_cnt[15:MAX_BURST_LEN_SIZE] >0)
            wburst_len <= M_AXI_MAX_BURST_LEN;
        else
            wburst_len <= fdma_wleft_cnt[MAX_BURST_LEN_SIZE-1:0];
    end
    else wburst_len <= wburst_len;
end

```

以上代码我们进行了详细的注释性分析。以下给出 FDMA 写操作源码部分的时序图。下图中一次传输以传输 262 个长度的数据为例，如果需要 MAX_BURST_LEN_SIZE 设置了最大值 256 看，那么 2 次 AXI4 BURST 才能完成，第一次传输 256 个长度数据，第二次传输 6 个长度的数据。



4:FDMA 的 AXI4-Master 读操作

以下代码中我们给出 axi4-master 读操作的代码分析注释

```
//fdma axi read-----
reg      [M_AXI_ADDR_WIDTH-1 : 0]      axi_araddr =0 ; //AXI4 读地址
reg                  axi_arvalid =1'b0; //AXI4 读地有效
wire                 axi_rlast ; //AXI4 读 LAST 信号
reg                  axi_rready = 1'b0;AXI4 读准备好
wire                  r_next      = (M_AXI_RVALID & M_AXI_RREADY); // 当 valid ready 信号都有效, 代表 AXI4 数据传输有效
reg [8 :0]            rburst_len = 1 ; //读传输的 axi burst 长度, 代码会自动计算每次 axi 传输的 burst 长度
reg [8 :0]            rburst_cnt = 0 ; /每次 axi bust 的计数器
reg [15:0]           rfdma_cnt = 0 ; //fdma 的读数据计数器
reg                  axi_rstart_locked =0; //axi 传输进行中, lock 住, 用于时序控制
wire [15:0] axi_rburst_size = rburst_len * AXI_BYTES; //axi 传输的地址长度计算

assign M_AXI_ARID      = M_AXI_ID; //读地址 ID, 用来标志一组写信号, M_AXI_ID 是通过参数接口定义
assign M_AXI_ARADDR     = axi_araddr;
assign M_AXI_ARLEN      = rburst_len - 1; //AXI4 burst 的长度
assign M_AXI_ARSIZE     = clogb2((AXI_BYTES)-1);
assign M_AXI_ARBURST   = 2'b01; //AXI4 的 busr 类型 INCR 模式, 地址递增
assign M_AXI_ARLOCK     = 1'b0; //不使用 cache, 不使用 buffer
assign M_AXI_ARCACHE    = 4'b0010;
assign M_AXI_ARPROT     = 3'h0;
assign M_AXI_ARQOS      = 4'h0;
```

```

assign M_AXI_ARVALID      = axi_arvalid;
assign M_AXI_RREADY       = axi_rready&&fdma_rready; //读数据准备好, 这里必须设置 fdma_rready 有效
assign fdma_rdata         = M_AXI_RDATA;
assign fdma_rvalid        = r_next;

//AXI4 FULL Read-----

reg     fdma_rstart_locked = 1'b0;
wire   fdma_rend;
wire   fdma_rstart;
assign  fdma_rbusy = fdma_rstart_locked ;
//在整个读过程中 fdma_rstart_locked 将保持有效, 直到本次 FDMA 写结束
always @(posedge M_AXI_ACLK)
  if(M_AXI_ARESETN == 1'b0 || fdma_rend == 1'b1)
    fdma_rstart_locked <= 1'b0;
  else if(fdma_rstart)
    fdma_rstart_locked <= 1'b1;
//产生 fdma_rstart 信号, 整个信号保持 1 个 M_AXI_ACLK 时钟周期
assign fdma_rstart = (fdma_rstart_locked == 1'b0 && fdma_rareq == 1'b1);

//AXI4 read burst lenth busrt addr -----
//当 fdma_rstart 信号有效, 代表一次新的 FDMA 传输, 首先把地址本次 fdma 的 burst 地址寄存到 axi_araddr 作为第一次 axi burst 的地址。如果 fdma 的数据长度大于 256, 那么当 axi_rlast 有效的时候, 自动计算下次 axi 的 burst 地址
always @(posedge M_AXI_ACLK)
  if(fdma_rstart == 1'b1)
    axi_araddr <= fdma_raddr;
  else if(axi_rlast == 1'b1)
    axi_araddr <= axi_araddr + axi_rburst_size ;
//AXI4 r_cycle_flag-----
//axi_rstart_locked_r1, axi_rstart_locked_r2 信号是用于时序同步
reg axi_rstart_locked_r1 = 1'b0, axi_rstart_locked_r2 = 1'b0;
always @(posedge M_AXI_ACLK)begin
  axi_rstart_locked_r1 <= axi_rstart_locked;
  axi_rstart_locked_r2 <= axi_rstart_locked_r1;
end
// axi_rstart_locked 的作用代表一次 axi 读 burst 操作正在进行中。
always @(posedge M_AXI_ACLK)
  if((fdma_rstart_locked == 1'b1) && axi_rstart_locked == 1'b0)
    axi_rstart_locked <= 1'b1;
  else if(axi_rlast == 1'b1 || fdma_rstart == 1'b1)
    axi_rstart_locked <= 1'b0;
//AXI4 addr valid and read addr-----
always @(posedge M_AXI_ACLK)
  if((axi_rstart_locked_r1 == 1'b1) && axi_rstart_locked_r2 == 1'b0)
    axi_arvalid <= 1'b1;
  else if((axi_rstart_locked == 1'b1 && M_AXI_ARREADY == 1'b1)|| axi_rstart_locked == 1'b0)
    axi_arvalid <= 1'b0;
//AXI4 read data-----
always @(posedge M_AXI_ACLK)
  if((axi_rstart_locked_r1 == 1'b1) && axi_rstart_locked_r2 == 1'b0)

```

```

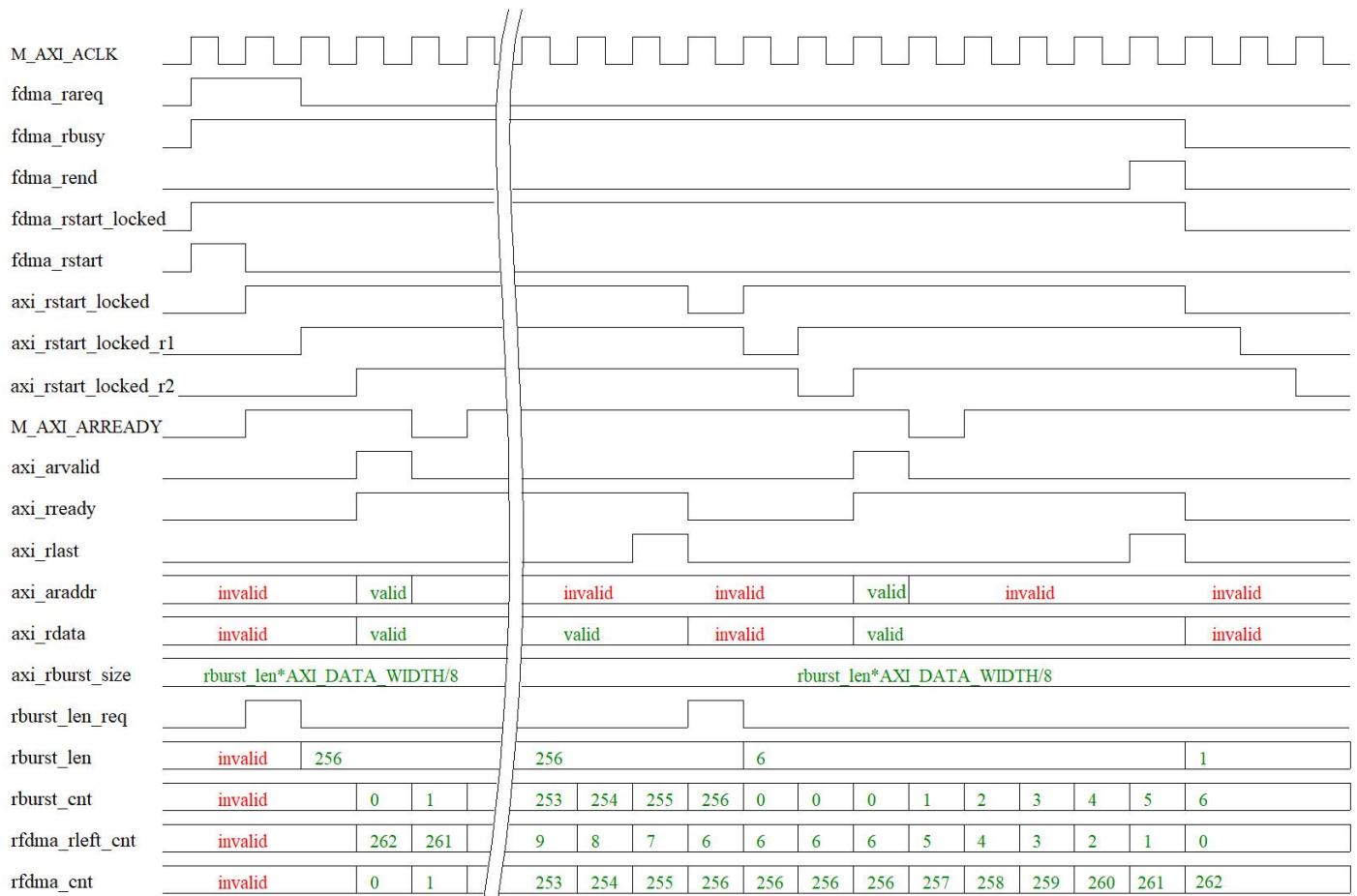
    axi_rready <= 1'b1;
  else if(axi_rlast == 1'b1 || axi_rstart_locked == 1'b0)
    axi_rready <= 1'b0;//

//AXI4 read data burst len counter-----
always @(posedge M_AXI_ACLK)
  if(axi_rstart_locked == 1'b0)
    rburst_cnt <= 'd0;
  else if(r_next)
    rburst_cnt <= rburst_cnt + 1'b1;
assign axi_rlast = (r_next == 1'b1) && (rburst_cnt == M_AXI_ARLEN);
//fdma read data burst len counter-----
reg rburst_len_req = 1'b0;
reg [15:0] fdma_rleft_cnt =16'd0;
// rburst_len_req 信号是自动管理每次 axi 需要 burst 的长度
always @(posedge M_AXI_ACLK)
  rburst_len_req <= fdma_rstart | axi_rlast;
// fdma_rleft_cnt 用于记录一次 FDMA 剩余需要传输的数据数量
always @(posedge M_AXI_ACLK)
  if(fdma_rstart )begin
    rfdma_cnt <= 1'd0;
    fdma_rleft_cnt <= fdma_rsize;
  end
  else if(r_next)begin
    rfdma_cnt <= rfdma_cnt + 1'b1;
    fdma_rleft_cnt <= (fdma_rsize - 1'b1) - rfdma_cnt;
  end
//当最后一个数据的时候，产生 fdma_rend 信号代表本次 fdma 传输结束
assign fdma_rend = r_next && (fdma_rleft_cnt == 1 );
//axi auto burst len caculate-----
//一次 axi 最大传输的长度是 256 因此当大于 256，自动拆分多次传输
always @(posedge M_AXI_ACLK)begin
  if(M_AXI_ARESETN == 1'b0)begin
    rburst_len <= 1;
  end
  else if(rburst_len_req)begin
    if(fdma_rleft_cnt[15:MAX_BURST_LEN_SIZE] >0)
      rburst_len <= M_AXI_MAX_BURST_LEN;
    else
      rburst_len <= fdma_rleft_cnt[MAX_BURST_LEN_SIZE-1:0];
  end
  else rburst_len <= rburst_len;
end

```

以上代码我们进行了详细的注释性分析。FDMA 的读写代码高度对称，以上源码和以下波形图都和写操作类似，理解起会提高很多效率。

以下给出 FDMA 读操作源码部分的时序图。下图中一次传输以传输 262 个长度的数据为例，如果需要 MAX_BURST_LEN_SIZE 设置了最大值 256 看，那么 2 次 AXI4 BURST 才能完成，第一次传输 256 个长度数据，第二次传输 6 个长度的数据。



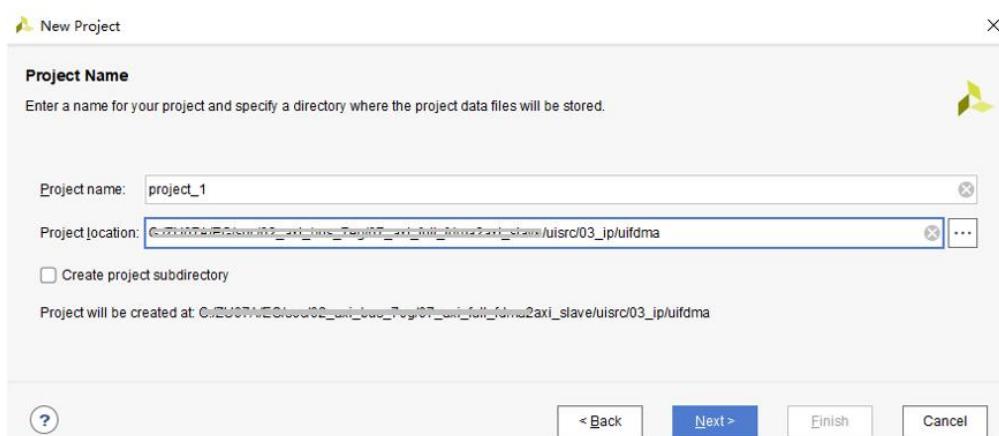
1.4 FDMA IP 的封装

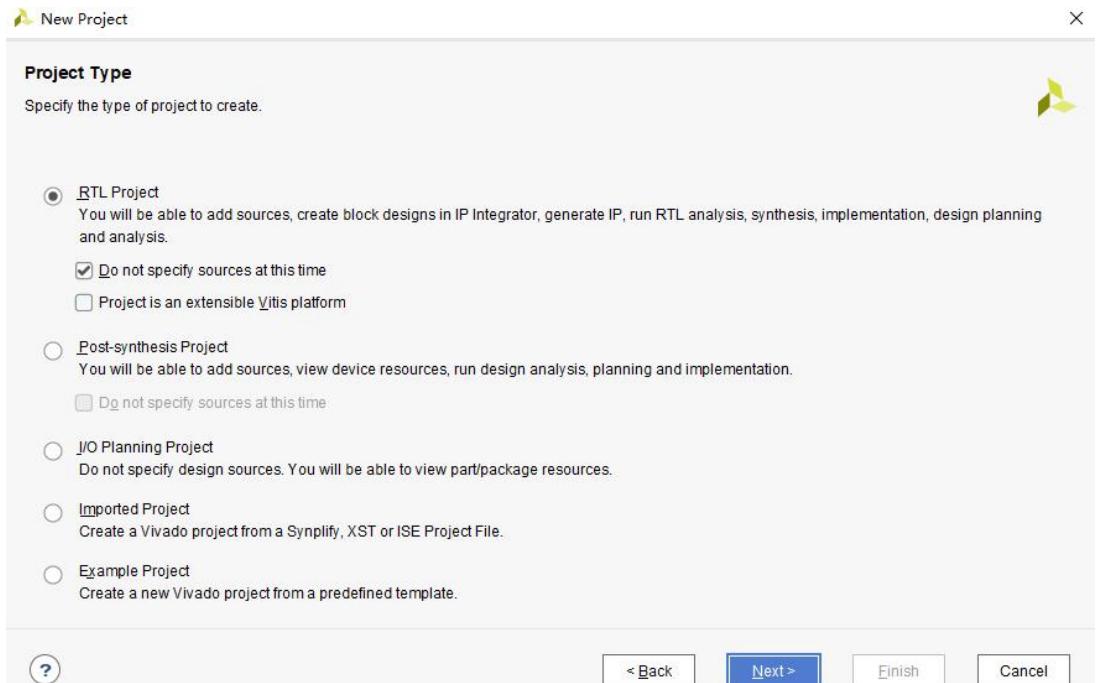
我先讲解如何封装 FDMA IP,之后再分析源码。封装 IP 少不了源码, 这里是利用已经编写好的 uiFDMA.v 进行封装。默认的源码路径在配套的工程 uisrc/uifdma 路径下

> ZU07A -> EC -> 000 -> 02_axi_boss_7sg -> 07_axi_full_fdma2axi_slave -> uisrc -> 03_ip -> uifdma

名称	修改日期	类型	大小
uiFDMA.v	2021/6/9 19:13	Verilog File	14 KB

创建一个新的空的 fpga 工程





Select Device

Filter, search, and browse parts by their resources. The selection will be applied.

Parts | Boards

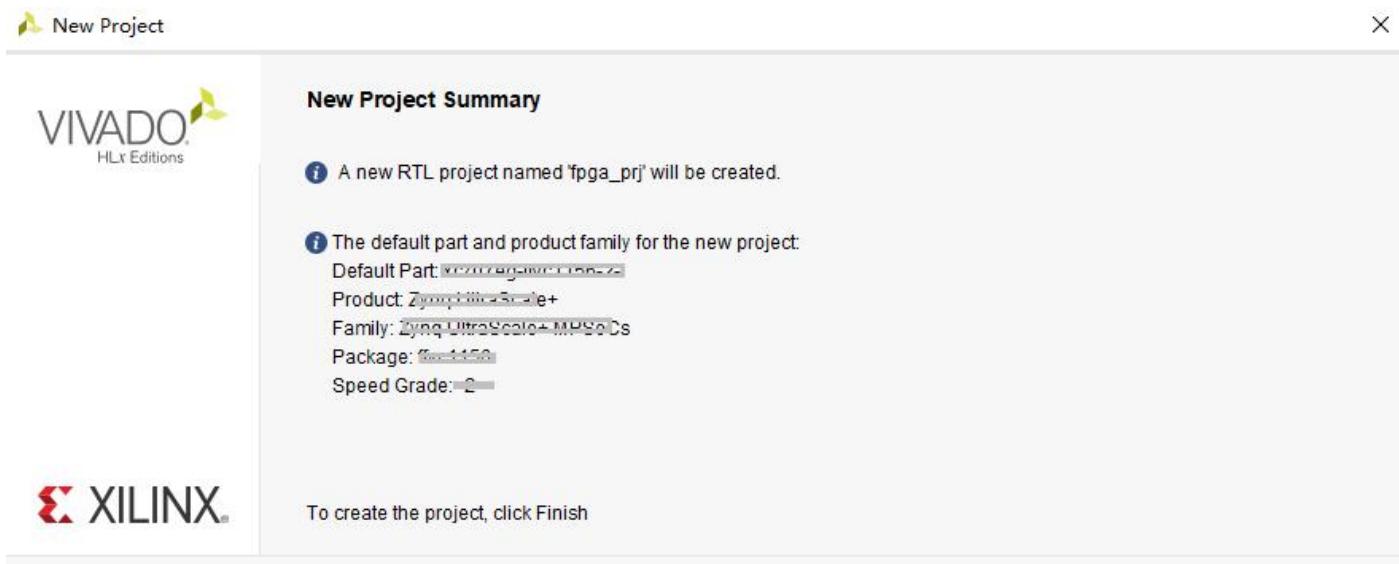
[Reset All Filters](#)

Category:	All	Package:	ffg676	Temperature:	All Remaining
Family:	Zynq-7000	Speed:	-2	Static power:	All Remaining

Search:

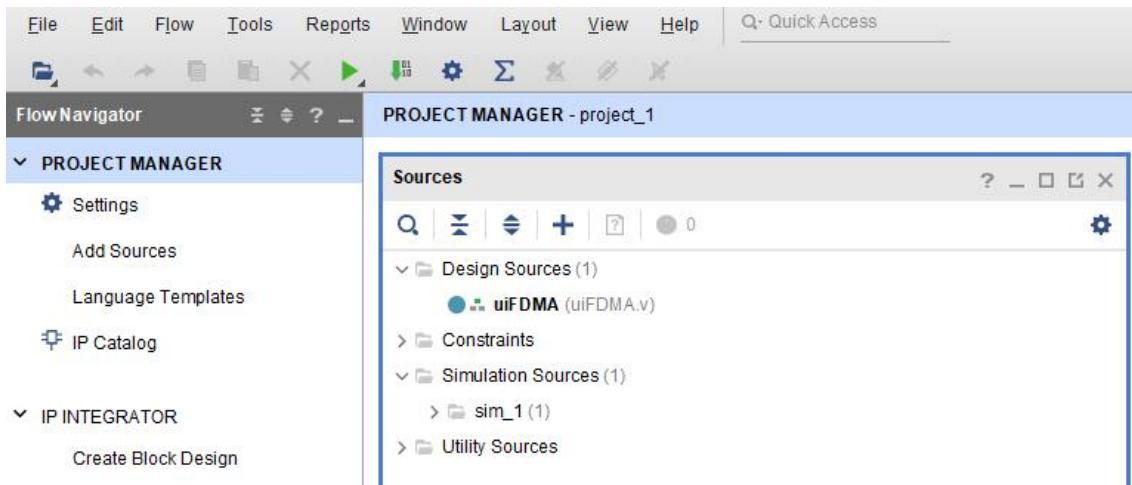
Part	I/O Pin C...	Available I...	LUT Ele...	FlipFlops	Block R...	Ultra R...	DSPs
xc7z030ffg676-2	676	250	78600	157200	265	0	400
xc7z035ffg676-2	676	250	171900	343800	500	0	900
xc7z045ffg676-2	676	250	218600	437200	545	0	900

? < > OK Cancel

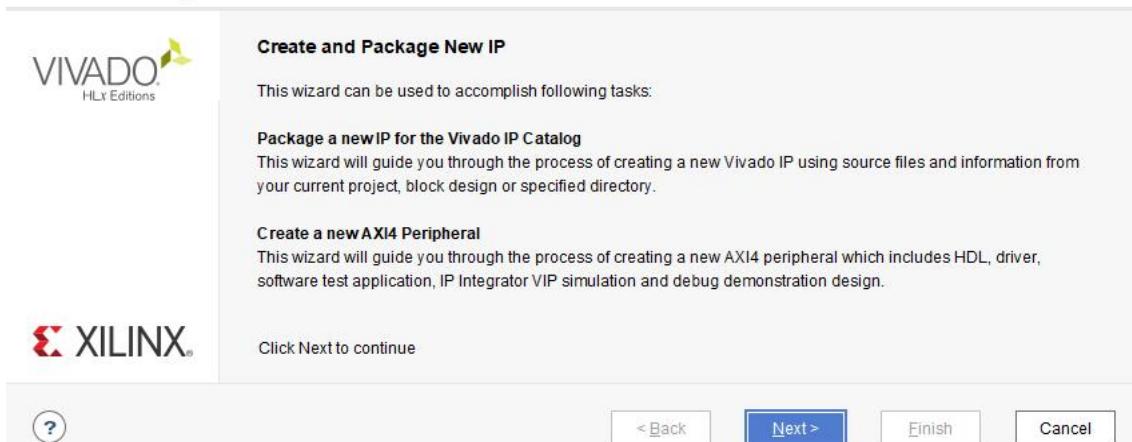
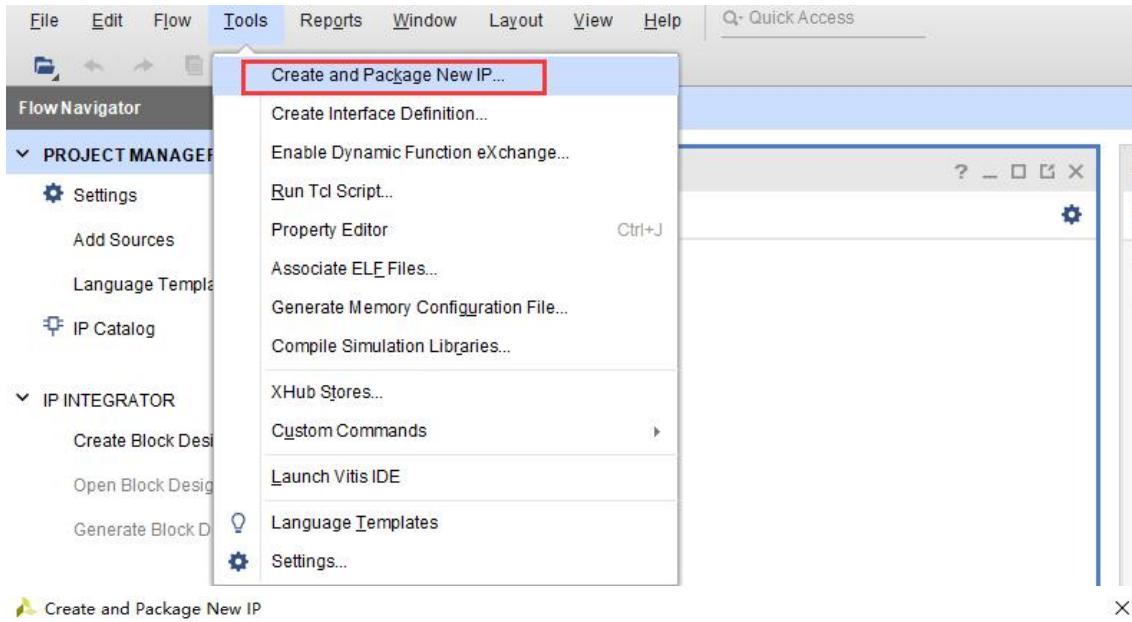


添加 uiFDMA.v 源码

The screenshot shows the Vivado IP Catalog interface. On the left, the 'PROJECT MANAGER' sidebar is open, with 'Add Sources' highlighted by a red box and a red arrow pointing to the 'Add Sources' dialog window. The main area displays the 'Sources' tab under 'Design Sources', showing a tree view with 'Constraints', 'Simulation Sources', and 'Utility Sources'. A sub-tree for 'sim_1' is expanded. The 'Add Sources' dialog is overlaid on the main window, titled 'Add Sources'. It contains instructions: 'This guides you through the process of adding and creating sources for your project'. Three radio buttons are shown: 'Add or create constraints' (unselected), 'Add or create design sources' (selected), and 'Add or create simulation sources' (unselected). Below the dialog is the Xilinx logo. At the bottom of the dialog are buttons for '?', '< Back', 'Next >', 'Finish', and 'Cancel'. The 'Add or Create Design Sources' sub-dialog is also visible, showing a table with columns 'Index', 'Name', 'Library', and 'Location'. A row for 'uiFDMA.v' is selected and highlighted with a red box. The 'Location' column shows a long path starting with 'C:/Vivado/...'. At the bottom of this sub-dialog are buttons for '+', 'Index', 'Name', 'Library', 'Location', 'Add Files', 'Add Directories', and 'Create File'. Below the table are three checkboxes: 'Scan and add RTL include files into project' (unchecked), 'Copy sources into project' (unchecked), and 'Add sources from subdirectories' (checked). The 'Finish' button at the bottom right of the sub-dialog is also highlighted with a red box.



创建 IP



选择 Package your current project

 Create and Package New IP 

Create Peripheral, Package IP or Package a Block Design

Please select one of the following tasks.



Packaging Options

- Package your current project
Use the project as the source for creating a new IP Definition.
- Package a block design from the current project
Choose a block design as the source for creating a new IP Definition.
- Package a specified directory
Choose a directory as the source for creating a new IP Definition.

Create AXI4 Peripheral

- Create a new AXI4 peripheral
Create an AXI4 IP, driver, software test application, IP Integrator AXI4 VIP simulation and debug demonstration design.

 Create and Package New IP 

Package Your Current Project

Select the directory where the IP Definition will be created and the associated options for packaging the current project.



IP location:  

 Create and Package New IP 



New IP Creation

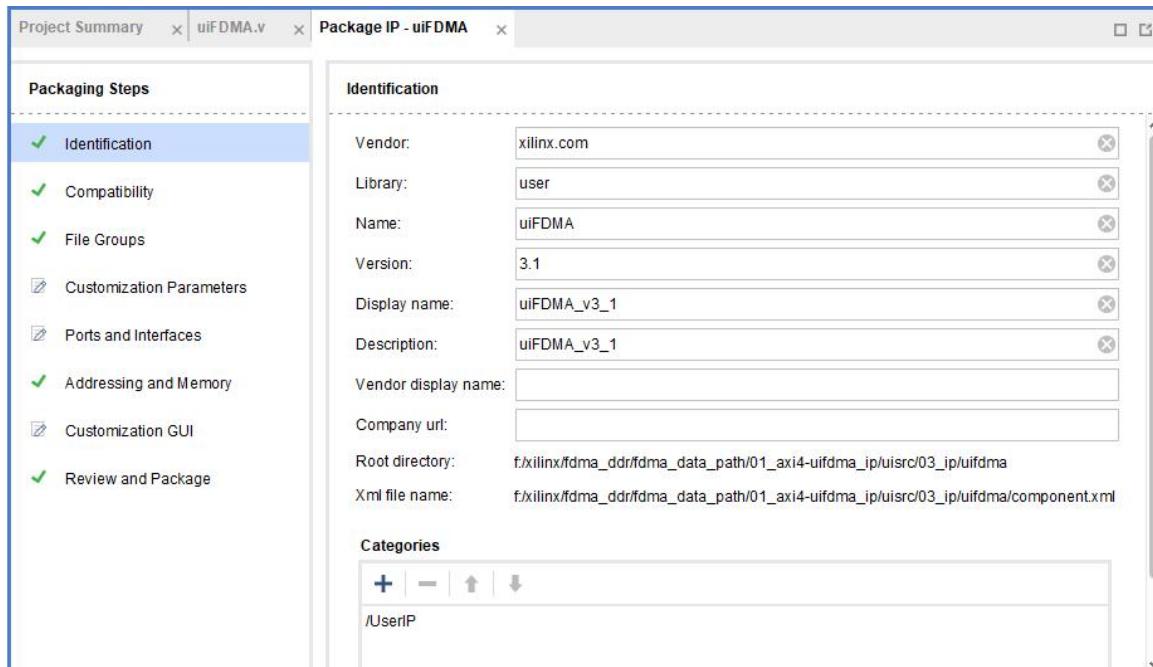
The following pieces of information will be gathered:

- Identification information based on top module name
- Family compatibility based on part in the project
- File(s) from Synthesis and Simulation file sets
- Ports from the file containing the top module
- Parameters from the file containing the top module
- Bus Interfaces based on port names
- Address Spaces and Memory Maps based on inferred bus interfaces

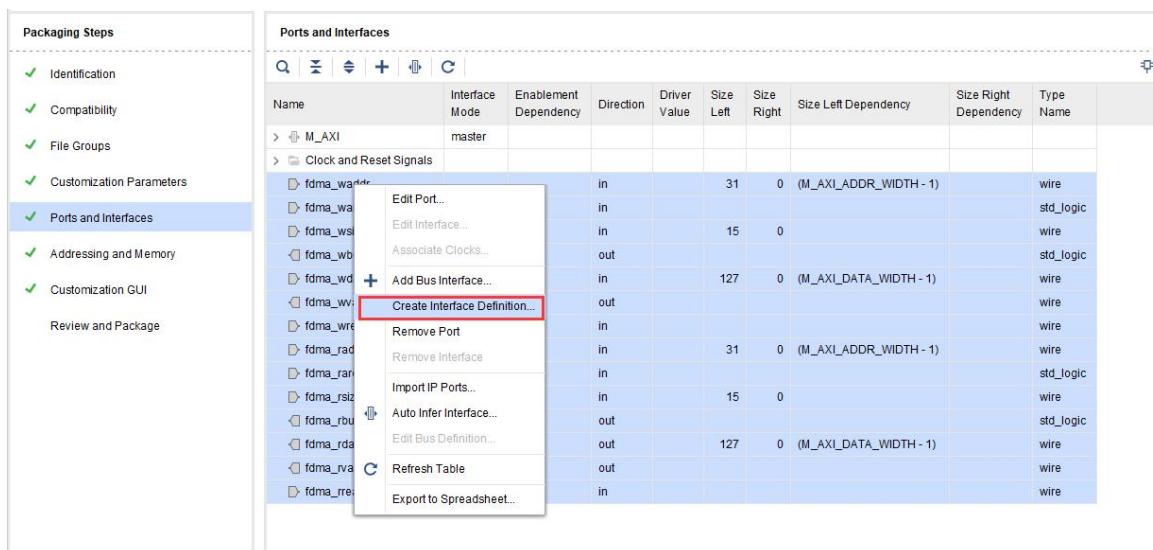
Following file will be created on disk along with corresponding customization files:
g:/zu07a/eg/soc/02_axi_bus_7eg/07_axi_full_fdma2axi_slave/uisrc/03_ip/uifdma/component.xml

Click Finish to continue

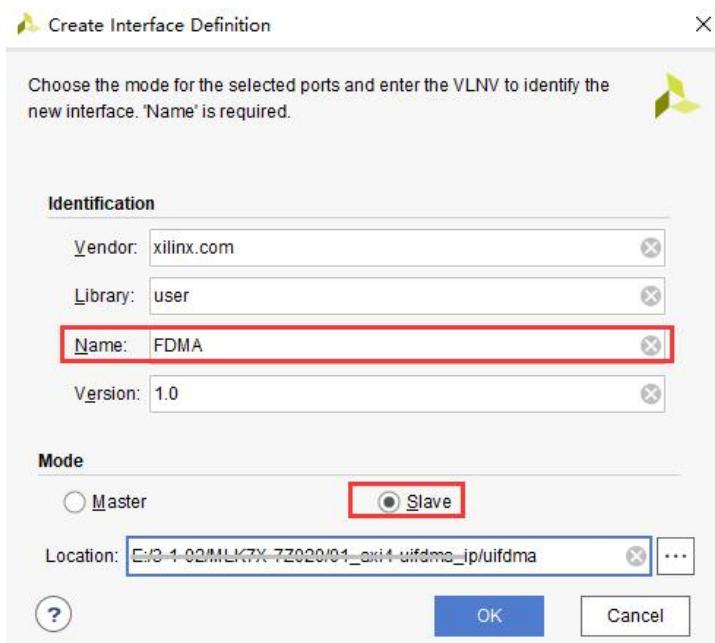
    



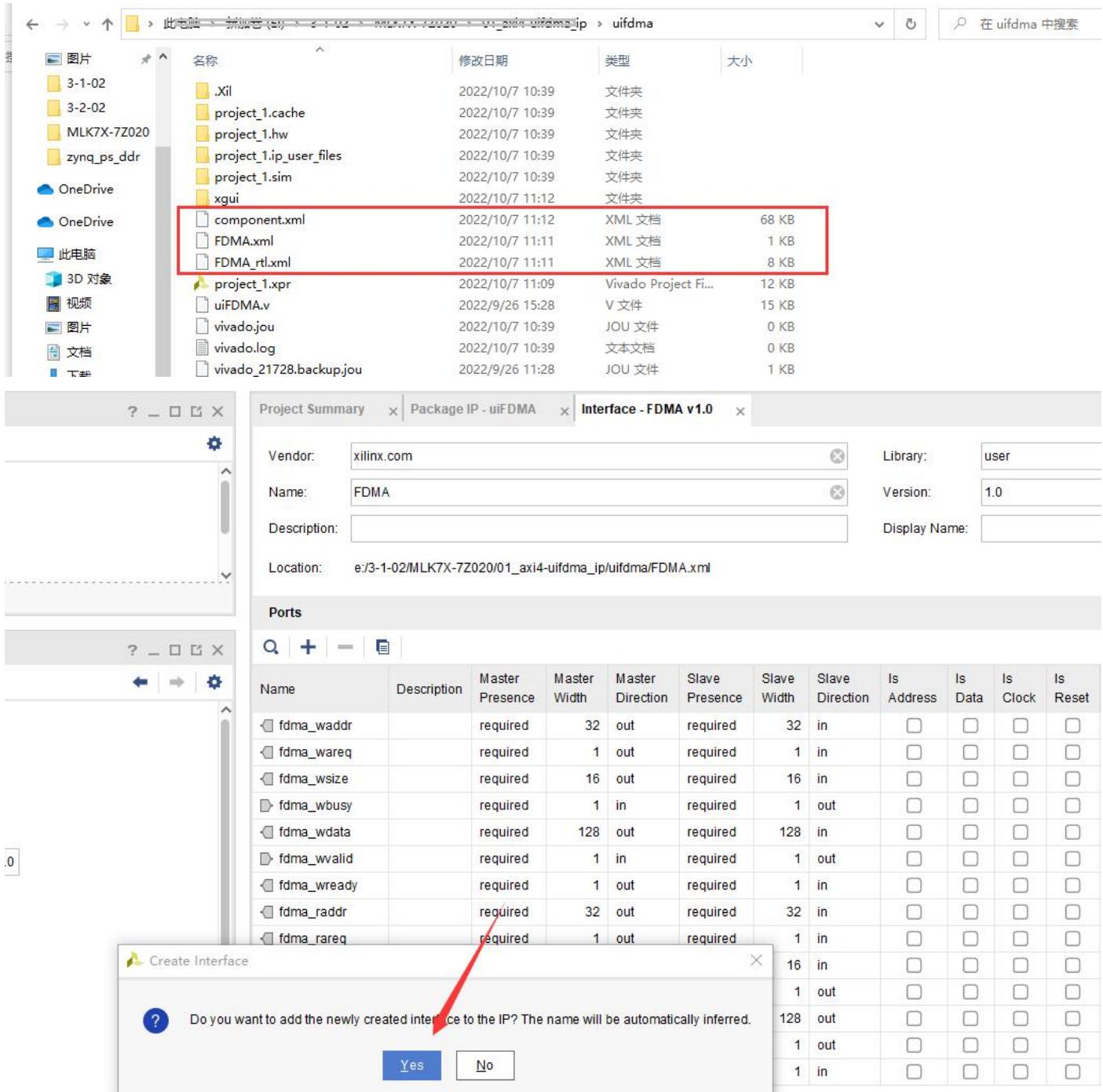
按住 shift 全选后，右击弹出菜单后选择 Create Interface Definition



接口定义为 slave，命名为 FDMA



设置完成，uisrc/03_ip/uifdma 路径下多出 2 个文件，这个两个文件就是定义了自定义的总线接口。



现在可以看到封装后的总线

The screenshot shows the Project Manager interface for a project named "project_1".

- Sources:** Shows two design sources: "uiFDMA (uiFDMA.v)" and "IP-XACT (1) component.xml".
- IP Bus Interface Properties:** For "FDMA_1", the Name is set to "FDMA_1".
- Ports and Interfaces:** A table lists the ports and interfaces:

Name	Interface Mode	Enablemen t Depend...	Direction	Driver Value	Size Left	Size Right	Size Left Depend...	Size Right Depend...	Type Name
> M_AXI	master								
> FDMA_1	slave								
> Clock and Reset Signals									

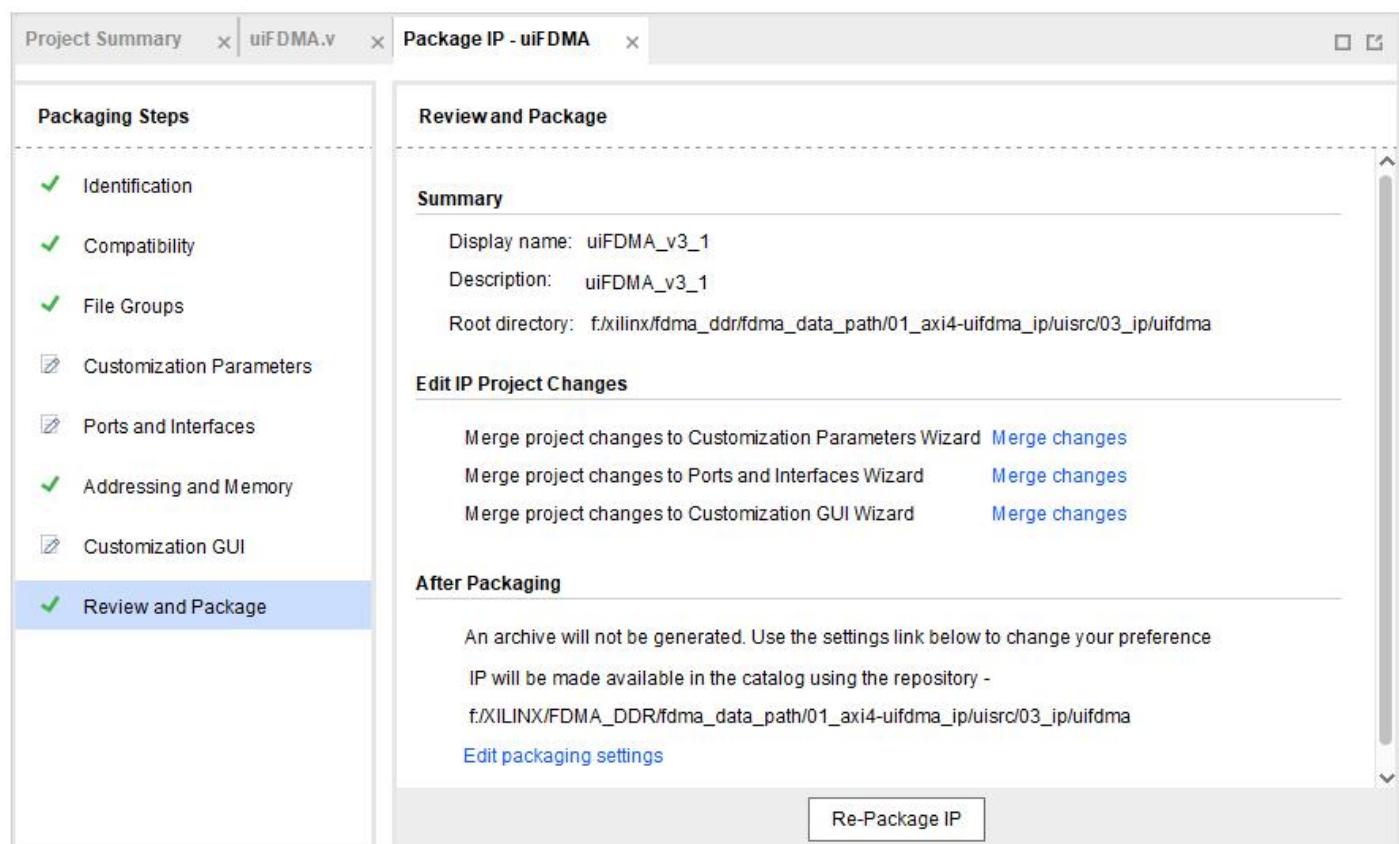
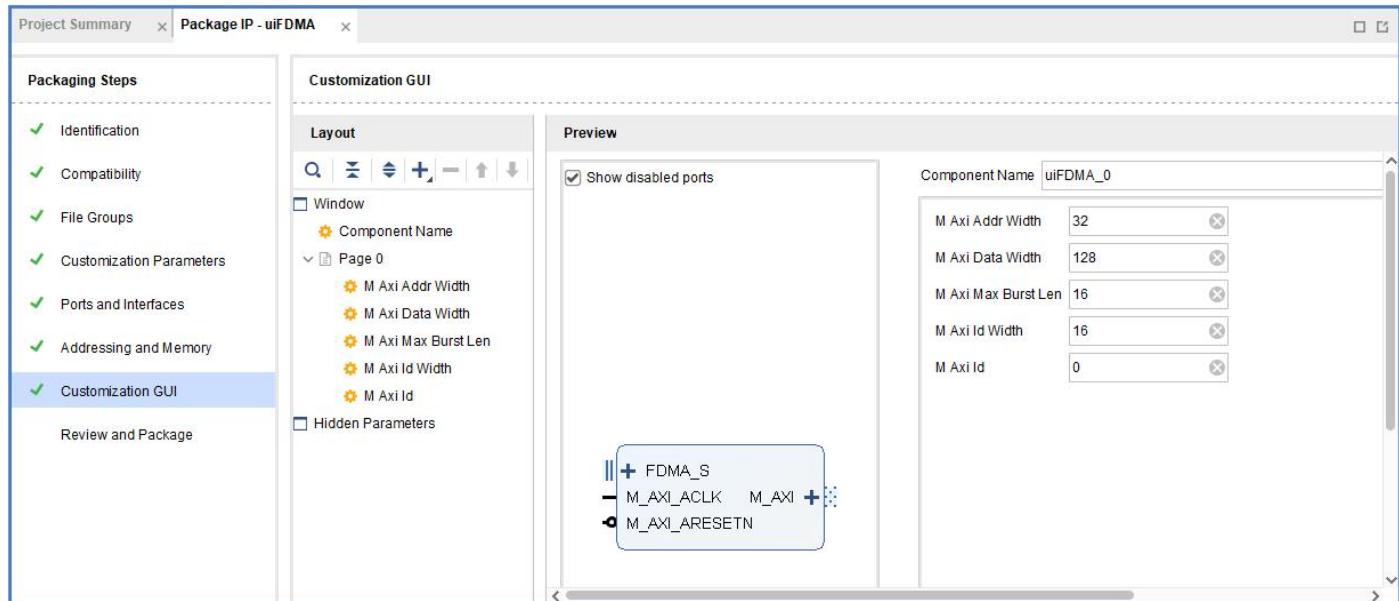
建议把名字改简洁一些

The screenshot shows the Project Manager interface for the same project.

- Sources:** Same as before.
- IP Bus Interface Properties:** For "FDMA_S", the Name is now set to "FDMA_S".
- Ports and Interfaces:** A table lists the ports and interfaces:

Name	Interface Mode	Enablemen t Depend...	Direction	Driver Value	Size Left	Size Right	Size Depen...
> M_AXI	master						
> FDMA_S	slave						
> Clock and Reset Signals							

可以看到封装好的接口，更加美观



02AXI4-FULL-uiFDMA IP 仿真验证

软件版本：vitis2021.1(vivado2021.1)

操作系统：WIN10 64bit

硬件平台：适用 XILINX A7/K7/Z7/ZU/KU 系列 FPGA

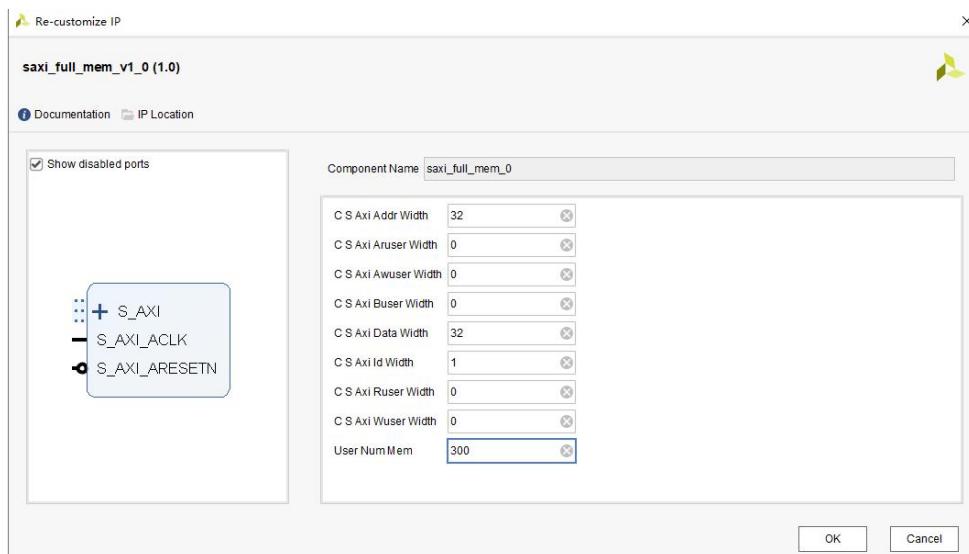
登录“米联客”FPGA 社区-www.uisrc.com 视频课程、答疑解惑！

2.1 概述

本文试验中对前面编写的 FDMA IP 进行仿真验证。

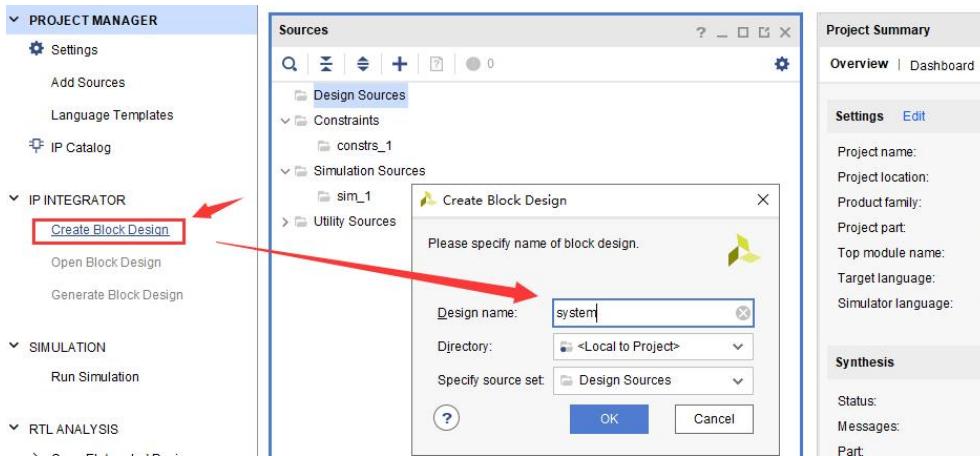
2.1saxi_full_mem IP 介绍

这个 IP 的源码可以基于 XILINX 提供的 axi-full-slave 的模板简单修改就可以实现，如果读者想要更加详细的学习 AXI 总线相关内容，可以阅读或者观看 “米联客 2022 版 AXI4 总线专题篇” 相关课程内容。本文实验使用我们已经修改好的代码来完成验证。

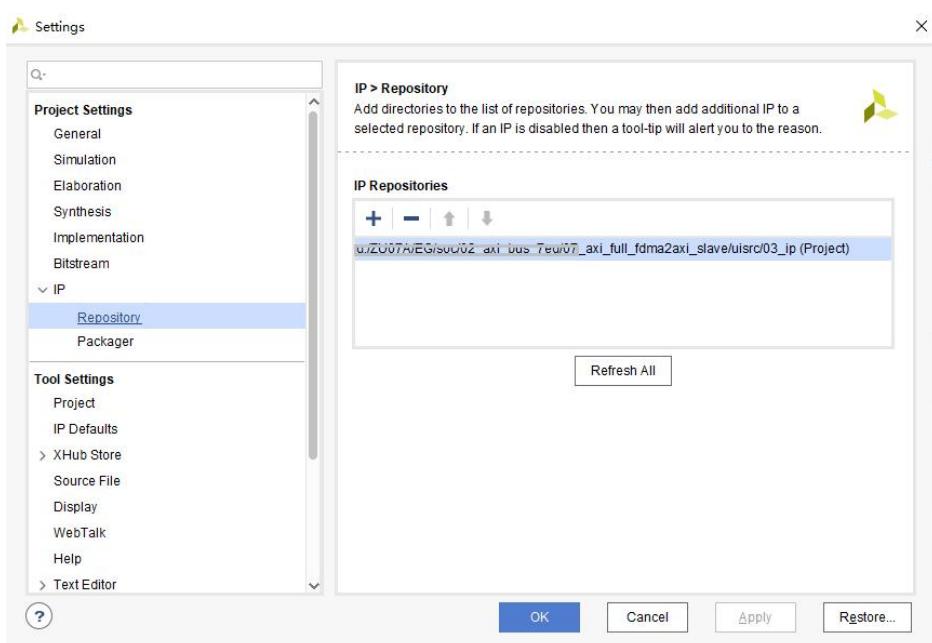
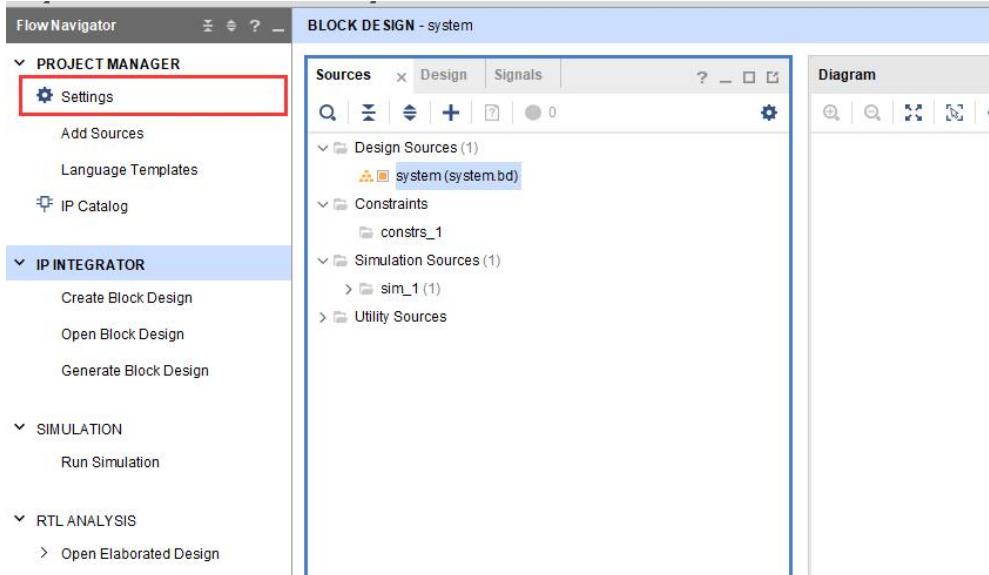


这个 IP 后面可以用于 AXI4 总线的仿真验证

2.2 创建 FPGA 逻辑工程



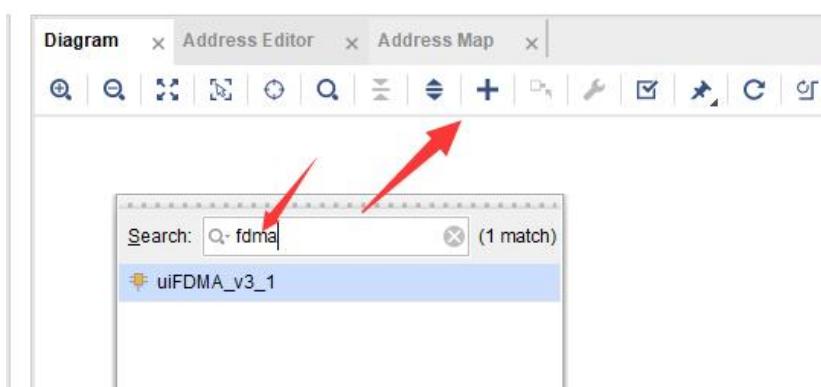
设置 IP 路径



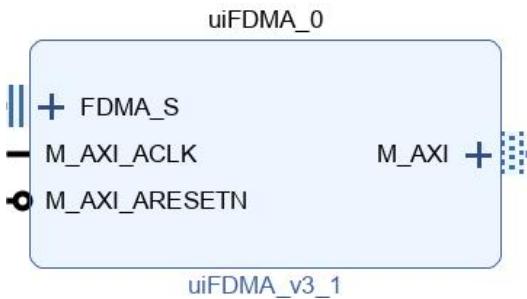
添加已经创建好的 IP



输入关键词 fdma，在最后可以看到，双击添加 Ip

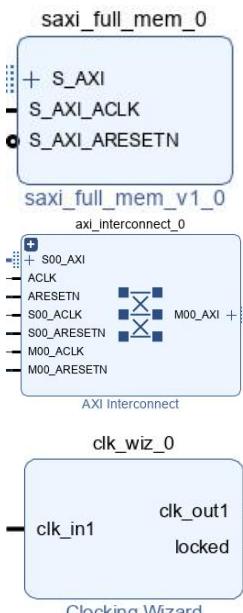


可以看到本文的 FDMA 版本升级到 3.1 版



完成连线

继续添加剩余 IP



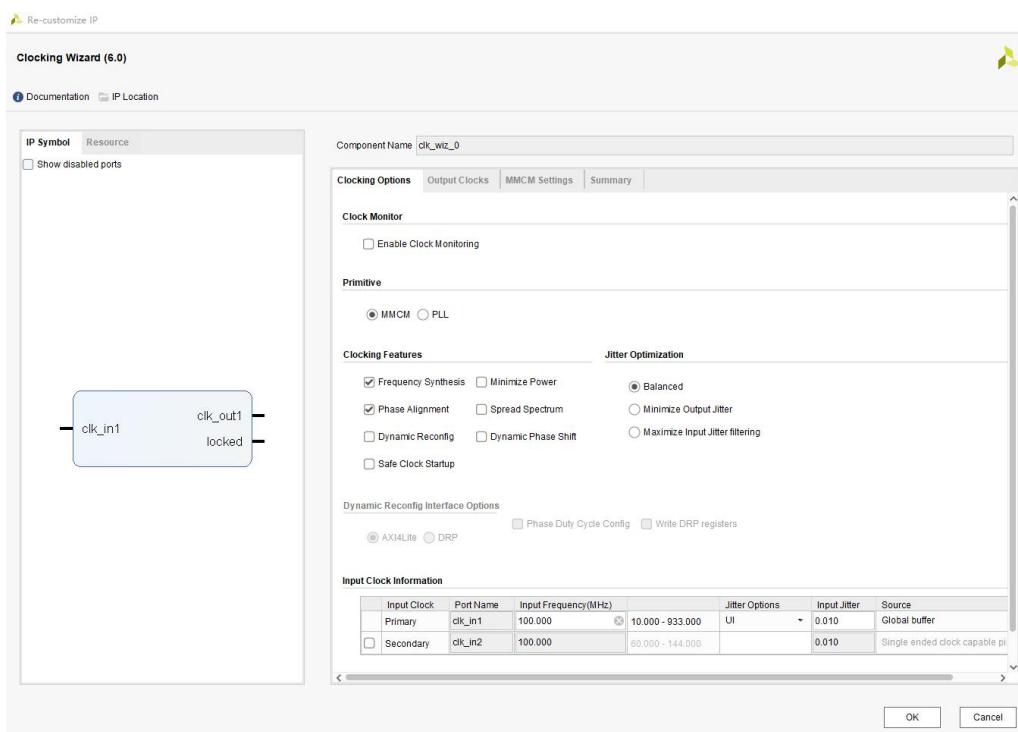
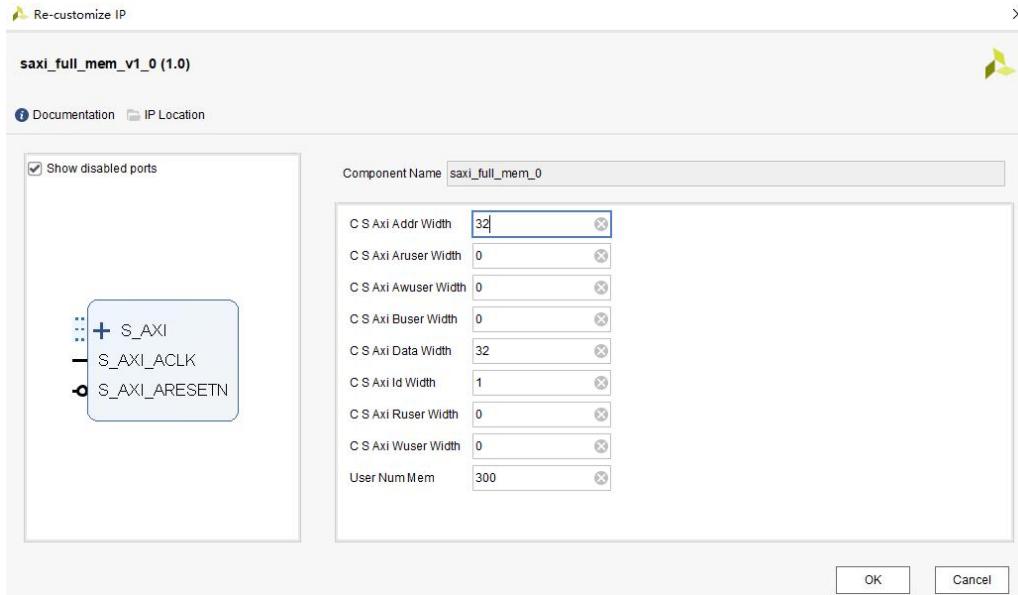
设置 IP 参数

uiFDMA_v3_1 (3.1)

Component Name: uiFDMA_0

Show disabled ports

M Axi Addr Width	32
M Axi Data Width	32
M Axi Max Burst Len	256
M Axi Id Width	1
M Axi Id	0



Component Name: clk_wiz_0

Clocking Options Output Clocks MMC Settings Summary

The phase is calculated relative to clk_out1.

Output Clock	Port Name	Output Freq (MHz)		Phase (degrees)		Duty Cycle (%)		Drives	Matched Routing	Max Freq. of buffer
		Requested	Actual	Requested	Actual	Requested	Actual			
<input checked="" type="checkbox"/> clk_out1	clk_out1	100.000	100.00000	0.000	0.000	50.000	50.0	Buffer	▼	<input type="checkbox"/> 775.194
<input type="checkbox"/> clk_out2	clk_out2	100.000	N/A	0.000	N/A	50.000	N/A	Buffer	▼	<input type="checkbox"/> 775.194
<input type="checkbox"/> clk_out3	clk_out3	100.000	N/A	0.000	N/A	50.000	N/A	Buffer	▼	<input type="checkbox"/> 775.194
<input type="checkbox"/> clk_out4	clk_out4	100.000	N/A	0.000	N/A	50.000	N/A	Buffer	▼	<input type="checkbox"/> 775.194
<input type="checkbox"/> clk_out5	clk_out5	100.000	N/A	0.000	N/A	50.000	N/A	Buffer	▼	<input type="checkbox"/> 775.194
<input type="checkbox"/> clk_out6	clk_out6	100.000	N/A	0.000	N/A	50.000	N/A	Buffer	▼	<input type="checkbox"/> 775.194
<input type="checkbox"/> clk_out7	clk_out7	100.000	N/A	0.000	N/A	50.000	N/A	Buffer	▼	<input type="checkbox"/> 775.194

USE CLOCK SEQUENCING

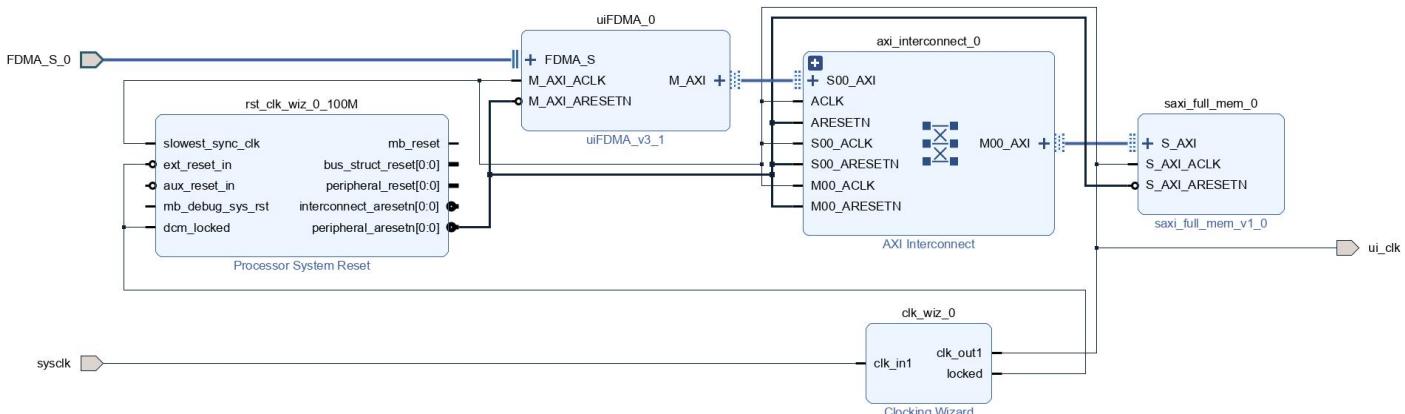
Output Clock	Sequence Number
clk_out1	1
clk_out2	1
clk_out3	1
clk_out4	1
clk_out5	1
clk_out6	1
clk_out7	1

Enable Optional Inputs / Outputs for MMCM/PLL

Reset Type: Active High (radio button selected), Active Low (radio button unselected), WAVEFORM (radio button unselected), LATENCY (radio button selected).

Locked: locked, clkfbstopped

完成连线



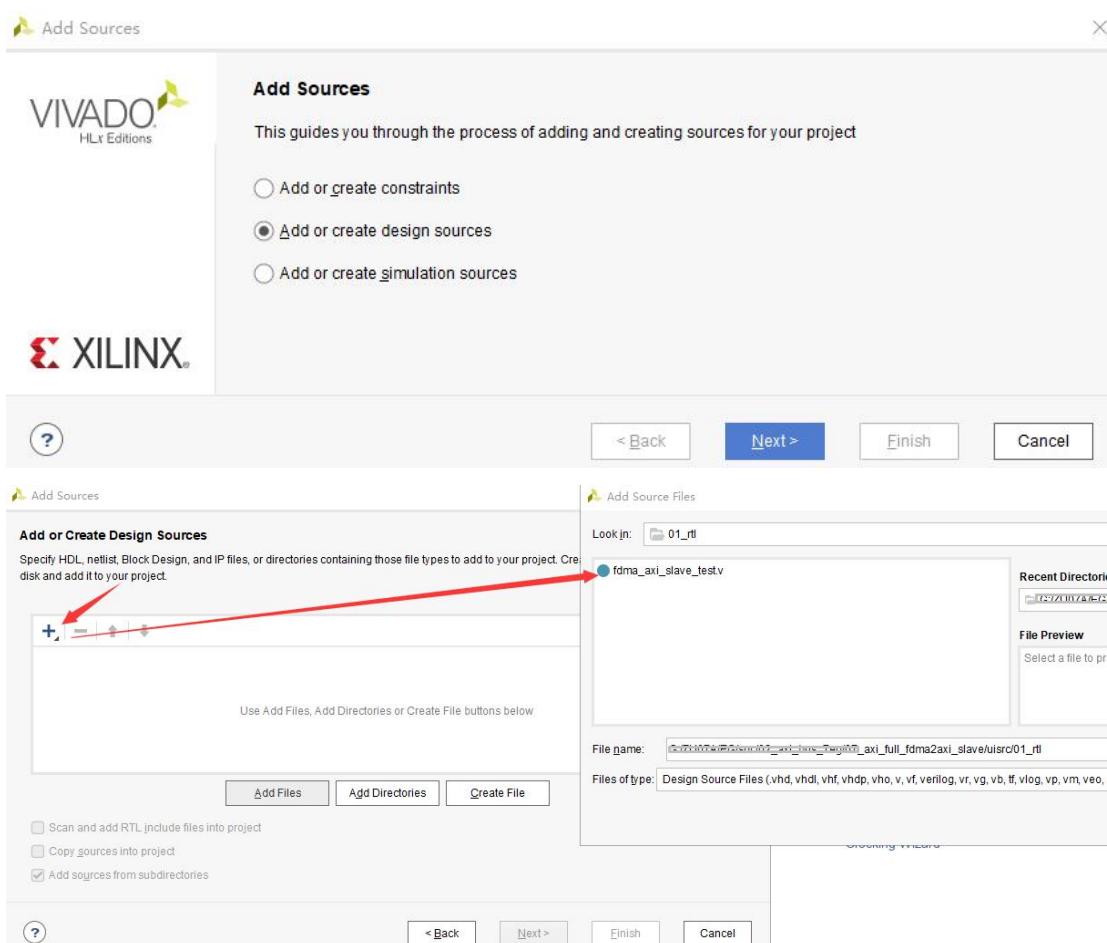
设置地址分配：

Diagram Address Editor Address Map

Assigned (1) Unassigned (0) Excluded (0) Hide All

Name	Interface	Slave Segment	Master Base Address	Range	Master High Address
Network 0					
/uiFDMA_0					
/uiFDMA_0/M_AXI (32 address bits : 4G)	S_AXI	reg0	0x0000_0000	4G	0xFFFF_FFFF
/saxi_full_mem_0/S_AXI					

2.3 添加 FDMA 接口控制代码



添加完成后如下图：



fdma_axi_slave_test.v 源码如下

```
/*
*****MILIANKE*****
*Company : MiLianKe Electronic Technology Co., Ltd.
*WebSite:https://www.milianke.com
*TechWeb:https://www.uisrc.com
*tmall-shop:https://milianke.tmall.com
*jd-shop:https://milianke.jd.com
```

```

*taobao-shop1: https://milianke.taobao.com
*Create Date: 2021/04/25
*Module Name:fdma_axi_slave_test
*File Name:fdma_axi_slave_test.v
*Description:
*The reference demo provided by Milianke is only used for learning.
*We cannot ensure that the demo itself is free of bugs, so users
*should be responsible for the technical problems and consequences
*caused by the use of their own products.
*Copyright: Copyright (c) MiLianKe
*All rights reserved.
*Revision: 1.0
*Signal description
*1) _i input
*2) _o output
*3) _n activ low
*4) _dg debug signal
*5) _r delay or register
*6) _s state machine
*****/
`timescale 1ns / 1ps

module fdma_axi_slave_test(
    input sysclk
);

wire [31:0] fdma_raddr;
reg         fdma_rreq;
wire         fdma_rbusy;
wire [31:0] fdma_rdata;
wire [15:0] fdma_rsize;
wire         fdma_rvalid;
wire [31:0] fdma_waddr;
reg         fdma_wreq;
wire         fdma_wbusy;
wire [31:0] fdma_wdata;
wire [15:0] fdma_wsize;
wire         fdma_wvalid;
wire         ui_clk;

parameter TEST_MEM_SIZE = 32'd20000; //测试内存的地址范围
parameter FDMA_BURST_LEN = 16'd500; //测试一次的长度
parameter ADDR_MEM_OFFSET = 0; //地址偏移量
parameter ADDR_INC = FDMA_BURST_LEN*4; //下一次 FDMA burst 的地址增加

parameter WRITE1 = 0;
parameter WRITE2 = 1;
parameter WAIT = 2;
parameter READ1 = 3;
parameter READ2 = 4;

```

```

reg [31: 0] t_data;
reg [31: 0] fdma_waddr_r;
reg [2 :0] T_S = 0;

assign fdma_waddr = fdma_waddr_r + ADDR_MEM_OFFSET; //设置偏移地址
assign fdma_raddr = fdma_waddr; //读写地址相同

assign fdma_wsize = FDMA_BURST_LEN; //设置 FDMA 控制器一次写 burst 的数据长度
assign fdma_rsize = FDMA_BURST_LEN; //设置 FDMA 控制器一次读 burst 的数据长度
assign fdma_wdata ={t_data,t_data,t_data,t_data};

////延迟复位
reg [8:0] rst_cnt = 0;
always @(posedge ui_clk)
  if(rst_cnt[8] == 1'b0)
    rst_cnt <= rst_cnt + 1'b1;
  else
    rst_cnt <= rst_cnt;

//FDMA 读写控制器，每次先写后读，读出后对比数据正确性
always @(posedge ui_clk)begin
  if(rst_cnt[8] == 1'b0)begin
    T_S <=0;
    fdma_wareq  <= 1'b0;
    fdma_rareq  <= 1'b0;
    t_data<=0;
    fdma_waddr_r <=0;
  end
  else begin
    case(T_S)
      WRITE1:begin
        if(fdma_waddr_r==TEST_MEM_SIZE) fdma_waddr_r<=0; //超出测试内存范围，重新测试
        if(!fdma_wbusy)begin//当 fdma 进入空闲， fdma_wbusy=0， 请求写
          fdma_wareq  <= 1'b1; //设置写请求
          t_data  <= 0; //设置初值
        end
        if(fdma_wareq&&fdma_wbusy)begin//当 fdma 响应请求后， fdma_wbusy=1， 进入下一个状态
          fdma_wareq  <= 1'b0;
          T_S          <= WRITE2;
        end
      end
      end
      WRITE2:begin
        if(!fdma_wbusy) begin//当 fdma 完成请求后， fdma_wbusy=0， 进入下一个状态
          T_S <= WAIT;
          t_data  <= 32'd0;
        end
        else if(fdma_wvalid) begin//当 fdma_wvalid 有效期间必须写入有效数据

```

```

    t_data <= t_data + 1'b1;
  end
end

WAIT:begin//not needed
  T_S <= READ1;
end

READ1:begin
  if(!fdma_rbusy)begin//当 fdma 进入空闲, fdma_rbusy=0, 请求读
    fdma_rreq  <= 1'b1; //设置读请求
    t_data    <= 0; //设置初值
  end
  if(fdma_rreq&&fdma_rbusy)begin//当 fdma 响应请求后, fdma_rbusy=1, 进入下一个状态
    fdma_rreq  <= 1'b0; //清除读请求
    T_S        <= READ2;
  end
end

READ2:begin
  if(!fdma_rbusy) begin//当 fdma 完成请求后, fdma_rbusy=0, 进入下一个状态
    T_S <= WRITE1;
    t_data  <= 32'd0;
    fdma_waddr_r <= fdma_waddr_r + ADDR_INC; //当本次读写周期完成增加地址, 地址以 BYTE 计算
  end
  else if(fdma_rvalid) begin//当 fdma_rvalid 有效期间读出的数据有效
    t_data <= t_data + 1'b1;
  end
end

default:
  T_S <= WRITE1;
endcase
end
end

//对比是否有错误数据
wire test_error = (fdma_rvalid && (t_data[15:0] != fdma_rdata[15:0]));

//ila_0 ila_dbg (
//  .clk(ui_clk),
//  .probe0({fdma_wdata[15:0],fdma_wareq,fdma_wvalid,fdma_wbusy}),
//  .probe1({fdma_rdata[15:0],t_data[15:0],fdma_rvalid,fdma_rbusy,T_S,test_error})
//);

system system_i
  (.FDMA_S_0_fdma_raddr(fdma_raddr),
  .FDMA_S_0_fdma_rreq(fdma_rreq),
  .FDMA_S_0_fdma_rbusy(fdma_rbusy),
  .FDMA_S_0_fdma_rdata(fdma_rdata),
  .FDMA_S_0_fdma_rready(1'b1),
  .FDMA_S_0_fdma_rsize(fdma_rsize),
  .FDMA_S_0_fdma_rvalid(fdma_rvalid),
  .FDMA_S_0_fdma_waddr(fdma_waddr),

```

```

.FDMA_S_0_fdma_wreq(fdma_wreq),
.FDMA_S_0_fdma_wbusy(fdma_wbusy),
.FDMA_S_0_fdma_wdata(fdma_wdata),
.FDMA_S_0_fdma_wready(1'b1),
.FDMA_S_0_fdma_wsize(fdma_wsize),
.FDMA_S_0_fdma_wvalid(fdma_wvalid),
.sysclk(sysclk),
.ui_clk(ui_clk)
);

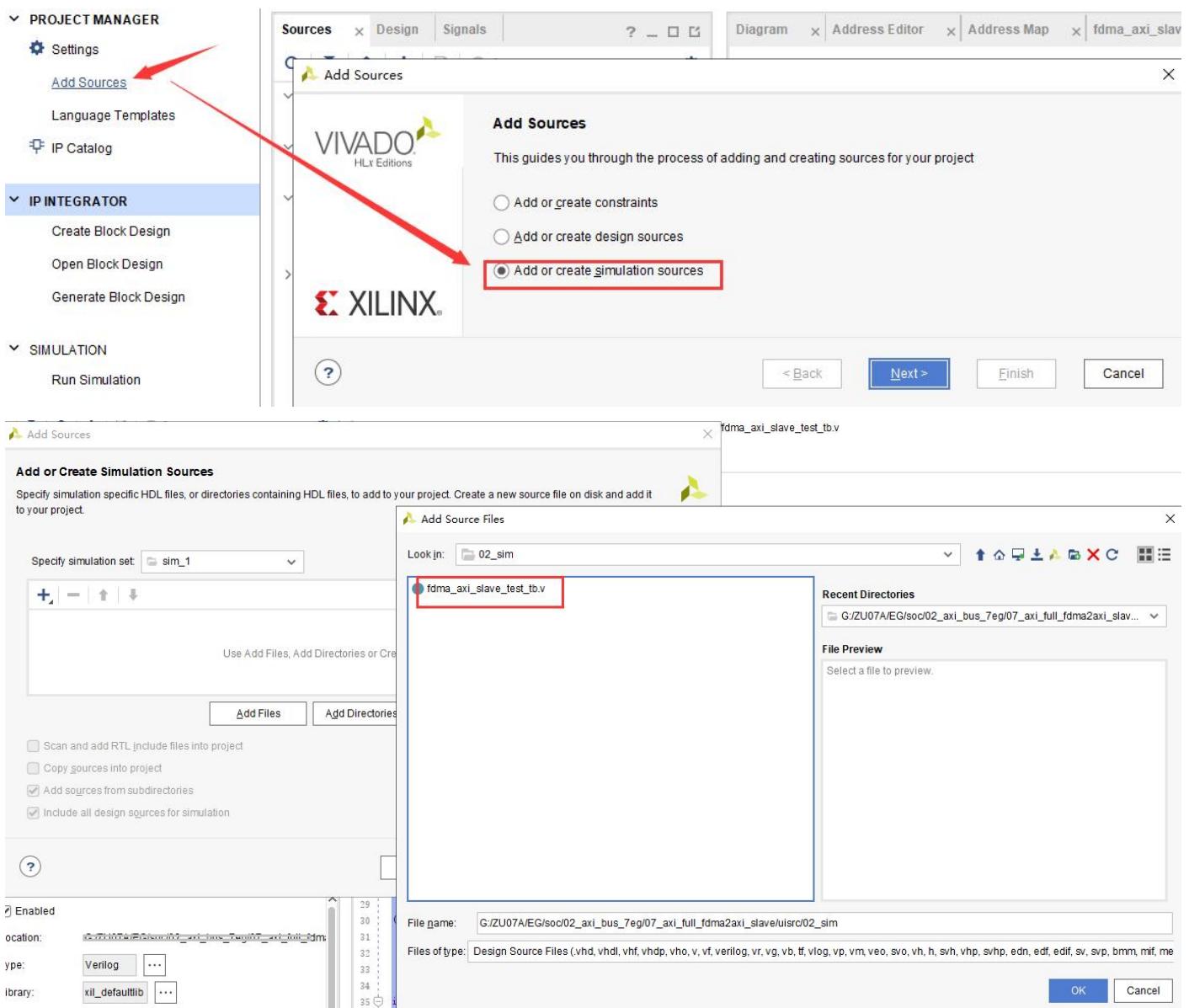
endmodule

```

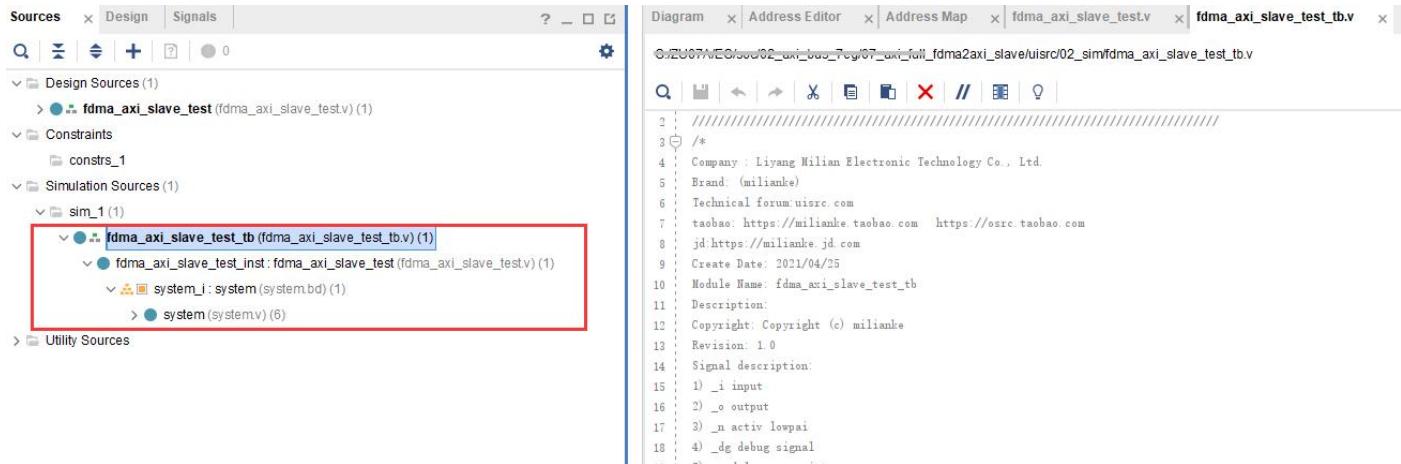
以上代码中调用的 system.bd 的图形代码接口。在状态机中，每次写 500 个长度 32bit 的数据，再读出来判断数据是否正确，因此传输 20000 字节的数据需要传输 10 次，每次 FDMA 传输的地址递增 2000。

2.4 仿真文件

添加仿真文件



添加完成后：



仿真文件非常简单，只要提供时钟激励就可以。

```

`timescale 1ns / 1ps

module fdma_axi_slave_test_tb();
reg sysclk;

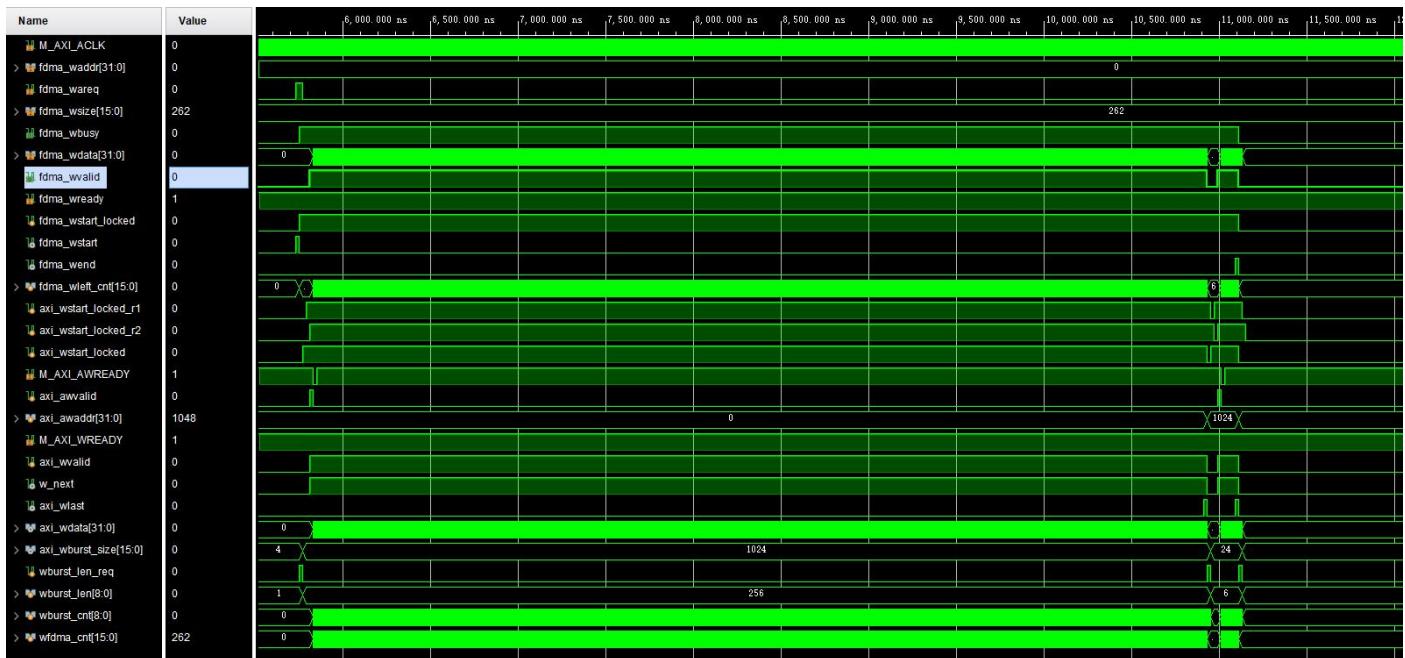
fdma_axi_slave_test fdma_axi_slave_test_inst
(
    .sysclk(sysclk)
);
initial begin
    sysclk = 0;
    #100;
end
always #10 sysclk = ~sysclk;
endmodule

```

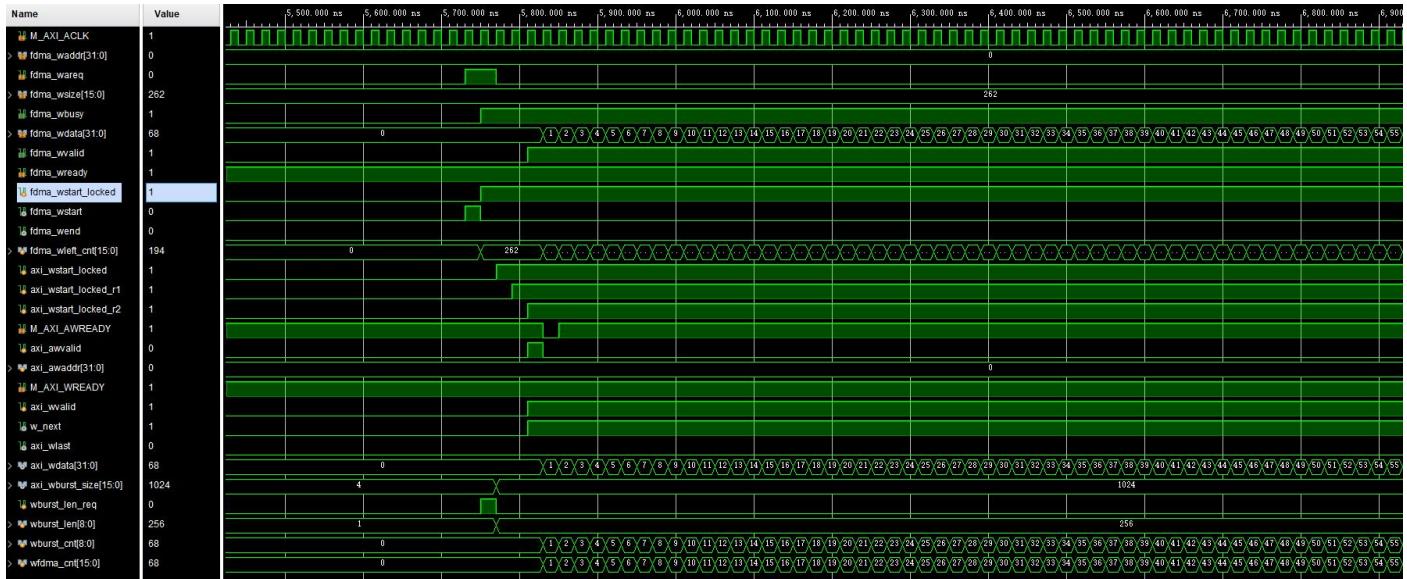
2.5 实验结果

FDMA 写操作仿真波形图，一次完成的 FDMA 写操作时序图如下：

这里一次 wburst_len_req 多产生一次，但是结果却不影响，大家可以思考下。如何设计出来和我们之前绘制的波形图一样。



一次 FDMA 写传输的起始时序

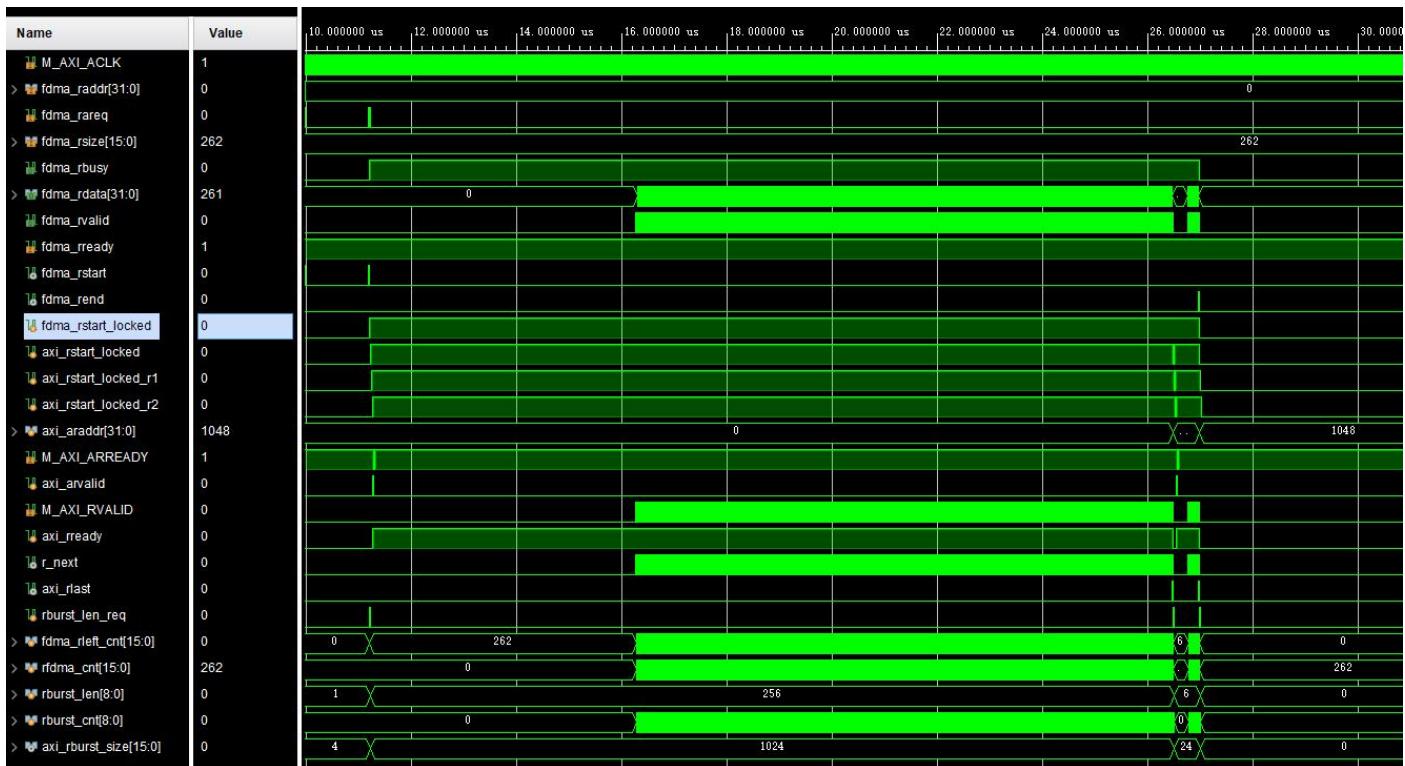


连续 burst,自动管理 burst 长度, 以及一次 FDMA 写传输结束时序

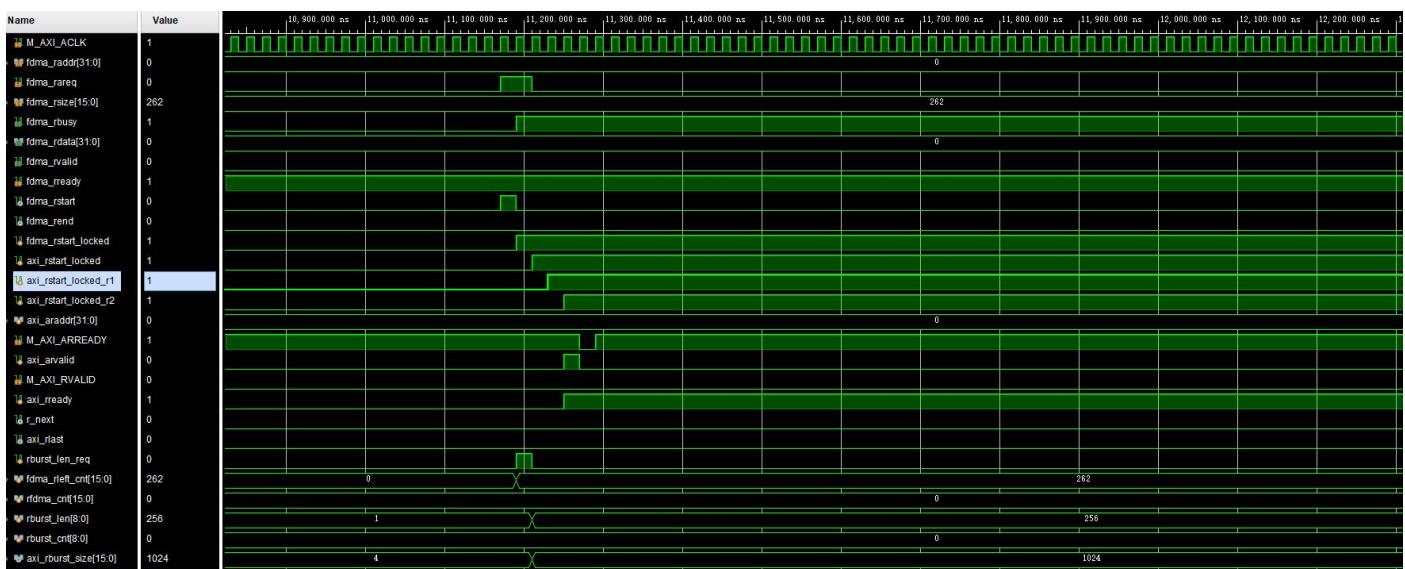


FDMA 读操作仿真波形图, 一次完成的 FDMA 读操作时序图如下:

这里一次 `rburst_len_req` 多产生一次, 但是结果却不影响, 大家可以思考下。如何设计出来和我们之前绘制的波形图一样。和写操作不同, 可以看到读操作的等待较长时间后才获取到数据。

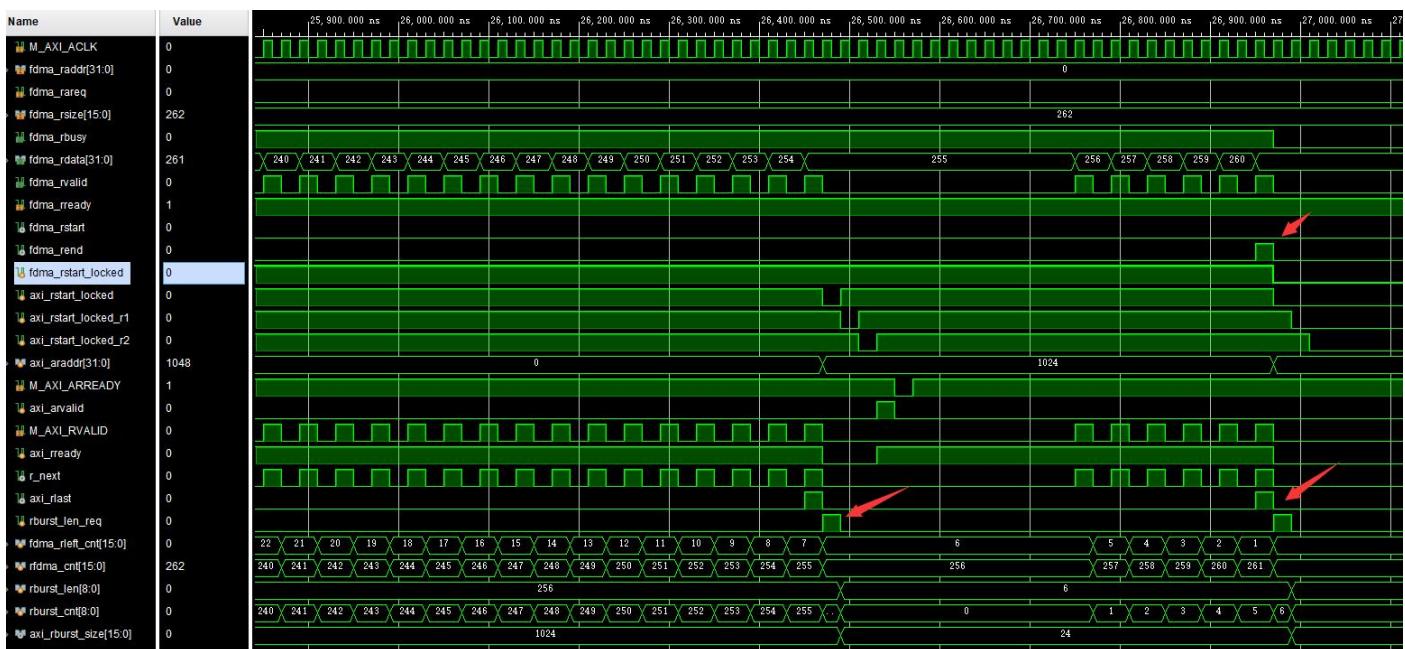


一次 FDMA 读传输的起始时序



连续 burst,自动管理 burst 长度, 以及一次 FDMA 读传输结束时序

另外放大后可以看到 rvalid 不是连续的, 这个读者也可以自己去优化 saxi_ful_mem ip 让这 IP 支持连续的 rvalid



03 使用 fdma 读写 axi-bram 测试

软件版本：vitis2021.1(vivado2021.1)

操作系统：WIN10 64bit

硬件平台：适用 XILINX A7/K7/Z7/ZU/KU 系列 FPGA

登录“米联客”FPGA 社区-www.uisrc.com 视频课程、答疑解惑！

3.1 概述

FDMA 是米联客的基于 AXI4 总线协议定制的一个 DMA 控制器。有了这个 IP 我们可以统一实现用 FPGA 代码直接读写 PL 的 DDR 或者 ZYNQ/ZYNQMP SOC PS 的 DDR 或者 BRAM。在米联客的数据交互方案中，FDMA IP CORE 已经广泛应用于 ZYNQ SOC/Artix7/Kintex7/ultrascale/ultrascale+系列 FPGA/SOC。

如果用过 ZYNQ/ZYNQMP SOC 的都知道，要直接操作 PS 的 DDR 通常是 DMA 或者 VDMA，然而用过 XILINX 的 DMA IP 和 VDMA IP，总有一种遗憾，那就是不够灵活，还需要对寄存器配置，真是麻烦。XILINX 的总线接口是 AXI4 总线，自定义 AXI4 IP 挂到总线上就能实现对内存地址空间的读写访问。因此，我们只要掌握 AXI4 协议就能完成不管是 PS 还是 PL DDR 的读写操作。

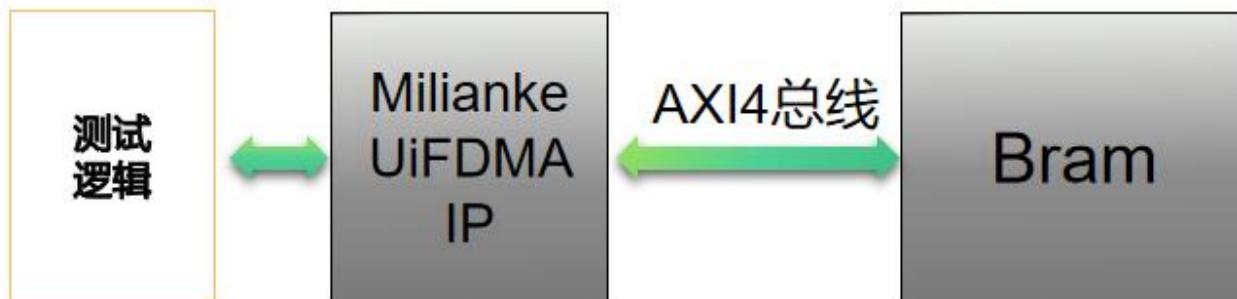
本文实验目的：

- 1:掌握基于 uiFDMA3.1 的 FPGA 工程设计
- 2:利用 uiFDMA3.1 提供的接口，编写 BRAM 测试程序
- 3:对 AXI-BRAM 读写仿真和测试

我们第一入门的 demo 选择对 BRAM 仿真测试，是因为不管是仿真还是编译测试，对 AXI-BRAM 速度都非常快，我们在下面一个 DEMO 中会给出对 DDR 的读写仿真测试。

3.2 系统框图

本系统中先将测试数据通过 Milianke UiFDMA 写入 Bram,再通过 Milianke UiFDMA 将 Bram 中数据读出。将读写数据进行对比。通过在线逻辑分析仪抓取读写数据测试读写正确性。



3.3 创建图形化逻辑工程

1:创建工程命名为 fpga_prj

双击 VIVADO 软件图标启动 VIVADO



设置工程路径，并且命名工程名为 fpga_prj

 New Project **Project Name**

Enter a name for your project and specify a directory where the project data files will be stored.

Project name: Project location:   Create project subdirectory

Project will be created at D:/Boards/fpga_prj

 < Back

Next >

Finish

Cancel

 New Project **Project Type**

Specify the type of project to create.

 RTL Project

You will be able to add sources, create block designs in IP Integrator, generate IP, run RTL analysis, synthesis, implementation, design planning and analysis.

 Do not specify sources at this time Project is an extensible Vitis platform Post-synthesis Project

You will be able to add sources, view device resources, run design analysis, planning and implementation.

 Do not specify sources at this time I/O Planning Project

Do not specify design sources. You will be able to view part/package resources.

 Imported Project

Create a Vivado project from a Synplify, XST or ISE Project File.

 Example Project

Create a new Vivado project from a predefined template.

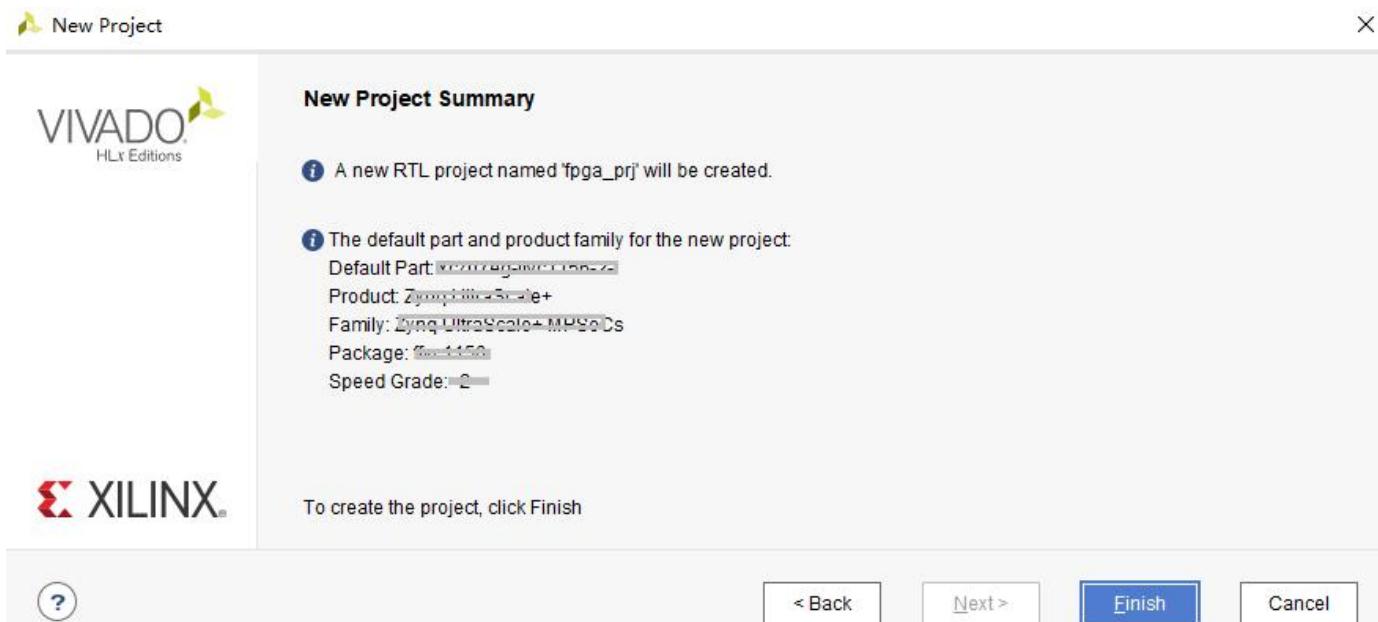
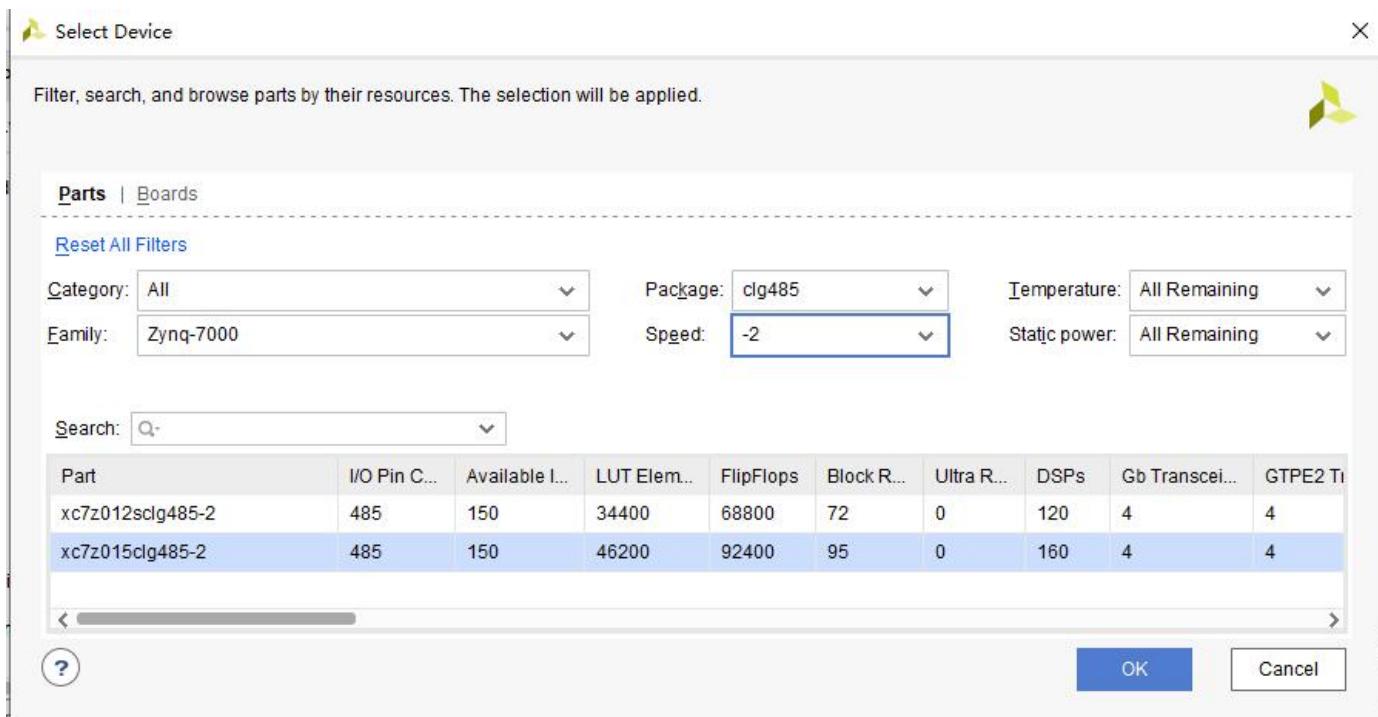
 < Back

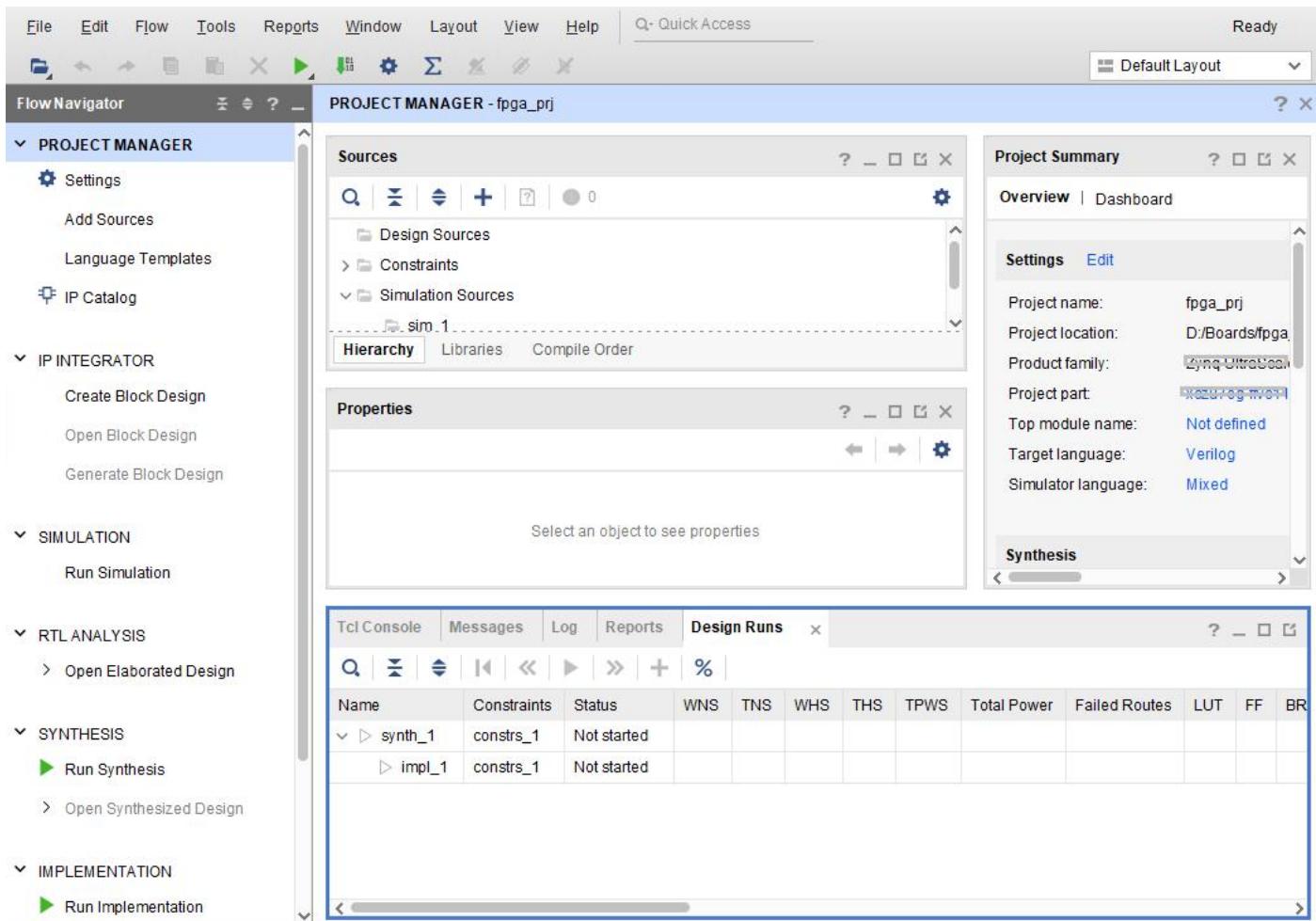
Next >

Finish

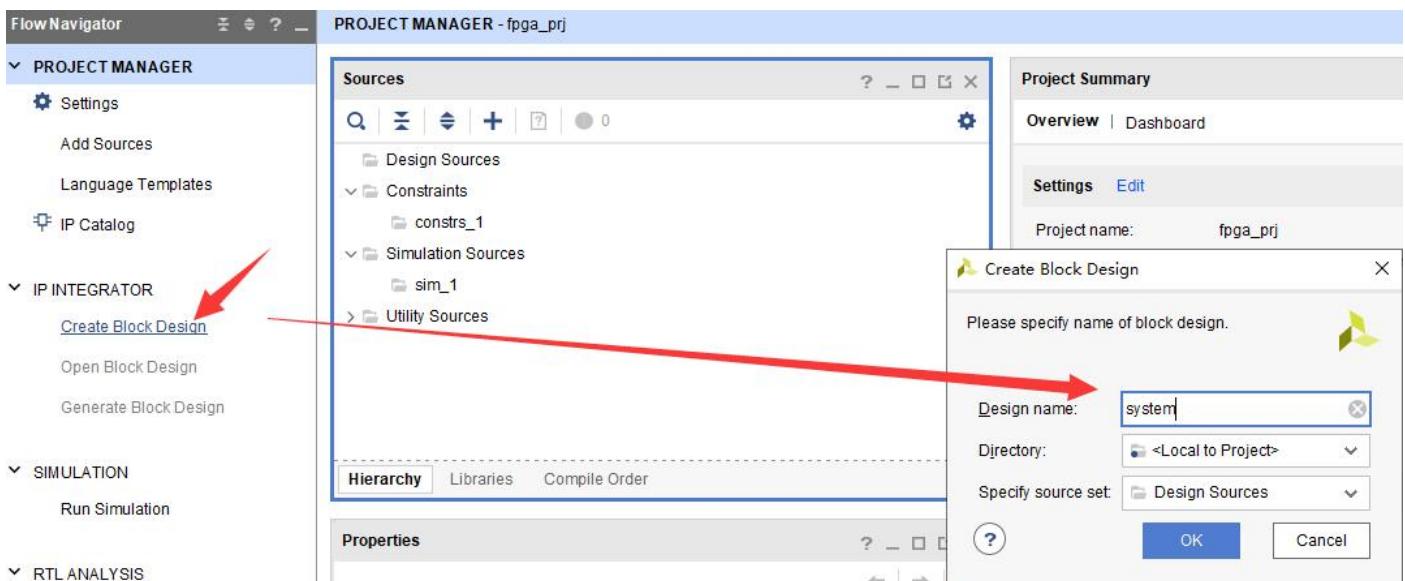
Cancel

以下设置 FPGA 或者 ZYNQ 或者 ZYNQ-MPSOC 芯片型号，必须和开发板保持一致，如果不清楚的请查阅自己开发板的硬件手册或者根据选型手册上参数确认：

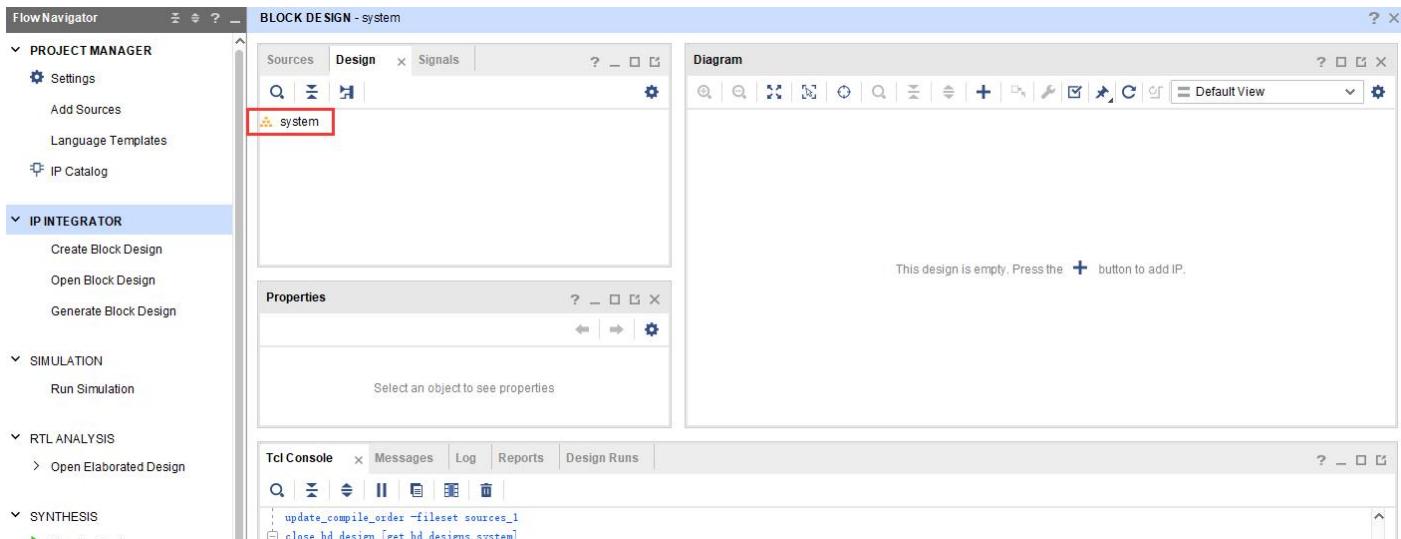




2: 创建 Block Design 并且命名为 system

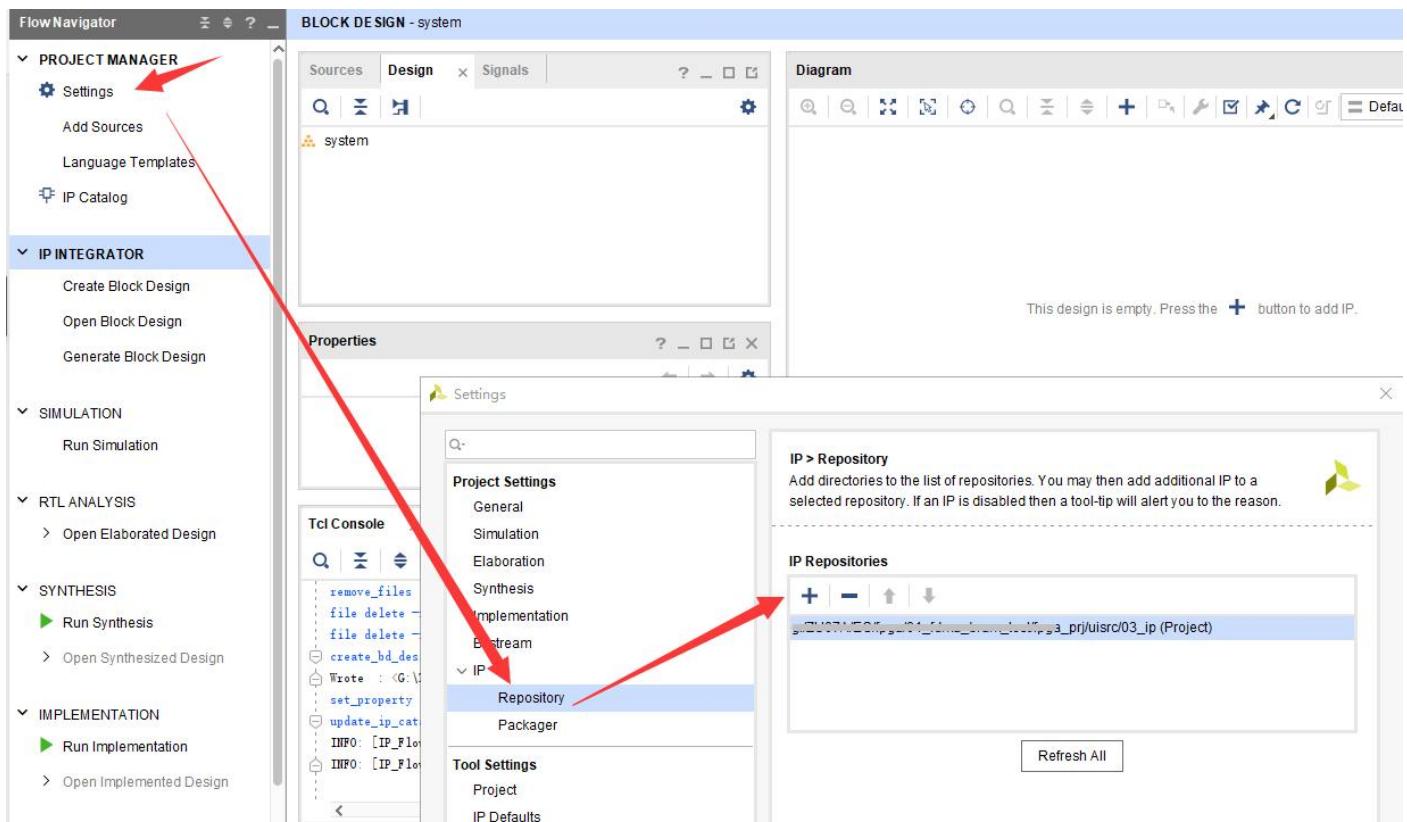


如下图所示，图形化 system 就是一个代码容器，接着我们要添加一些图像化的 IP

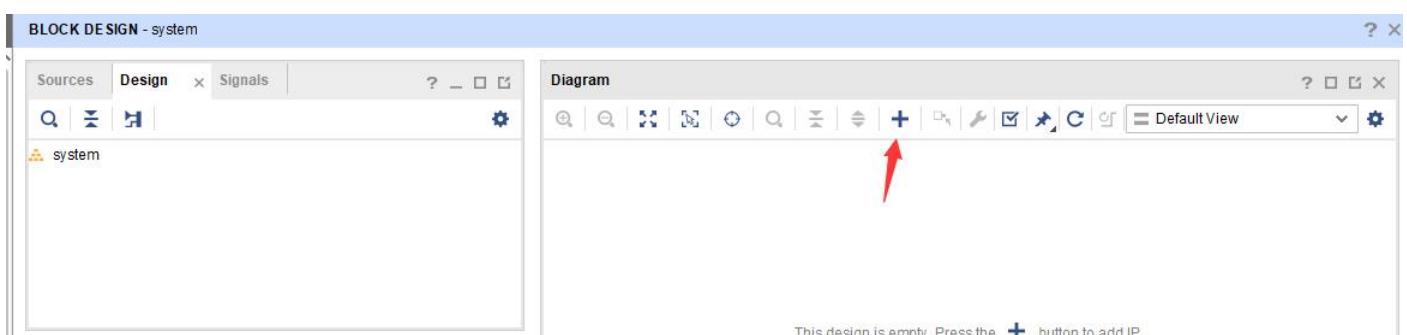


3:添加图形化 FPGA IP

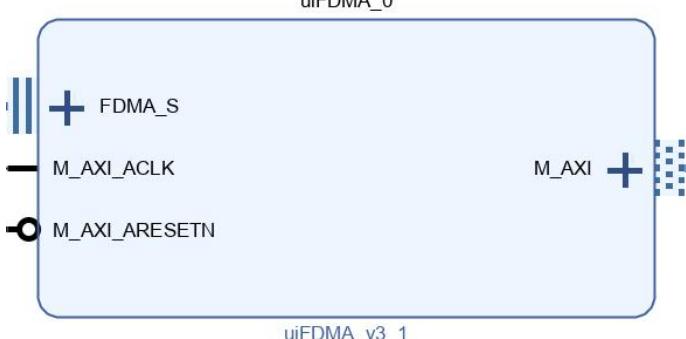
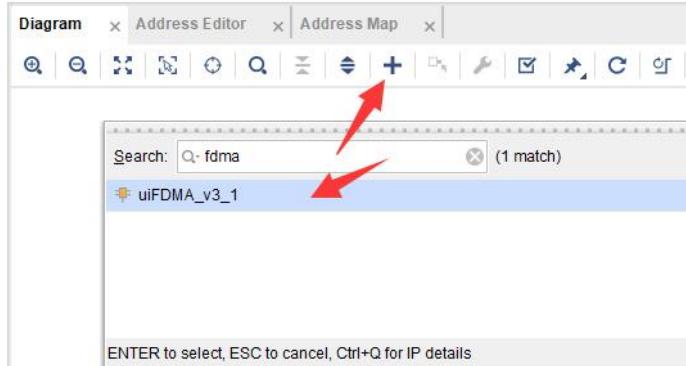
首先设置自定义 IP 的路径，这里读者可以把我们配套工程根路径下的 uisrc 文件夹复制到目前的工程根路径，单击 Settings 在弹出的 Settings 窗口选择 IP->Repository 设置如下路径



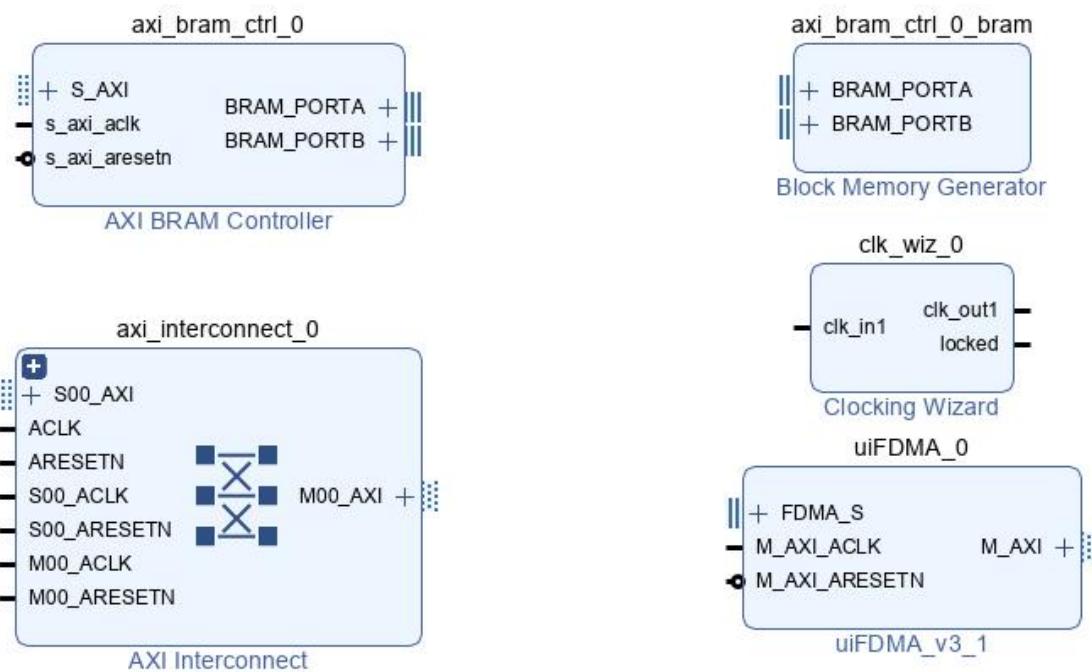
添加十号添加 IP



比如输入关键词 FDMA 就可以搜索到我们米联客 uiFDMA IP



用类似的方法添加必要的 IP 如下图所示:

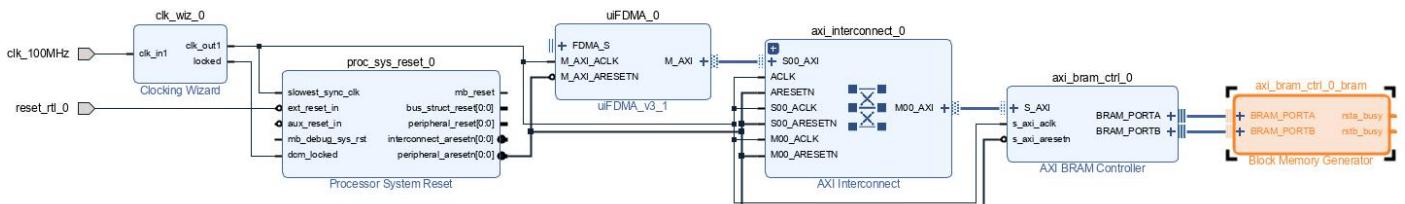


4:完成 IP 之间的信号自动

这种简单的工程可以先让软件先自动化线，然后根据结果手动一些调整



可以看到连线结果

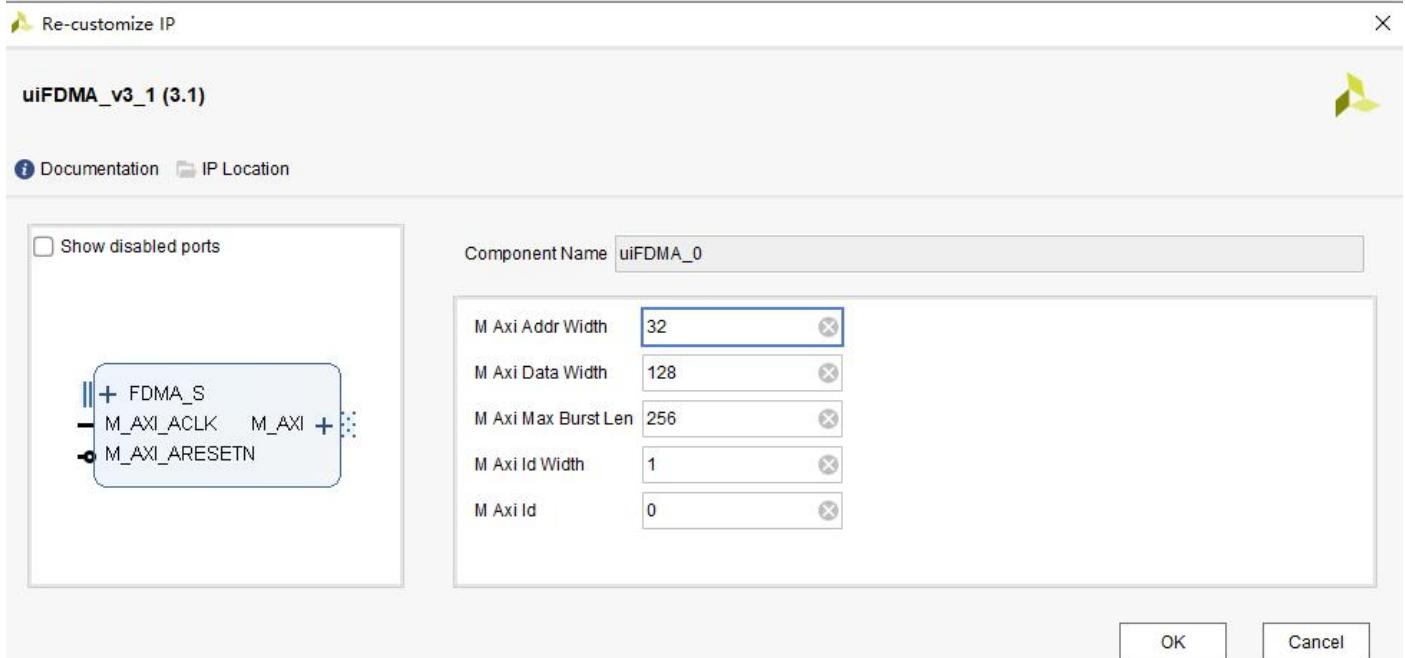


5:调整 IP 参数

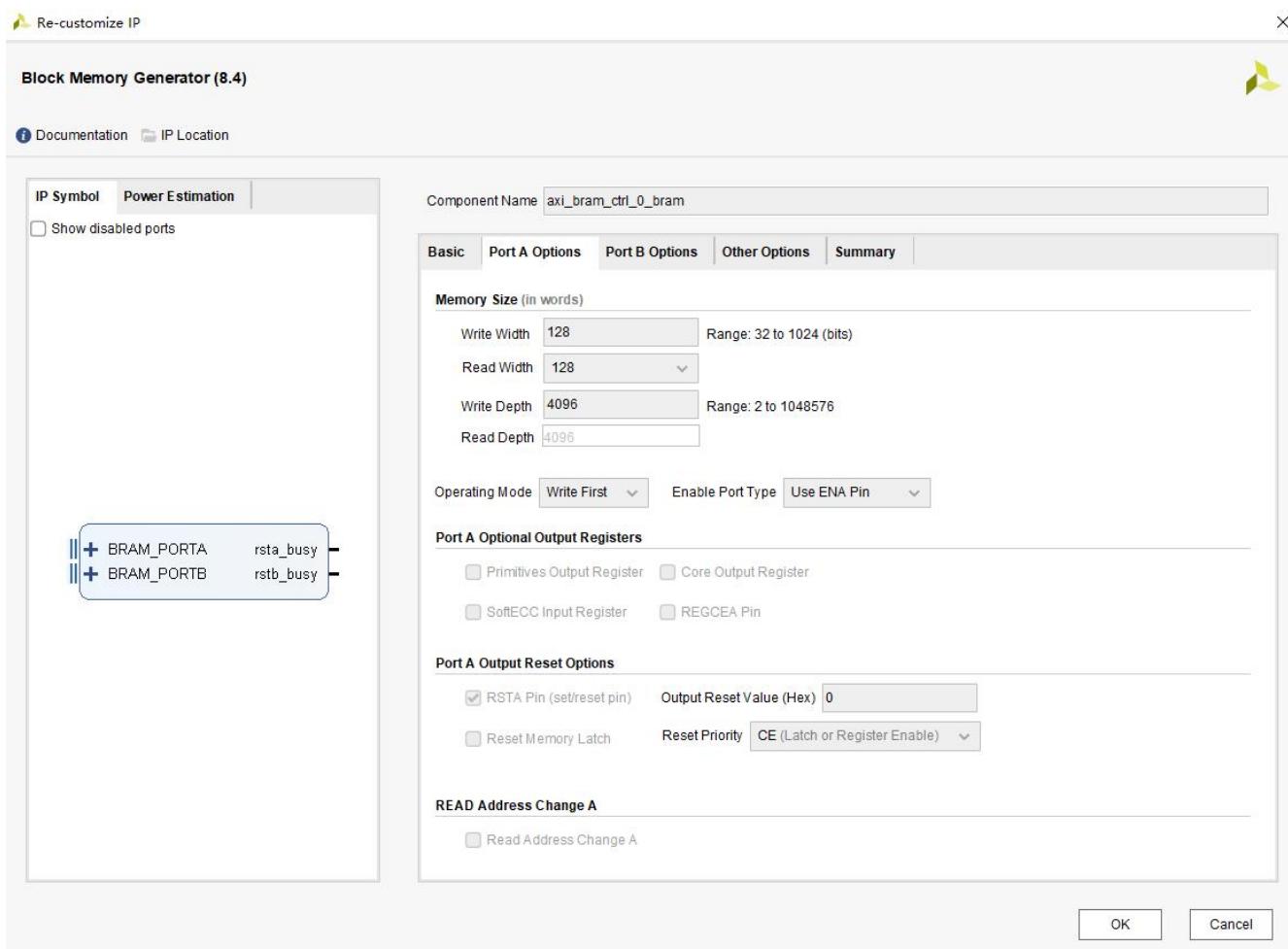
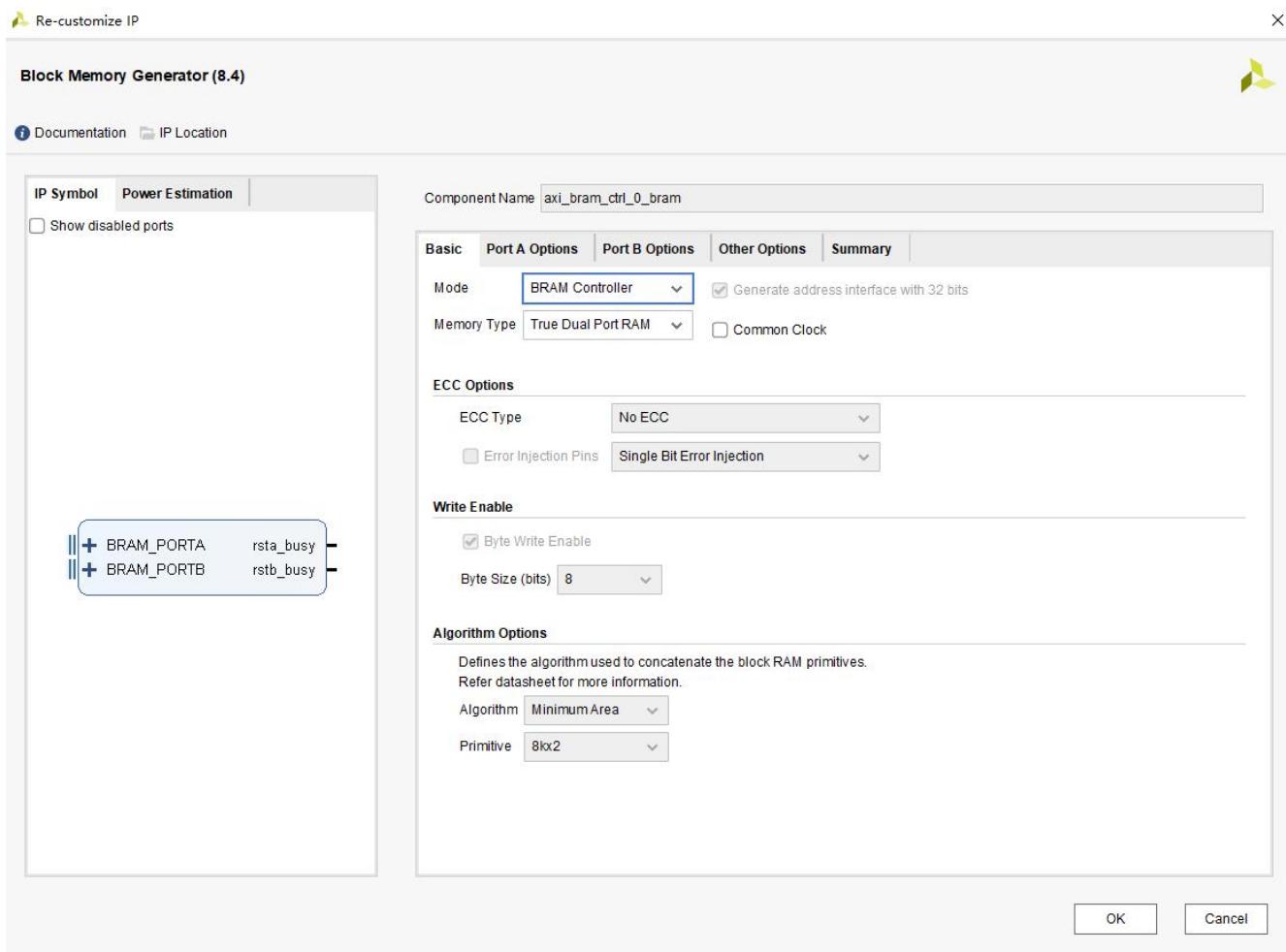
5-1:BRAM 参数设置

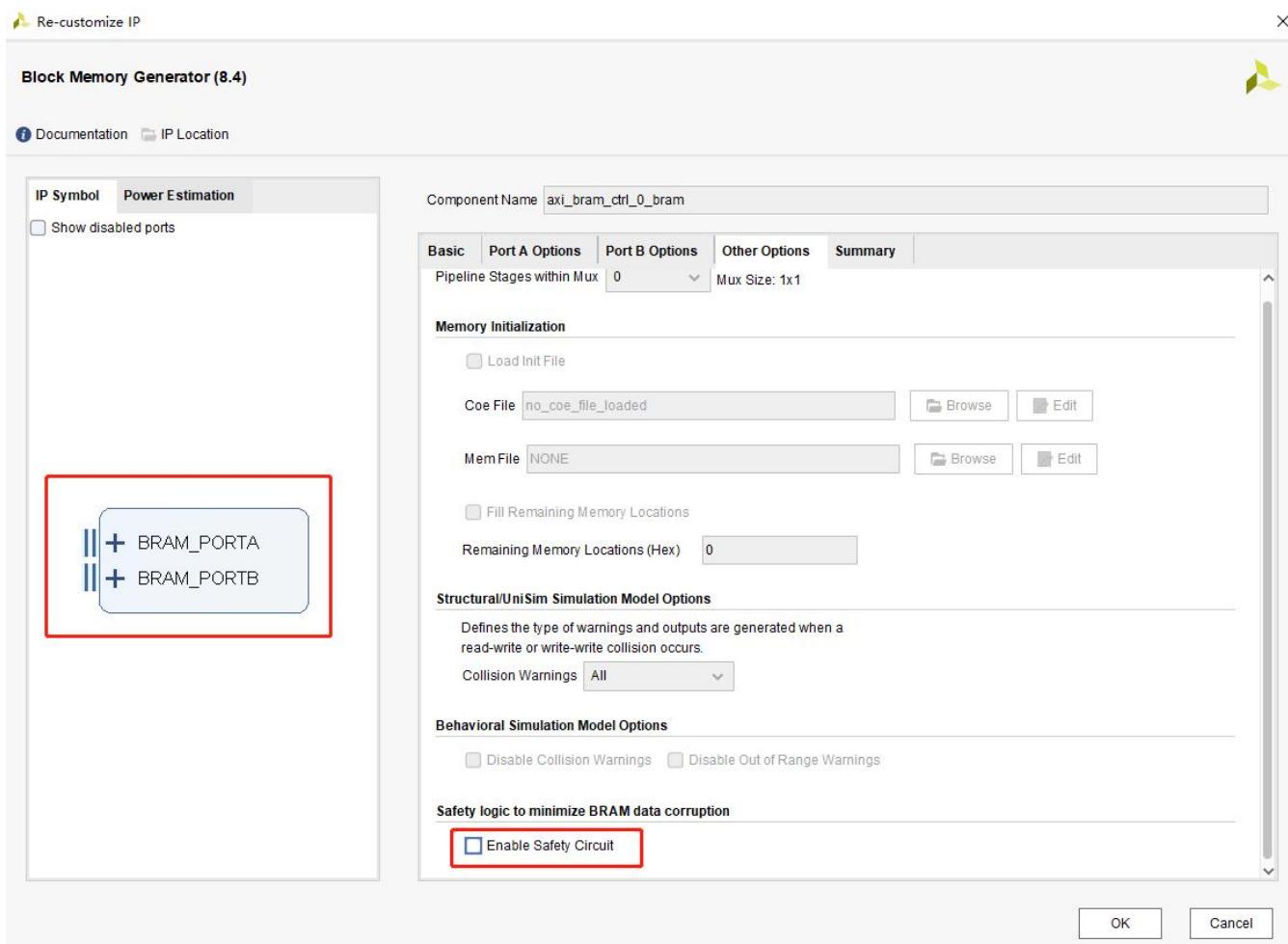
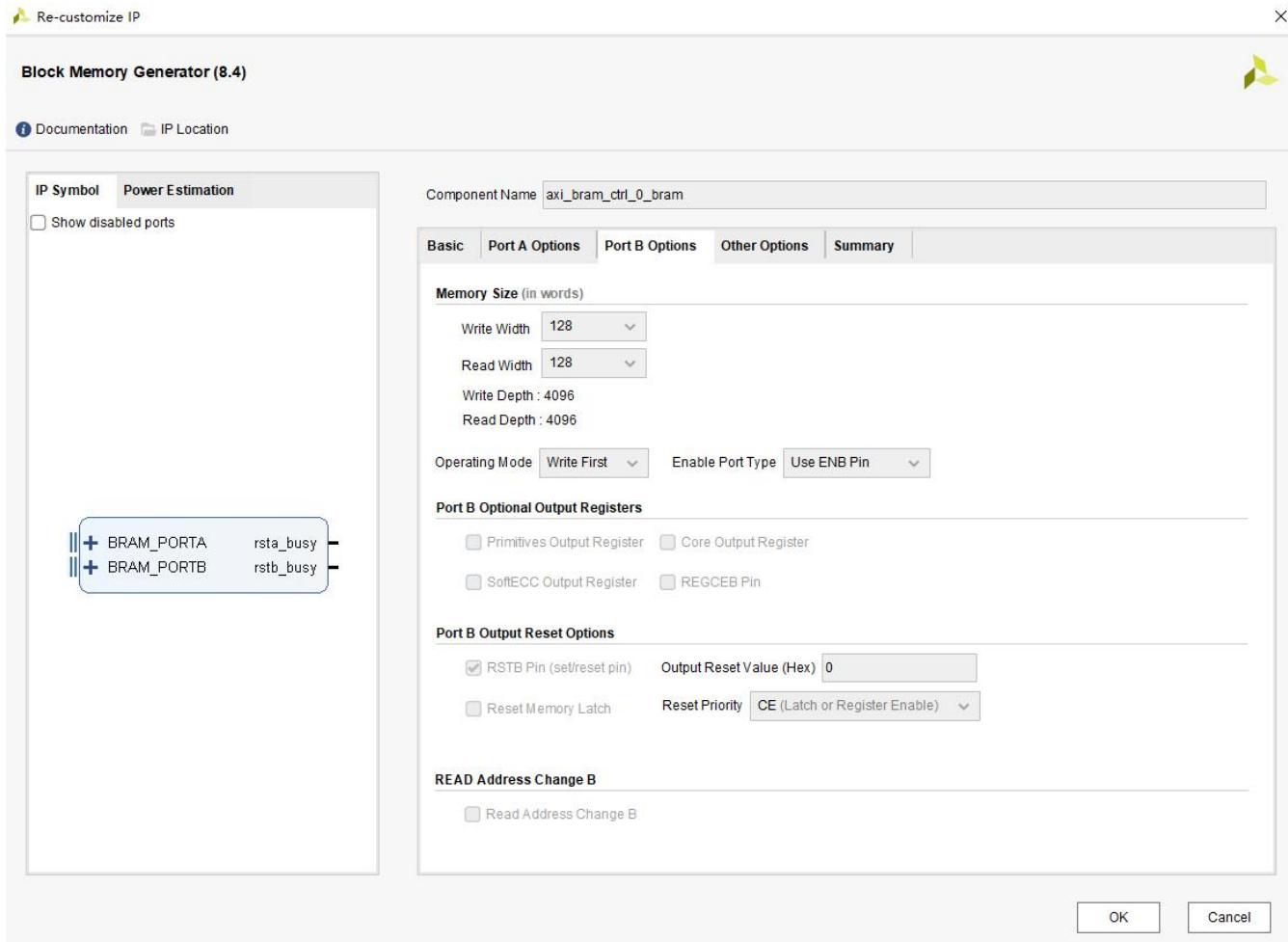
首先把 IP 的配置参数修改下，双击需要设置的 IP 可以进行参数设置

FDMA 设置数据位宽 128bit 可以访问内存地址位宽 32bit，最大 AXI 最大 burst 的长度为 256



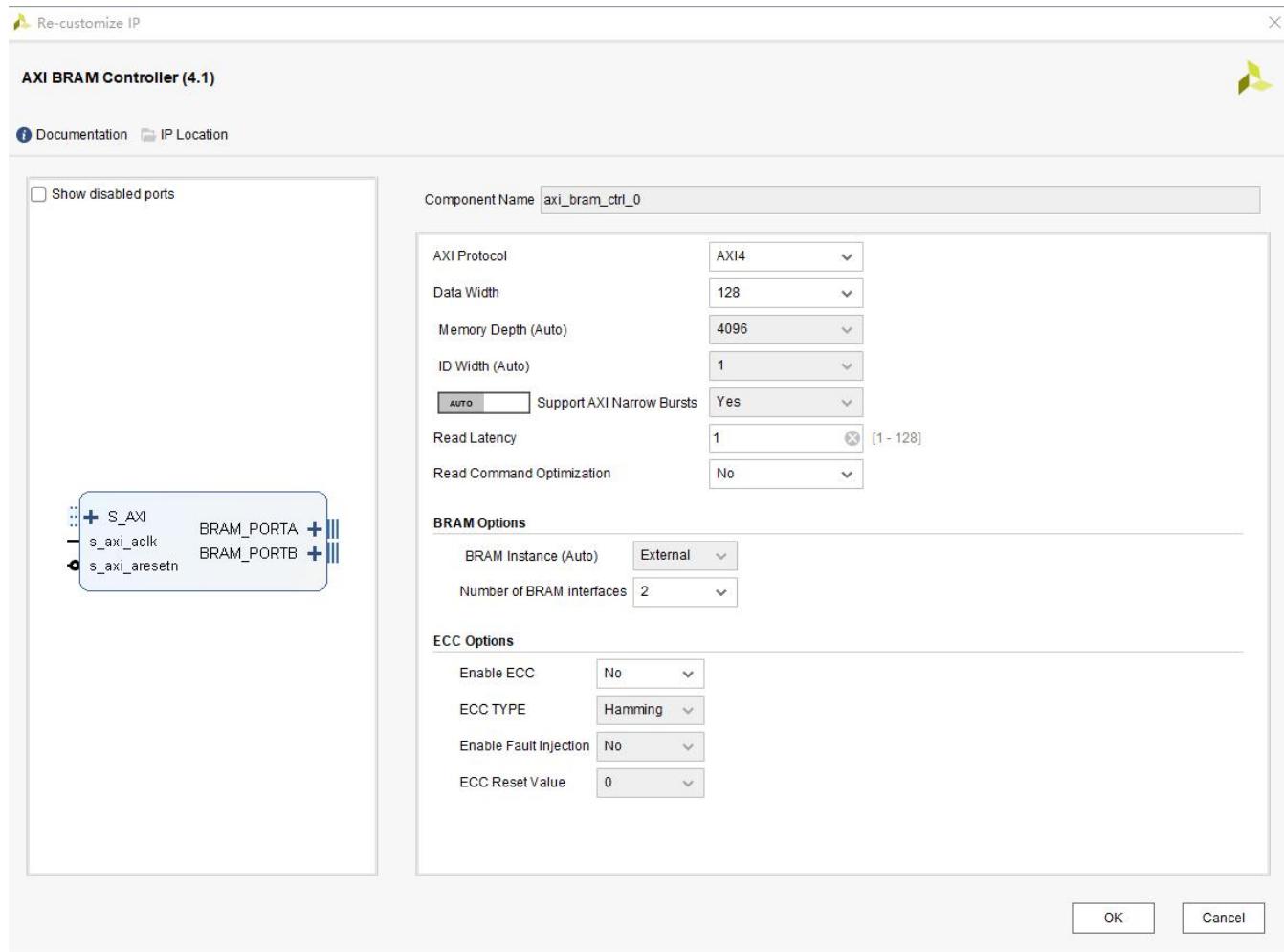
BRAM 设置，使用 BRAM Controller 为真双口 RAM



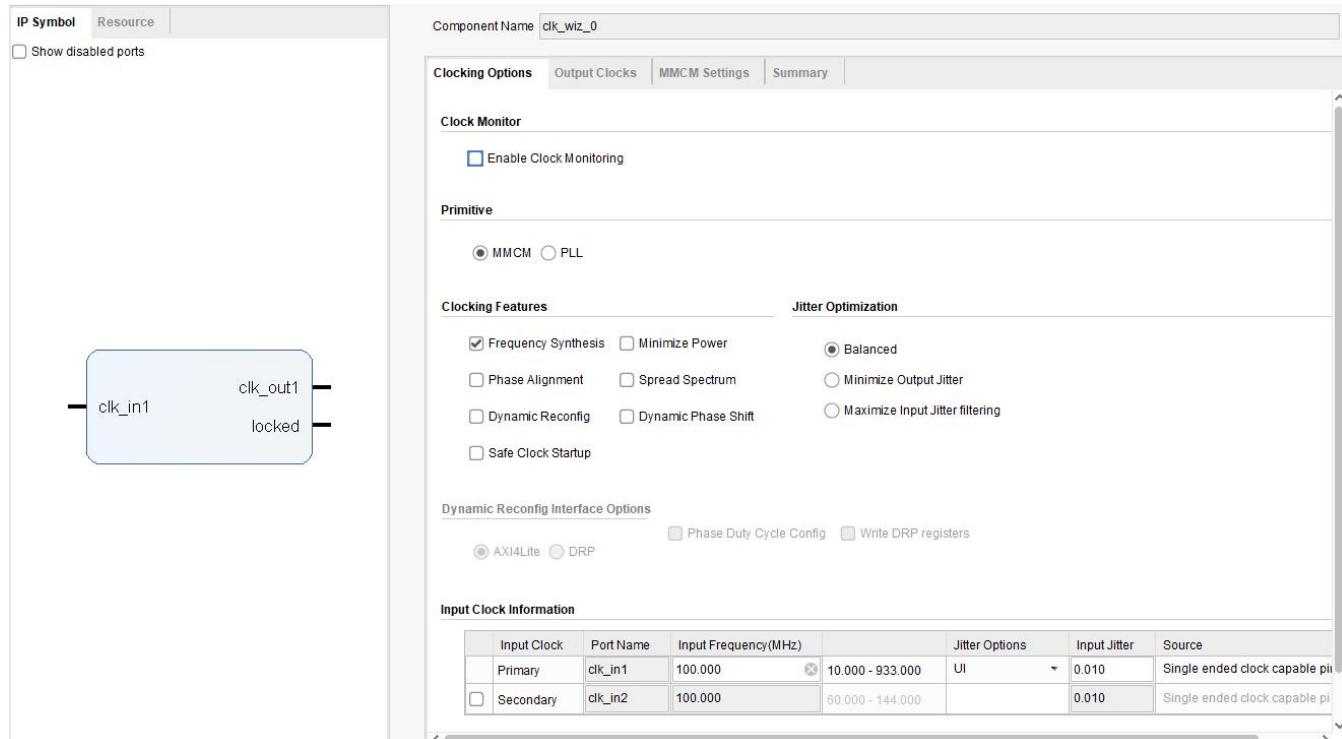


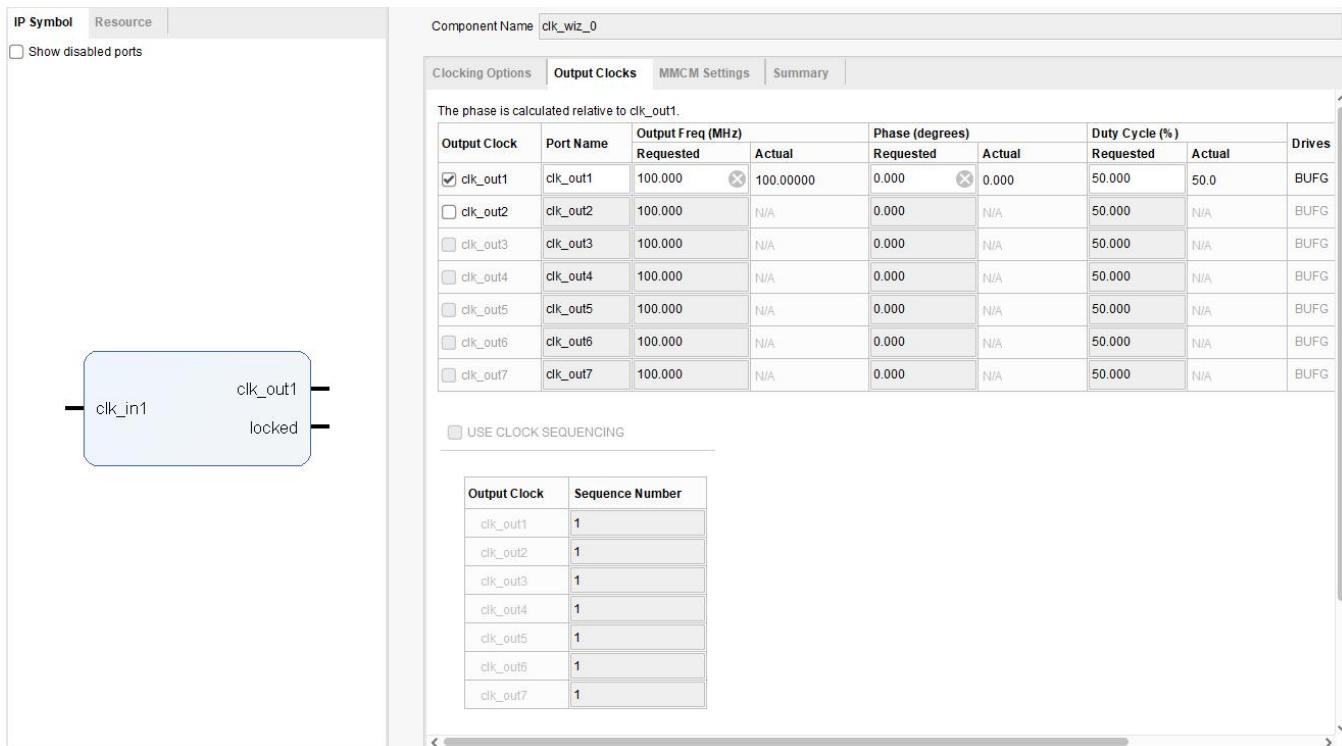
5-2:BRAM Controller 参数设置

AXI BRAM Controller 设置 axi4 协议，数据位宽 128bit 读延迟 1 个时钟



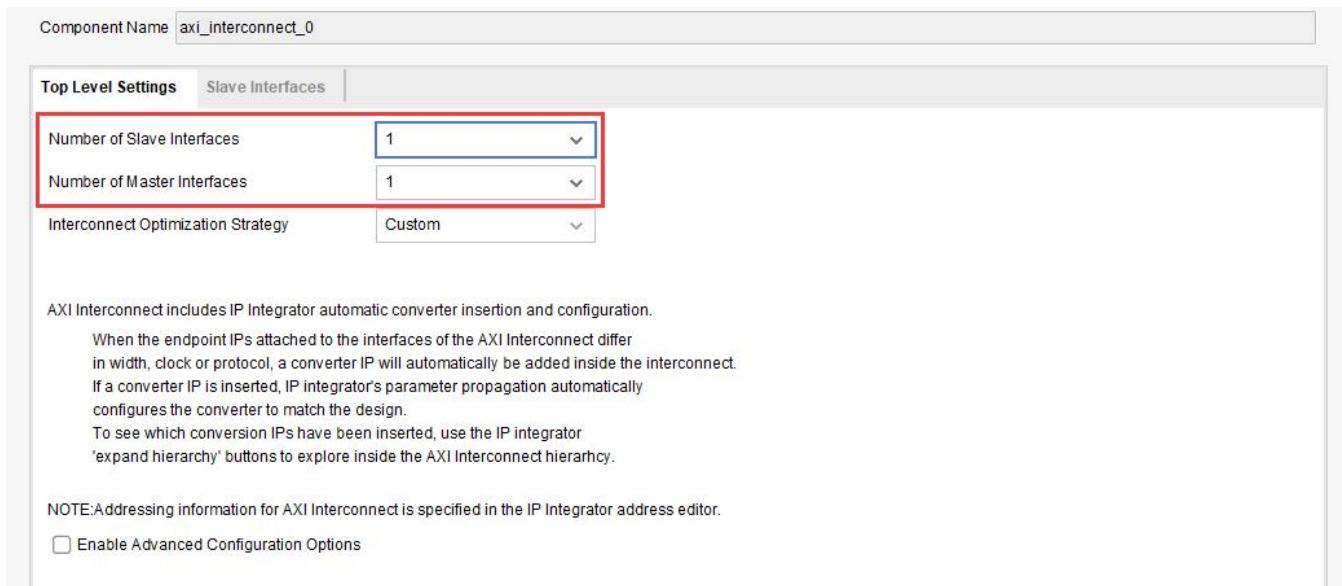
5-3:Clocking Wizard 参数设置





5-4:AXI Interconnect IP 设置

双击 AXI Interconnect IP 进行设置

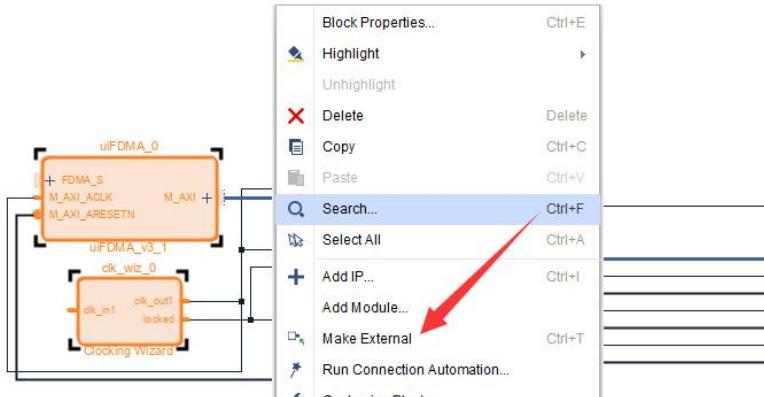


设置 AXI Interconnect IP 的性能参数，其中 Enable Register Slice 用于改善时序，Enable Data FIFO 用于增加 FIFO 大小，增加数据传输效率

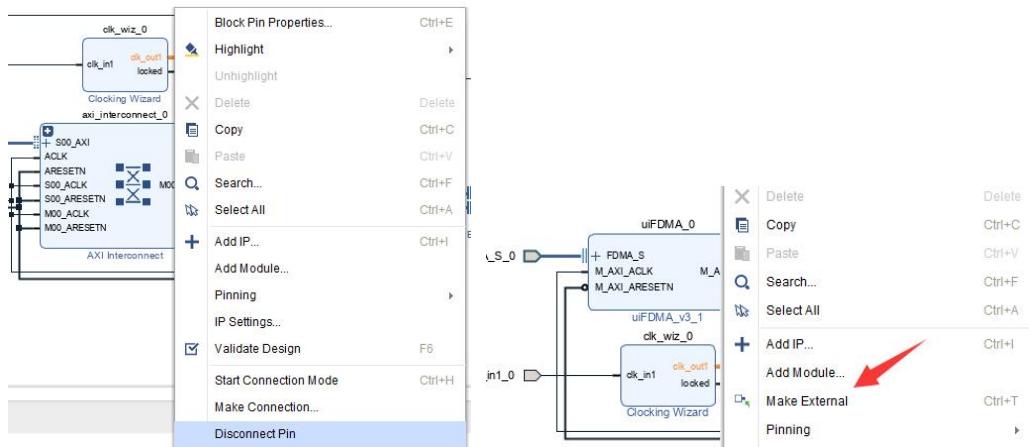


6:引出 FPGA 接口信号

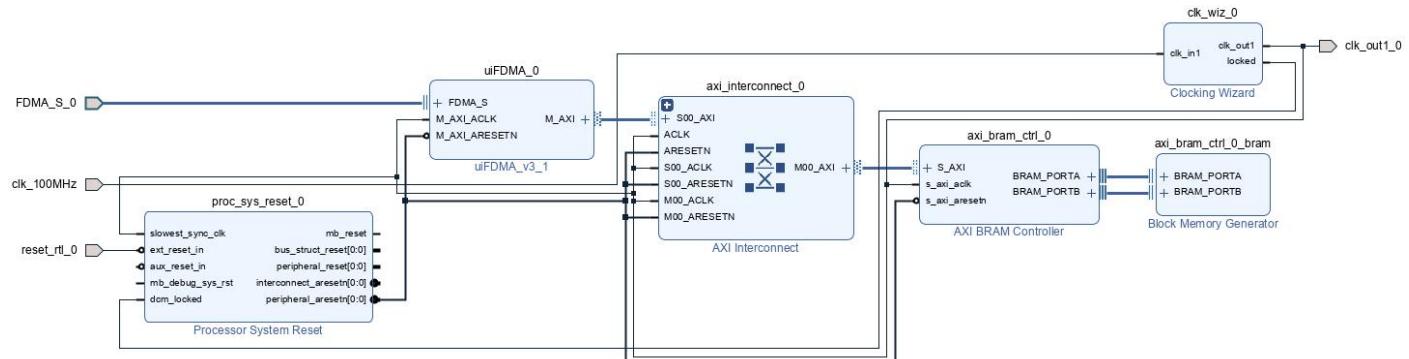
分别右击下图 2 个 IP,然后选择 Make External, 把需要引出到顶层的 FPGA 信号引出



为了引出时钟需先右击信号 PIN 脚断开连接, 然后 Make External, 之后重新连接

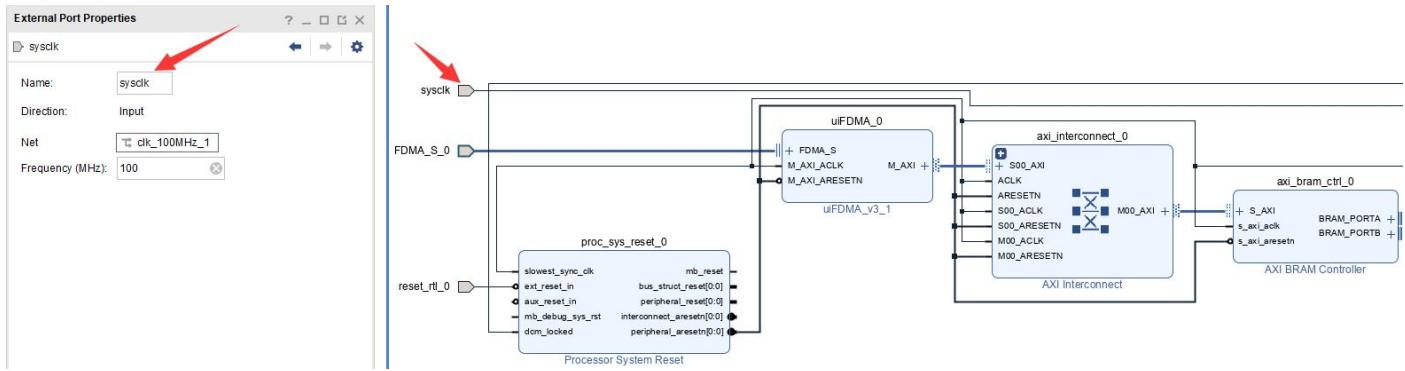


修改后重新连接时钟

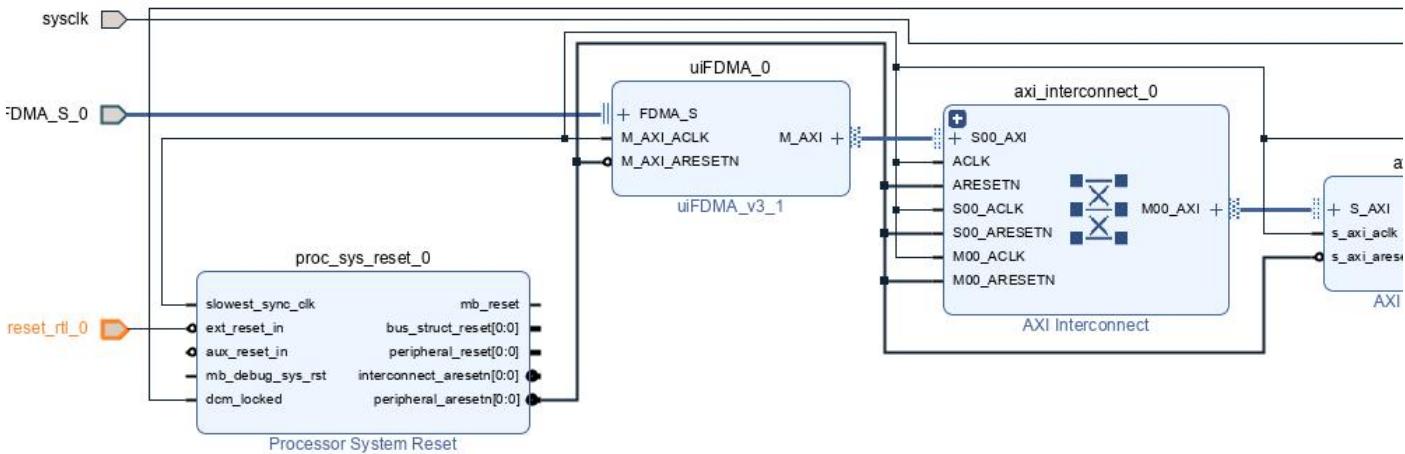


7:修改复位和信号名字

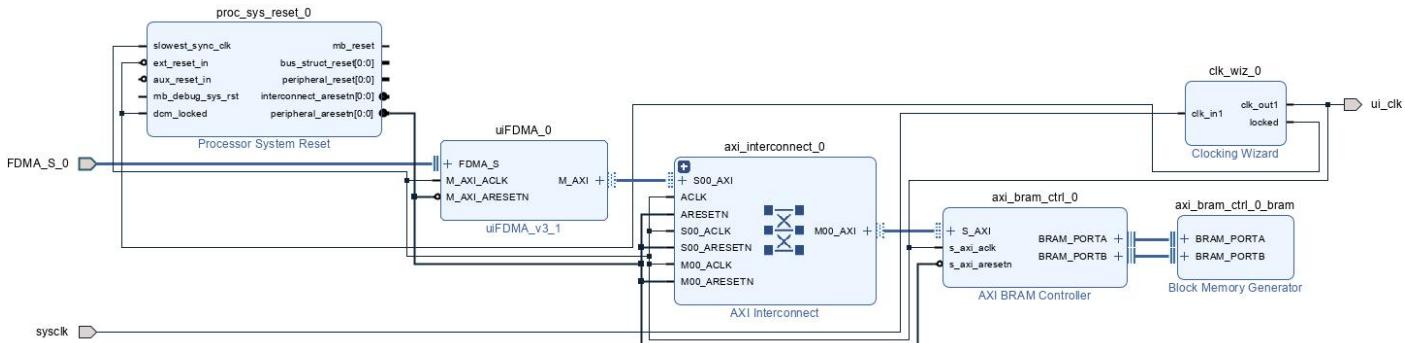
修改默认的时钟名字,以相同的方法修改其他信号



信号名修改后，修改复位脚的接线

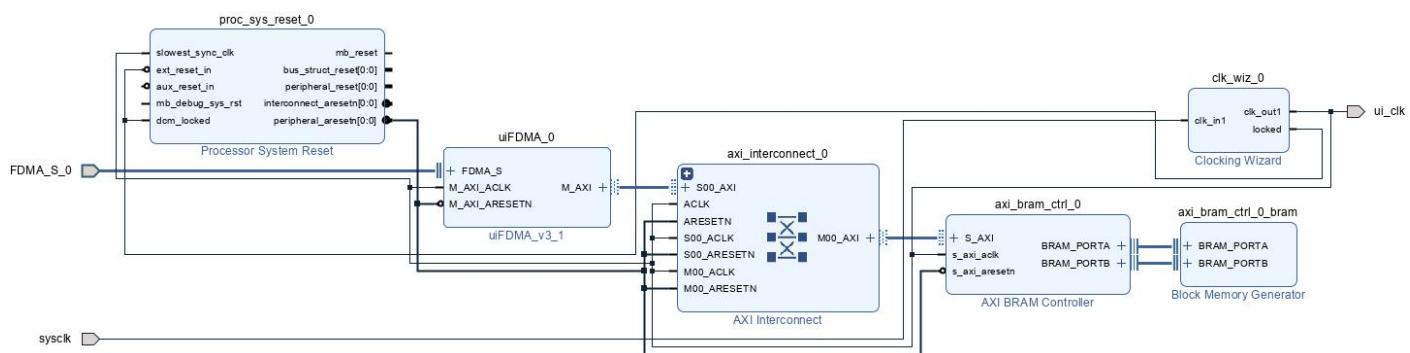
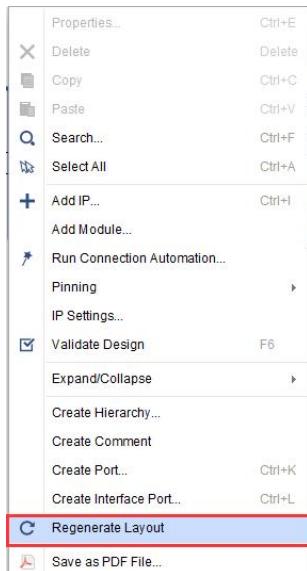


修改完成后



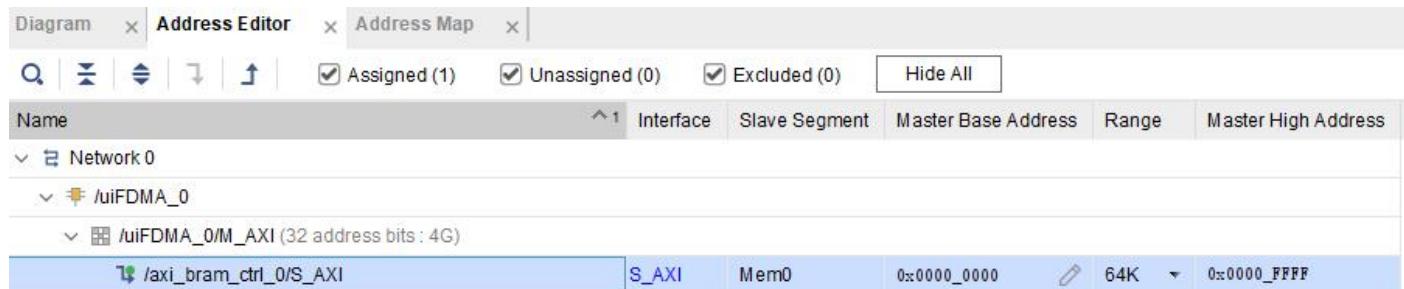
8:视图优化

右击空白处，弹出菜单选择 Regenerate Layout 优化下视图



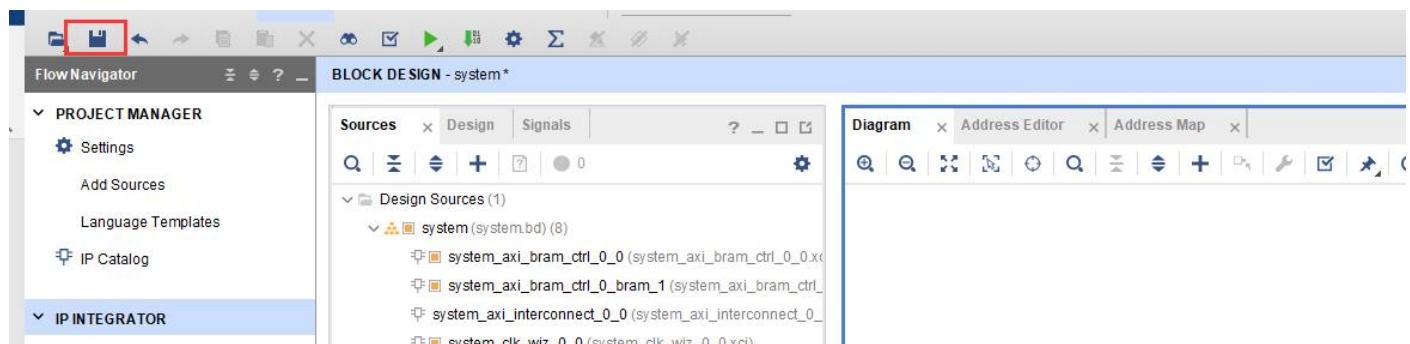
9:地址分配

AXI 总线必须分配地址，设置 uiFDMA 的地址空间分配，起始地址可以任意设置，我们设置从 0x0000_0000 开始，大小 64KB



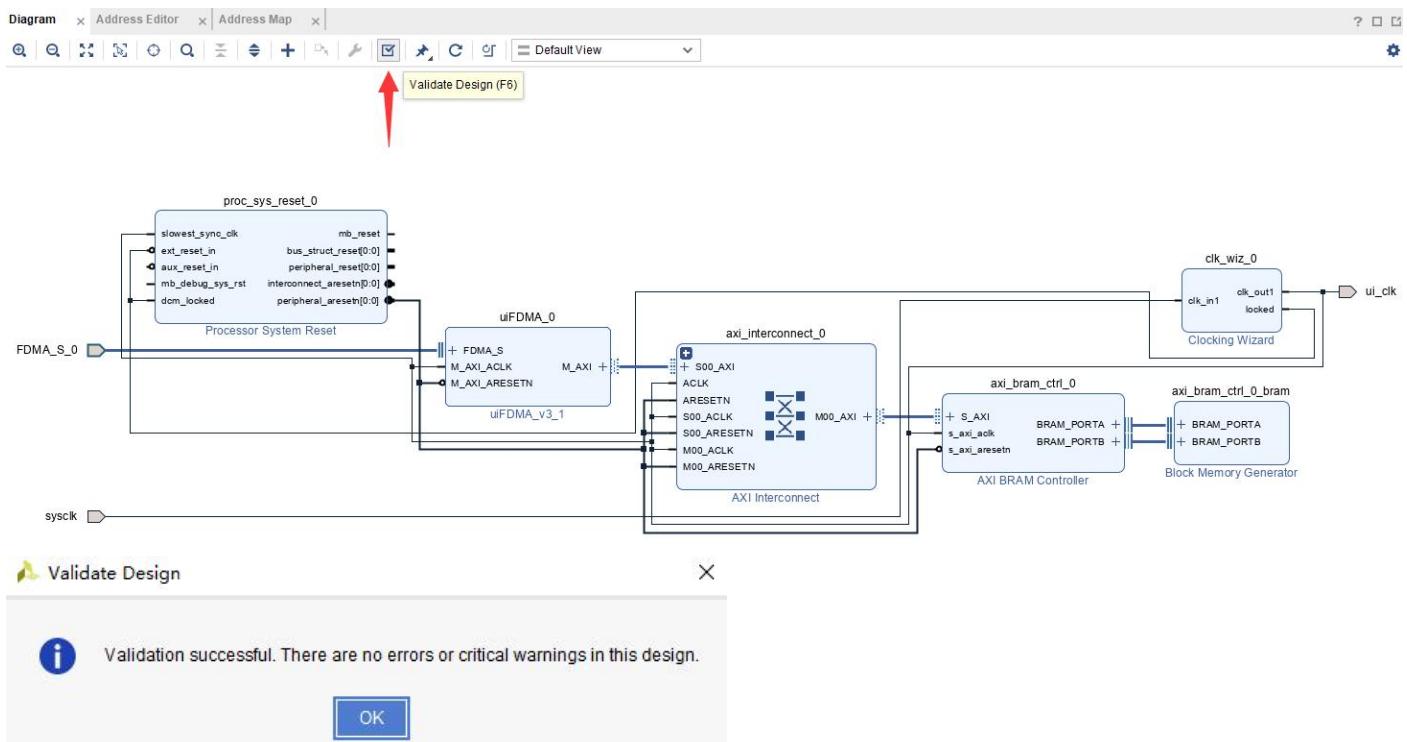
10:自动校验

保存工程



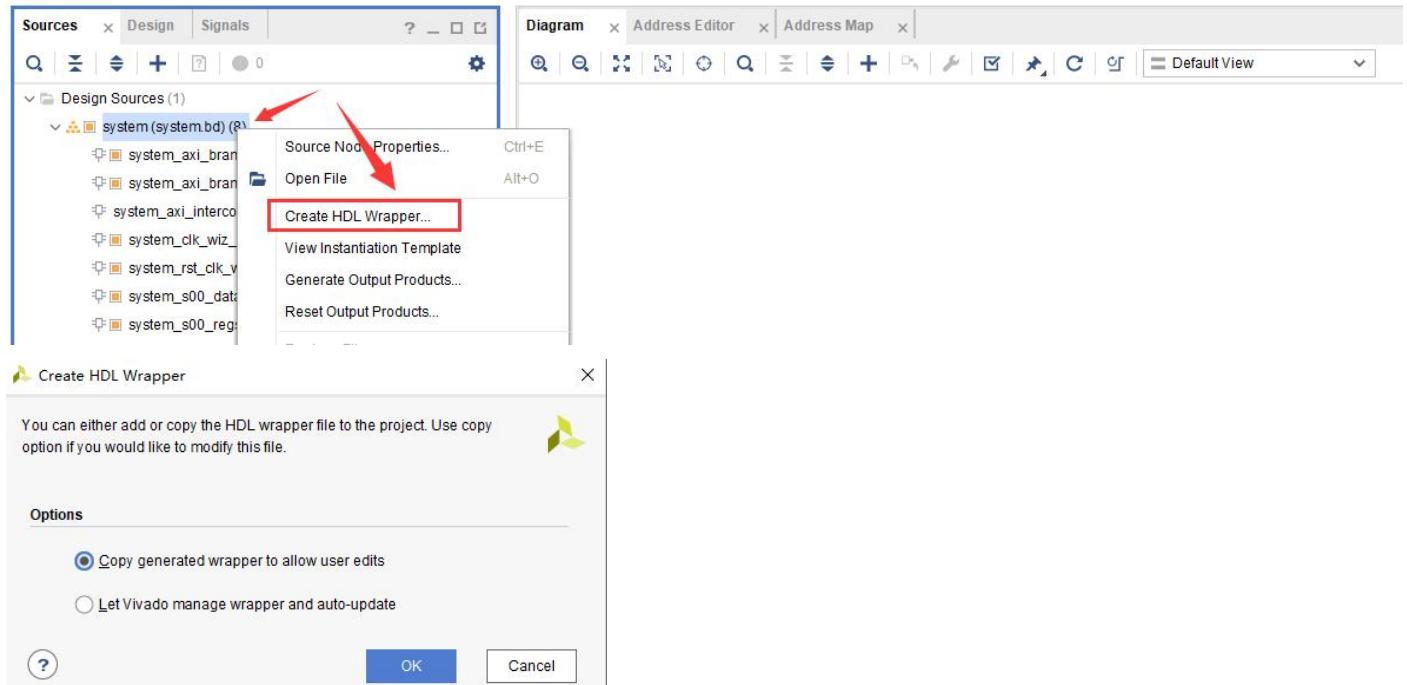
单击下图中的控件可以对图形化编程进行校验

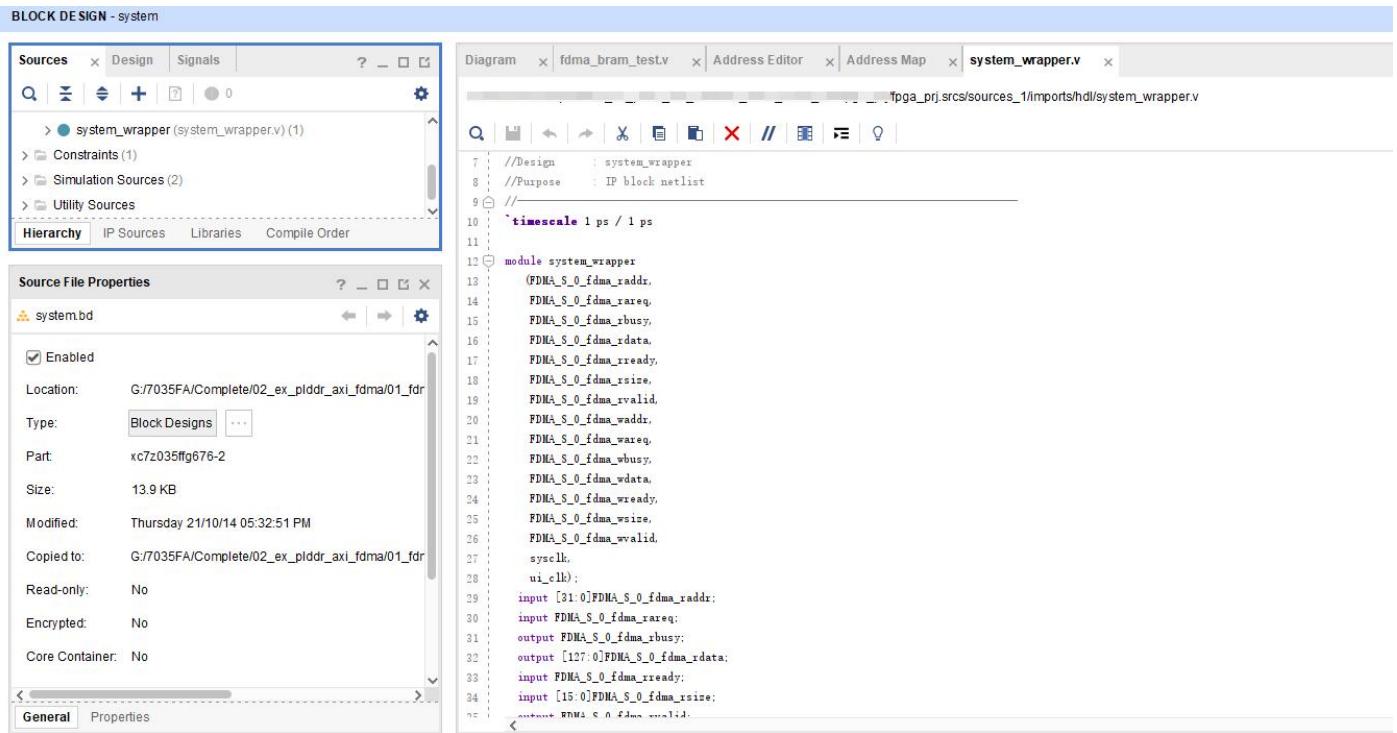
11:校验图形逻辑工程



12:自动产生顶层文件

右击 system,在弹出的菜单中选择 Create HDL Wrapper





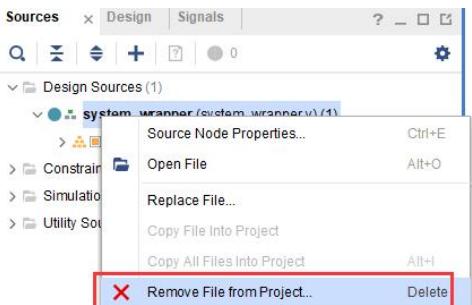
3.4 编写 FDMA 的 BRAM 测试代码

刚刚以上自动产生的顶层文件，只是引出的信号接口，并不能完成对 FDMA 的控制，所以我们需要自定义一个顶层文件，可以复制刚才产生的文件，修改，这样可以省一些编写调用接口的时间。

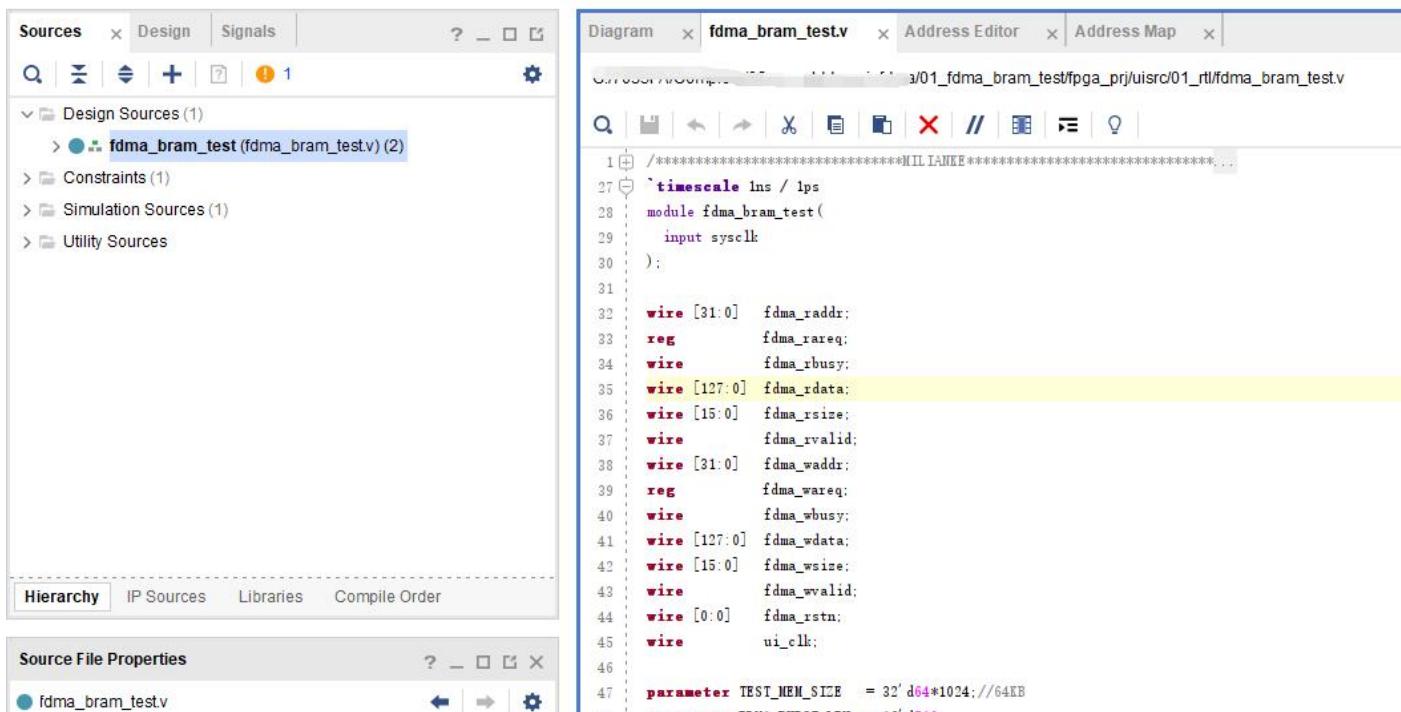
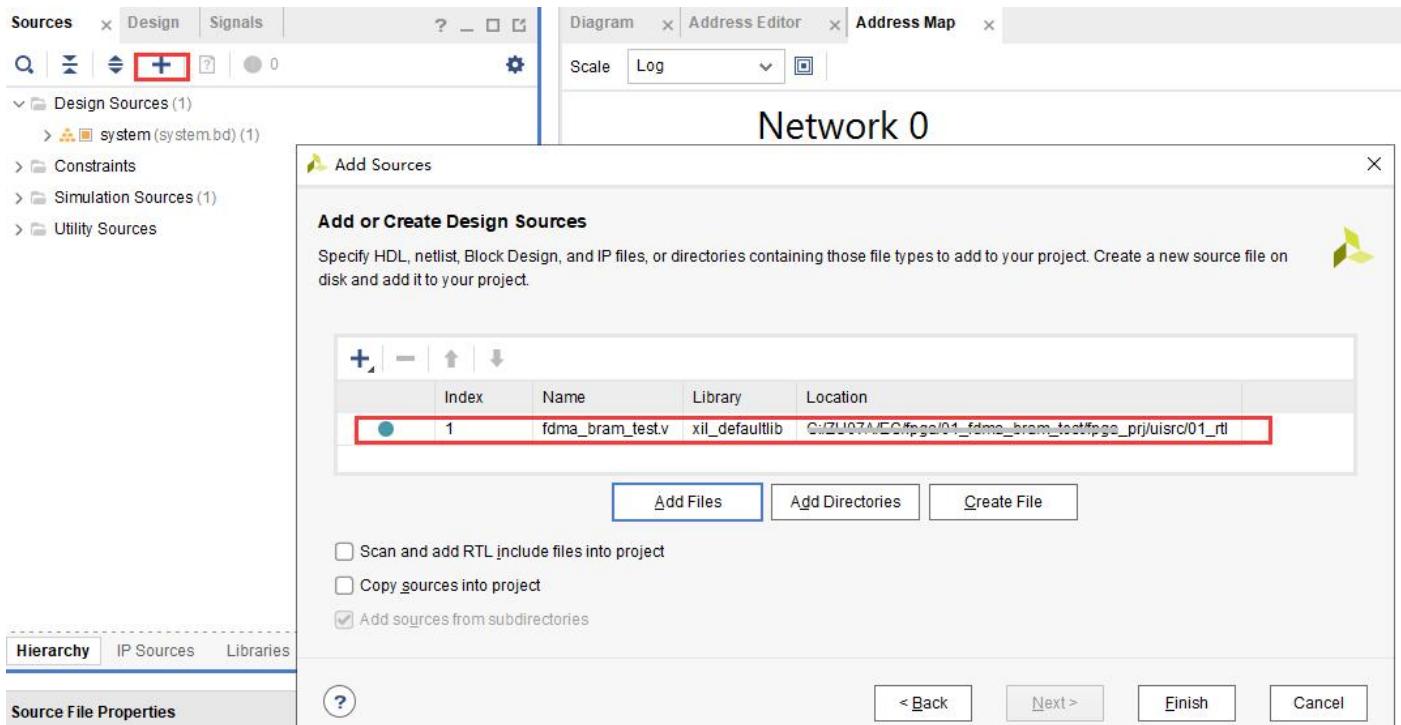
为了方便文件的管理，我们新建一个 fdma_bram_test.v 文件，并且复制以上代码，到这个文件夹。



右击删除刚刚自动产生的文件



添加 fdma_bram_test.v 文件



fdma_bram_test.v 文件

```
*****MILIANKE*****
*Company : MilianKe Electronic Technology Co., Ltd.
*WebSite:https://www.milianke.com
*TechWeb:https://www.uisrc.com
*tmall-shop:https://milianke.tmall.com
*jd-shop:https://milianke.jd.com
*taobao-shop1: https://milianke.taobao.com
*Create Date: 2021/04/25
*Module Name:fdma_bram_test
*File Name:fdma_bram_test.v
*Description:
*The reference demo provided by Milianke is only used for learning.
```

```

*We cannot ensure that the demo itself is free of bugs, so users
*should be responsible for the technical problems and consequences
*caused by the use of their own products.

*Copyright: Copyright (c) MiLianKe
*All rights reserved.

*Revision: 1.0

*Signal description

*1) _i input
*2) _o output
*3) _n activ low
*4) _dg debug signal
*5) _r delay or register
*6) _s state machine
******/



`timescale 1ns / 1ps

module fdma_bram_test(
    input sysclk
);

wire [31:0] fdma_raddr;
reg         fdma_rareq;
wire         fdma_rbusy;
wire [127:0] fdma_rdata;
wire [15:0]  fdma_rsize;
wire         fdma_rvalid;
wire [31:0]  fdma_waddr;
reg          fdma_wreq;
wire         fdma_wbusy;
wire [127:0] fdma_wdata;
wire [15:0]  fdma_wsize;
wire         fdma_wvalid;
wire [0:0]   fdma_rstn;
wire         ui_clk;

parameter TEST_MEM_SIZE = 32'd64*1024;//64KB
parameter FDMA_BURST_LEN = 16'd512;      //测试一次的长度
parameter ADDR_MEM_OFFSET = 0;           //地址偏移量
parameter ADDR_INC = FDMA_BURST_LEN * 16;

parameter WRITE1 = 0;
parameter WRITE2 = 1;
parameter WAIT   = 2;
parameter READ1  = 3;
parameter READ2  = 4;

reg [31: 0] t_data;
reg [31: 0] fdma_waddr_r;
reg [2 :0] T_S = 0;

```

```

assign fdma_waddr = fdma_waddr_r + ADDR_MEM_OFFSET; //设置偏移地址
assign fdma_raddr = fdma_waddr; //读写地址相同

assign fdma_wsize = FDMA_BURST_LEN; //设置 FDMA 控制器一次写 burst 的数据长度
assign fdma_rsize = FDMA_BURST_LEN; //设置 FDMA 控制器一次读 burst 的数据长度
assign fdma_wdata = {t_data,t_data,t_data,t_data};

//delay reset
reg [8:0] rst_cnt = 0;
always @(posedge ui_clk)
if(rst_cnt[8] == 1'b0)
    rst_cnt <= rst_cnt + 1'b1;
else
    rst_cnt <= rst_cnt;

//FDMA 读写控制器，每次先写后读，读出后对比数据正确性
always @(posedge ui_clk)begin
if(rst_cnt[8] == 1'b0)begin
    T_S <=0;
    fdma_wareq <= 1'b0;
    fdma_rareq <= 1'b0;
    t_data<=0;
    fdma_waddr_r <=0;
end
else begin
    case(T_S)
        WRITE1:begin
            if(fdma_waddr_r==TEST_MEM_SIZE) fdma_waddr_r<=0; //超出测试内存范围，重新测试
            if(!fdma_wbusy)begin //当 fdma 进入空闲，fdma_wbusy=0，请求写
                fdma_wareq <= 1'b1; //设置写请求
                t_data <= 0; //设置初值
            end
            if(fdma_wareq&&fdma_wbusy)begin //当 fdma 响应请求后，fdma_wbusy=1，进入下一个状态
                fdma_wareq <= 1'b0; //清除写请求
                T_S      <= WRITE2;
            end
        end
        WRITE2:begin
            if(!fdma_wbusy) begin //当 fdma 完成请求后，fdma_wbusy=0，进入下一个状态
                T_S <= WAIT;
                t_data <= 32'd0;
            end
            else if(fdma_wvalid) begin //当 fdma_wvalid 有效期间必须写入有效数据
                t_data <= t_data + 1'b1;
            end
        end
        WAIT:begin//not needed
            T_S <= READ1;
        end
    endcase
end
end

```

```

end

READ1:begin
    if(!fdma_rbusy)begin //当 fdma 进入空闲, fdma_rbusy=0, 请求读
        fdma_rreq <= 1'b1; //设置读请求
        t_data <= 0; //设置初值
    end
    if(fdma_rreq&&fdma_rbusy)begin //当 fdma 响应请求后, fdma_rbusy=1, 进入下一个状态
        fdma_rreq <= 1'b0; //清除读请求
        T_S <= READ2;
    end
end

READ2:begin
    if(!fdma_rbusy) begin //当 fdma 完成请求后, fdma_rbusy=0, 进入下一个状态
        T_S <= WRITE1;
        t_data <= 32'd0;
        fdma_waddr_r <= fdma_waddr_r + ADDR_INC;//128/8=16 //当本次读写周期完成增加地址, 地址以 BYTE 计算
    end
    else if(fdma_rvalid) begin //当 fdma_rvalid 有效期间读出的数据有效
        t_data <= t_data + 1'b1;
    end
end
default:
    T_S <= WRITE1;
endcase
end
end

//对比是否有错误数据

wire test_error = (fdma_rvalid && (t_data[15:0] != fdma_rdata[15:0]));
//ila 在线调试核的调用用于观察 fdma 运行的情况

ila_0 ila_dbg (
    .clk(ui_clk),
    .probe0({fdma_wdata[15:0],fdma_wareq,fdma_wvalid,fdma_wbusy}),
    .probe1({fdma_rdata[15:0],t_data[15:0],fdma_rvalid,fdma_rbusy,T_S,test_error})
);
system system_i
(
    .FDMA_S_0_fdma_raddr(fdma_raddr),
    .FDMA_S_0_fdma_rreq(fdma_rreq),
    .FDMA_S_0_fdma_rbusy(fdma_rbusy),
    .FDMA_S_0_fdma_rdata(fdma_rdata),
    .FDMA_S_0_fdma_rrready(1'b1),
    .FDMA_S_0_fdma_rsize(fdma_rsize),
    .FDMA_S_0_fdma_rvalid(fdma_rvalid),
    .FDMA_S_0_fdma_waddr(fdma_waddr),
    .FDMA_S_0_fdma_wareq(fdma_wareq),
    .FDMA_S_0_fdma_wbusy(fdma_wbusy),
    .FDMA_S_0_fdma_wdata(fdma_wdata),
    .FDMA_S_0_fdma_wready(1'b1),
    .FDMA_S_0_fdma_wsize(fdma_wsize),
    .FDMA_S_0_fdma_wvalid(fdma_wvalid),

```

```

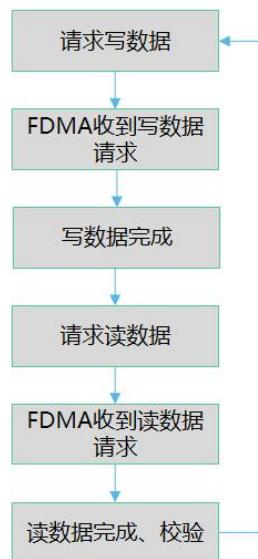
    .sysclk(sysclk),
    .ui_clk(ui_clk)
);
endmodule

```

3.5 程序分析

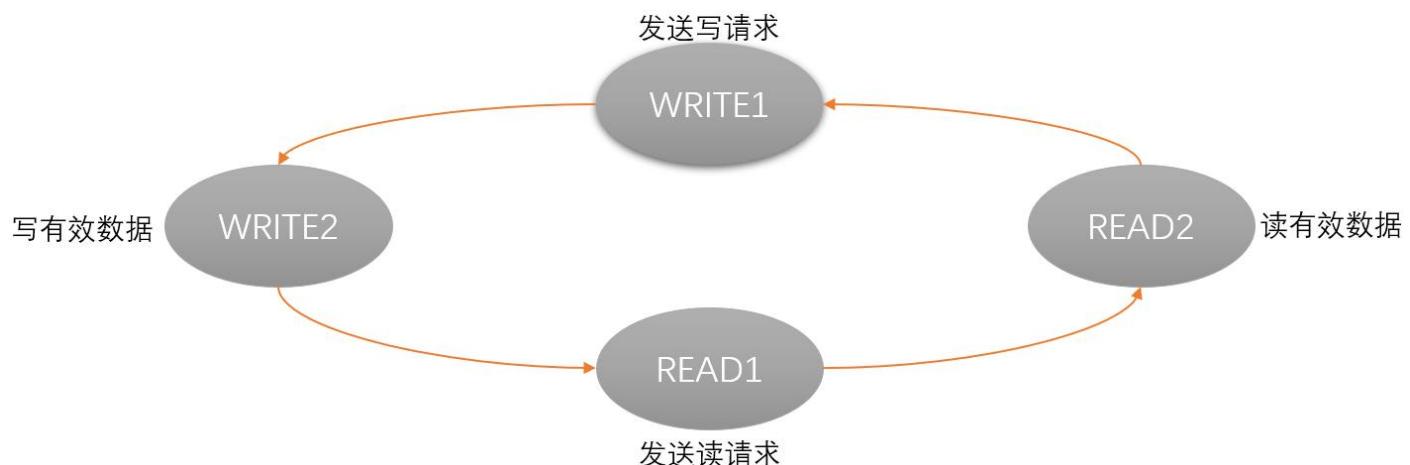
1:总流程图

如下图所示，本文的程序工作流程如下，包括请求写数据、FDMA 收到写数据请求、写数据完成、请求读数据、FDMA 收到读数据请求和读数据完成、校验。

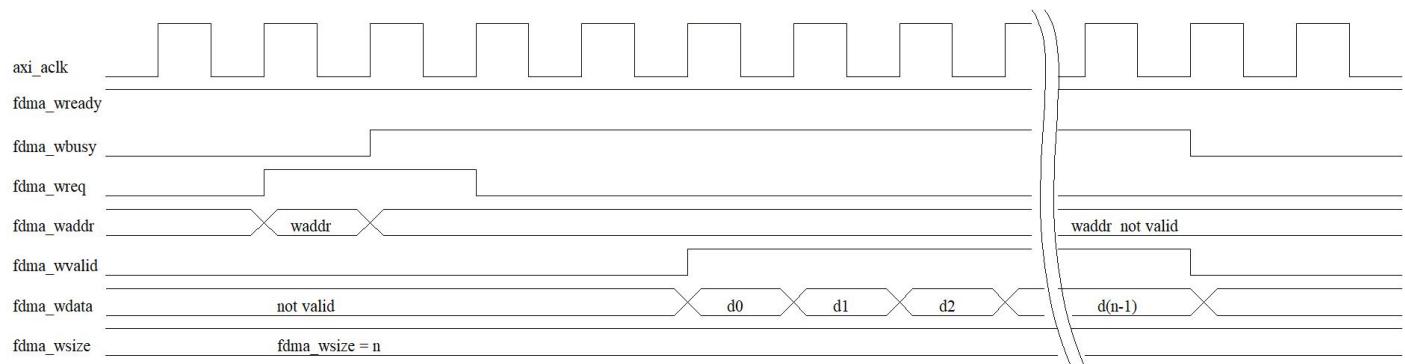


2:FDMA 的读写时序

以下是 fdma_test.v 中读写 fdma ip 接口的状态机，由于读写代码对称，对 fdma 的读写操作可以分为 2 步完成，分别是：1:发送读/写请求 2:读/写有效数据。



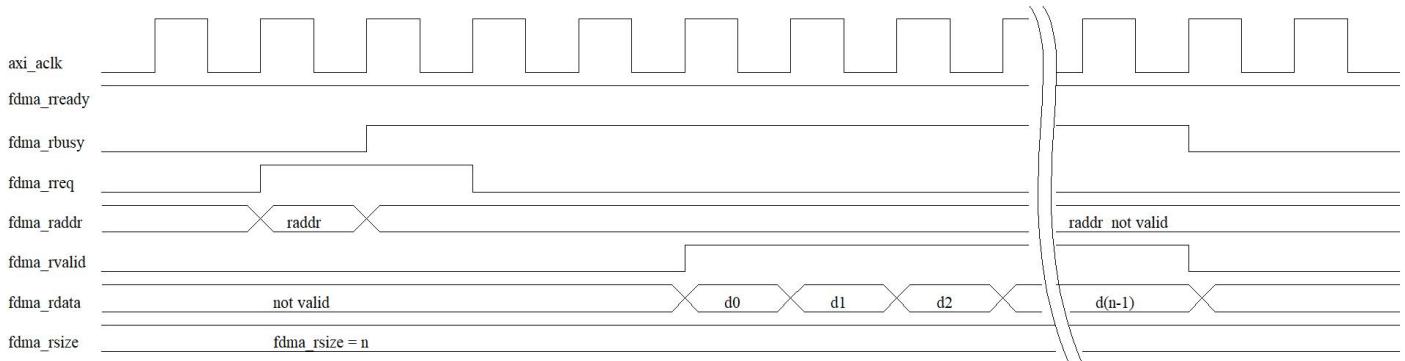
2-1:FDMA 的写时序



fdma_wready 设置为 1，当 fdma_wbusy=0 的时候代表 FDMA 的总线非忙，可以进行一次新的 FDMA 传输，这个时候可以设置 fdma_wreq=1，同时设置 fdma burst 的起始地址和 fdma_wszie 本次需要传输的数据大小(以 bytes 为单位)。当 fdma_wvalid=1 的时候需要给出有效的数据，写入 AXI 总线。当最后一个数写完后，fdma_wvalid 和 fdma_wbusy 变为 0。

AXI4 总线最大的 burst length 是 256，而经过封装后，用户接口的 fdma_size 可以任意大小的，fdma ip 内部代码控制每次 AXI4 总线的 Burst 长度，这样极大简化了 AXI4 总线协议的使用。

2-2:FDMA 的读时序



fdma_rready 设置为 1，当 fdma_rbusy=0 的时候代表 FDMA 的总线非忙，可以进行一次新的 FDMA 传输，这个时候可以设置 fdma_rreq=1，同时设置 fdma burst 的起始地址和 fdma_rsize 本次需要传输的数据大小(以 bytes 为单位)。当 fdma_rvalid=1 的时候需要给出有效的数据，写入 AXI 总线。当最后一个数写完后，fdma_rvalid 和 fdma_rbusy 变为 0。

同样对于 AXI4 总线的读操作，AXI4 总线最大的 burst length 是 256，而经过封装后，用户接口的 fdma_size 可以任意大小的，fdma ip 内部代码控制每次 AXI4 总线的 Burst 长度，这样极大简化了 AXI4 总线协议的使用。

3.6RTL 仿真

1:仿真 tb 文件

编写仿真 tb 文件，fdma_bram_test_tb.v，非常简单，只需要给一个时钟

```
*****MILIANKE*****
*Company : MiLianKe Electronic Technology Co., Ltd.
*WebSite:https://www.milianke.com
*TechWeb:https://www.uisrc.com
*tmall-shop:https://milianke.tmall.com
*jd-shop:https://milianke.jd.com
*taobao-shop1: https://milianke.taobao.com
*Create Date: 2021/04/25
*Module Name:fdma_bram_test_tb
```

```
*File Name:fdma_bram_test_tb.v

*Description:
*The reference demo provided by Milianke is only used for learning.
*We cannot ensure that the demo itself is free of bugs, so users
*should be responsible for the technical problems and consequences
*caused by the use of their own products.

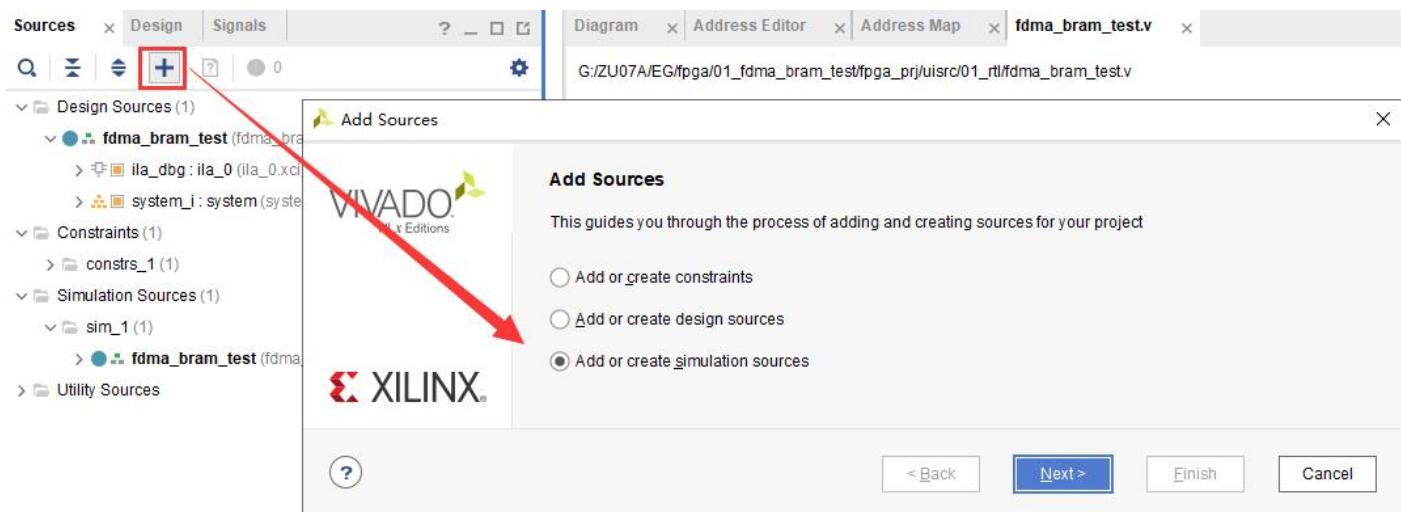
*Copyright: Copyright (c) MiLianKe
*All rights reserved.

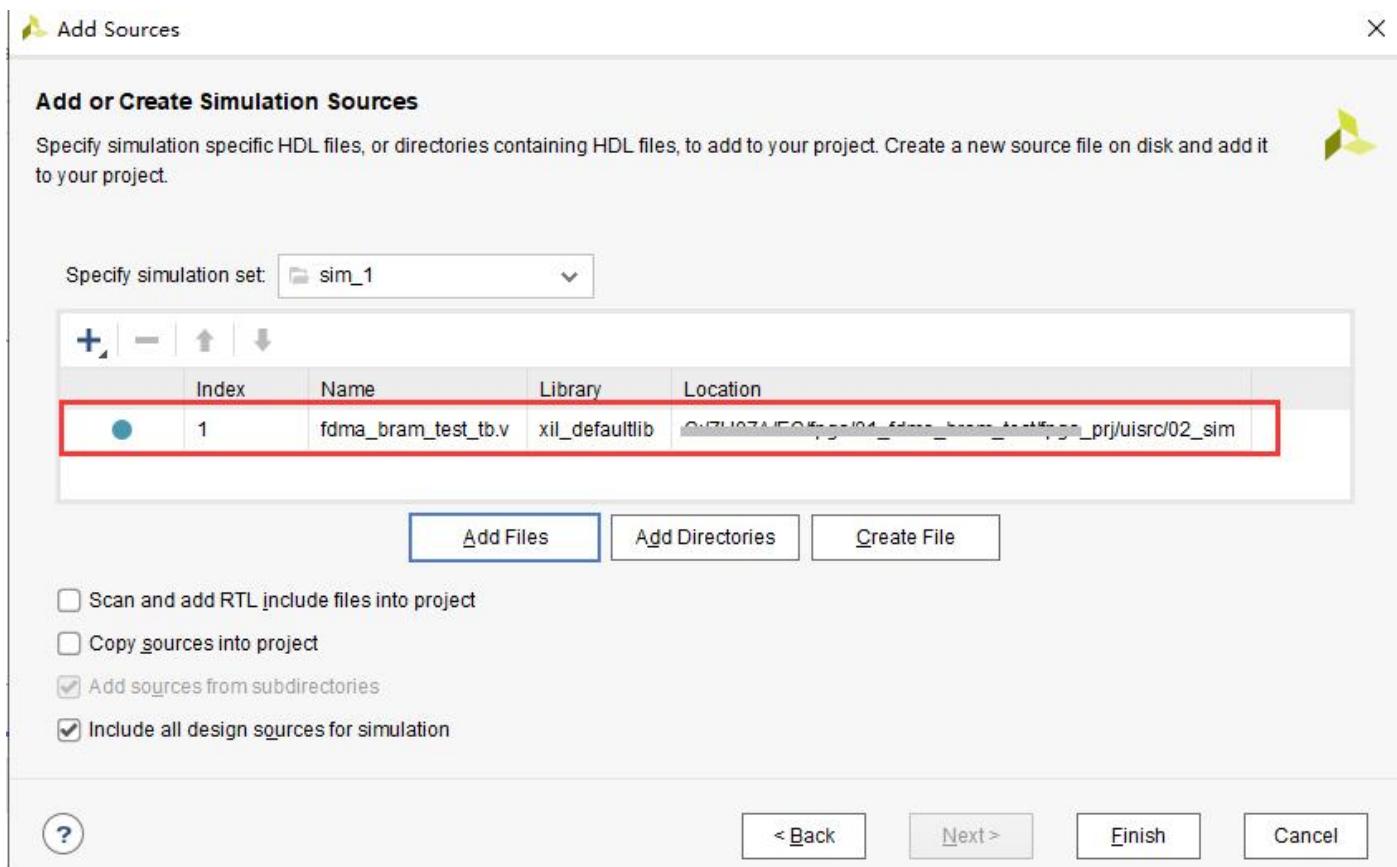
*Revision: 1.0

*Signal description
*1) _i input
*2) _o output
*3) _n activ low
*4) _dg debug signal
*5) _r delay or register
*6) _s state machine
*****
`timescale 1ns / 1ps

module fdma_bram_test_tb();
reg sysclk;
fdma_bram_test fdma_bram_test_inst
(
    .sysclk(sysclk)
);
initial begin
    sysclk = 0;
    #100;
end
always #10 sysclk = ~sysclk;
endmodule
```

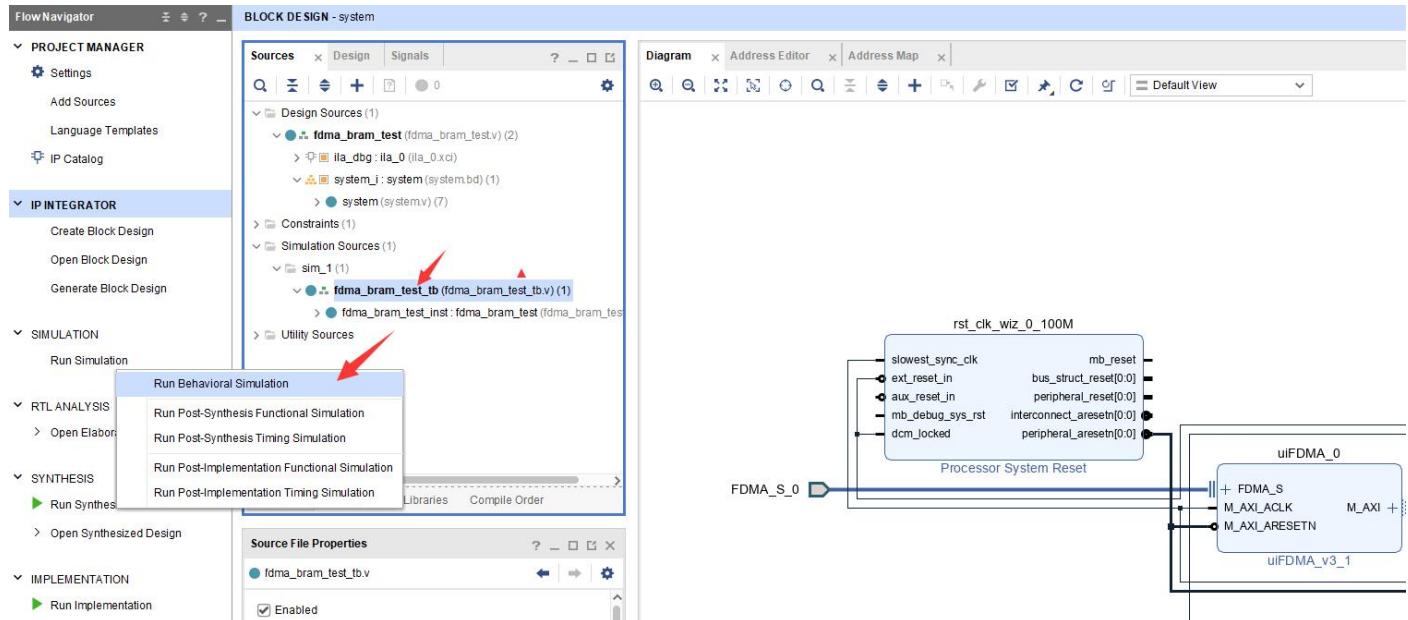
添加 fdma_bram_test_tb.v 到工程中





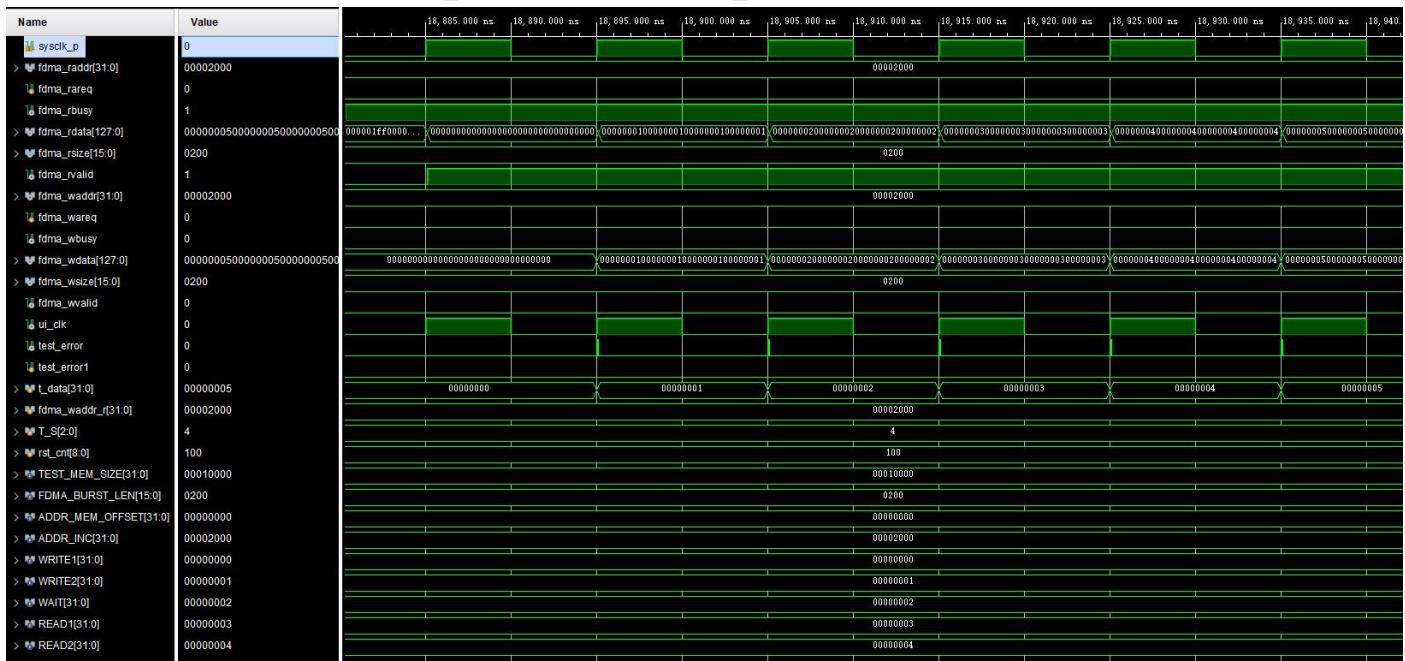
2:仿真测试

进行 RTL 行为仿真





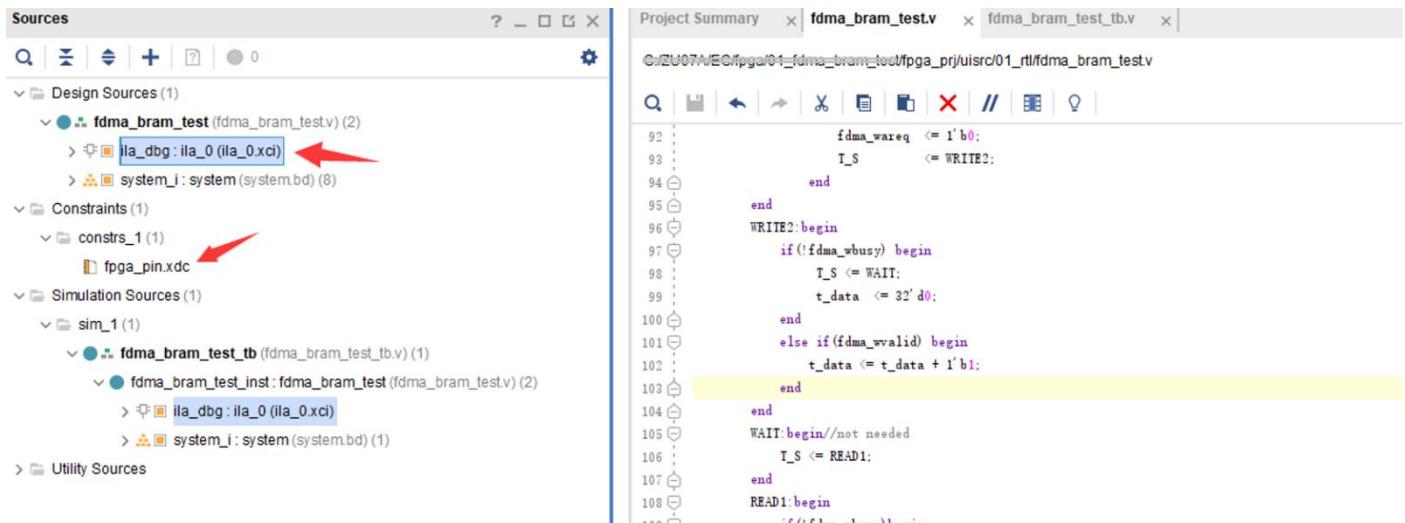
放大后观察数据，我们可以看到 test_error 有毛刺，但是 test_error 打拍后，毛刺就没有了。



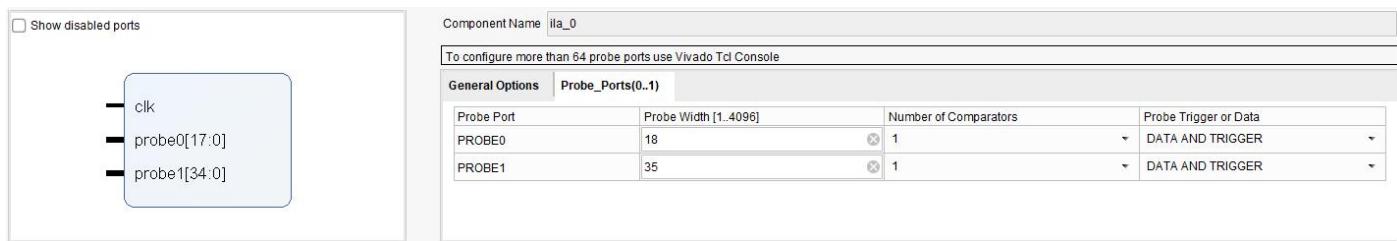
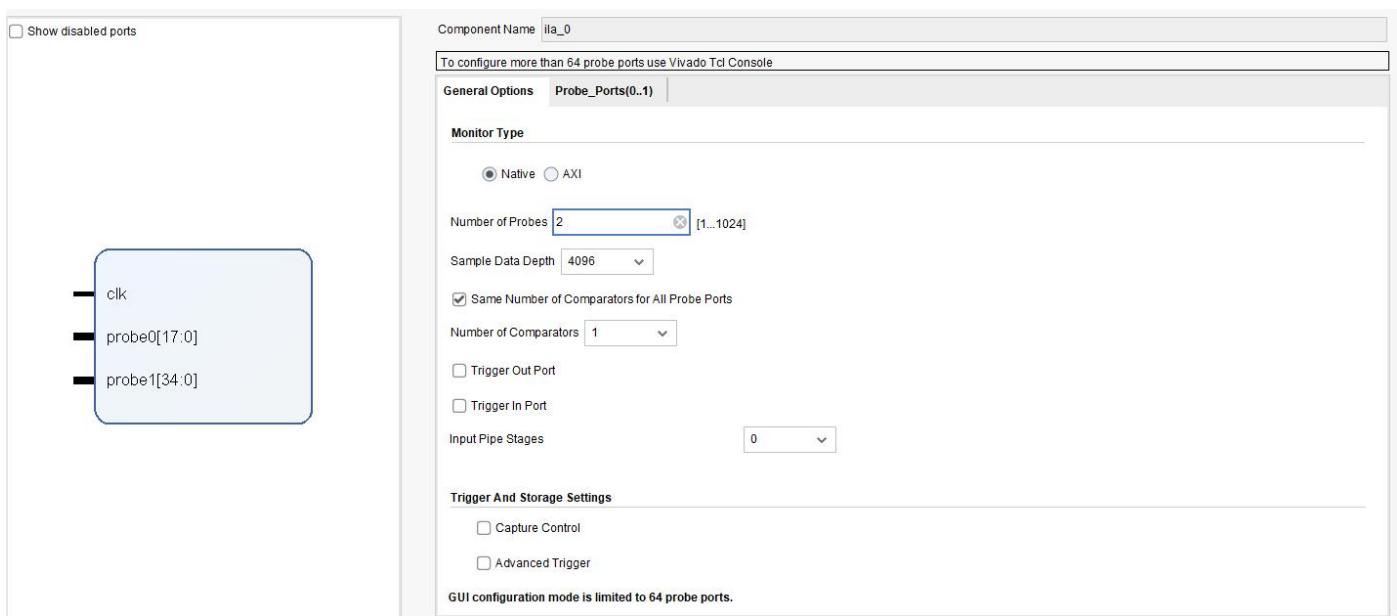
FDMA 源码部分的分析，在前面课程中已经分析过，这里不再重复。

3.7 编译测试

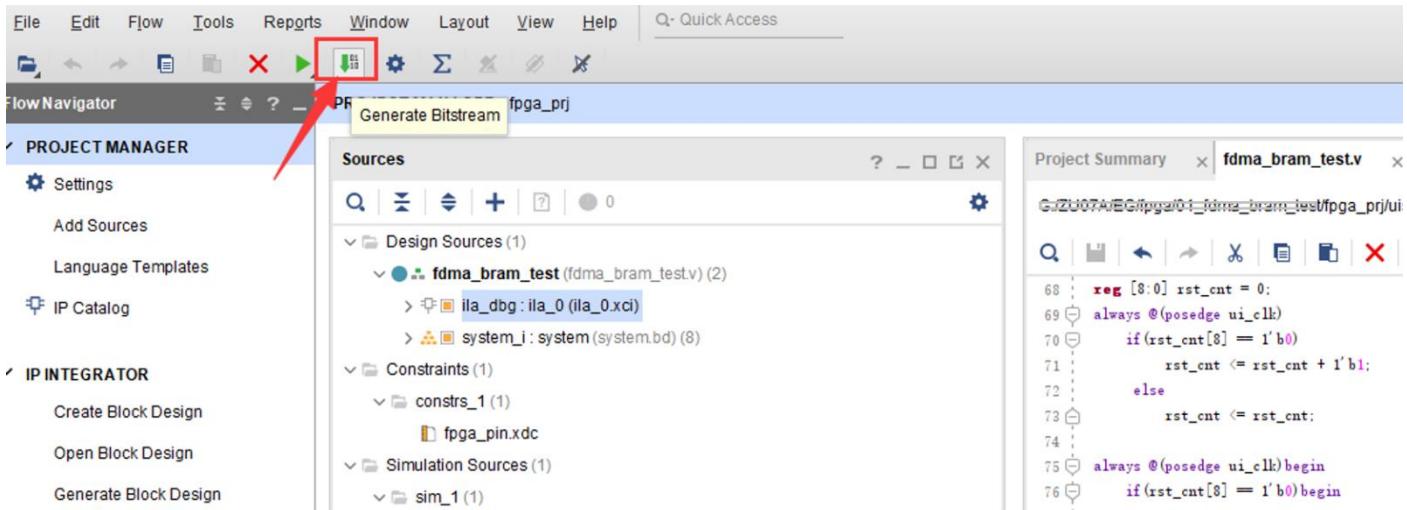
添加 ila IP CORE 和 fpga pin 定义



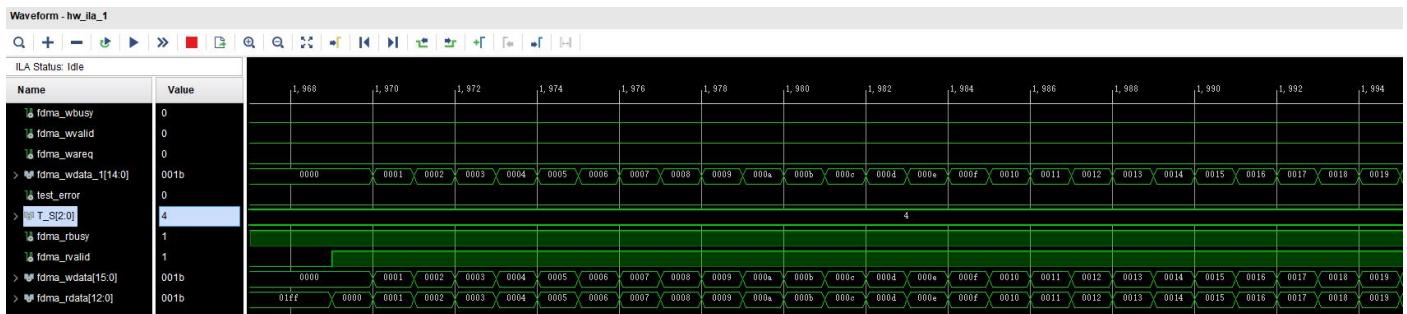
ila 的设置如下



编译产生 bit



下载程序到开发板测试，通过在线逻辑分析仪观察



04 使用 fdma 读写 DDR

软件版本：vitis2021.1(vivado2021.1)

操作系统：WIN10 64bit

硬件平台：适用 XILINX A7/K7/Z7/ZU/KU 系列 FPGA

登录“米联客”FPGA 社区-www.uisrc.com 视频课程、答疑解惑！

4.1 概述

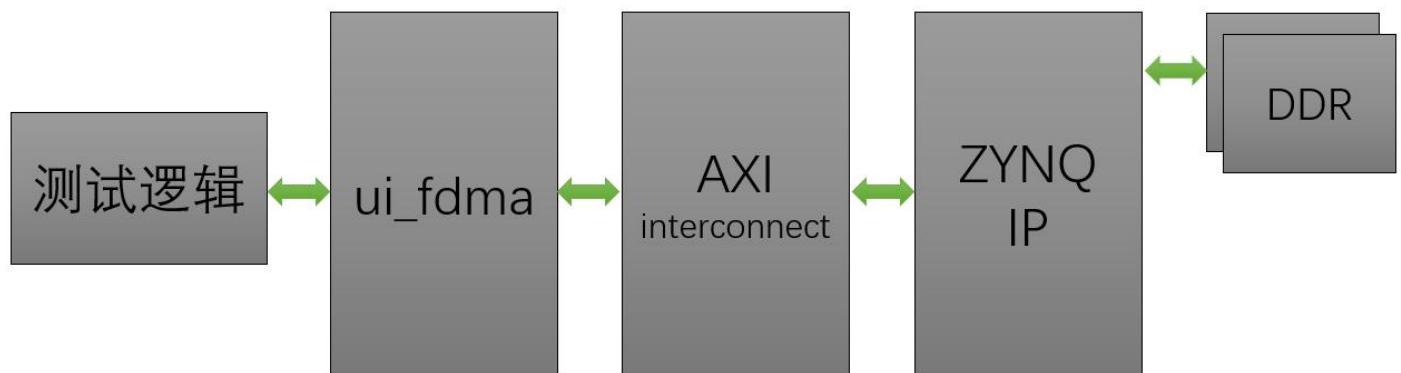
在前文的实验中我们详细介绍了 FDMA 的使用方法，以及使用了 AXI-BRAM 演示了 FDMA 的使用，现在我们已经掌握了 FDMA 的使用，本文我们继续使用 FDMA 实现对 PS DDR 的读写。由于 FDMA 的读写操作都是基于 AXI 总线，所以用户代码部分一致性也非常好，我们的状态机都不需要做修改，基本上只要把前文的 BRAM IP 换成 ZYNQ IP 并且把 AXI4 数据接口对接到 ZYNQ 的 HP 接口即可。

本文实验目的：

- 1:利用 uiFDMA3.1 提供的接口，编写 DDR 测试程序
- 2:由于 ZYNQ IP 核方便不方便，不提供仿真验证，直接上板验证

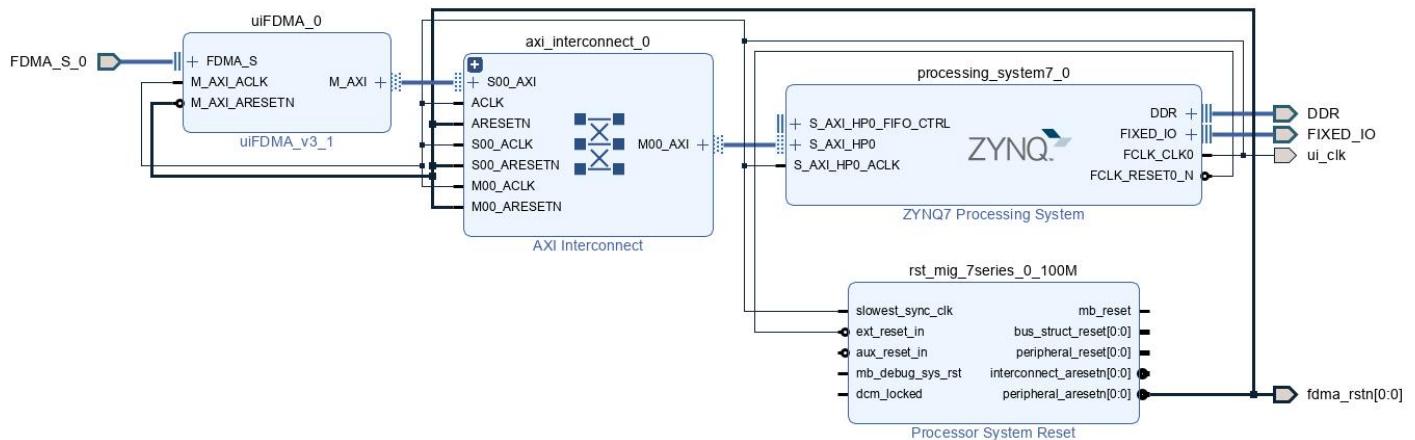
4.2 系统框图

本系统中先将测试数据通过 AXI-FDMA 写入 DDR,再通过 AXI-FDMA 将 DDR 中数据读出。将读写数据进行对比。通过在线逻辑分析仪抓取读写数据测试读写正确性。



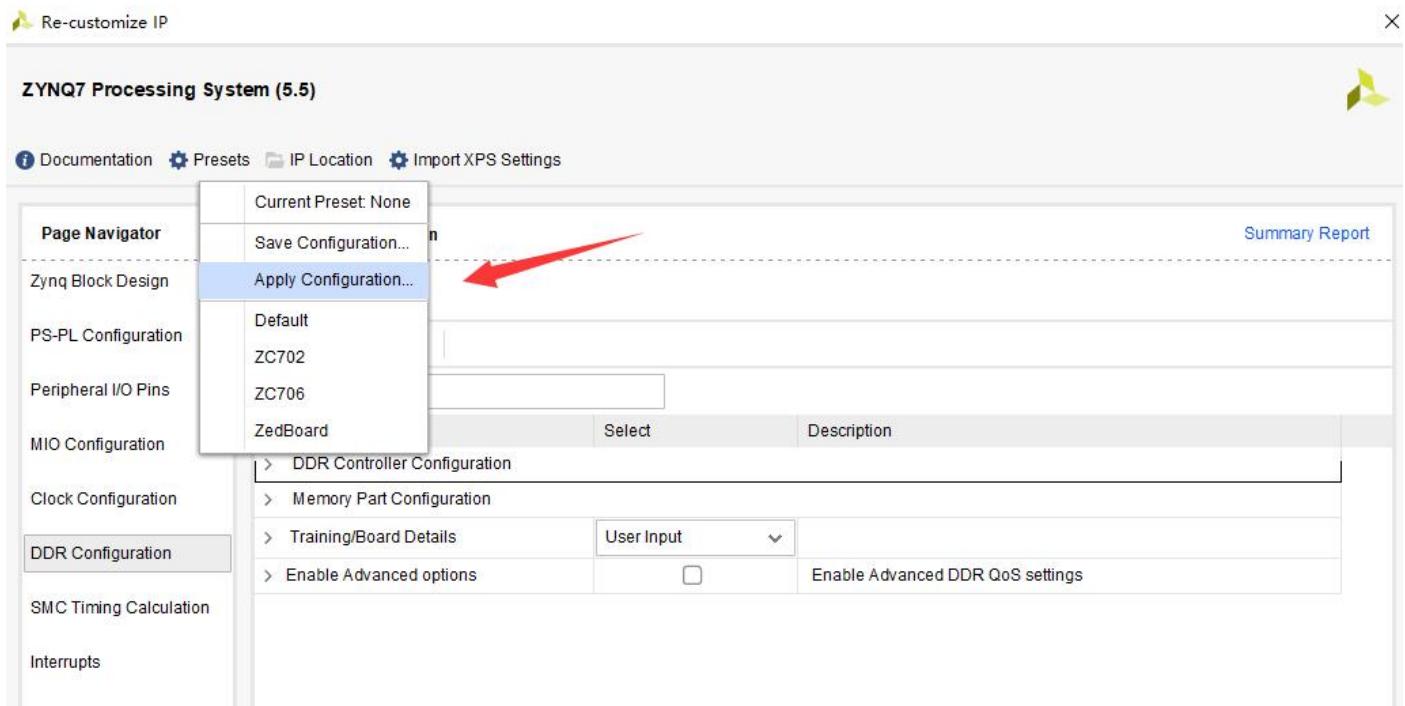
4.3 基于图形化的逻辑设计

详细的搭建过程这里不再重复，对于初学读者如果还不清楚如何创建 SOC 工程的，请学习“3-1-01 米联客 2022 版 ZynqSocSDK 入门篇”中第一个工程“01Vitis Soc 开发入门”这个实验。

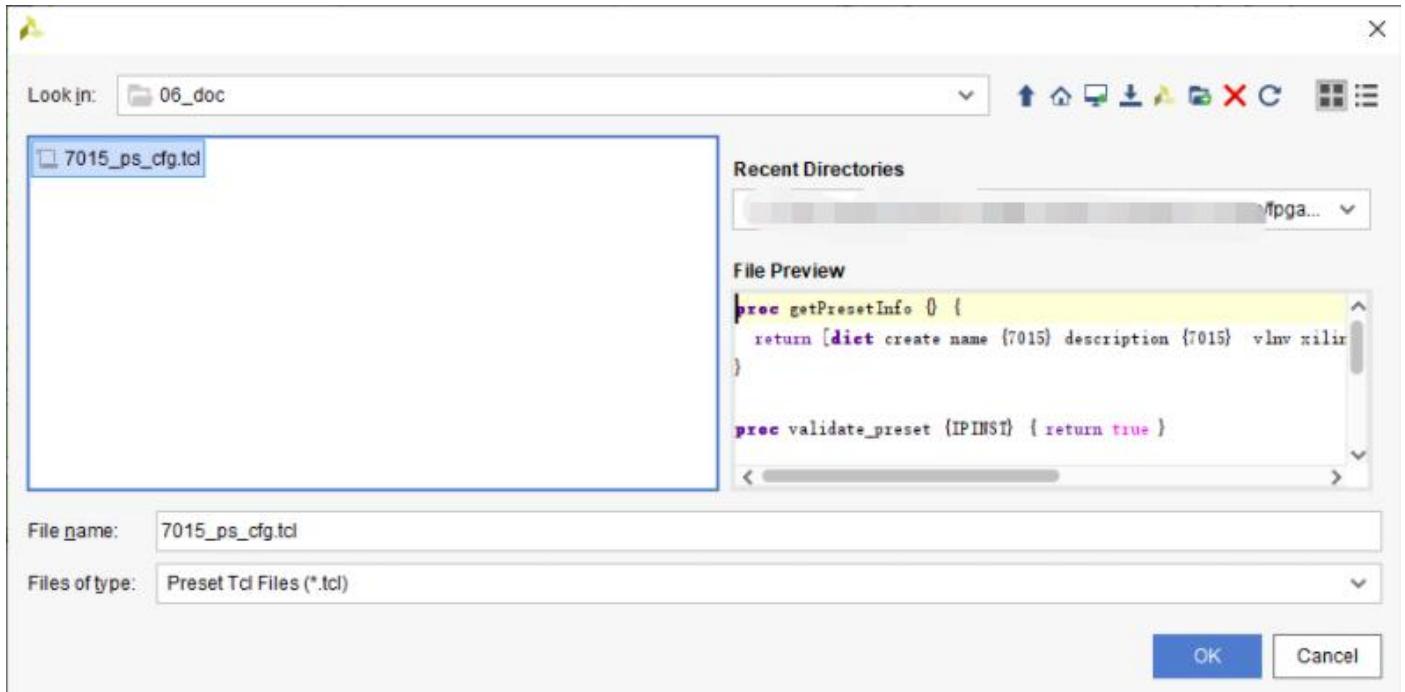


关于 ZYNQ 的 IP 配置可以直接导入我们已经提供的 m1k7x_ps_cfg.tcl 脚本文件, 该配置文件会完成包括: DDR 配置、MIO 配置、输入时钟配置等必要的配置, 在此基础上继续配置 IP。

导入配置参数方法如下, 双击 ZYNQ IP, 导入



选择对应开发板的配置文件单击 OK



我们主要看下 ZYNQ IP 中的本实验需要的关键配置

1:PS 复位设置

The screenshot shows the PS-PL Configuration page with the following settings:

Name	Select	Description
FLCK_CLKTRIG2	<input type="checkbox"/>	Enables PL clock trigger signal 2 used to halt the PL clock when counting
FLCK_CLKTRIG3	<input type="checkbox"/>	Enables PL clock trigger signal 3 used to halt the PL clock when counting
Enable Clock Resets		
FCLK_RESET0_N	<input checked="" type="checkbox"/>	Enables general purpose reset signal 0 for PL logic
FCLK_RESET1_N	<input type="checkbox"/>	Enables general purpose reset signal 1 for PL logic
FCLK_RESET2_N	<input type="checkbox"/>	Enables general purpose reset signal 2 for PL logic
FCLK_RESET3_N	<input type="checkbox"/>	Enables general purpose reset signal 3 for PL logic
AXI Non Secure Enablement		
1		Enable AXI Non Secure Transaction
GP Master AXI Interface		
M AXI GP0 interface	<input type="checkbox"/>	Enables General purpose AXI master interface 0
Static remap	0	Enables static remap for GP0 interface

2:设置 PS HP Slave 接口

The screenshot shows the PS-PL Configuration page with the following settings:

Name	Select	Description
General		
AXI Non Secure Enablement	0	Enable AXI Non Secure Transaction
GP Slave AXI Interface		
S AXI GP0 interface	<input type="checkbox"/>	Enables General purpose 32-bit AXI Slave interface 0
S AXI GP1 interface	<input type="checkbox"/>	Enables General purpose 32-bit AXI Slave interface 1
HP Slave AXI Interface		
> S AXI HP0 interface	<input checked="" type="checkbox"/>	Enables AXI high performance slave interface 0
> S AXI HP1 interface	<input type="checkbox"/>	Enables AXI high performance slave interface 1
> S AXI HP2 interface	<input type="checkbox"/>	Enables AXI high performance slave interface 2
> S AXI HP3 interface	<input type="checkbox"/>	Enables AXI high performance slave interface 3

3:PL 输出时钟设置

The screenshot shows the 'Clock Configuration' interface in Vivado. On the left, the 'Page Navigator' lists various design components like Zynq Block Design, PS-PL Configuration, Peripheral I/O Pins, MIO Configuration, and Clock Configuration (which is selected). The main area displays 'Basic Clocking' settings with an input frequency of 33.333333 MHz and a CPU clock ratio of 6:2:1. Below this is a search bar and a table titled 'Component' listing various clock sources and their properties. The table includes columns for Component, Clock Source, Requested Freq..., Actual Frequency..., and Range(MHz). Key entries include:

Component	Clock Source	Requested Freq...	Actual Frequency...	Range(MHz)
FCLK_CLK0	IO PLL	100	100.000000	0.100000 : 250.000000
FCLK_CLK1	IO PLL	25	10.000000	0.100000 : 250.000000
FCLK_CLK2	IO PLL	50	10.000000	0.100000 : 250.000000
FCLK_CLK3	IO PLL	50	10.000000	0.100000 : 250.000000

4:地址空间分配

配置完成后需要注意地址空间分配，FDMA IP 的内存起始地址从 0 开始

The screenshot shows the 'Address Map' window in Vivado. The top navigation bar includes tabs for Diagram, Address Editor (selected), and Address Map. Below the navigation bar are filter buttons for Assigned (1), Unassigned (0), and Excluded (0), along with a Hide All button. The main pane displays a hierarchical tree of memory regions. Under the '/uiFDMA_0' node, there is a single entry for the '/processing_system7_0/S_AXI_HP0' interface, which is assigned to the 'HP0_DDR_LOWOCM' slave segment. The memory range is set from 0x0000_0000 to 0x3FFF_FFFF, with a size of 1G.

5:编写 FDMA 的 DDR 测试代码

以下程序中是 fdma 读写操作的具体实现，先写入一定数据到 DDR 中，然后再读出，对比是否有错误，几个关键参数：

TEST_MEM_SIZE: 定义了测试内从空间的大小，以 byte 为单位,是整数倍的 FDMA_BURST_LEN *(fdma_wdata/8)。

FDMA_BURST_LEN: 定义每次 FDMA 传输的长度，这个长度是整数倍的 fdma_wdata 或者 fdma_rdata。

ADDR_MEM_OFFSET: 代码了内从访问的起始地址。

```
*****MILIANKE*****
*Company : MiLianKe Electronic Technology Co., Ltd.
*WebSite:https://www.milianke.com
*TechWeb:https://www.uisrc.com
*tmall-shop:https://milianke.tmall.com
*jd-shop:https://milianke.jd.com
*taobao-shop1: https://milianke.taobao.com
```

```

*Create Date: 2021/10/15
*Module Name:
*File Name:
*Description:
*config sensor resgister
*The reference demo provided by Milianke is only used for learning.
*We cannot ensure that the demo itself is free of bugs, so users
*should be responsible for the technical problems and consequences
*caused by the use of their own products.
*Copyright: Copyright (c) MiLianKe
*All rights reserved.
*Revision: 1.0
*Signal description
*1) _i input
*2) _o output
*3) _n activ low
*4) _dg debug signal
*5) _r delay or register
*6) _s state machine
*****/*
`timescale 1ns / 1ps

module fdma_test#
(
parameter TEST_MEM_SIZE    = 32'd409600,
parameter FDMA_BURST_LEN   = 16'd1024,
parameter ADDR_MEM_OFFSET = 1024*1024*20
)
(
input          ui_clk,
input          fdma_rstn,
output [31: 0] fdma_waddr,
output reg     fdma_wareq,
output [15: 0] fdma_wsize,
input          fdma_wbusy,
output reg [127:0] fdma_wdata,
input          fdma_wvalid,
output         fdma_wready,

output [31: 0] fdma_raddr,
output reg     fdma_rareq,
output [15: 0] fdma_rsize,
input          fdma_rbusy,
input [127:0]  fdma_rdata,
input          fdma_rvalid,
output         fdma_rready
);

assign fdma_rready = 1'b1;

```

```

assign fdma_wready = 1'b1;

parameter WRITE1 = 0;
parameter WRITE2 = 1;
parameter READ1 = 2;
parameter READ2 = 3;
parameter ADDR_INC = FDMA_BURST_LEN*16;

reg [31: 0] fdma_waddr_r;
reg [16 :0] fdma_rcnt = 0;
reg [2 :0] T_S = 0;

assign fdma_waddr = fdma_waddr_r + ADDR_MEM_OFFSET;
assign fdma_raddr = fdma_waddr;

assign fdma_wsize = FDMA_BURST_LEN;
assign fdma_rsize = FDMA_BURST_LEN;

reg [10:0] rst_cnt = 0;

always @(posedge ui_clk)
  if(fdma_rstn == 1'b0)begin
    rst_cnt <=0;
  end
  else begin
    if(rst_cnt[10] == 1'b0)
      rst_cnt <= rst_cnt + 1'b1;
    else
      rst_cnt <= rst_cnt;
  end
end

//FDMA 读写控制器，每次先写后读，读出后对比数据正确性
always @(posedge ui_clk)begin
  if(rst_cnt[8] == 1'b0)begin
    T_S <=0;
    fdma_wareq <= 1'b0;
    fdma_rareq <= 1'b0;
    t_data<=0;
    fdma_waddr_r <=0;
  end
  else begin
    case(T_S)
      WRITE1:begin
        if(fdma_waddr_r==TEST_MEM_SIZE) fdma_waddr_r<=0; //超出测试内存范围，重新测试
        if(!fdma_wbusy)begin //当 fdma 进入空闲，fdma_wbusy=0，请求写
          fdma_wareq <= 1'b1; //设置写请求
          t_data <= 0; //设置初值
        end
      end
    endcase
  end
end

```

```

    end

    if(fdma_wareq&&fdma_wbusy)begin //当 fdma 响应请求后, fdma_wbusy=1, 进入下一个状态
        fdma_wareq <= 1'b0; //清除写请求
        T_S <= WRITE2;
    end
end

WRITE2:begin
    if(!fdma_wbusy) begin //当 fdma 完成请求后, fdma_wbusy=0, 进入下一个状态
        T_S <= WAIT;
        t_data <= 32'd0;
    end
    else if(fdma_wvalid) begin //当 fdma_wvalid 有效期间必须写入有效数据
        t_data <= t_data + 1'b1;
    end
end

WAIT:begin//not needed
    T_S <= READ1;
end

READ1:begin
    if(!fdma_rbusy)begin //当 fdma 进入空闲, fdma_rbusy=0, 请求读
        fdma_rreq <= 1'b1; //设置读请求
        t_data <= 0; //设置初值
    end
    if(fdma_rreq&&fdma_rbusy)begin //当 fdma 响应请求后, fdma_rbusy=1, 进入下一个状态
        fdma_rreq <= 1'b0; //清除读请求
        T_S <= READ2;
    end
end

READ2:begin
    if(!fdma_rbusy) begin //当 fdma 完成请求后, fdma_rbusy=0, 进入下一个状态
        T_S <= WRITE1;
        t_data <= 32'd0;
        fdma_waddr_r <= fdma_waddr_r + ADDR_INC;//128/8=16 //当本次读写周期完成增加地址, 地址以 BYTE 计算
    end
    else if(fdma_rvalid) begin //当 fdma_rvalid 有效期间读出的数据有效
        t_data <= t_data + 1'b1;
    end
end

default:
    T_S <= WRITE1;
endcase
end
end

wire test_error = ((fdma_rready&&fdma_rvalid) && (fdma_rcnt[15:0] != fdma_rdata[15:0]));
ila_0 ila_dbg (
    .clk(ui_clk),
    .probe0({fdma_waddr[15:0],fdma_wdata[15:0],fdma_wareq,fdma_wvalid,fdma_wready,fdma_wbusy}),

```

```

    .probe1({fdma_rdata[15:0],fdma_rcnt[15:0],fdma_rvalid,fdma_rready,fdma_rbusy,T_S,test_error})
);

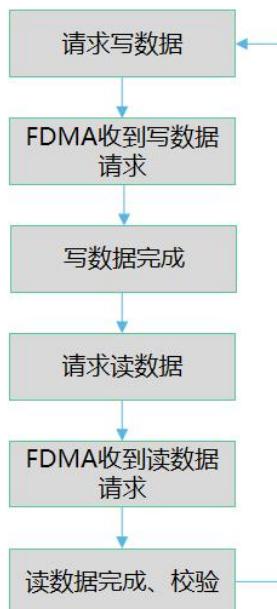
endmodule

```

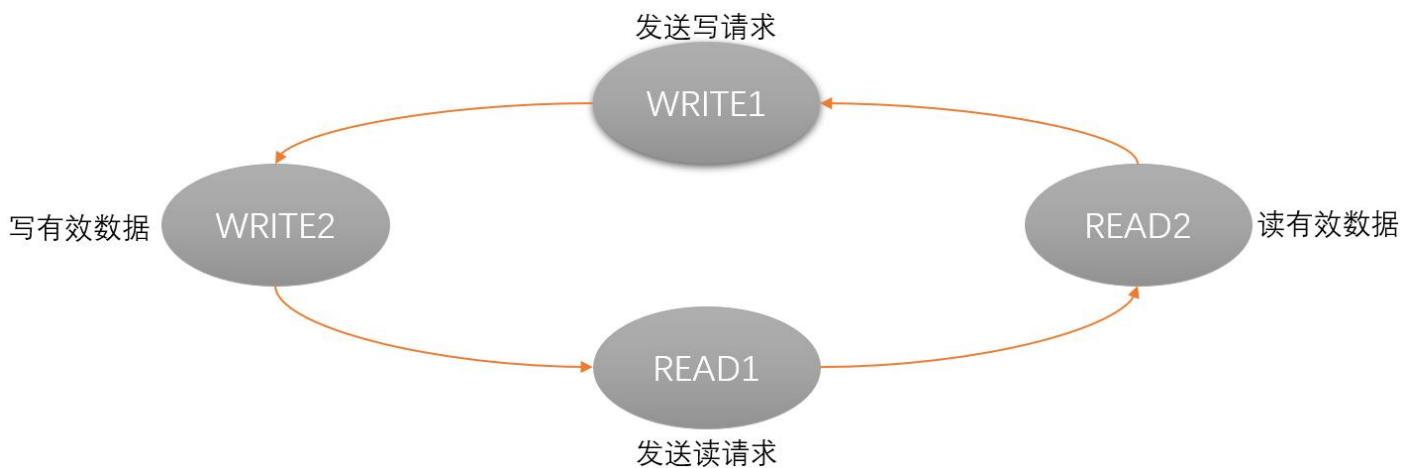
6:程序分析

6-1:总流程图

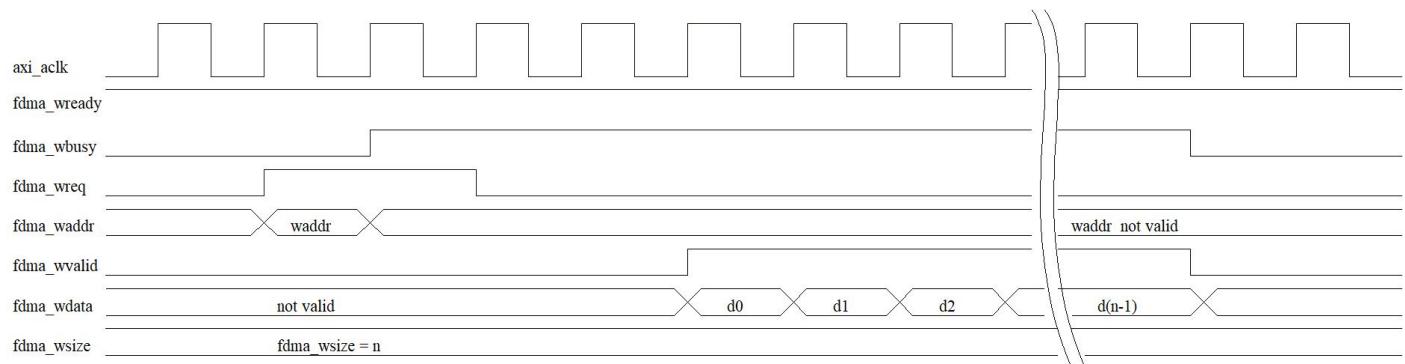
如下图所示，本文的程序工作流程如下，包括请求写数据、FDMA 收到写数据请求、写数据完成、请求读数据、FDMA 收到读数据请求和读数据完成、校验。



以下是 fdma_test.v 中读写 fdma ip 接口的状态机，由于读写代码对称，对 fdma 的读写操作可以分为 2 步完成，分别是：1:发送读/写请求 2:读/写有效数据。



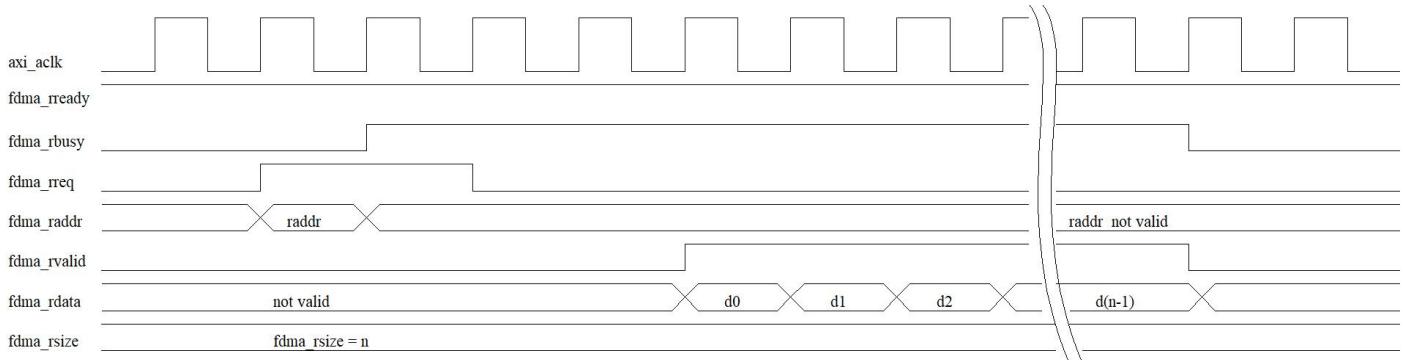
6-2:FDMA 的写时序



fdma_wready 设置为 1，当 fdma_wbusy=0 的时候代表 FDMA 的总线非忙，可以进行一次新的 FDMA 传输，这个时候可以设置 fdma_wreq=1，同时设置 fdma burst 的起始地址和 fdma_wszie 本次需要传输的数据大小(以 bytes 为单位)。当 fdma_wvalid=1 的时候需要给出有效的数据，写入 AXI 总线。当最后一个数写完后，fdma_wvalid 和 fdma_wbusy 变为 0。

AXI4 总线最大的 burst length 是 256，而经过封装后，用户接口的 fdma_size 可以任意大小的，fdma ip 内部代码控制每次 AXI4 总线的 Burst 长度，这样极大简化了 AXI4 总线协议的使用。

6-3:FDMA 的读时序



fdma_rready 设置为 1，当 fdma_rbusy=0 的时候代表 FDMA 的总线非忙，可以进行一次新的 FDMA 传输，这个时候可以设置 fdma_rreq=1，同时设置 fdma burst 的起始地址和 fdma_rsize 本次需要传输的数据大小(以 bytes 为单位)。当 fdma_rvalid=1 的时候需要给出有效的数据，写入 AXI 总线。当最后一个数写完后，fdma_rvalid 和 fdma_rbusy 变为 0。

同样对于 AXI4 总线的读操作，AXI4 总线最大的 burst length 是 256，而经过封装后，用户接口的 fdma_size 可以任意大小的，fdma ip 内部代码控制每次 AXI4 总线的 Burst 长度，这样极大简化了 AXI4 总线协议的使用。

7:编译并导出平台文件

1:单击 Block 文件 → 右键 → Generate the Output Products → Global → Generate。

2:单击 Block 文件 → 右键 → Create a HDL wrapper(生成 HDL 顶层文件) → Let vivado manager wrapper and auto-update(自动更新)。

3:生成 Bit 文件。

4:导出到硬件: File → Export Hardware → Include bitstream

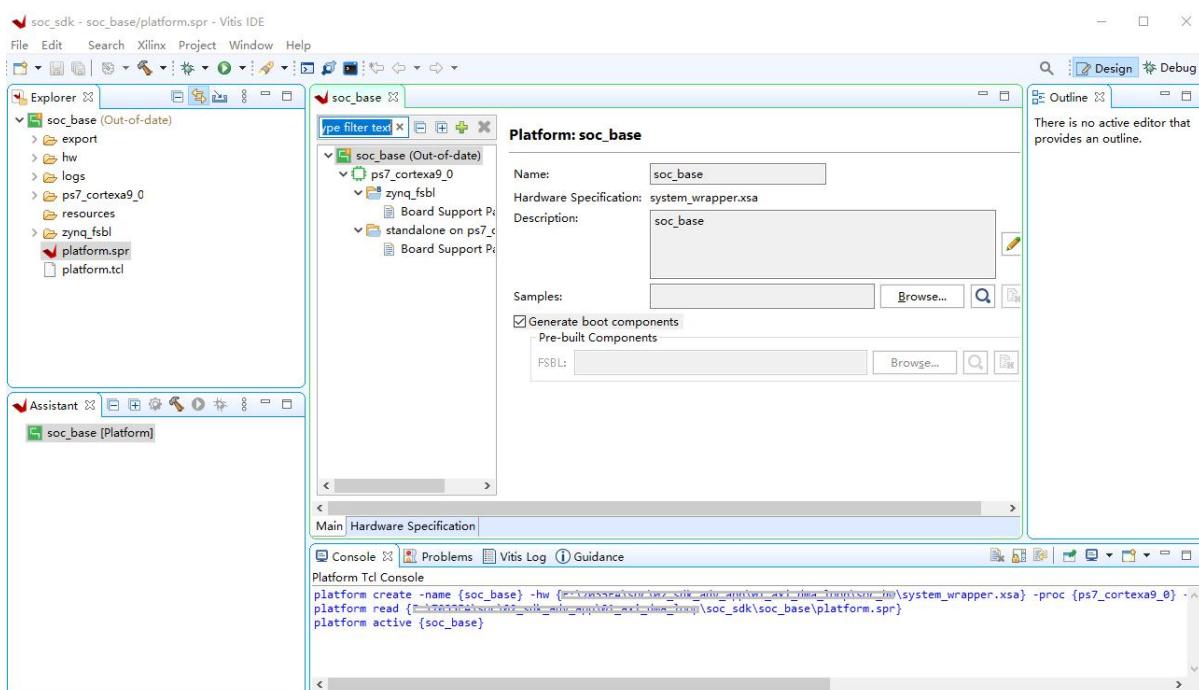
5:导出完成后，对应工程路径的 soc_hw 路径下有硬件平台文件: system_wrapper.xsa 的文件。根据硬件平台文件 system_wrapper.xsa 来创建需要 Platform 平台。



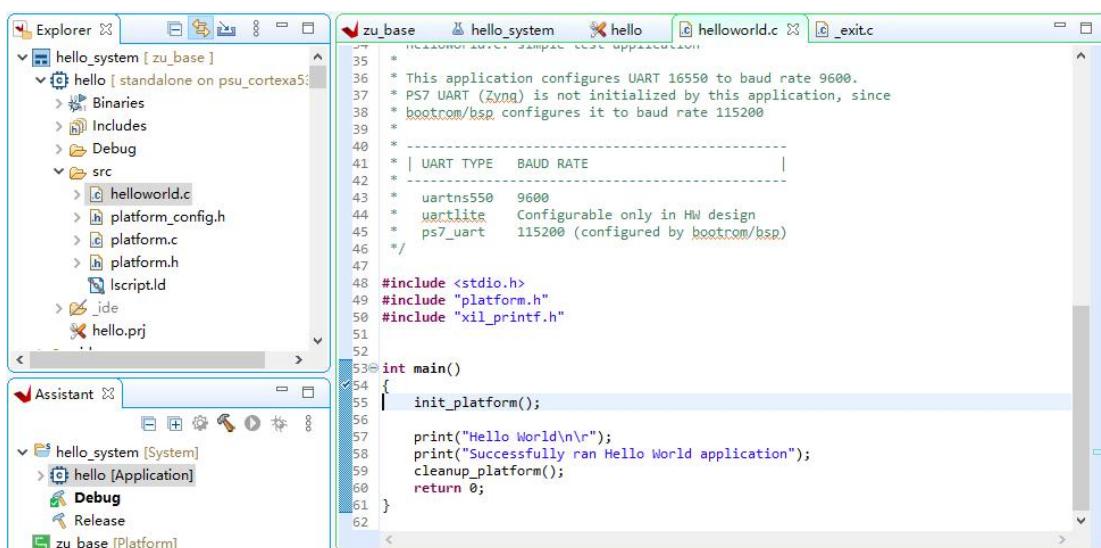
4.4 搭建 Vitis-sdk 工程

创建 soc_base sdk platform 和 APP 工程的过程不再重复，对于初学读者如果还不清楚如何创建 soc_sdk 工程的，请学习“3-1-01 米联客 2022 版 ZynqSocSDK 入门篇”中第一个工程“01Vitis Soc 开发入门”这个实验。

1: 创建 SDK Platform 工程



2: 创建 hello APP 工程



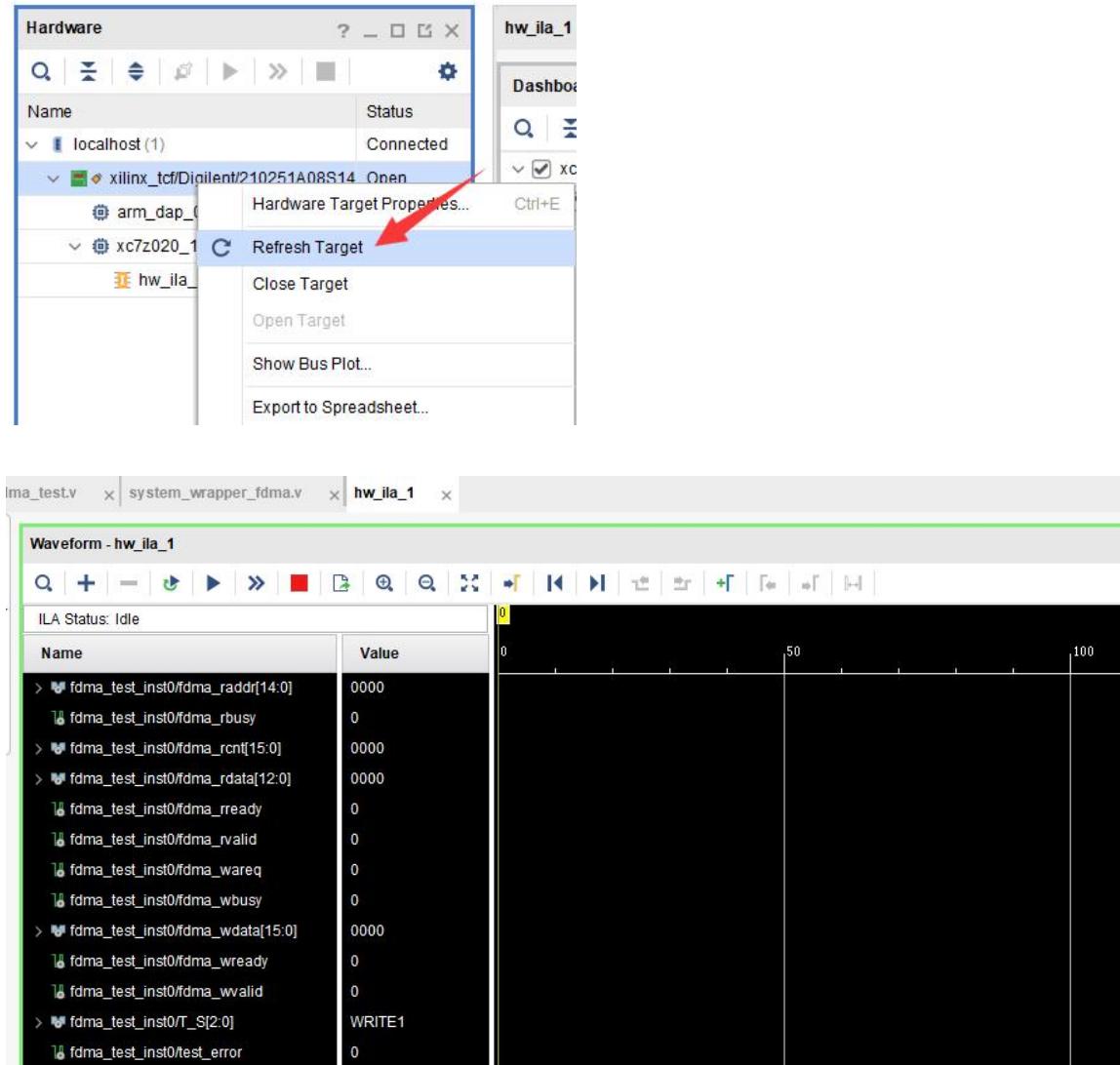
4.5 实验结果

1:由于 PS DDR 需要先运行 ARM 因此，先运行 hello app

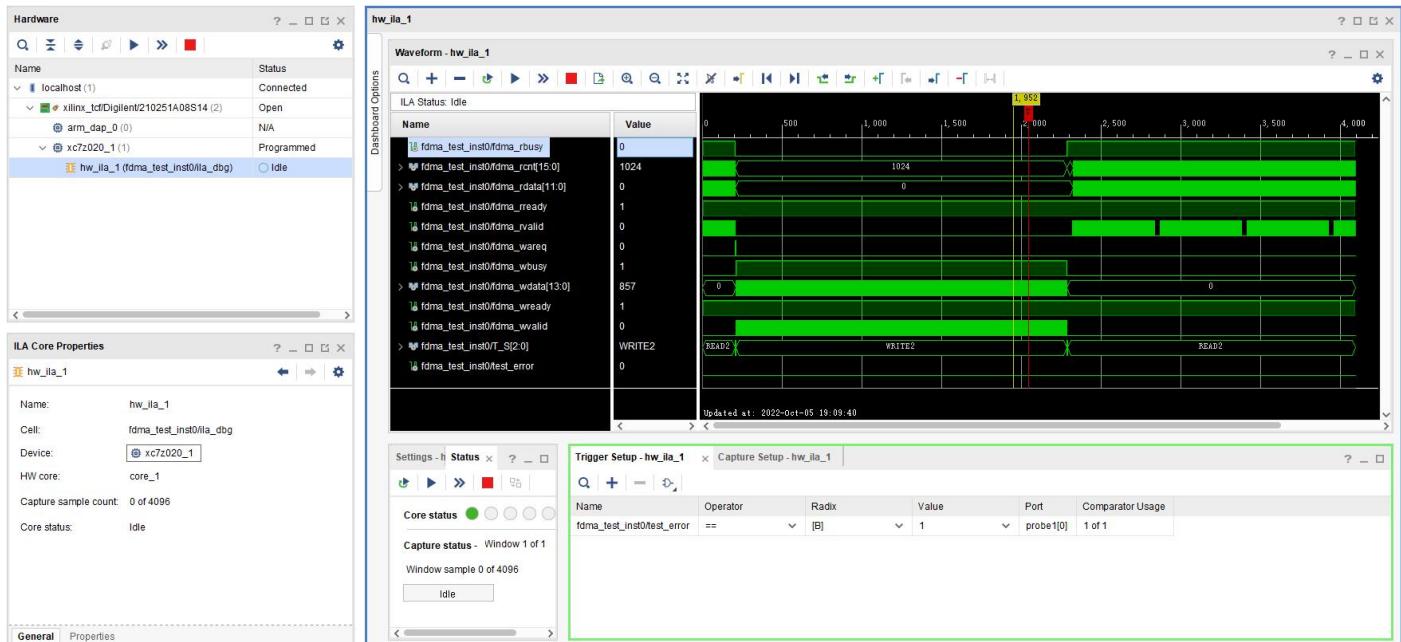
2:打开设备



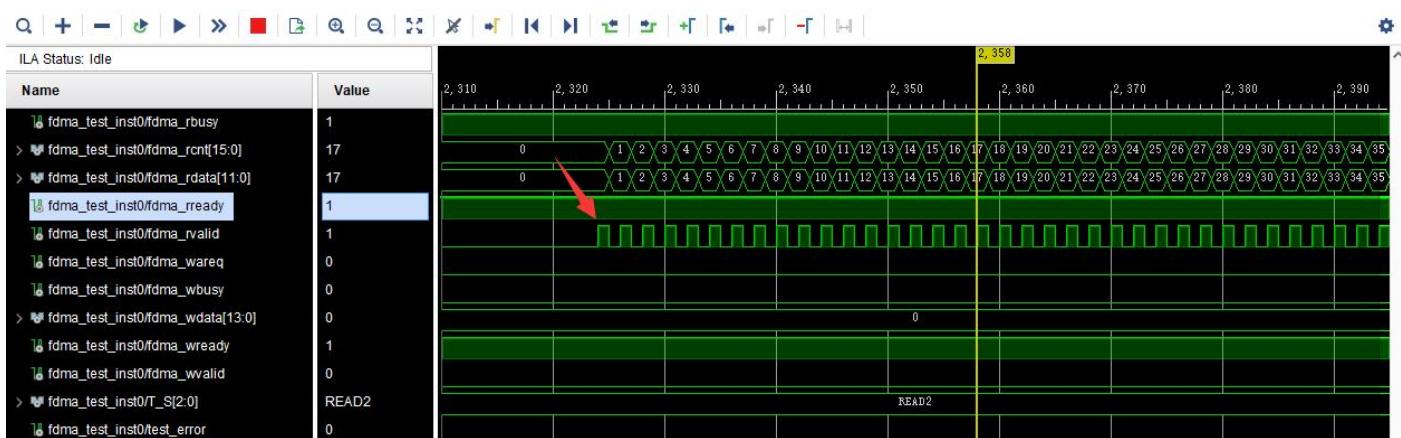
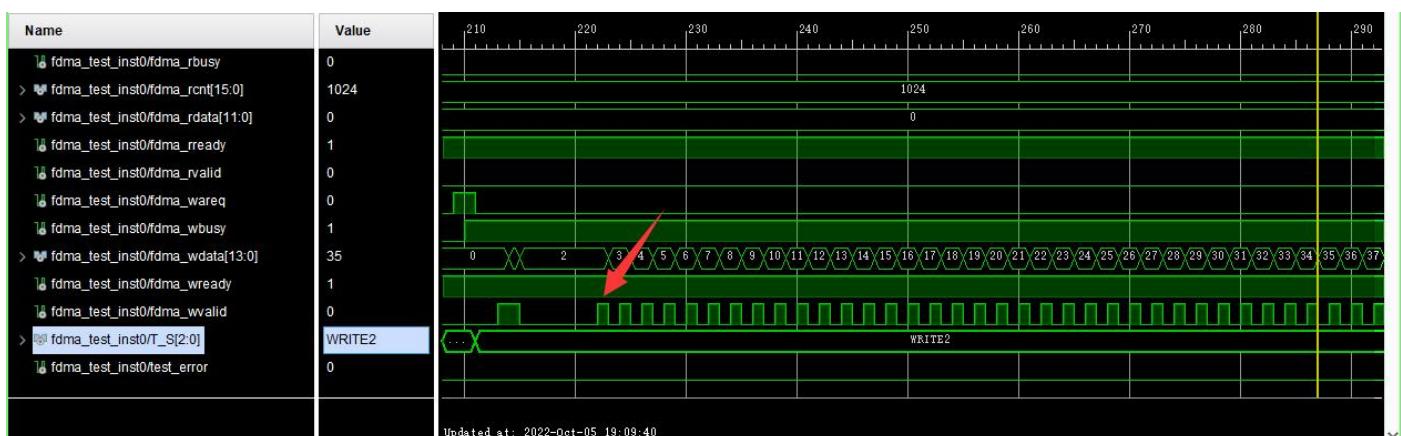
3:如果没有出来，右击芯片型号，选择刷新(芯片型号根据自己的开发板而定，以下芯片为 xc7z020 仅作参考)，



通过在线逻辑分析仪查看结果，可以看到 test_error 信号一直低电平



可以看到 fdma_wvalid fdma_rvalid 并不是连续的，这是由于我们 FDMA 的数据位宽这里设置的 128bits, 大于 ZYNQ IP 的 HP 接口的 AXI4 的 64bits



05uifdma_dbuf 3.0 IP 介绍

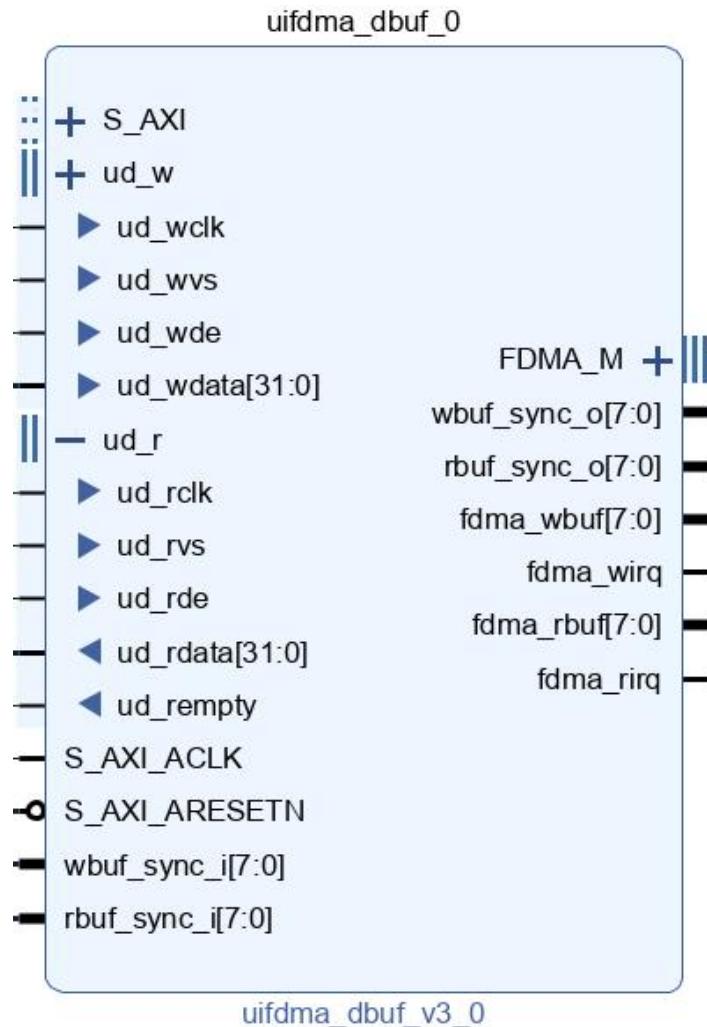
软件版本：vitis2021.1(vivado2021.1)

操作系统：WIN10 64bit

硬件平台：适用 XILINX A7/K7/Z7/ZU/KU 系列 FPGA

登录“米联客”FPGA 社区-www.uisrc.com 视频课程、答疑解惑！

5.1 概述



uifdma_dbuf IP 是米联客研发用于配合 FDMA 完成数据传输控制的 IP 模块。FDMA-DBUF IP 代码采用“对称设计”方法，读写代码对称，好处是代码结构清晰，读写过程一致，代码效率高，更加容易维护。

uifdma_dbuf 的信号接口包含了：

AXI-LITE 接口: 用于 PS 端获取当前的中断帧号;

FDMA_M 接口: 和 FDMA 相连接的数据接口

ud_wx 接口: uifdma_dbuf 写数据通路接口

ud_rx 接口: uifdma_dbuf 读数据通路接口

以下只介绍 uifdma_dbuf 的 ud 写数据通道和 ud 读数据通道

5.2 uifdma_dbuf 的 ud 信号定义

写数据接口信号定义

信号名称	方向	位宽	功能描述
ud_wclk	input	1	写数据通路时钟
ud_wvs	input	1	写数据帧同步信号，当使能视频功能，每个 ud_wvs 的上升沿进行帧同步，否则该值设置为 1 代表数据开始

			传输
ud_wde	input	1	写数据数据有效, 高电平有效
ud_wdata	input	32~128	写数据
ud_wfull	output	1	写数据 FIFO 满, 当 FIFO 满, 继续写入会导致数据溢出
wbuf_sync_o	output	7	写数据帧同步输出, 代表了正在操作的缓存号
wbuf_sync_i	input	7	写数据帧同步输入, 帧同步关系到内部缓存地址切换
fdma_wbuf	output	7	当 fdma_wirq 中断产生后, 代表了当前已经写到 DDR 完成的帧缓存号
fdma_wirq	output	1	写数据完成中断

读数据接口信号定义

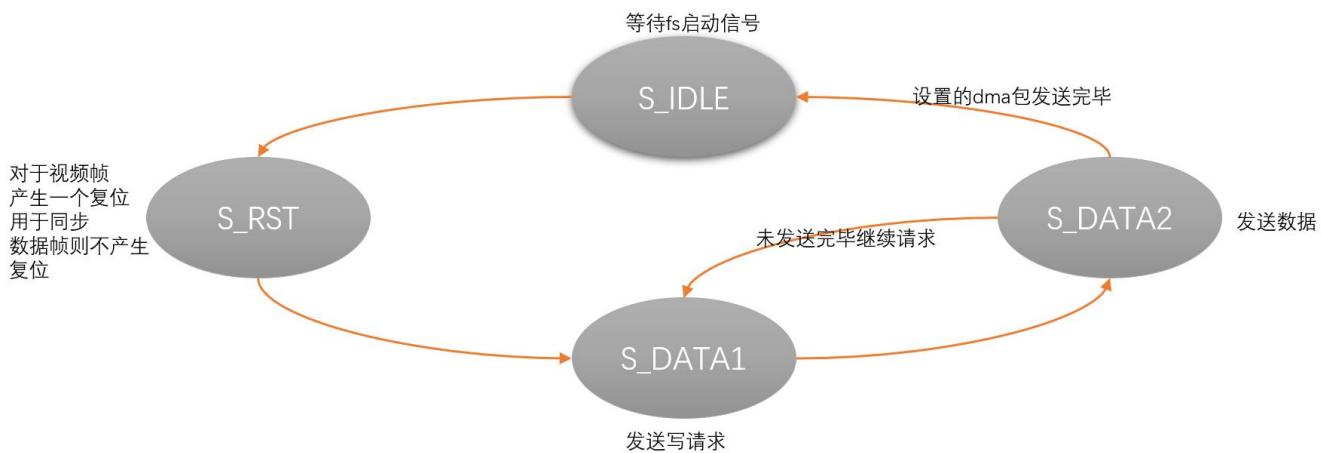
信号名称	方向	位宽	功能描述
ud_rclk	input	1	读数据通路时钟
ud_rvs	input	1	读数据帧同步信号, 当使能视频功能, 每个 ud_rvs 的上升沿进行帧同步, 否则该值设置为 1 代表数据开始传输
ud_rde	input	1	读数据数据有效, 高电平有效
ud_rdata	input	32~128	读数据
ud_rempty	output	1	读数据 FIFO 空, 当 FIFO 非空, 可以读出数据, 当 fdma 工作在非视频模式下, 可以用该信号去使能 ud_rde
rbuf_sync_o	output	7	读数据帧同步输出, 代表了正在操作的缓存号
rbuf_sync_i	input	7	读数据帧同步输入, 帧同步关系到内部缓存地址切换
fdma_rbuf	output	7	当 fdma_rirq 中断产生后, 代表了当前已经读到 DDR 完成的帧缓存号
fdma_rirq	output		读数据完成中断

5.3 FDMA-DBUF IP 代码分析

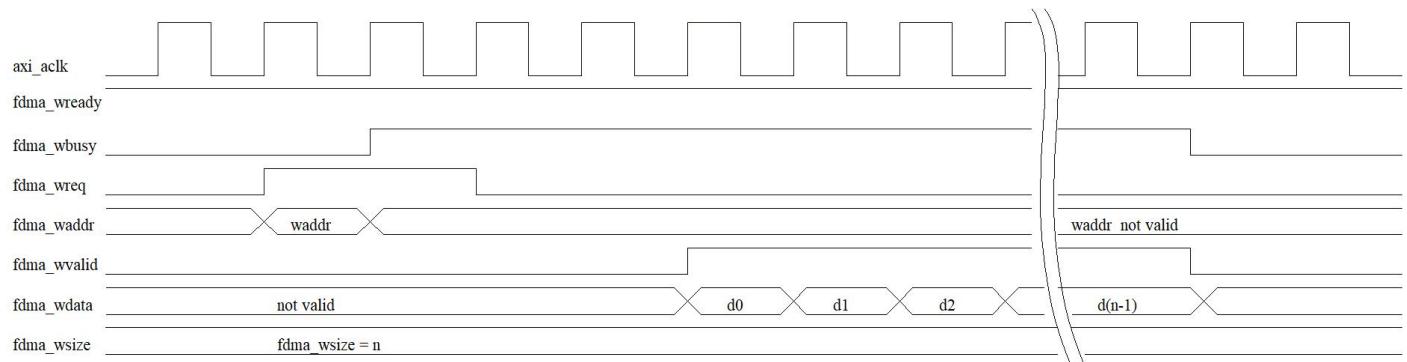
FDMA-DBUF IP 代码采用“对称设计”方法, 读写代码对称, 好处是代码结构清晰, 读写过程一致, 代码效率高, 更加容易维护。

1:FDMA-DBUF 写状态机

为了配合 AXI-FDMA IP 发送数据到 PS, 我们写了一个 uifdmdbuf ip, 通过这个 IP 把用户编写的数据时序, 转为 AXI-FMDA 接口数据流。该 IP 支持视频格式的帧同步, 每一帧都进行同步, 也支持没有帧同步的数据流方式传输。



2:FDMA 的写时序波形图

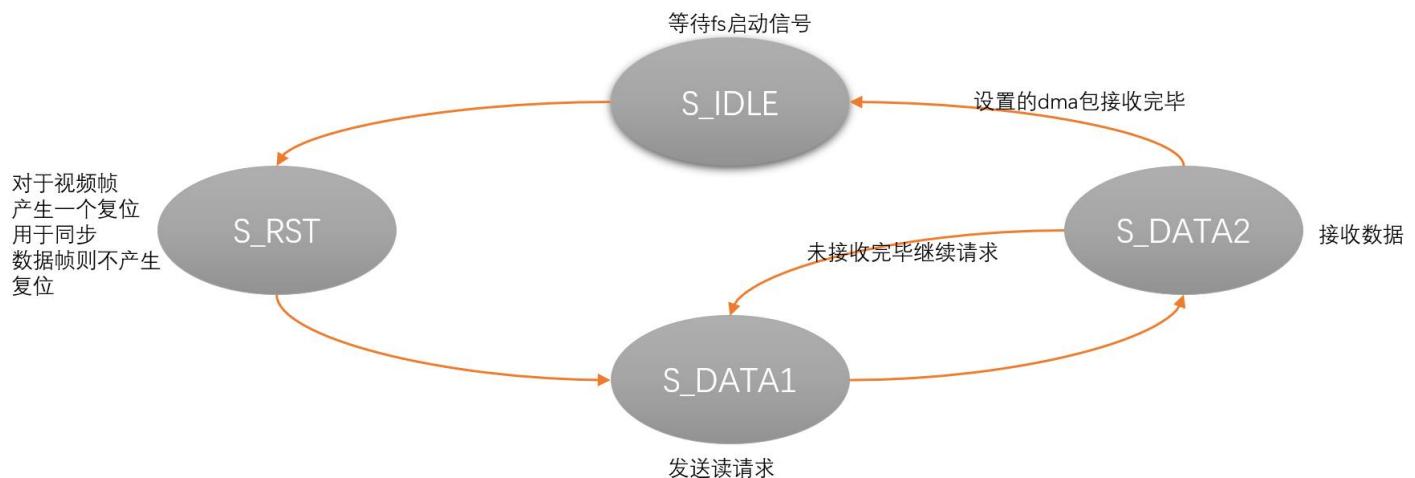


fdma_wready 设置为 1, 当 fdma_wbusy=0 的时候代表 FDMA 的总线非忙, 可以进行一次新的 FDMA 传输, 这个时候可以设置 fdma_wreq=1, 同时设置 fdma burst 的起始地址和 fdma_wszie 本次需要传输的数据大小(以 bytes 为单位)。当 fdma_wvalid=1 的时候需要给出有效的数据, 写入 AXI 总线。当最后一个数写完后, fdma_wvalid 和 fdma_wbusy 变为 0。

AXI4 总线最大的 burst length 是 256, 而经过封装后, 用户接口的 fdma_size 可以任意大小的, fdma ip 内部代码控制每次 AXI4 总线的 Burst 长度, 这样极大简化了 AXI4 总线协议的使用。

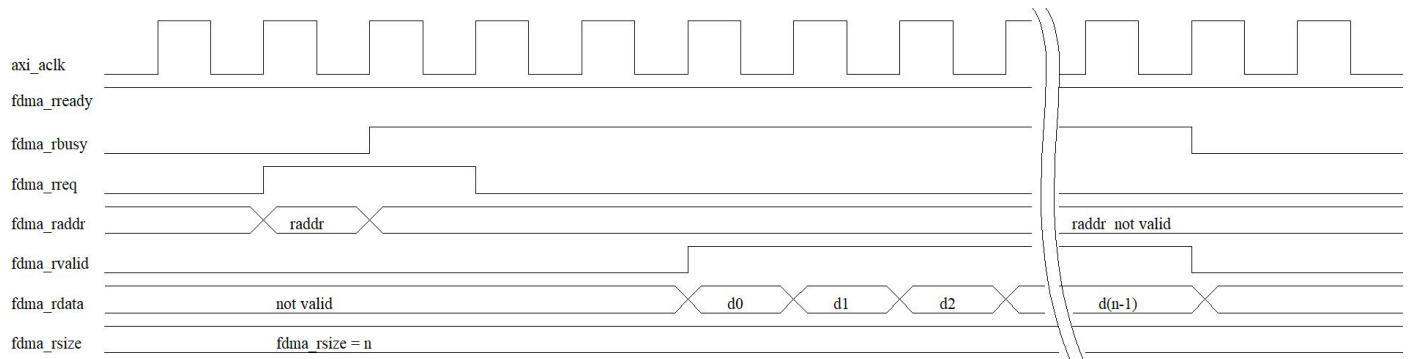
3:FDMA-DBUF 写状态机

读数据的过程和写数据的过程是对称的, 状态机如下:



为了配合 AXI-FDMA IP 发送数据到 PS, 我们写了一个 uifdmdbuf ip, 通过这个 IP 把用户编写的数据时序, 转为 AXI-FDMA 接口数据流。该 IP 支持视频格式的帧同步, 每一帧都进行同步, 也支持没有帧同步的数据流方式传输。

4:FDMA 的读时序波形图



fdma_rready 设置为 1, 当 fdma_rbusy=0 的时候代表 FDMA 的总线非忙, 可以进行一次新的 FDMA 传输, 这个时候可以设置 fdma_rreq=1, 同时设置 fdma burst 的起始地址和 fdma_rsize 本次需要传输的数据大小(以 bytes 为单位)。当 fdma_rvalid=1 的时候需要给出有效的数据, 写入 AXI 总线。当最后一个数写完后, fdma_rvalid 和 fdma_rbusy 变

为 0。

同样对于 AXI4 总线的读操作，AXI4 总线最大的 burst length 是 256，而经过封装后，用户接口的 fdma_size 可以任意大小的，fdma ip 内部代码控制每次 AXI4 总线的 Burst 长度，这样极大简化了 AXI4 总线协议的使用。

5.4 源码分析

米联客自定义 IP 路径在配套的 soc_prj/uisrc/03_ip 路径下，如下图所示：

名称	修改日期	类型	大小
project_1	2021/10/27 9:40	文件夹	
uifdma_dbuf_v1_0_project	2021/10/27 9:40	文件夹	
xgui	2021/10/27 9:40	文件夹	
component.xml	2021/9/19 14:19	XML 文档	75 KB
FDMA.xml	2021/6/16 13:50	XML 文档	1 KB
FDMA_rtl.xml	2021/6/16 13:50	XML 文档	8 KB
fs_cap.v	2021/8/29 18:43	Verilog File	2 KB
uidbuf.v	2021/9/23 15:45	Verilog File	18 KB
uidbufirq.v	2021/8/29 19:54	Verilog File	10 KB
uifdma_dbuf.v	2021/9/3 10:25	Verilog File	8 KB

代码的层次结构如下：

```



```

源码部分一共有包括 4 个文件：

fs_cap.v 该文件用于帧同步信号的抓取，采样边沿抓取方式。

uidbufirq.v 文件用来保存一帧数据发送完毕后产生的中断，ps 部分可以通过 axi-lite 接口读取中断值知道哪一个地址完成数据传输。

uidbuf.v 文件是完成用户数据到 FDMA 接口数据转换的关键代码，同时该代码完成了中断控制，帧缓存控制。

uifdma_dbuf.v 该文件是该 IP 的顶层文件

1:uidbuf.v

```

/*
*****MILIANKE*****
*Company : MiLianKe Electronic Technology Co., Ltd.
*WebSite:https://www.milianke.com
*TechWeb:https://www.uisrc.com
*tmall-shop:https://milianke.tmall.com
*jd-shop:https://milianke.jd.com
*taobao-shop1: https://milianke.taobao.com
*Create Date: 2021/10/15
*File Name: uidbuf.v
*Description:
*Declaration:
*The reference demo provided by Milianke is only used for learning.
*We cannot ensure that the demo itself is free of bugs, so users
*should be responsible for the technical problems and consequences
*caused by the use of their own products.
*Copyright: Copyright (c) MiLianKe
>All rights reserved.
*Revision: 1.0
*Signal description
*1) _i input
*2) _o output
*3) _n activ low
*4) _dg debug signal

```

```

*5) _r delay or register
*6) _s state machine
*****`timescale 1ns / 1ps

module uidbuf#(
parameter integer VIDEO_ENABLE = 1, //使能视频帧支持功能
parameter integer ENABLE_WRITE = 1, //使能写通道
parameter integer ENABLE_READ = 1, //使能读通道

parameter integer AXI_DATA_WIDTH = 128, //AXI 总线数据位宽
parameter integer AXI_ADDR_WIDTH = 32, //AXI 总线地址位宽

parameter integer W_BUFDEPTH = 2048, //写通道 AXI 设置 FIFO 缓存大小
parameter integer W_DATAWIDTH = 32, //写通道 AXI 设置数据位宽大小
parameter [AXI_ADDR_WIDTH -1'b1: 0] W_BASEADDR = 0, //写通道设置内存起始地址
parameter integer W_DSIZEBITS = 24, //写通道设置缓存数据的增量地址大小, 用于 FDMA DBUF 计算帧缓存起始地址
parameter integer W_XSIZE = 1920, //写通道设置 X 方向的数据大小, 代表了每次 FDMA 传输的数据长度
parameter integer W_XSTRIDE = 1920, //写通道设置 X 方向的 Stride 值, 主要用于图形缓存应用
parameter integer W_YSIZE = 1080, //写通道设置 Y 方向值, 代表了进行了多少次 XSIZE 传输
parameter integer W_XDIV = 2, //写通道对 X 方向数据拆分为 XDIV 次传输, 减少 FIFO 的使用
parameter integer W_BUFSIZE = 3, //写通道设置帧缓存大小, 目前最大支持 128 帧, 可以修改参数支持更缓存数

parameter integer R_BUFDEPTH = 2048, //读通道 AXI 设置 FIFO 缓存大小
parameter integer R_DATAWIDTH = 32, //读通道 AXI 设置数据位宽大小
parameter [AXI_ADDR_WIDTH -1'b1: 0] R_BASEADDR = 0, //读通道设置内存起始地址
parameter integer R_DSIZEBITS = 24, //读通道设置缓存数据的增量地址大小, 用于 FDMA DBUF 计算帧缓存起始地址
parameter integer R_XSIZE = 1920, //读通道设置 X 方向的数据大小, 代表了每次 FDMA 传输的数据长度
parameter integer R_XSTRIDE = 1920, //读通道设置 X 方向的 Stride 值, 主要用于图形缓存应用
parameter integer R_YSIZE = 1080, //读通道设置 Y 方向值, 代表了进行了多少次 XSIZE 传输
parameter integer R_XDIV = 2, //读通道对 X 方向数据拆分为 XDIV 次传输, 减少 FIFO 的使用
parameter integer R_BUFSIZE = 3 //读通道设置帧缓存大小, 目前最大支持 128 帧, 可以修改参数支持更缓存数
)
(
input wire ui_clk, //和 FDMA AXI 总线时钟一致
input wire ui_rstn, //和 FDMA AXI 复位一致
//sensor input -W_FIFO-----
input wire W_wclk_i, //用户写数据接口时钟
input wire W_FS_i, //用户写数据接口同步信号, 对于非视频帧一般设置为 1
input wire W_wren_i, //用户写数据使能
input wire [W_DATAWIDTH-1'b1 : 0] W_data_i, //用户写数据
output reg [7 :0] W_sync_cnt_o = 0, //写通道 BUF 帧同步输出
input wire [7 :0] W_buf_i, //写通道 BUF 帧同步输入
output wire W_full, //写通道 BUF 帧满标志

//-----fdma signals write-----
output wire [AXI_ADDR_WIDTH-1'b1: 0] fdma_waddr, //FDMA 写通道地址
output wire fdma_wreq, //FDMA 写通道请求
output wire [15 :0] fdma_wsize, //FDMA 写通道一次 FDMA 的传输大小
input wire fdma_wbusy, //FDMA 处于 BUSY 状态, AXI 总线正在写操作
output wire fdma_wdata, //FDMA 写数据
output wire fdma_wvalid, //FDMA 写有效
output wire fdma_wready, //FDMA 写准备好, 用户可以写数据
output wire fdma_wbuf = 0, //FDMA 的写帧缓存号输出
output wire fdma_wirq, //FDMA 一次写完成的数据传输完成后, 产生中断。
//-----fdma signals read-----
input wire R_rclk_i, //用户读数据接口时钟
input wire R_FS_i, //用户读数据接口同步信号, 对于非视频帧一般设置 1
input wire R_rden_i, //用户读数据使能
output wire [R_DATAWIDTH-1'b1 : 0] R_data_o, //用户读数据
output reg [7 :0] R_sync_cnt_o = 0, //读通道 BUF 帧同步输出
input wire [7 :0] R_buf_i, //写通道 BUF 帧同步输入
output wire R_empty, //读通道 BUF 帧空标志

```

```

output wire      [AXI_ADDR_WIDTH-1'b1: 0]      fdma_raddr, // FDMA 读通道地址
output wire      [15: 0]                      fdma_rreq, // FDMA 读通道请求
output wire      [15: 0]                      fdma_rsize, // FDMA 读通道一次 FDMA 的传输大小
input  wire      [AXI_DATA_WIDTH-1'b1:0]       fdma_rbusy, // FDMA 处于 BUSY 状态, AXI 总线正在读操作
input  wire      [AXI_DATA_WIDTH-1'b1:0]       fdma_rdata, // FDMA 读数据
input  wire      [15: 0]                      fdma_rvalid, // FDMA 读有效
output wire      [7 :0]                       fdma_rready, // FDMA 读准备好, 用户可以写数据
output reg      [7 :0]                       fdma_rbuf =0, // FDMA 的读帧缓存号输出
output wire      fdma_rirq // FDMA 一次读完成的数据传输完成后, 产生中断
);

// 计算 Log2
function integer clog2;
    input integer value;
    begin
        value = value-1;
        for (clog2=0; value>0; clog2=clog2+1)
            value = value>>1;
    end
endfunction

//FDMA 读写状态机的状态值, 一般 4 个状态值即可
localparam S_IDLE  = 2'd0;
localparam S_RST   = 2'd1;
localparam S_DATA1 = 2'd2;
localparam S_DATA2 = 2'd3;

// 通过设置通道使能, 可以优化代码的利用率
generate if(ENABLE_WRITE == 1)begin : FDMA_WRITE_ENABLE

localparam WFIFO_DEPTH = W_BUFDEPTH; //写通道 FIFO 深度
localparam W_WR_DATA_COUNT_WIDTH = clog2(WFIFO_DEPTH)+1; //计算 FIFO 的写通道位宽
localparam W_RD_DATA_COUNT_WIDTH = clog2(WFIFO_DEPTH*W_DATAWIDTH/AXI_DATA_WIDTH)+1;//clog2(WFIFO_DEPTH/(AXI_DATA_WIDTH/W_DATAWIDTH))+1;

localparam WYBUF_SIZE          = (W_BUFSIZE - 1'b1); //写通道需要完成多少次 XSIZE 操作
localparam WY_BURST_TIMES     = (W_YSIZE*W_XDIV); //写通道需要完成的 FDMA burst 操作次数, XDIV 用于把 XSIZE 分解多次传输
localparam FDMA_WX_BURST      = (W_XSIZE*W_DATAWIDTH/AXI_DATA_WIDTH)/W_XDIV; //FDMA_BURST 一次的大小
localparam WX_BURST_ADDR_INC  = (W_XSIZE*(W_DATAWIDTH/8))/W_XDIV; //FDMA 每次 burst 之后的地址增加
localparam WX_LAST_ADDR_INC   = (W_XSTRIDE-W_XSIZE)*(W_DATAWIDTH/8) + WX_BURST_ADDR_INC; //根据 stride 值计算出来最后一次地址

(*mark_debug = "true") (* KEEP = "TRUE" *) wire W_wren_ri = W_wren_i;

assign                               fdma_wready = 1'b1;
reg                                fdma_wareq_r= 1'b0;
reg                                W_FIFO_Rst=0;
(*mark_debug = "true") (* KEEP = "TRUE" *)wire W_FS;
(*mark_debug = "true") (* KEEP = "TRUE" *)reg [1 :0] W_MS=0;
reg [W_DSIZEBITS-1'b1:0]           W_addr=0;
(*mark_debug = "true") (* KEEP = "TRUE" *)reg [15:0] W_bcnt=0;
(*mark_debug = "true") (* KEEP = "TRUE" *)wire [W_RD_DATA_COUNT_WIDTH-1'b1 :0] W_rcnt;
(*mark_debug = "true") (* KEEP = "TRUE" *)reg W_REQ=0;
(*mark_debug = "true") (* KEEP = "TRUE" *)reg [5 :0] wirq_dly_cnt =0;
reg [3 :0]                           wdiv_cnt =0;
reg [7 :0]                           wrst_cnt =0;
reg [7 :0]                           fdma_wbufn;

(*mark_debug = "true") (* KEEP = "TRUE" *) wire wirq= fdma_wirq;

assign fdma_wsize = FDMA_WX_BURST;
assign fdma_wirq = (wirq_dly_cnt>0);

```

```

assign fdma_waddr = W_BASEADDR + {fmda_wbufn,W_addr};//由于 FPGA 逻辑做乘法比较复杂，因此通过设置高位地址实现缓存设置

reg [1:0] W_MS_r =0;
always @(posedge ui_clk) W_MS_r <= W_MS;

//每次 FDMA DBUF 完成一帧数据传输后，产生中断，这个中断持续 60 个周期的 uiclk,这里的延迟必须足够 ZYNQ IP 核识别到这个中断
always @(posedge ui_clk) begin
    if(ui_rstn == 1'b0)begin
        wirq_dly_cnt <= 6'd0;
        fmda_wbuf <=0;
    end
    else if((W_MS_r == S_DATA2) && (W_MS == S_IDLE))begin
        wirq_dly_cnt <= 60;
        fmda_wbuf <= fmda_wbufn;
    end
    else if(wirq_dly_cnt >0)
        wirq_dly_cnt <= wirq_dly_cnt - 1'b1;
end

//帧同步，对于视频有效
fs_cap #
(
    .VIDEO_ENABLE(VIDEO_ENABLE)
)
fs_cap_W0
(
    .clk_i(ui_clk),
    .rstn_i(ui_rstn),
    .vs_i(W_FS_i),
    .fs_cap_o(W_FS)
);

assign fdma_wareq = fdma_wareq_r;

//写通道状态机，采用 4 个状态值描述
always @(posedge ui_clk) begin
    if(!ui_rstn)begin
        W_MS      <= S_IDLE;
        W_FIFO_Rst <= 0;
        W_addr     <= 0;
        W_sync_cnt_o <= 0;
        W_bcnt      <= 0;
        wrst_cnt    <= 0;
        wdiv_cnt    <= 0;
        fmda_wbufn   <= 0;
        fdma_wareq_r <= 1'd0;
    end
    else begin
        case(W_MS)
            S_IDLE:begin
                W_addr <= 0;
                W_bcnt <= 0;
                wrst_cnt <= 0;
                wdiv_cnt <=0;
                if(W_FS) begin //帧同步，对于非视频数据一般常量为 1
                    W_MS <= S_RST;
                    if(W_sync_cnt_o < WBUF_SIZE) //输出帧同步计数器
                        W_sync_cnt_o <= W_sync_cnt_o + 1'b1;
                    else
                        W_sync_cnt_o <= 0;
                end
            end
            S_RST:begin//帧同步，对于非视频数据直接跳过，对于视频数据，会同步每一帧，并且复位数据 FIFO

```

```

    fdma_wbufn <= W_buf_i;
    wrst_cnt <= wrst_cnt + 1'b1;
    if((VIDEO_ENABLE == 1) && (wrst_cnt < 40))
        W_FIFO_Rst <= 1;
    else if((VIDEO_ENABLE == 1) && (wrst_cnt < 100))
        W_FIFO_Rst <= 0;
    else if(fdma_wirq == 1'b0) begin
        W_MS <= S_DATA1;
    end
end
S_DATA1:begin //发送写 FDMA 请求
    if(fdma_wbusy == 1'b0 && W_REQ )begin
        fdma_wareq_r <= 1'b1;
    end
    else if(fdma_wbusy == 1'b1) begin
        fdma_wareq_r <= 1'b0;
        W_MS <= S_DATA2;
    end
end
S_DATA2:begin //写有效数据
    if(fdma_wbusy == 1'b0)begin
        if(W_bcnt == WY_BURST_TIMES - 1'b1) //判断是否传输完毕
            W_MS <= S_IDLE;
        else begin
            if(wdiv_cnt < W_XDIV - 1'b1)begin//如果对 XSIZE 做了分次传输，一个 XSIZE 也需要 XDIV 次 FDMA 完成传输
                W_addr <= W_addr + WX_BURST_ADDR_INC; //计算地址增量
                wdiv_cnt <= wdiv_cnt + 1'b1;
            end
            else begin
                W_addr <= W_addr + WX_LAST_ADDR_INC; //计算最后一次地址增量，最后一次地址根据 stride 计算
                wdiv_cnt <= 0;
            end
            W_bcnt <= W_bcnt + 1'b1;
            W_MS <= S_DATA1;
        end
    end
end
default: W_MS <= S_IDLE;
endcase
end
end

//写通道的数据 FIFO，采用了原语调用 xpm_fifo_async fifo，当 FIFO 存储的数据阈值达到一定量，一般满足一次 FDMA 的 burst 即可发出请求
wire W_rbusy;
always@(posedge ui_clk)
    W_REQ <= (W_rcnt > FDMA_WX_BURST - 2)&&(~W_rbusy);

xpm_fifo_async # (
    .FIFO_MEMORY_TYPE      ("auto"),           //string; "auto", "block", or "distributed";
    .ECC_MODE              ("no_ecc"),          //string; "no_ecc" or "en_ecc";
    .RELATED_CLOCKS         (0),                //positive integer; 0 or 1
    .FIFO_WRITE_DEPTH       (WFIFO_DEPTH),      //positive integer
    .WRITE_DATA_WIDTH       (W_DATAWIDTH),       //positive integer
    .WR_DATA_COUNT_WIDTH   (W_WR_DATA_COUNT_WIDTH), //positive integer
    .PROG_FULL_THRESH       (20),               //positive integer
    .FULL_RESET_VALUE       (0),                //positive integer; 0 or 1
    .USE_ADV_FEATURES       ("0707"),            //string; "0000" to "1F1F";
    .READ_MODE              ("fwft"),             //string; "std" or "fwft";
    .FIFO_READ_LATENCY      (0),                //positive integer;
    .READ_DATA_WIDTH        (AXI_DATA_WIDTH),     //positive integer
    .RD_DATA_COUNT_WIDTH   (W_RD_DATA_COUNT_WIDTH), //positive integer
    .PROG_EMPTY_THRESH       (10),               //positive integer
    .DOUT_RESET_VALUE       ("0"),               //string

```

```

.CDC_SYNC_STAGES          (2),           //positive integer
.WAKEUP_TIME              (0)            //positive integer; 0 or 2;
) xpm_fifo_w_inst (
    .rst      ((ui_rstn == 1'b0) || (W_FIFO_Rst == 1'b1)),
    .wr_clk   (W_wclk_i),
    .wr_en    (W_wren_i),
    .din      (W_data_i),
    .full     (W_full),
    .overflow  (),
    .prog_full(),
    .wr_data_count(),
    .almost_full(),
    .wr_ack   (),
    .wr_rst_busy(),
    .rd_clk   (ui_clk),
    .rd_en    (fdma_wvalid),
    .dout     (fdma_wdata),
    .empty    (),
    .underflow(),
    .rd_rst_busy(W_rbusy),
    .prog_empty(),
    .rd_data_count(W_rcnt),
    .almost_empty(),
    .data_valid(W_dvalid),
    .sleep    (1'b0),
    .injectsbiterr(1'b0),
    .injectdbiterr(1'b0),
    .sbiterr   (),
    .dbiterr   ()
);

end
else begin : FDMA_WRITE_DISABLE

//-----fdma signals write-----
assign fdma_waddr = 0;
assign fdma_wareq = 0;
assign fdma_wsize = 0;
assign fdma_wdata = 0;
assign fdma_wready = 0;
assign fdma_wirq = 0;
assign W_full = 0;

end
endgenerate

generate if(ENABLE_READ == 1)begin : FDMA_READ// 通过设置通道使能, 可以优化代码的利用率
localparam RYBUF_SIZE        = (R_BUFSIZE - 1'b1); //读通道需要完成多少次 XSIZE 操作
localparam RY_BURST_TIMES    = (R_YSIZE*R_XDIV); //读通道需要完成的 FDMA burst 操作次数, XDIV 用于把 XSIZE 分解多次传输
localparam FDMA_RX_BURST     = (R_XSIZE*R_DATAWIDTH/AXI_DATA_WIDTH)/R_XDIV; //FDMA BURST 一次的大小
localparam RX_BURST_ADDR_INC = (R_XSIZE*(R_DATAWIDTH/8))/R_XDIV; //FDMA 每次 burst 之后的地址增加
localparam RX_LAST_ADDR_INC  = (R_XSTRIDE-R_XSIZE)*(R_DATAWIDTH/8) + RX_BURST_ADDR_INC; //根据 stride 值计算出来最后一次地址

localparam RFIFO_DEPTH = R_BUFDEPTH*R_DATAWIDTH/AXI_DATA_WIDTH;//R_BUFDEPTH/(AXI_DATA_WIDTH/R_DATAWIDTH);
localparam R_WR_DATA_COUNT_WIDTH = clog2(RFIFO_DEPTH)+1; //读通道 FIFO 输入部分深度
localparam R_RD_DATA_COUNT_WIDTH = clog2(R_BUFDEPTH)+1; //写通道 FIFO 输出部分深度

assign                                     fdma_rready = 1'b1;
reg                                         fdma_rareq_r= 1'b0;
reg                                         R_FIFO_Rst=0;
wire                                         R_FS;
reg [1 :0]                                    R_MS=0;
reg [R_DSIZEBITS-1'b1:0]                      R_addr=0;

```

```

reg [15:0] R_bcnt=0;
wire[R_WR_DATA_COUNT_WIDTH-1'b1 :0] R_wcnt;
reg R_REQ=0;
reg [5 :0] rirq_dly_cnt =0;
reg [3 :0] rdiv_cnt =0;
reg [7 :0] rrst_cnt =0;
reg [7 :0] fmda_rbufn;

assign fdma_rsize = FDMA_RX_BURST;
assign fdma_rirq = (rirq_dly_cnt>0);

assign fdma_raddr = R_BASEADDR + {fmda_rbufn,R_addr};//由于 FPGA 逻辑做乘法比较复杂，因此通过设置高位地址实现缓存设置

reg [1:0] R_MS_r =0;
always @(posedge ui_clk) R_MS_r <= R_MS;

//每次 FDMA DBUF 完成一帧数据传输后，产生中断，这个中断持续 60 个周期的 uiclk,这里的延迟必须足够 ZYNQ IP 核识别到这个中断
always @(posedge ui_clk) begin
  if(ui_rstn == 1'b0)begin
    rirq_dly_cnt <= 6'd0;
    fmda_rbuf <=0;
  end
  else if((R_MS_r == S_DATA2) && (R_MS == S_IDLE))begin
    rirq_dly_cnt <= 6'd0;
    fmda_rbuf <= fmda_rbufn;
  end
  else if(rirq_dly_cnt >0)
    rirq_dly_cnt <= rirq_dly_cnt - 1'b1;
end

//帧同步，对于视频有效
fs_cap #
(
  .VIDEO_ENABLE(VIDEO_ENABLE)
)
fs_cap_R0
(
  .clk_i(ui_clk),
  .rstn_i(ui_rstn),
  .vs_i(R_FS_i),
  .fs_cap_o(R_FS)
);

assign fdma_rareq = fdma_rareq_r;

//读通道状态机，采用 4 个状态值描述
always @(posedge ui_clk) begin
  if(!ui_rstn)begin
    R_MS      <= S_IDLE;
    R_FIFO_Rst <= 0;
    R_addr    <= 0;
    R_sync_cnt_o <= 0;
    R_bcnt    <= 0;
    rrst_cnt  <= 0;
    rdiv_cnt   <= 0;
    fmda_rbufn <= 0;
    fdma_rareq_r <= 1'd0;
  end
  else begin
    case(R_MS) //帧同步，对于非视频数据一般常量为 1
      S_IDLE:begin
        R_addr <= 0;
        R_bcnt <= 0;
        rrst_cnt <= 0;
      end
    endcase
  end
end

```

```

rdiv_cnt <= 0;
if(R_FS) begin
    R_MS <= S_RST;
    if(R_sync_cnt_o < RYBUF_SIZE) //输出帧同步计数器, 当需要用读通道做帧同步的时候使用
        R_sync_cnt_o <= R_sync_cnt_o + 1'b1;
    else
        R_sync_cnt_o <= 0;
end
S_RST:begin//帧同步, 对于非视频数据直接跳过,对于视频数据, 会同步每一帧, 并且复位数据 FIFO
    fdma_rbufn <= R_buf_i;
    rrst_cnt <= rrst_cnt + 1'b1;
    if((VIDEO_ENABLE == 1) && (rrst_cnt < 40))
        R_FIFO_Rst <= 1;
    else if((VIDEO_ENABLE == 1) && (rrst_cnt < 100))
        R_FIFO_Rst <= 0;
    else if(fdma_rirq == 1'b0) begin
        R_MS <= S_DATA1;
    end
end
S_DATA1:begin
    if(fdma_rbusy == 1'b0 && R_REQ)begin
        fdma_rreq_r <= 1'b1;
    end
    else if(fdma_rbusy == 1'b1) begin
        fdma_rreq_r <= 1'b0;
        R_MS <= S_DATA2;
    end
end
S_DATA2:begin //写有效数据
    if(fdma_rbusy == 1'b0)begin
        if(R_bcnt == RY_BURST_TIMES - 1'b1) //判断是否传输完毕
            R_MS <= S_IDLE;
        else begin
            if(rdiv_cnt < R_XDIV - 1'b1)begin//如果对 XSIZE 做了分次传输, 一个 XSIZE 也需要 XDIV 次 FDMA 完成传输
                R_addr <= R_addr + RX_BURST_ADDR_INC; //计算地址增量
                rdiv_cnt <= rdiv_cnt + 1'b1;
            end
            else begin
                R_addr <= R_addr + RX_LAST_ADDR_INC; //计算最后一次地址增量, 最后一次地址根据 stride 计算
                rdiv_cnt <= 0;
            end
            R_bcnt <= R_bcnt + 1'b1;
            R_MS <= S_DATA1;
        end
    end
end
default:R_MS <= S_IDLE;
endcase
end
end

//写通道的数据 FIFO, 采用了原语调用 xpm_fifo_async fifo, 当 FIFO 存储的数据阈值达到一定量, 一般满足一次 FDMA 的 burst 即可发出请求
wire R_wbusy;
always@(posedge ui_clk)
    R_REQ <= (R_wcnt < FDMA_RX_BURST - 2)&&(~R_wbusy);

xpm_fifo_async # (
    .FIFO_MEMORY_TYPE      ("auto"),           //string; "auto", "block", or "distributed";
    .ECC_MODE              ("no_ecc"),          //string; "no_ecc" or "en_ecc";
    .RELATED_CLOCKS         (0),                //positive integer; 0 or 1
    .FIFO_WRITE_DEPTH       (RFIFO_DEPTH),      //positive integer
    .WRITE_DATA_WIDTH       (AXI_DATA_WIDTH),    //positive integer

```

```

.WR_DATA_COUNT_WIDTH      (R_WR_DATA_COUNT_WIDTH),           //positive integer
.PROG_FULL_THRESH         (20),                      //positive integer
.FULL_RESET_VALUE          (0),                      //positive integer; 0 or 1
.USE_ADV_FEATURES          ("0707"),                //string; "0000" to "1F1F";
.READ_MODE                 ("fwft"),                //string; "std" or "fwft";
.FIFO_READ_LATENCY         (0),                      //positive integer;
.READ_DATA_WIDTH           (R_DATAWIDTH),           //positive integer
.RD_DATA_COUNT_WIDTH       (R_RD_DATA_COUNT_WIDTH),           //positive integer
.PROG_EMPTY_THRESH          (10),                    //positive integer
.DOUT_RESET_VALUE          ("0"),                    //string
.CDC_SYNC_STAGES           (2),                      //positive integer
.WAKEUP_TIME                (0)                      //positive integer; 0 or 2;

) xpm_fifo_R_inst (
    .rst          ((ui_rstn == 1'b0) || (W_FIFO_Rst == 1'b1)),
    .wr_clk       (ui_clk),
    .wr_en        (fdma_rvalid),
    .din          (fdma_rdata),
    .full         (),
    .overflow     (),
    .prog_full    (),
    .wr_data_count (R_wcnt),
    .almost_full  (),
    .wr_ack       (),
    .wr_rst_busy  (R_wbusy),
    .rd_clk       (R_rclk_i),
    .rd_en        (R_rden_i),
    .dout         (R_data_o),
    .empty        (R_empty),
    .underflow    (),
    .rd_rst_busy  (),
    .prog_empty   (),
    .rd_data_count (R_wcnt),
    .almost_empty (),
    .data_valid   (),
    .sleep        (1'b0),
    .injectsbiterr (1'b0),
    .injectdbiterr (1'b0),
    .sbiterr      (),
    .dbiterr      ()
);

end
else begin : FDMA_READ_DISABLE

assign fdma_raddr = 0;
assign fdma_rareq = 0;
assign fdma_rsize = 0;
assign fdma_rdata = 0;
assign fdma_rready = 0;
assign fdma_rirq = 0;
assign R_empty = 1'b0;

end
endgenerate

endmodule

```

2:fs_cap.v

```

*Company : MiLianKe Electronic Technology Co., Ltd.
*WebSite:https://www.milianke.com
*TechWeb:https://www.uisrc.com
*tmall-shop:https://milianke.tmall.com
*jd-shop:https://milianke.jd.com
*taobao-shop1: https://milianke.taobao.com
*Create Date: 2021/10/15
*Module Name:fs_cap
*File Name:fs_cap.v
*Description:
*The reference demo provided by Milianke is only used for learning.
*We cannot ensure that the demo itself is free of bugs, so users
*should be responsible for the technical problems and consequences
*caused by the use of their own products.
*Copyright: Copyright (c) MiLianKe
>All rights reserved.
*Revision: 1.0
*Signal description
*1) _i input
*2) _o output
*3) _n activ low
*4) _dg debug signal
*5) _r delay or register
*6) _s state machine
*****/*
`timescale 1ns / 1ps

module fs_cap#
parameter integer VIDEO_ENABLE = 1
)
(
input clk_i,
input rstn_i,
input vs_i,
output reg fs_cap_o
);

reg[4:0]CNT_FS = 6'b0;
reg[4:0]CNT_FS_n = 6'b0;
reg FS = 1'b0;

//异步信号定义，告诉工具异步信号不用分析时序
(* ASYNC_REG = "TRUE" *) reg vs_i_r1;
(* ASYNC_REG = "TRUE" *) reg vs_i_r2;
(* ASYNC_REG = "TRUE" *) reg vs_i_r3;
(* ASYNC_REG = "TRUE" *) reg vs_i_r4;

//异步转同步
always@(posedge clk_i) begin
    vs_i_r1 <= vs_i;
    vs_i_r2 <= vs_i_r1;
    vs_i_r3 <= vs_i_r2;
    vs_i_r4 <= vs_i_r3;
end

```

```

    vs_i_r3 <= vs_i_r2;
    vs_i_r4 <= vs_i_r3;
end

//同步 vs 信号, 只有当使能了 VIDEO_ENABLE 才会启用
always@(posedge clk_i) begin
    if(!rstn_i)begin
        fs_cap_o <= 1'd0;
    end
    else if(VIDEO_ENABLE == 1)begin //当 VIDEO_ENABLE=1, vs 的上升沿有效, 用于视频同步
        if({vs_i_r4,vs_i_r3} == 2'b01)begin
            fs_cap_o <= 1'b1;
        end
        else begin
            fs_cap_o <= 1'b0;
        end
    end
    else begin//当 VIDEO_ENABLE=0, 直接寄存一次 vs_i_r4
        fs_cap_o <= vs_i_r4;
    end
end
endmodule

```

3:uidbufirq.v

这个部分的功能只有在 PS 需要获取中断后的帧缓存通道才会用到, PS 通过 axi-lite 接口可以读取到寄存器的值。

关于更多 AXI4 总线相关知识可以阅读 “3-2-02_axi_bus_7035fa.pdf” 这个章节, 这个章节专讲 AXI4 总线, 其中也详细讲解了 FDMA 的代码分析。(**注意: 7035fa 代表了 zynq 系列中 MZ7035FA 这个子型号**)

这里主要看代码的最后, 根据中断信号寄存内存的中断号

```

// 当写中断产生的时候, 寄存当前的帧缓存号到 fdma_wbuf_irq 寄存器
always @(posedge S_AXI_ACLK) fdma_wirq_r <= fdma_wirq;

always @(posedge S_AXI_ACLK)begin
if( S_AXI_ARESETN == 1'b0)
    fdma_wbuf_irq <= 0;
else if(fdma_wirq_r == 1'b0 & fdma_wirq == 1'b1)
    fdma_wbuf_irq <= fdma_wbuf;
end

// 当读中断产生的时候, 寄存当前的帧缓存号到 fdma_wbuf_irq 寄存器
always @(posedge S_AXI_ACLK) fdma_rirq_r <= fdma_rirq;

always @(posedge S_AXI_ACLK)begin
if( S_AXI_ARESETN == 1'b0)
    fdma_rbuf_irq <= 0;
else if(fdma_rirq_r == 1'b0 & fdma_rirq == 1'b1)
    fdma_rbuf_irq <= fdma_rbuf;
end

// User logic ends
endmodule

```

06 uifdma_dbuf+fdma 实现数据流方案

软件版本：vitis2021.1(vivado2021.1)

操作系统：WIN10 64bit

硬件平台：适用 XILINX A7/K7/Z7/ZU/KU 系列 FPGA

登录“米联客”FPGA 社区-www.uisrc.com 视频课程、答疑解惑！

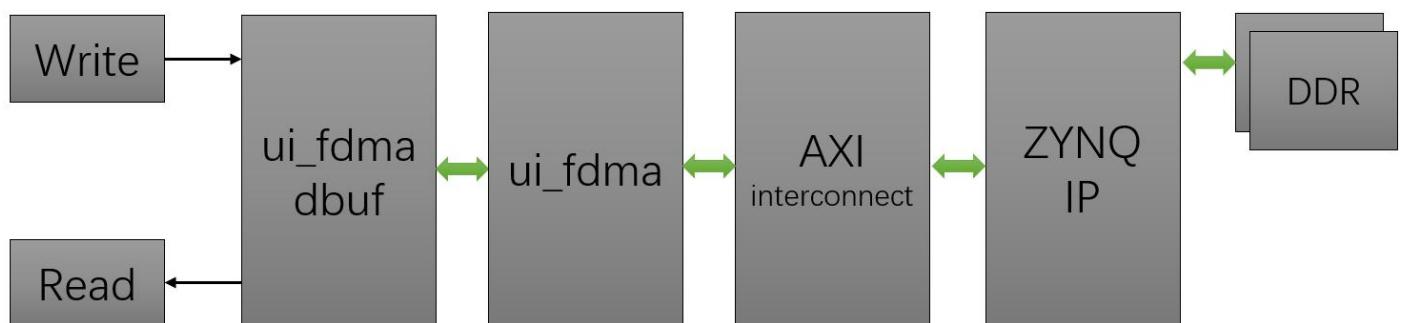
6.1 概述

uifdma_dbuf3.0 开始支持了 full 和 empty 信号， ud_wfull 信号为 0 代表 FIFO 未满，写通道可以写入数据，uifdma_dbuf 的 ud_rempy 信号为 0 代表读通道的 FIFO 中有数据可以读出。

当 uifdma_dbuf 设置非视频模式的时候，ud_vs 信号仅仅用于 uifdma_dbuf 内部的传输状态机的启动和停止。当 ud_vs 为高电平，数据传输开始后，通过 de 控制数据的读或者写。本方案在实际的应用中需要注意，ud_rvs 有效的时候，只要 uifdma_dbuf 中读控制部分的 FIFO 准备好，就会从 DDR 读数据写入到 uifdma_dbuf 的读控制部分的 fifo，这样就存在一个问题，必须确保读出该地址空间的内存数据都是有效的。

所以在本方案中设计的测试环境是写入数据流速度，和读出数据流速度是一致的，并且读操作会晚于写操作一段时间，这样可以确保读的内存地址空间里面的数据都是有效的。

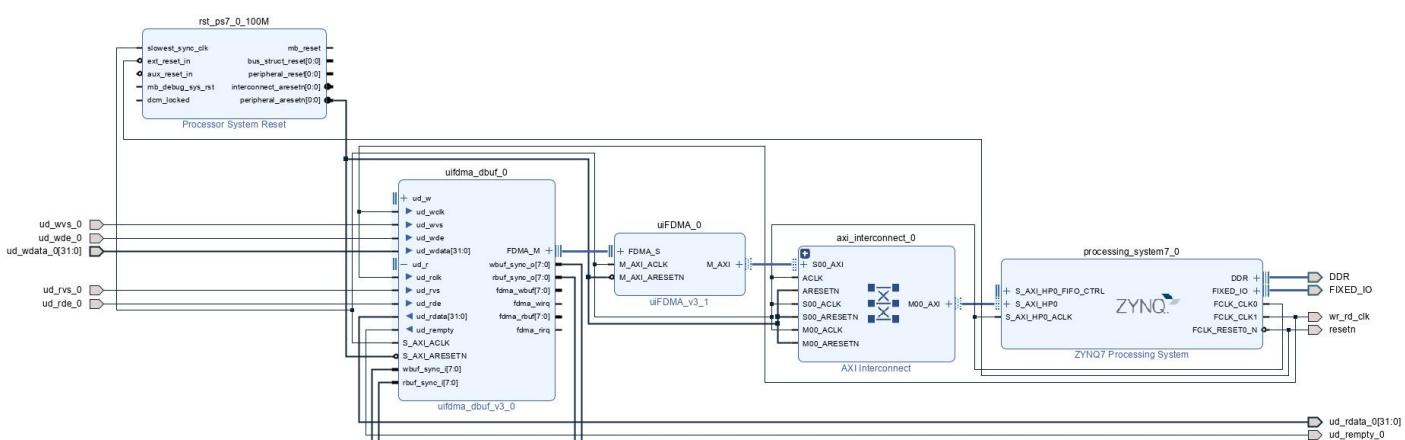
6.2 系统框图



6.3 基于图形化逻辑设计

基于图形化的编程思想主要是把所有的开发都进行标准化的 IP 化，通过可视化的绘制 IP 之间的连线即可完成逻辑部分的设计工作。基于这套思想，我们可以把所有一切可以标准化的代码制作成 IP，这样以后需要使用的时候只需要通过调用图形化的 IP 即可完成编程设计。实现 FPGA 所见即所得的可视化编程。

以下代码中，我们把数据的输入，输出接口引出到顶层模块，由用户自由控制。



通过把 uifdma_dbuf 配置为非视频模式，可以无需 vs 进行帧同步，即可进行数据的传输。当 uifdma_dbuf 配置为非视频模式的时候，vs 信号高电平时，可以进行 fdma 数据搬运动输。如下图所示，写通道和读通道，我们都设置了 3 个缓存地址，每个缓存的大小为 1024*1024*32bit=4MB。ud 写数据端接口数据位宽为 32，ud 读数据接口

数据位宽为 32。

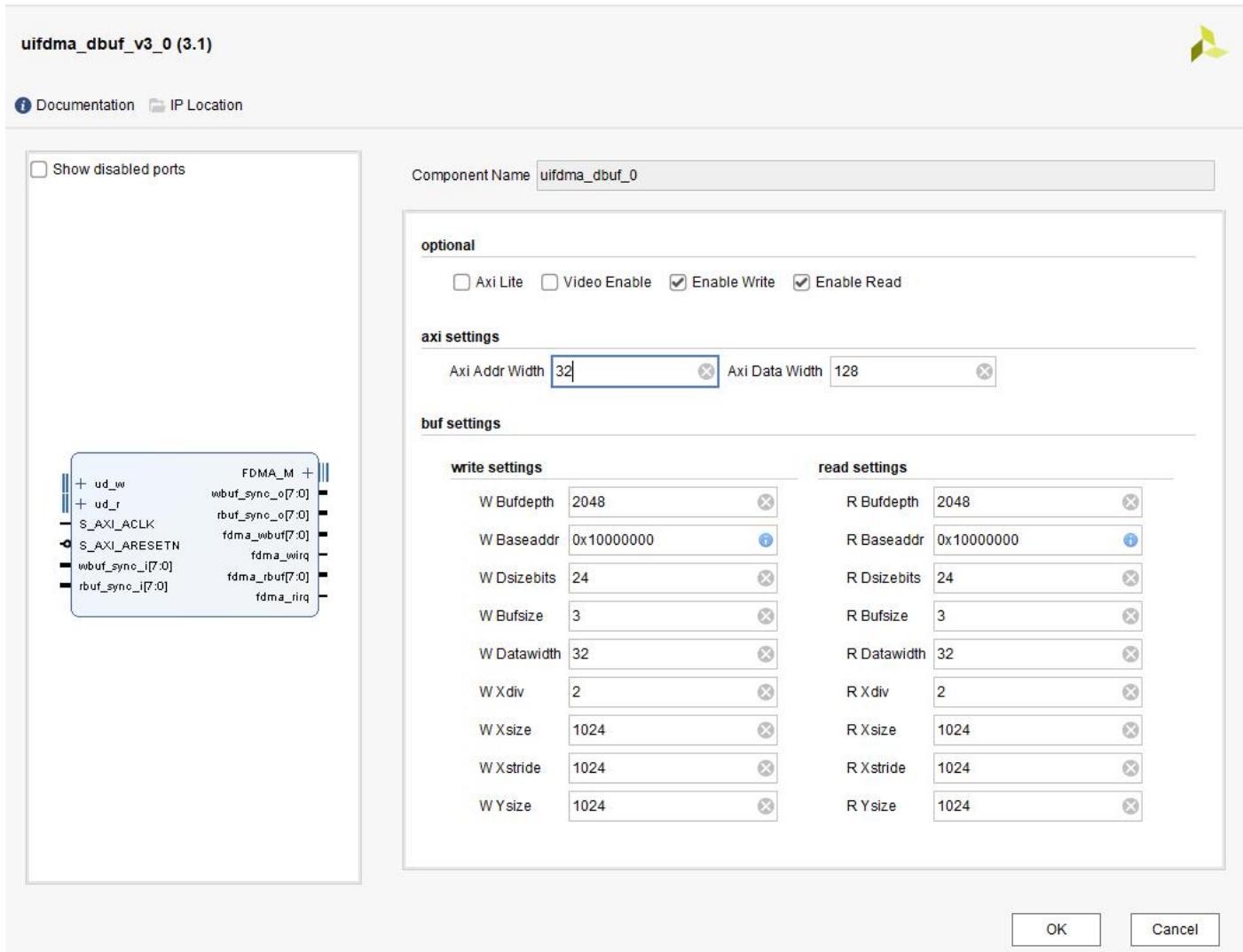
缓存 1 的起始地址为:0x10000000+2^24*0

缓存 2 的起始地址为:0x10000000+2^24*1

缓存 3 的起始地址为:0x10000000+2^24*2

注意:这里的起始地址是从 256MB 处开始的,对于 7010-A 核心板总共才 256M 内存,那么可以起始地址可以改为 0x01000000

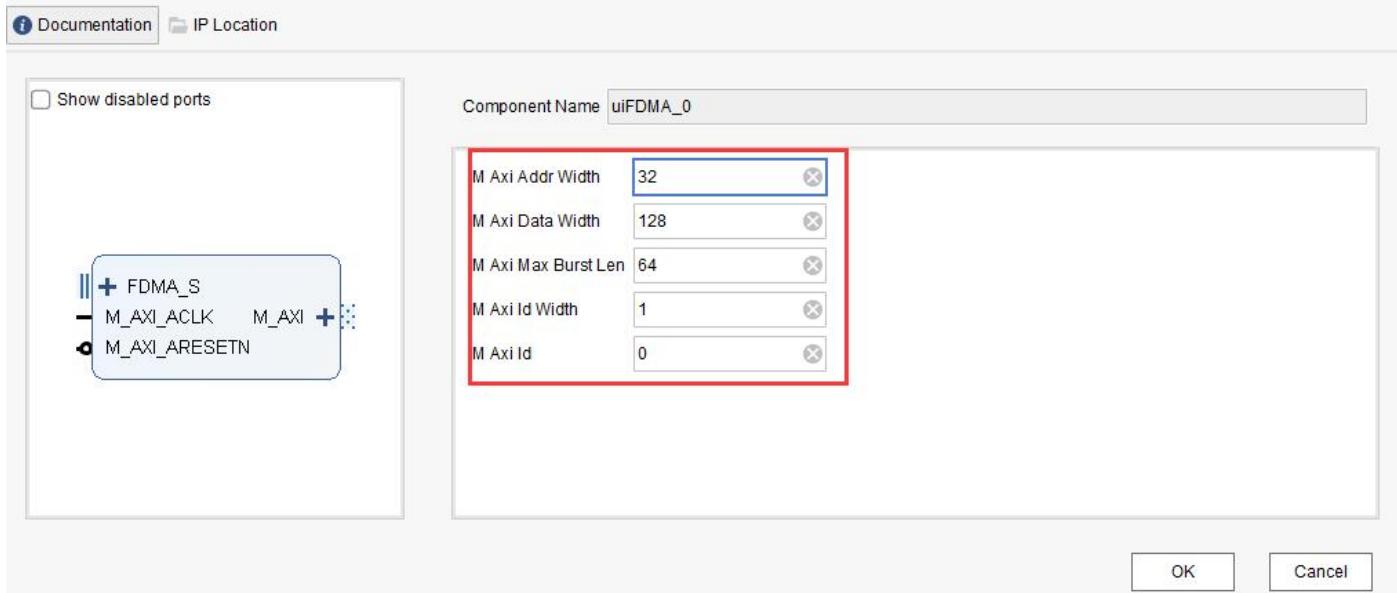
1:uifdma_dbuf 设置



2:uifdma 设置

我们这里演示的 demo 输入输出的数据时钟为 50M,数据速度为 50M*32bit,而 ZYNQ HP 接口出来的时钟远高于 50M 并且这里配置 AXI4 总线位宽是 64bit 所以总线带宽非常充裕,我们可以设置每次 burst 的长度为合理值,无需太大,我们这里设置 64

uiFDMA_v3_1 (3.1)



3 编写测试代码

本方案中，演示从 DDR 中连续写入数据，之后连续读出，并且写入的速度和读出的速度一致。

```

`timescale 1ns / 1ps
//*****************************************************************************
*Company : MiLianKe Electronic Technology Co., Ltd.
*WebSite:https://www.milianke.com
*TechWeb:https://www.uisrc.com
*tmall-shop:https://milianke.tmall.com
*jd-shop:https://milianke.jd.com
*taobao-shop1: https://milianke.taobao.com
*Create Date: 2021/10/15
*Module Name:
*File Name:
*Description:
*config sensor register
/The reference demo provided by Milianke is only used for learning.
*We cannot ensure that the demo itself is free of bugs, so users
*should be responsible for the technical problems and consequences
*caused by the use of their own products.
*Copyright: Copyright (c) MiLianKe
>All rights reserved.
*Revision: 1.0
*Signal description
*1) _i input
*2) _o output
*3) _n activ low
*4) _dg debug signal
*5) _r delay or register
*6) _s state machine

```

```
*****/*
```

```
module fdma_ddr_test(
    inout [14:0]DDR_addr,
    inout [2:0]DDR_ba,
    inout DDR_cas_n,
    inout DDR_ck_n,
    inout DDR_ck_p,
    inout DDR_cke,
    inout DDR_cs_n,
    inout [3:0]DDR_dm,
    inout [31:0]DDR_dq,
    inout [3:0]DDR_dqs_n,
    inout [3:0]DDR_dqs_p,
    inout DDR_odt,
    inout DDR_ras_n,
    inout DDR_reset_n,
    inout DDR_we_n,
    inout FIXED_IO_ddr_vrn,
    inout FIXED_IO_ddr_vrp,
    inout [53:0]FIXED_IO_mio,
    inout FIXED_IO_ps_clk,
    inout FIXED_IO_ps_porb,
    inout FIXED_IO_ps_srstb
);

wire resetn;
wire wr_rd_clk;
wire ud_rempy_0,error;
reg ud_wvs_0, ud_wde_0 ,ud_rvs_0 ;
wire [31:0]ud_wdata_0 , ud_rdata_0;
reg [15:0]wr_cnt , rd_cnt, delay_cnt;

assign ud_rde_0 = !ud_rempy_0;
assign ud_wdata_0 = {16'd0,wr_cnt};

always @(posedge wr_rd_clk)begin
    if(resetn == 1'b0)begin
        delay_cnt <= 0;
        ud_rvs_0 <= 1'b0;
    end
    else if(delay_cnt[13:12] == 2'b11) begin //延迟启动读，通过 delay_cnt 控制写入和读出的开始时机
        delay_cnt <= delay_cnt;
        ud_rvs_0 <= 1'b1;
    end
    else begin
        delay_cnt <= delay_cnt + 1'b1;
    end
end

```

```

    end
end

always @(posedge wr_rd_clk)begin //延迟启动写, 通过 delay_cnt 控制写入和读出的开始时机
    if(delay_cnt[13] == 1'b0)begin
        ud_wvs_0 <= 1'b0;
        ud_wde_0 <= 1'b0;
    end
    else begin
        ud_wvs_0 <= 1'b1;
        ud_wde_0 <= 1'b1;
    end
end

//写入 16bits 计数器值到缓存中
always @(posedge wr_rd_clk)begin
    if(resetn == 1'b0)begin
        wr_cnt <= 0;
    end
    else begin
        if(ud_wde_0)
            wr_cnt <= wr_cnt + 1'b1;
    end
end

//当读信号 ud_rde 有效, 通过计数器计数
always @(posedge wr_rd_clk)begin
    if(resetn == 1'b0)begin
        rd_cnt <= 0;
    end
    else begin
        if(ud_rde_0)
            rd_cnt <= rd_cnt + 1'b1;
        else
            rd_cnt <= rd_cnt;
    end
end

//通过读出的数据和我们读计数器的值是否一致判断数据是否正确
assign error = ud_rde_0&&(rd_cnt != ud_rdata_0[15:0]); //比较是否有错误

//在线逻辑分析仪
ila_0 ila_wdg (
    .clk(wr_rd_clk), // input wire clk
    .probe0({ud_rde_0,rd_cnt[15:0],ud_rdata_0[15:0],ud_wde_0,ud_wdata_0[15:0],ud_rempy_0, error})// input wire
    [17:0] probe0
);

```

system system_i

```

(.DDR_addr(DDR_addr),
 .DDR_ba(DDR_ba),
 .DDR_cas_n(DDR_cas_n),
 .DDR_ck_n(DDR_ck_n),
 .DDR_ck_p(DDR_ck_p),
 .DDR_cke(DDR_cke),
 .DDR_cs_n(DDR_cs_n),
 .DDR_dm(DDR_dm),
 .DDR_dq(DDR_dq),
 .DDR_dqs_n(DDR_dqs_n),
 .DDR_dqs_p(DDR_dqs_p),
 .DDR_odt(DDR_odt),
 .DDR_ras_n(DDR_ras_n),
 .DDR_reset_n(DDR_reset_n),
 .DDR_we_n(DDR_we_n),
 .FIXED_IO_ddr_vrn(FIXED_IO_ddr_vrn),
 .FIXED_IO_ddr_vrp(FIXED_IO_ddr_vrp),
 .FIXED_IO_mio(FIXED_IO_mio),
 .FIXED_IO_ps_clk(FIXED_IO_ps_clk),
 .FIXED_IO_ps_porb(FIXED_IO_ps_porb),
 .FIXED_IO_ps_srstb(FIXED_IO_ps_srstb),

.ud_rdata_0(ud_rdata_0),
.ud_rde_0(ud_rde_0),
.ud_rempy_0(ud_rempy_0),
.ud_rvs_0(ud_rvs_0),

.ud_wdata_0(ud_wdata_0),
.ud_wde_0(ud_wde_0),
.ud_wvs_0(ud_wvs_0),

.wr_rd_clk(wr_rd_clk),
.resetn(resetn)
);

```

endmodule

4:地址空间分配

配置完成后需要注意地址空间分配，FDMA IP 的内存起始地址从 0 开始

Name	Interface	Slave Segment	Master Base Address	Range	Master High Address
Network 0					
/uiFDMA_0					
/uiFDMA_0/M_AXI (32 address bits : 4G)	S_AXI_HP0	HP0_DDR_LOWOCM	0x0000_0000	1G	0x3FFF_FFFF

5:编译并导出平台文件

1:单击 Block 文件→右键→Generate the Output Products→Global→Generate。

2: 单击 Block 文件 → 右键 → Create a HDL wrapper(生成 HDL 顶层文件) → Let vivado manager wrapper and auto-update(自动更新)。

3:生成 Bit 文件。

4:导出到硬件: File→Export Hardware→Include bitstream

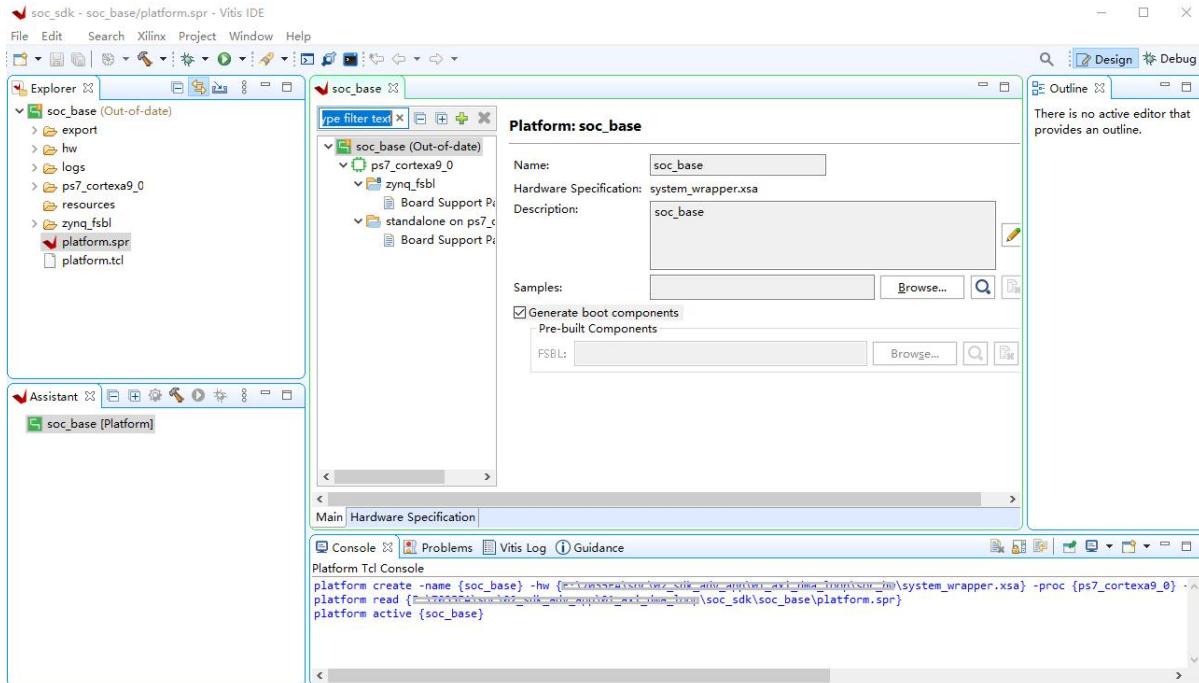
5:导出完成后, 对应工程路径的 soc_hw 路径下有硬件平台文件: system_wrapper.xsa 的文件。根据硬件平台文件 system_wrapper.xsa 来创建需要 Platform 平台。



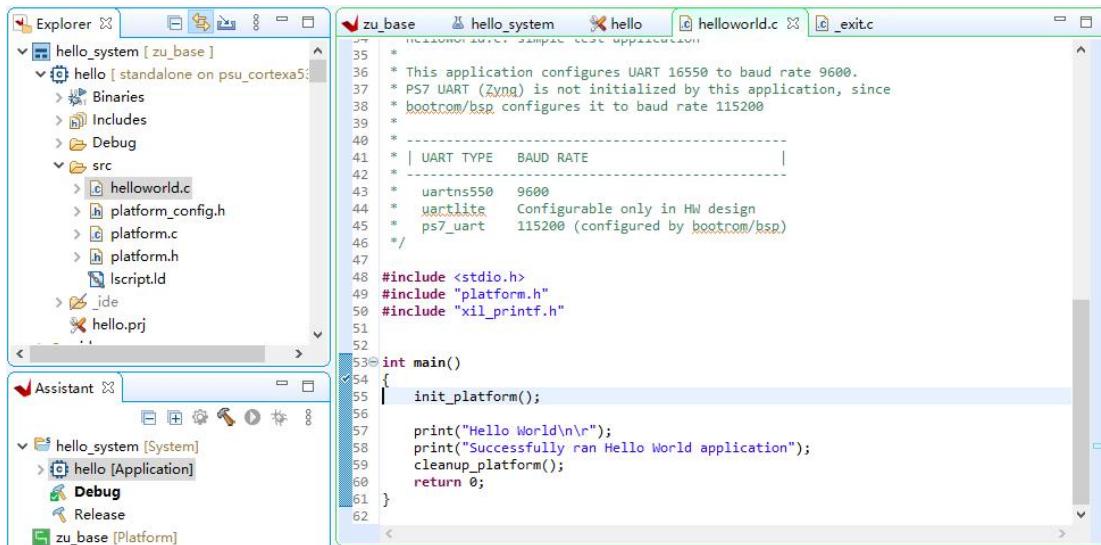
6.4 搭建 Vitis-sdk 工程

创建 soc_base sdk platform 和 APP 工程的过程不再重复, 对于初学读者如果还不清楚如何创建 soc_sdk 工程的, 请学习“3-1-01 米联客 2022 版 ZynqSocSDK 入门篇”中第一个工程“01Vitis Soc 开发入门”这个实验。

1:创建 SDK Platform 工程



2:创建 hello APP 工程



6.5 实验结果

1:由于 PS DDR 需要先运行 ARM 因此，先运行 hello app

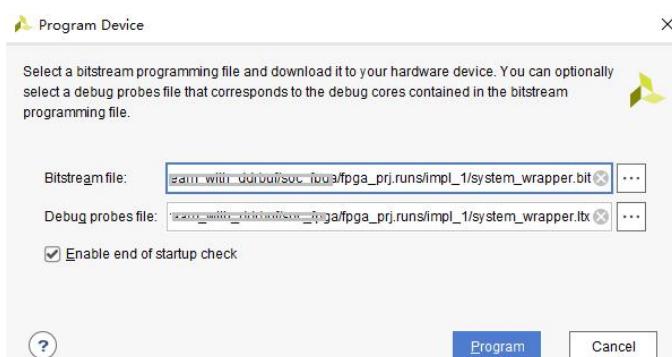
2:打开设备



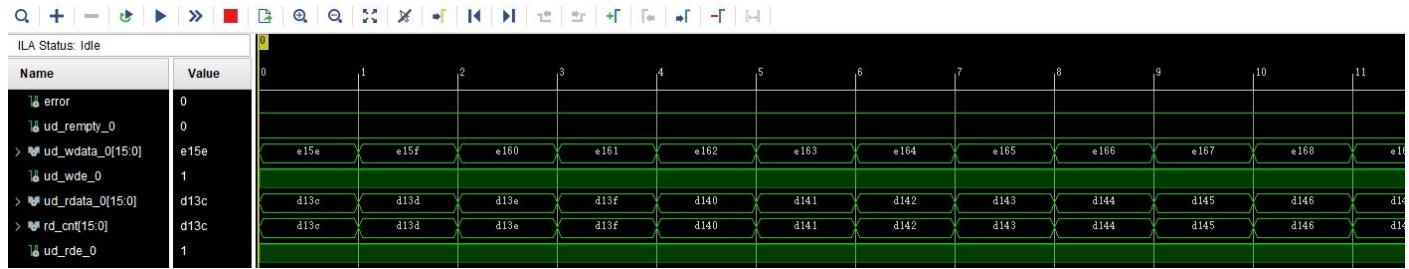
3:如果没有出来，右击芯片型号(芯片型号根据自己的开发板而定)，右击刷新



如果还是没出来逻辑分析仪可以下载一次 FPGA 程序



通过在线逻辑分析仪观察结果



07 基于 fdma 多路视频缓存数据构架方案

软件版本：vitis2021.1(vivado2021.1)

操作系统：WIN10 64bit

硬件平台：适用 XILINX A7/K7/Z7/ZU/KU 系列 FPGA

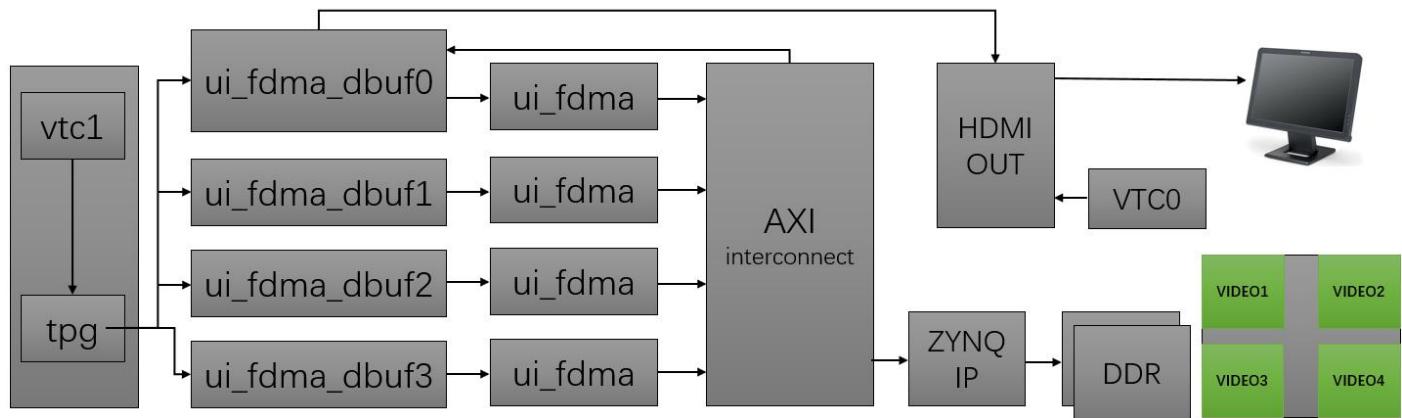
登录“米联客”FPGA 社区-www.uisrc.com 视频课程、答疑解惑！

7.1 概述

基于 AXI 总线可以使用 axi_interconnect 的仲裁机制，同时接入多个基于 AXI 总线的 IP,米联客的 fdma 采用的是 AXI4 总线接口，因此基于 AXI 总线的多数据通路方案实现起来会非常轻松。每一个通路都可以独立工作，总裁都交给 axi_interconnect IP 来完成。

本方案实现了 4 路视频测试数据的输入，以及 1 路视频的输出。

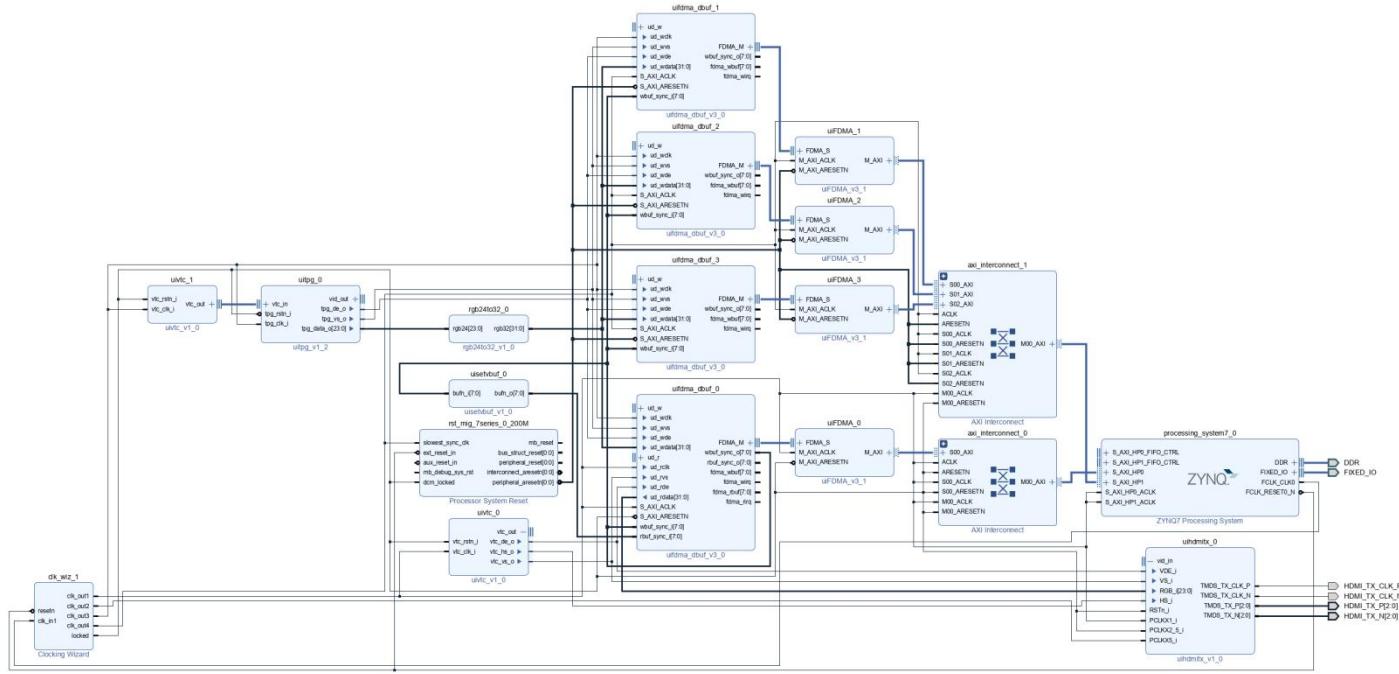
7.2 系统框图



7.3 基于图形化的逻辑设计

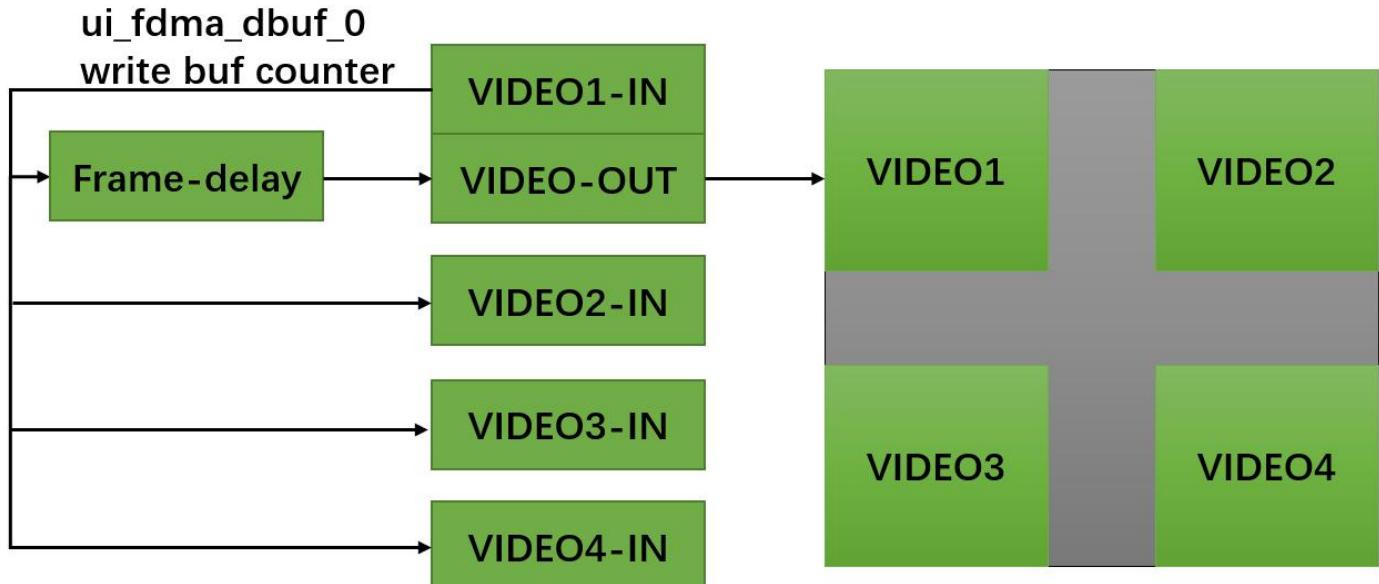
本方案中通过 VTC0 产生 1080P 视频时序，通过 VTC1 产生 480P 时序，通过 TPG 模块产生测试图形。测试视频数据通过 uifdma_dbuf0, uifdma_dbuf1, uifdma_dbuf2, uifdma_dbuf3 IP 的写通道，再经过 FDMA IP 后，经过 AXI interconnect IP 后进入 DDR 缓存。缓存后的数据，通过 uifdma_dbuf0 的读通道输出到 HDMI IP 显示。

注意：关于时钟部分的设置，请打开配套工程里面的 IP 设置查看

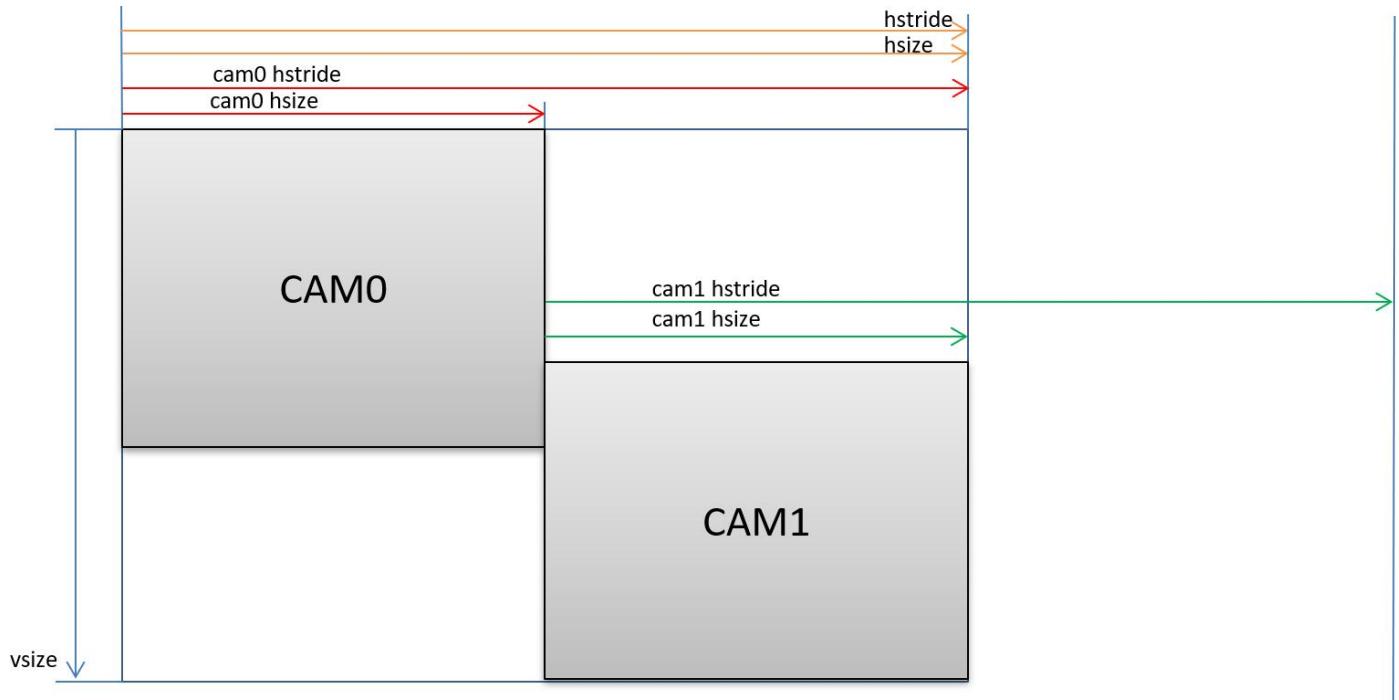


1:帧同步设计

对于多路视频传输的场合，需要正确设置同步。uifdma_dbuf0 的写通道输出帧同步计数器直接接入 uifdma_dbuf0, uifdma_dbuf1, uifdma_dbuf2, uifdma_dbuf3 的写通道同步计数输入。uifdma_dbuf0 的读通道，延时 1 帧于 uifdma_dbuf0 的写通道帧计数器。



2:多路视频的同屏显示原理



以把 2 个摄像头 CAM0 和 CAM1 输出到同一个显示器上为列，为了把 2 个图像显示到 1 个显示器，首先得搞清楚以下关系：

hszie: 每 1 行图像实际在内存中占用的有效空间，以 32bit 表示一个像素的时候占用内存大小为 $hszie * 4$

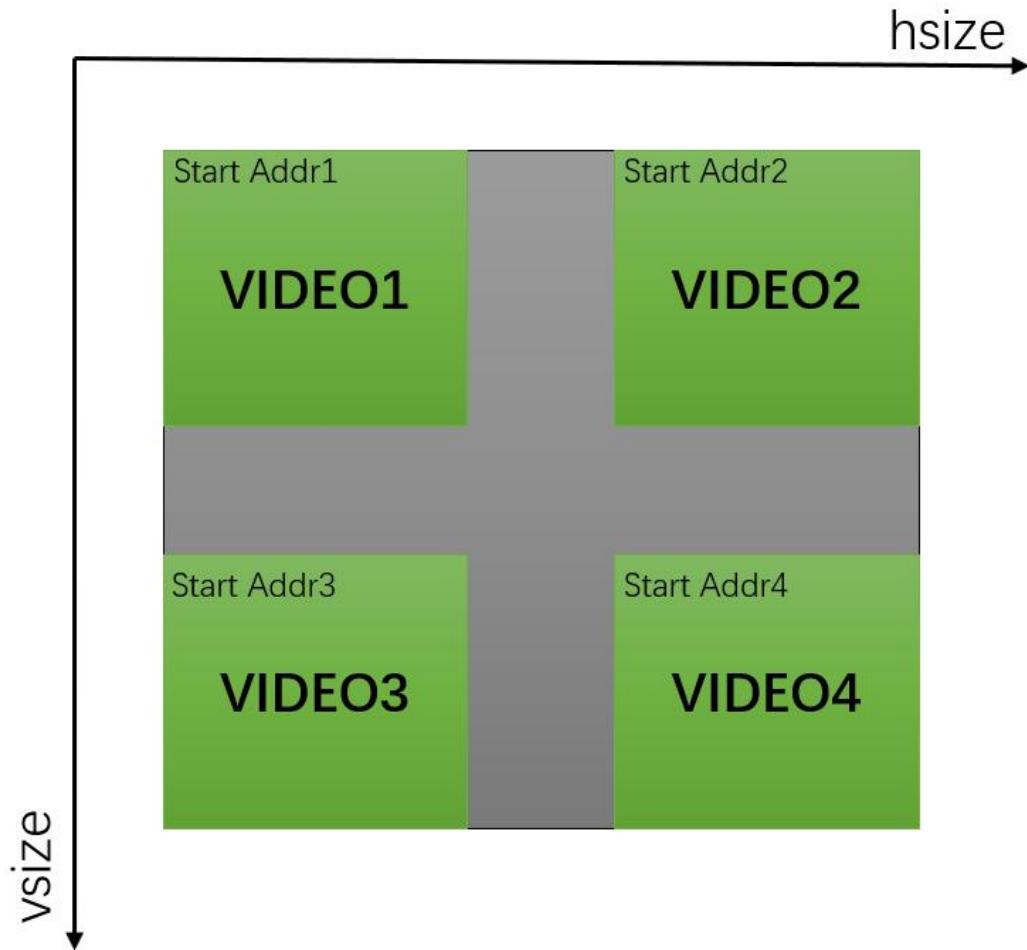
hstride: 用于设置每行图像第一个像素的地址,以 32bit 表示一个像素的时候 $v_cnt * hstride * 4$

vszie: 有效的行

因此很容易得出 cam0 的每行第一个像素的地址也是 $v_cnt * hstride * 4$

同理如果我们需要把 cam1 在 hszie 和 vszie 空间的任何位置显示，我们只要关心 cam1 每一行图像第一个像素的地址，可以用以下公式 $v_cnt * hstride * 4 + offset$

这里 4 个通道的 640*480 分辨率视频，在同一个 1920*1080 的视频输出显示。video1 在左上角，video2 在右上角，video3 在左下角，video4 在右下角。



uifdma_dbuf 支持 stride 参数设置, stride 参数可以设置输入数据 X(hsize)方向每一行数据的第一个像素到下一个起始像素的间隔地址, 利用 stride 参数可以非常方便地摆放输入视频到内存中的排列方式。

video1 的 start addr1=0x10000000(第一个图像的起始地址对于 PL DDR 可以从 0 地址开始, 低于 PS DDR 建议偏移 20MB)

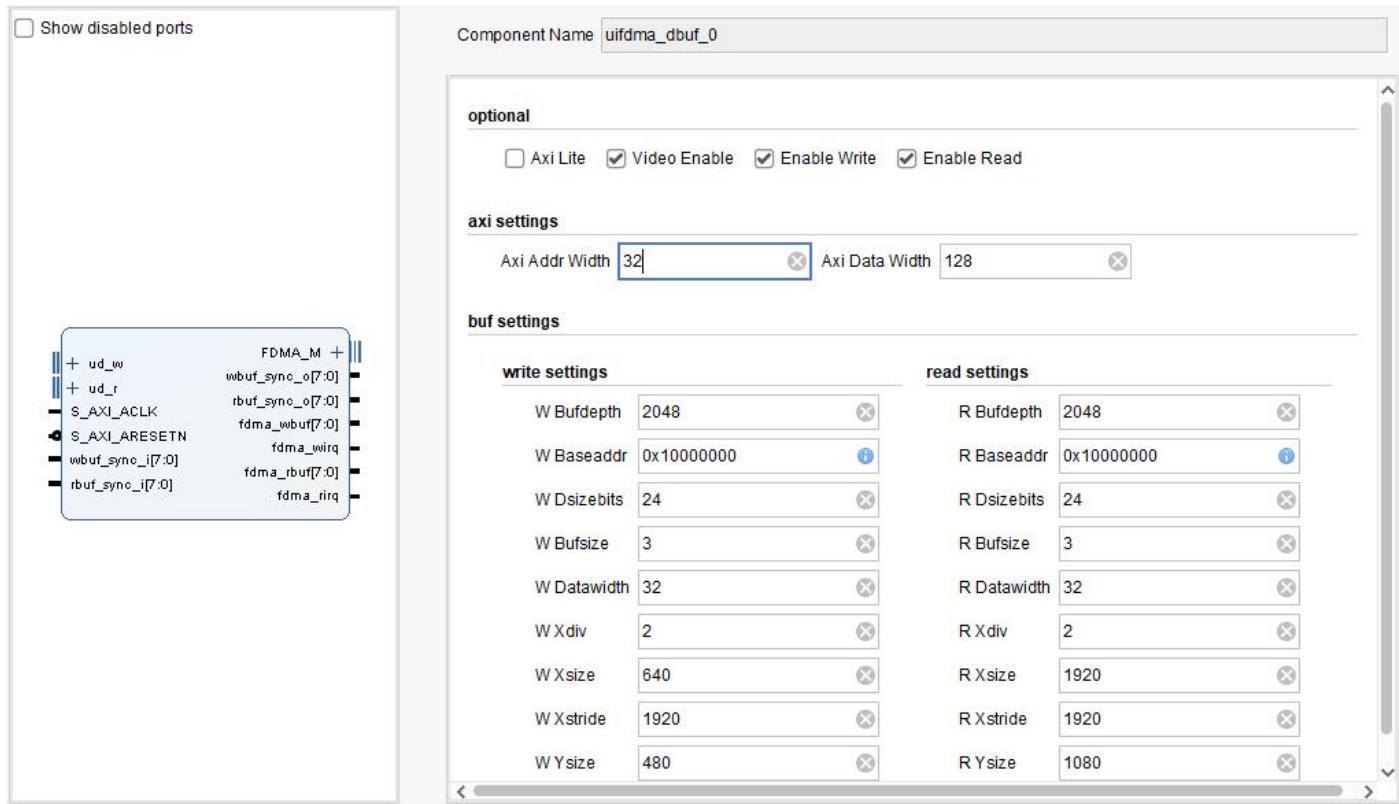
video2 的 start addr2=0x10000000+(1920-640)*4

video3 的 start addr3=0x10000000+(1080-480)*1920*4

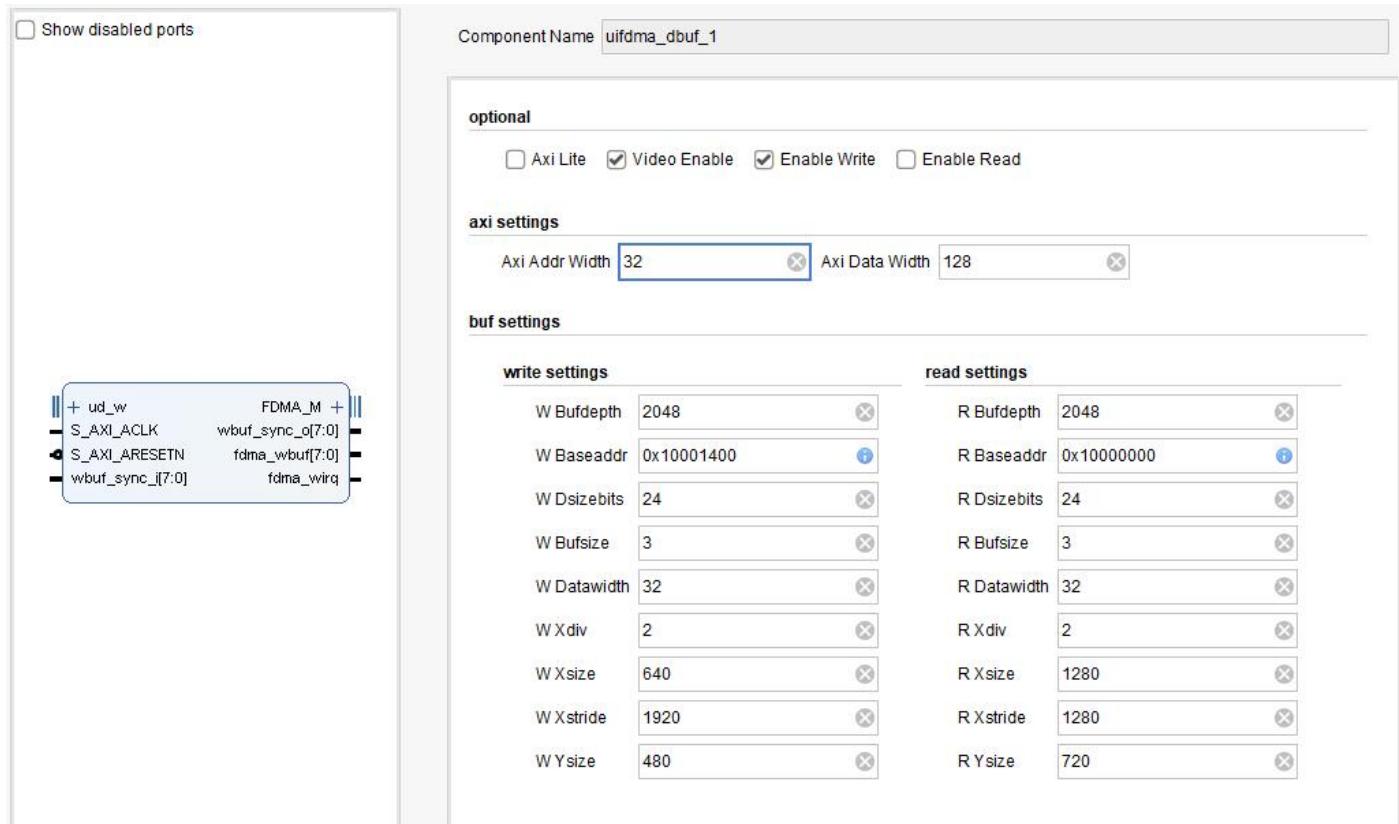
video4 的 start addr4=0x10000000+(1080-480)*1920*4+(1920-640)*4

注意:这里的起始地址是从 256MB 处开始的, 对于 7010-A 核心板总共才 256M 内存, 那么可以起始地址可以改为 0x01000000

3:uifdma_dbuf0 的参数设置



4:uifdma_dbuf1 的参数设置



5:uifdma_dbuf2 的参数设置

Show disabled ports

Component Name: uifdma_dbuf_2

optional

Axi Lite Video Enable Enable Write Enable Read

axi settings

Axi Addr Width: 32 Axi Data Width: 128

buf settings

write settings		read settings	
W Bufdepth	2048 <input type="button" value="X"/>	R Bufdepth	2048 <input type="button" value="X"/>
W Baseaddr	0x10465000 <input type="button" value="i"/>	R Baseaddr	0x10000000 <input type="button" value="i"/>
W Dsizebits	24 <input type="button" value="X"/>	R Dsizebits	24 <input type="button" value="X"/>
W Bufsize	3 <input type="button" value="X"/>	R Bufsize	3 <input type="button" value="X"/>
W Datawidth	32 <input type="button" value="X"/>	R Datawidth	32 <input type="button" value="X"/>
W Xdiv	2 <input type="button" value="X"/>	R Xdiv	2 <input type="button" value="X"/>
W Xsize	640 <input type="button" value="X"/>	R Xsize	1280 <input type="button" value="X"/>
W Xstride	1920 <input type="button" value="X"/>	R Xstride	1280 <input type="button" value="X"/>
W Ysize	480 <input type="button" value="X"/>	R Ysize	720 <input type="button" value="X"/>

6:uifdma_dbuf3 的参数设置

Show disabled ports

Component Name: uifdma_dbuf_3

optional

Axi Lite Video Enable Enable Write Enable Read

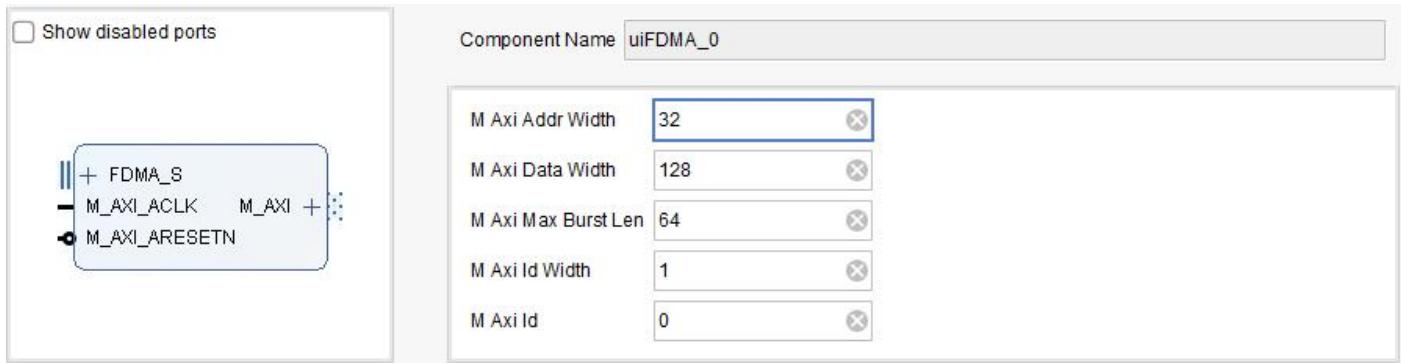
axi settings

Axi Addr Width: 32 Axi Data Width: 128

buf settings

write settings		read settings	
W Bufdepth	2048 <input type="button" value="X"/>	R Bufdepth	2048 <input type="button" value="X"/>
W Baseaddr	0x10466400 <input type="button" value="i"/>	R Baseaddr	0x10000000 <input type="button" value="i"/>
W Dsizebits	24 <input type="button" value="X"/>	R Dsizebits	24 <input type="button" value="X"/>
W Bufsize	3 <input type="button" value="X"/>	R Bufsize	3 <input type="button" value="X"/>
W Datawidth	32 <input type="button" value="X"/>	R Datawidth	32 <input type="button" value="X"/>
W Xdiv	2 <input type="button" value="X"/>	R Xdiv	2 <input type="button" value="X"/>
W Xsize	640 <input type="button" value="X"/>	R Xsize	1280 <input type="button" value="X"/>
W Xstride	1920 <input type="button" value="X"/>	R Xstride	1280 <input type="button" value="X"/>
W Ysize	480 <input type="button" value="X"/>	R Ysize	720 <input type="button" value="X"/>

7:uiFDMA 的参数设置



8:地址空间分配

配置完成后需要注意地址空间分配，FDMA IP 的内存起始地址从 0 开始

Name	Interface	Slave Segment	Master Base Address	Range	Master High Address
Network 0					
uiFDMA_0	/processing_system7_0/S_AXI_HP0	S_AXI_HP0 HP0_DDR_LOWOCM	0x0000_0000	1G	0x3FFF_FFFF
uiFDMA_1	/processing_system7_0/S_AXI_HP1	S_AXI_HP1 HP1_DDR_LOWOCM	0x0000_0000	1G	0x3FFF_FFFF
uiFDMA_2	/processing_system7_0/S_AXI_HP1	S_AXI_HP1 HP1_DDR_LOWOCM	0x0000_0000	1G	0x3FFF_FFFF
uiFDMA_3	/processing_system7_0/S_AXI_HP1	S_AXI_HP1 HP1_DDR_LOWOCM	0x0000_0000	1G	0x3FFF_FFFF

9:编译并导出平台文件

1:单击 Block 文件 → 右键 → Generate the Output Products → Global → Generate。

2: 单击 Block 文件 → 右键 → Create a HDL wrapper(生成 HDL 顶层文件) → Let vivado manager wrapper and auto-update(自动更新)。

3:生成 Bit 文件。

4:导出到硬件: File → Export Hardware → Include bitstream

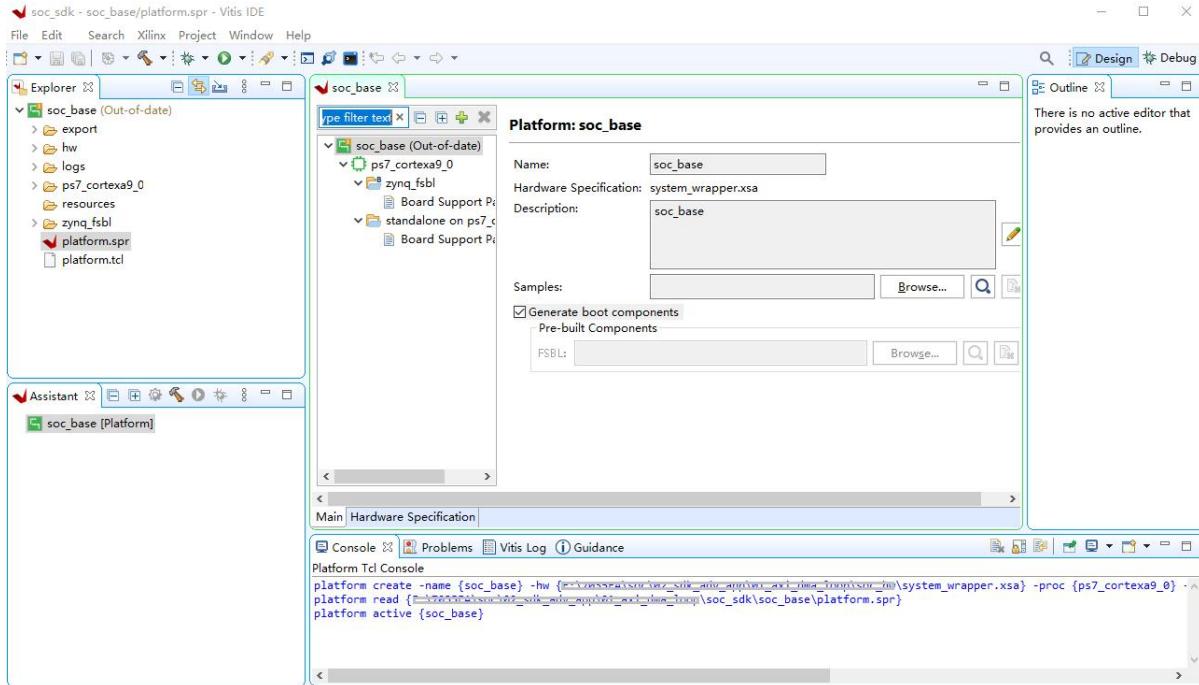
5:导出完成后，对应工程路径的 soc_hw 路径下有硬件平台文件：system_wrapper.xsa 的文件。根据硬件平台文件 system_wrapper.xsa 来创建需要 Platform 平台。

10_data_path > milianke7x_020 > 04_ddr_test > soc_hw			
名称	修改日期	类型	大小
system_wrapper.xsa	2022/10/5 19:04	XSA 文件	430 KB

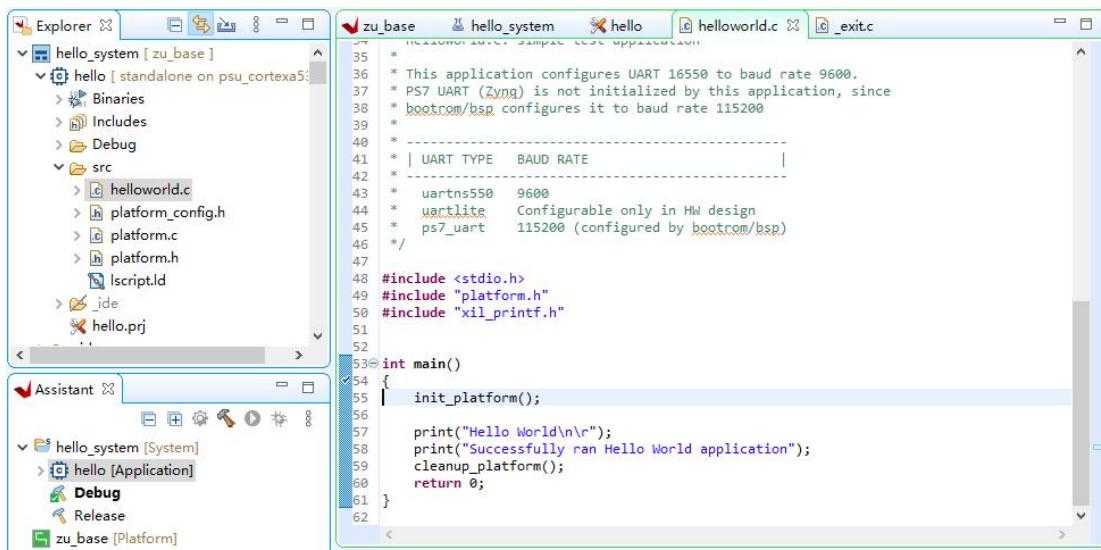
7.4 搭建 Vitis-sdk 工程

创建 soc_base sdk platform 和 APP 工程的过程不再重复，对于初学读者如果还不清楚如何创建 soc_sdk 工程的，请学习“3-1-01 米联客 2022 版 ZynqSocSDK 入门篇”中第一个工程“01Vitis Soc 开发入门”这个实验。

1: 创建 SDK Platform 工程

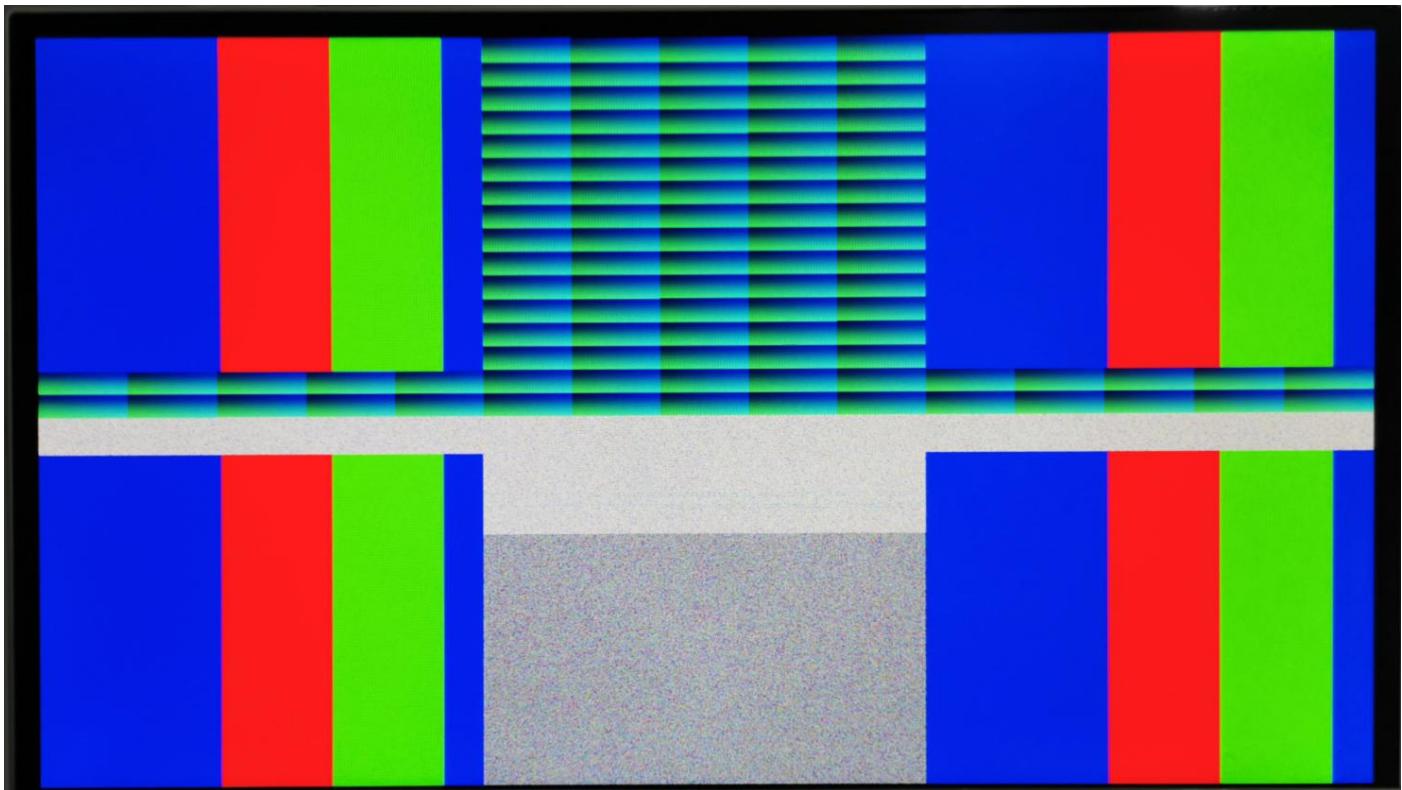


2: 创建 hello APP 工程



7.5 实验结果

由于 PS DDR 需要先运行 ARM 因此，先运行 hello app



08fdma 数据通路加入 sobel 算法 IP 方案

软件版本：vitis2021.1(vivado2021.1)

操作系统：WIN10 64bit

硬件平台：适用 XILINX A7/K7/Z7/ZU/KU 系列 FPGA

登录“米联客”FPGA 社区-www.uisrc.com 视频课程、答疑解惑！

8.1 概述

本文实验目的：

1:掌握 2 个 uifdma_dbuf IP 的同时使用，以及读写通道之间的同步设计

2:实现 1 路数据实时显示，1 路数据实时 sobel 计算通过 uifdma_dbuf+fdma3.1 保持到内存中。

3:掌握 uifdma_dbuf IP stride 参数的设置，实现一个屏幕实时显示当前摄像头视频以及实时计算的 sobel 视频

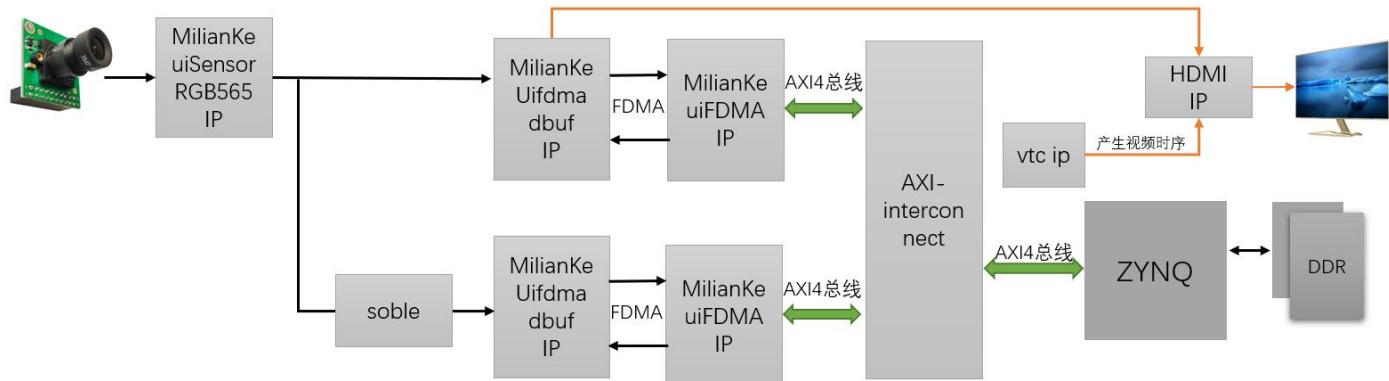
4:掌握图像处理中多通道读写的多缓存机制

8.2 系统框图

本系统中通过 milianke uifdma_dbuf ip 的写通道将摄像头采集到的数据通过 AXI intercomment 写入 DDR3，同时使用 milianke uifdma_dbuf ip 的读通道读取 DDR3 中的视频数据通过 HDMI 接口输出至显示屏。

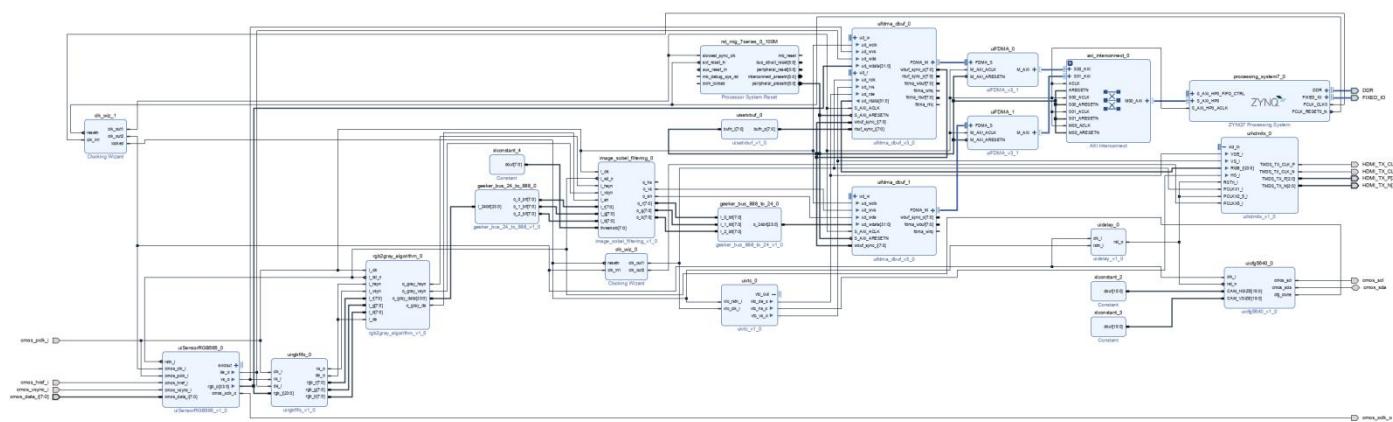
为 2 个输入图像在一个显示器输出，而不出现撕裂，关键就是在于 uifdmdbuf 的同步机制，这里让 uifdmdbuf1 的 wbuf_sync_o 作为同步信号，分别给 uifdmdbuf1 的读和写通道，以及 uifdmdbuf2 的写通道使用。uifdmdbuf2 的读通道没有使用。

将其中一路数据进行 soble 计算，为了完成 sobel 计算，RGB565 to RGB888 输出的 de 信号不是连续的，需要通过 uirgbfifo 转换成连续的 de 信号。



8.3 基于图形化的逻辑设计

详细的搭建过程这里不再重复，对于初学读者如果还不清楚如何创建 SOC 工程的，请学习“3-1-01 米联客 2022 版 ZynqSocSDK 入门篇”中第一个工程“01Vitis Soc 开发入门”这个实验。

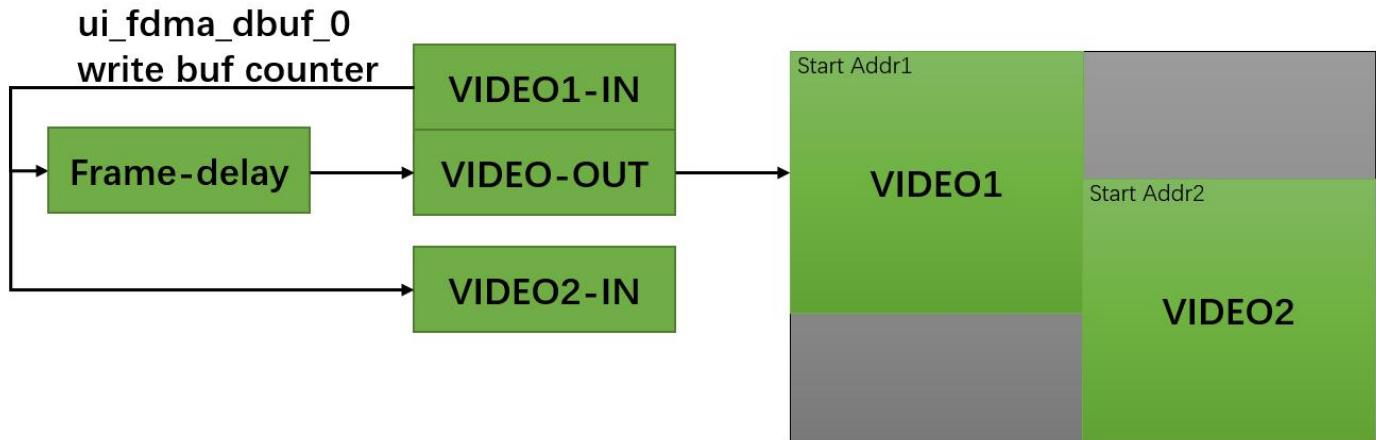


注意：关于时钟部分的设置，请打开配套工程里面的 IP 设置查看

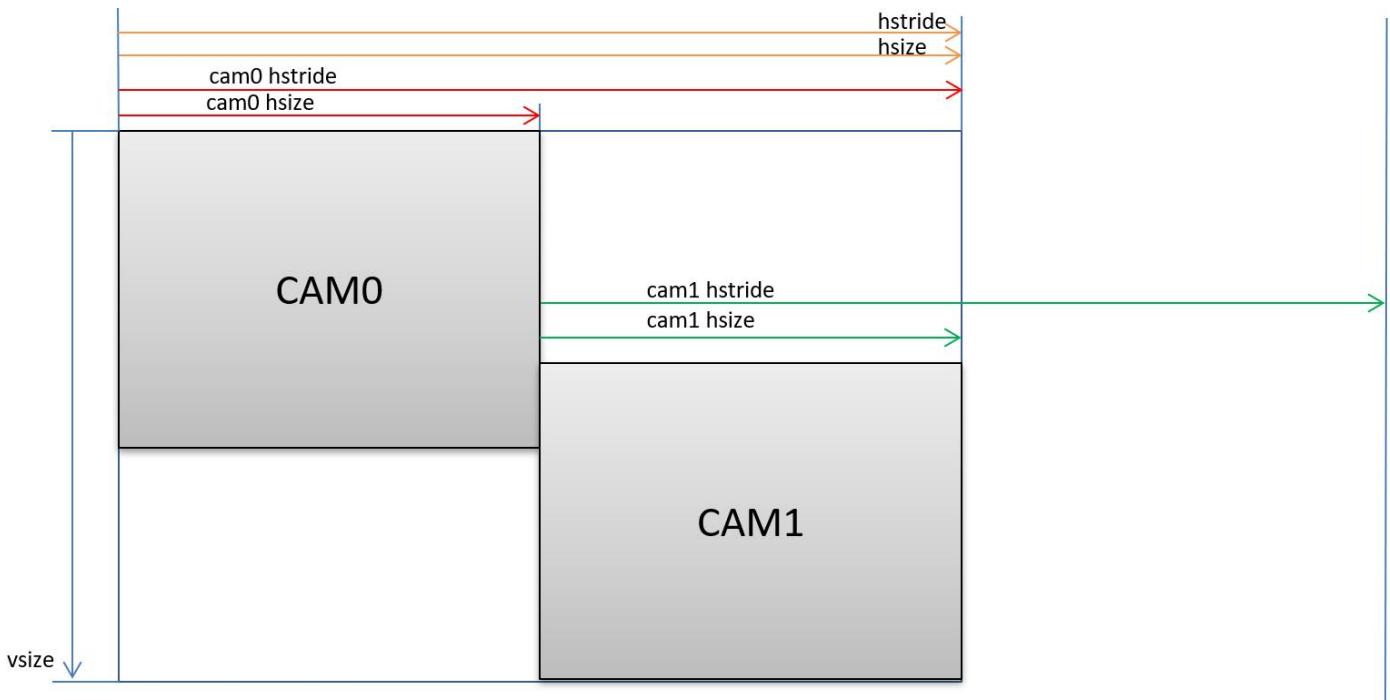
1:帧同步设计

对于多路视频传输的场合，需要正确设置同步。 uifdma_dbuf0 的写通道输出帧同步计数器直接接入

uifdma_dbuf0, uifdma_dbuf1 的写通道同步计数输入。uifdma_dbuf0 的读通道，延迟 1 帧于 uifdma_dbuf0 的写通道帧计数器。



2:多路视频的同屏显示原理



以把 2 个摄像头 CAM0 和 CAM1 输出到同一个显示器上为列，为了把 2 个图像显示到 1 个显示器，首先得搞清楚以下关系：

hszie: 每 1 行图像实际在内存中占用的有效空间，以 32bit 表示一个像素的时候占用内存大小为 $hszie * 4$

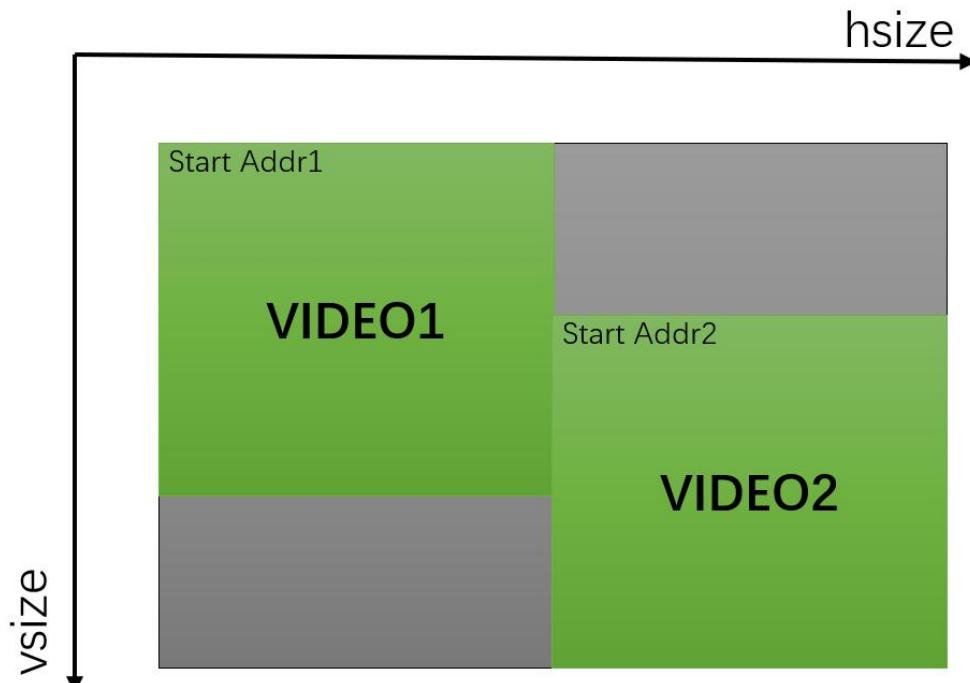
hstride: 用于设置每行图像第一个像素的地址,以 32bit 表示一个像素的时候 $v_cnt * hstride * 4$

vszie: 有效的行

因此很容易得出 cam0 的每行第一个像素的地址也是 $v_cnt * hstride * 4$

同理如果我们需要把 cam1 在 hszie 和 vszie 空间的任何位置显示，我们只要关心 cam1 每一行图像第一个像素的地址，可以用以下公式 $v_cnt * hstride * 4 + offset$

这里 2 个通道的 640*480 分辨率视频，在同一个 1280*720 的视频输出显示。VIDEO1 没有进行 soble 计算的原始图像在左上角,VIDEO2 是计算后的图像在右下角。



uifdma_dbuf 支持 stride 参数设置, stride 参数可以设置输入数据 X(hsize) 方向每一行数据的第一个像素到下一个起始像素的间隔地址, 利用 stride 参数可以非常方便地摆放输入视频到内存中的排列方式。

VIDEO1 的 start addr1=0x10000000(第一个图像的起始地址对于 PL DDR 可以从 0 地址开始, 低于 PS DDR 建议偏移 20MB)

VIDEO2 的 start addr1=0x10000000+(640-480)*1280*4+(1280-640)*4

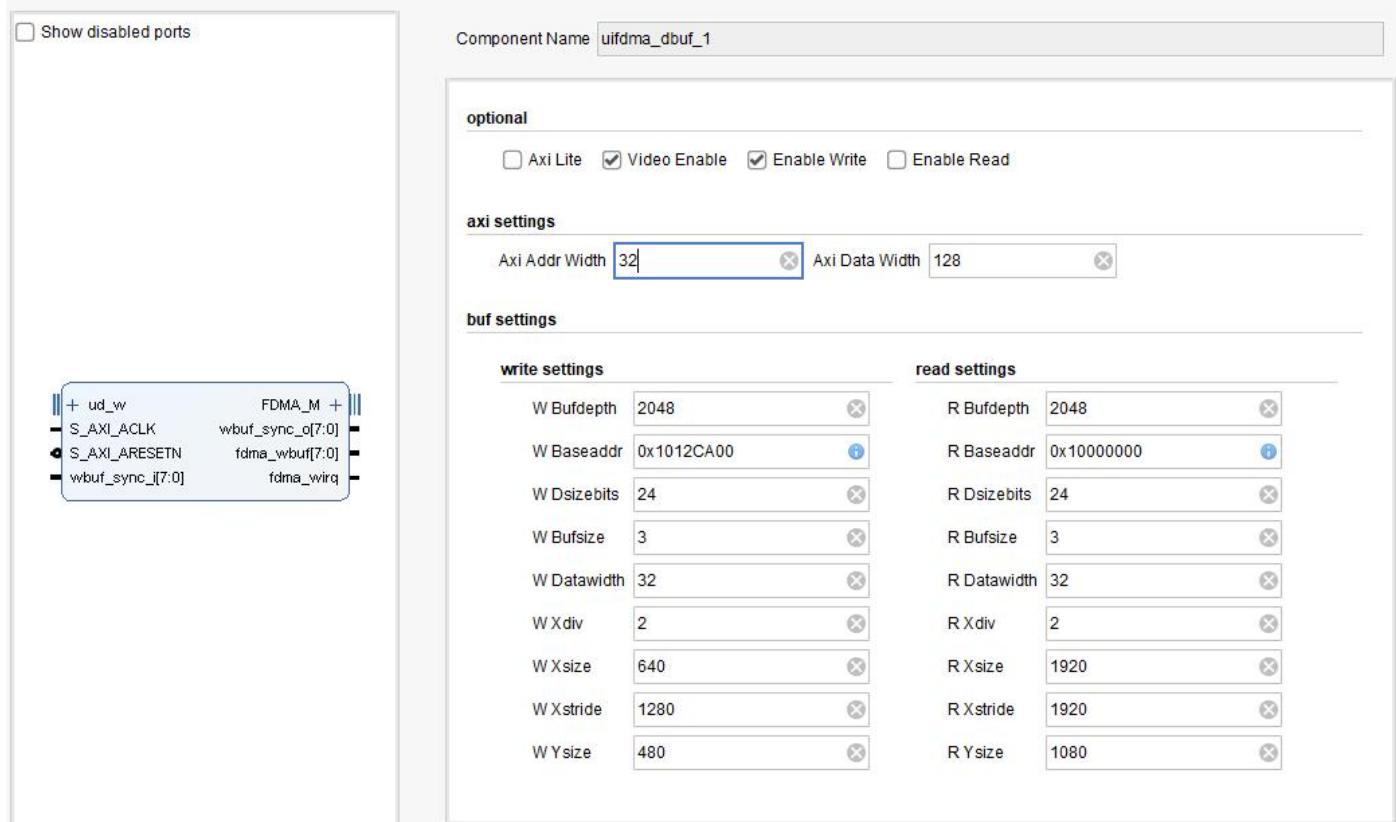
下面重点看下 uifdma_dbuf 的设置。

注意:这里的起始地址是从 256MB 处开始的, 对于 7010-A 核心板总共才 256M 内存, 那么可以起始地址可以改为 0x01000000

3:uifdma_dbuf_0 的配置

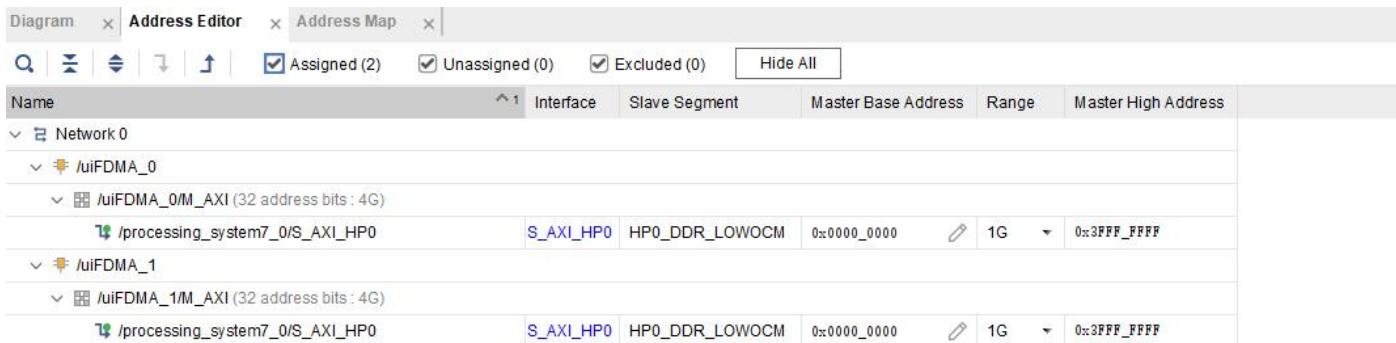
write settings		read settings	
W Bufdepth	2048	R Bufdepth	2048
W Baseaddr	0x10000000	R Baseaddr	0x10000000
W Dsizebits	24	R Dsizebits	24
W Bufsize	3	R Bufsize	3
W Datawidth	32	R Datawidth	32
W Xdiv	2	R Xdiv	2
W Xsize	640	R Xsize	1280
W Xstride	1280	R Xstride	1280
W Ysize	480	R Ysize	720

4:uifdma_dbuf_1 的配置



这里是把摄像头的输出分辨率设置为 640*480，并且把相同的摄像头数据分别输入到 uifdma_dbuf0 和 uifdma_dbuf1 的写通道。输出到显示器的分辨率为 1280*720

5:地址空间分配



6:编译并导出平台文件

1:单击 Block 文件→右键→Generate the Output Products→Global→Generate。

2:单击 Block 文件→右键→ Create a HDL wrapper(生成 HDL 顶层文件)→ Let vivado manager wrapper and auto-update(自动更新)。

3:生成 Bit 文件。

4:导出到硬件: File→Export Hardware→Include bitstream

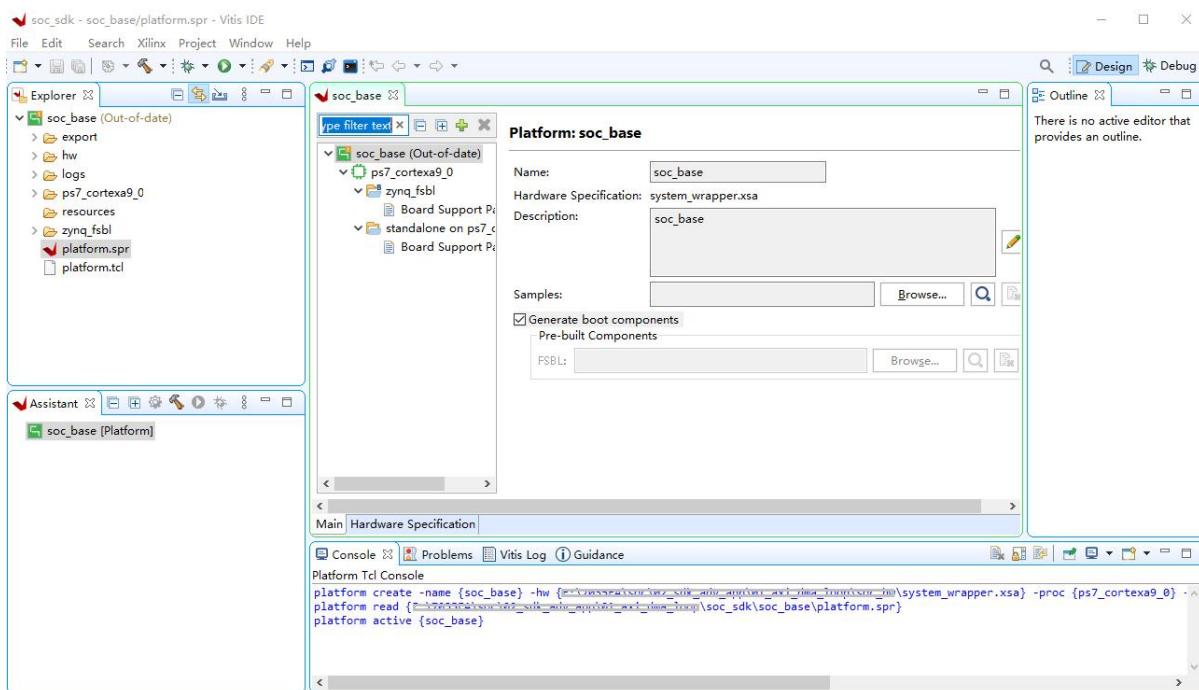
5:导出完成后，对应工程路径的 soc_hw 路径下有硬件平台文件: system_wrapper.xsa 的文件。根据硬件平台文件 system_wrapper.xsa 来创建需要 Platform 平台。



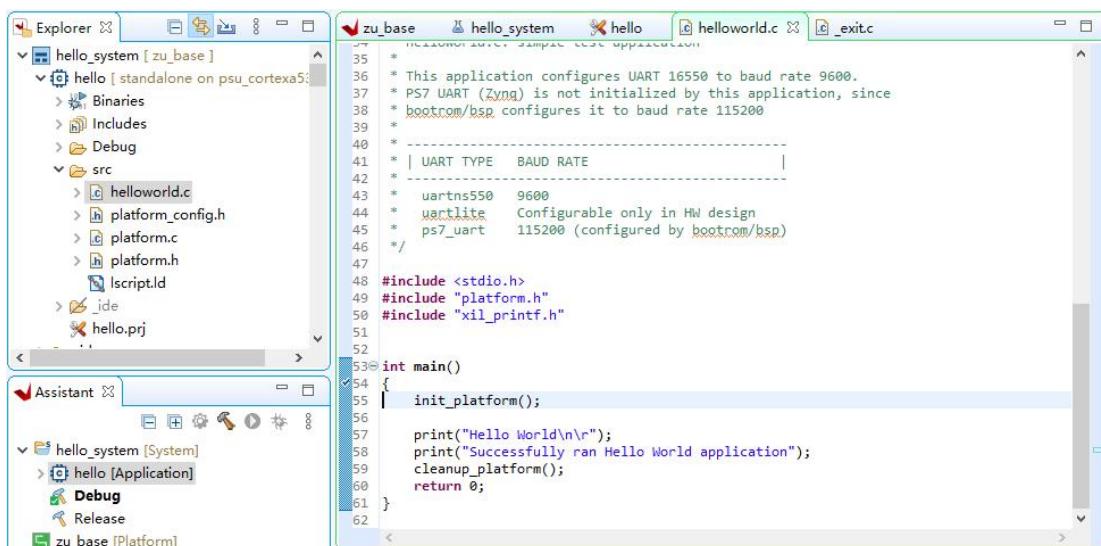
8.4 搭建 Vitis-sdk 工程

创建 soc_base sdk platform 和 APP 工程的过程不再重复，对于初学读者如果还不清楚如何创建 soc_sdk 工程的，请学习“3-1-01 米联客 2022 版 ZynqSocSDK 入门篇”中第一个工程“01Vitis Soc 开发入门”这个实验。

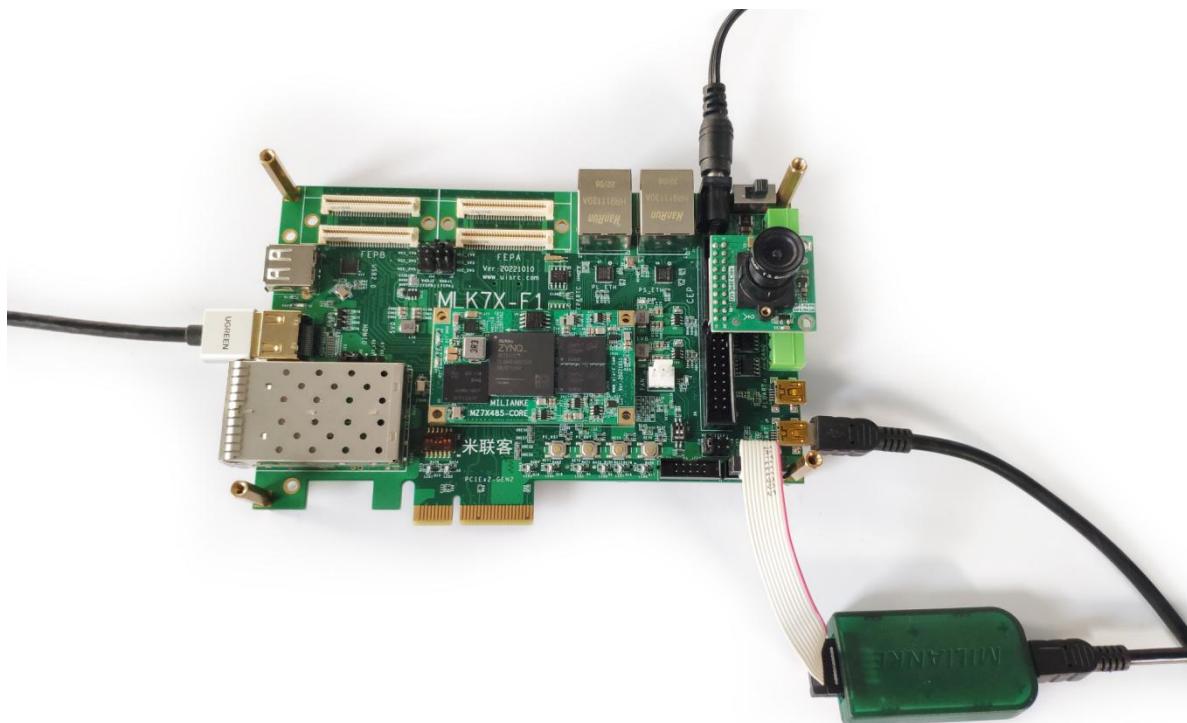
1: 创建 SDK Platform 工程



2: 创建 hello APP 工程

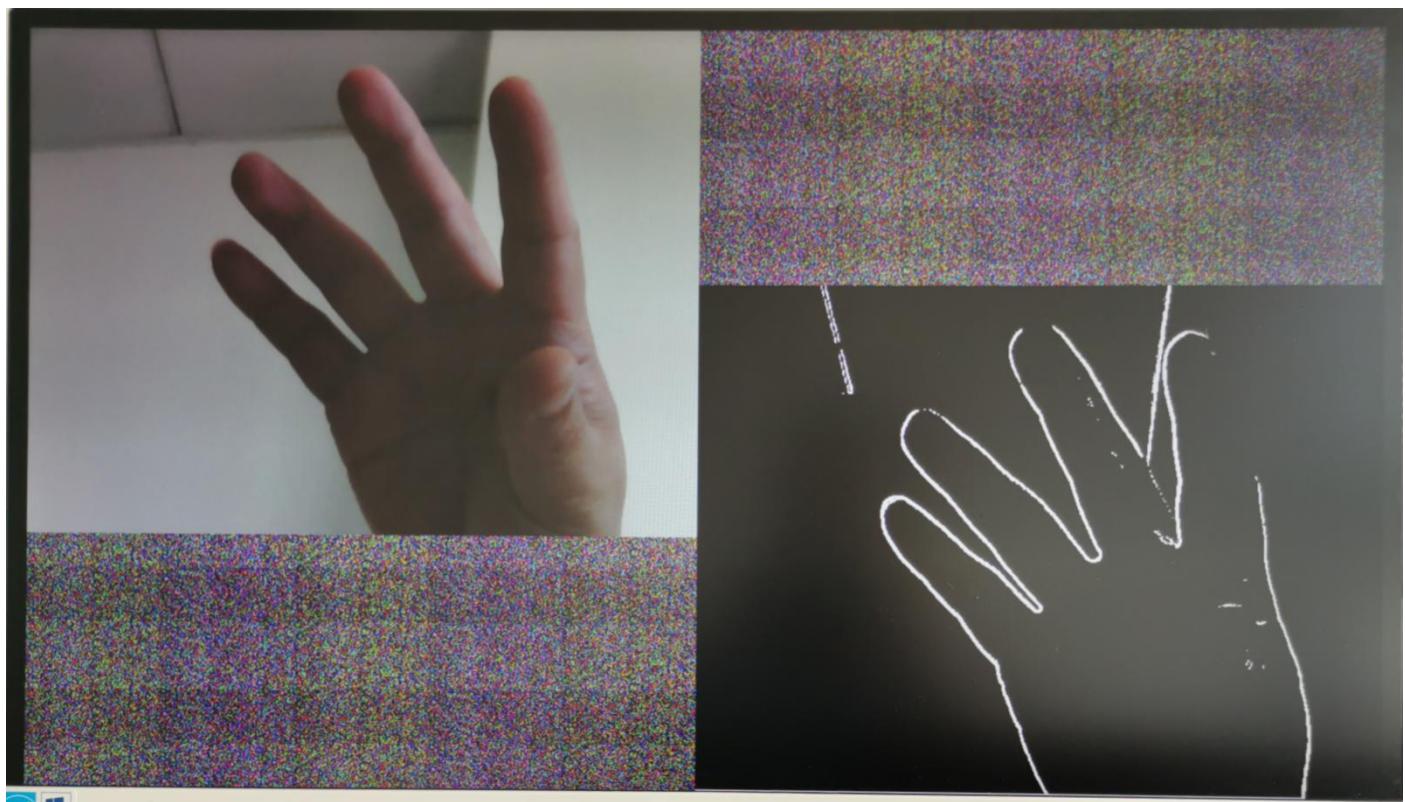


8.5 硬件接线



8.6 实验结果

由于 PS DDR 需要先运行 ARM 因此，先运行 hello app



附录 1:更多关于米联客 FDMA IP 的方案介绍

“米联客 2022 版 FPGA PCIE 通信方案”中使用到了 FDMA IP 实现和 PCIE-XDMA IP 共享内存数据
具体包括：

- “PCIE 数据卡 BRAM 缓存中断采集”
- “PCIE 数据卡 DDR 缓存中断采集”
- “PCIE 图像采集卡 HDMI 输入”
- “PCIE 摄像头图像采集卡”
- “PCIE 图片输出到 HDMI 显示器”
- “PCIE 实现同屏输出到 HDMI 显示器”
- “PCIE 与 PL 数据交互加速构架方案”
- “PCIE 与 ZYNQ PS 数据交互方案”

“米联客 2022 版 ZynqSocSDK 高级篇”中使用到了 FDMA IP 实现和 PS DDR 共享内存数据

- “PL 使用 PS 的 DDR 方案”
- “PL 发数据到 PS 方案(DBUF+FDMA)”
- “双摄像头采集显示方案(FDMA+VDMA)”
- “摄像头拍照方案(FDMA+VDMA)”
- “PS 图像数据共享给 PL 方案(FDMA)”
- “PS 灰度算法演示方案”
- “DAQ7606 波形显示方案(FDMA+VDMA)”
- “DAQ7606 以太网 LWIP TCP 传输方案(FDMA)”
- “DAQ9767-PS 波形数据共享给 PL 方案(FDMA)”
- “基于 LWIP TCP 的网络摄像头方案”
- “基于 LWIP UDP 的网络摄像头方案”

“米联客 2022 版 FPGA 图像处理课程”中的所有演示 demo 都用到 FDMA 进行数据交互

附录 2:常见问题

1 联系方式

技术交流群网址: <https://www.uisrc.com/f-380.html> 查看最新可以加入的 QQ 群

技术微信: 18951232035

技术电话: 18951232035

官方微博公众号 (新微信公众号) :



2 售后

1、7 天无理由退货(人为原因除外)

2、质保期限: 本公司产品自快递签收之日起, 提供一年质保服务(主芯片, 比如 FPGA 或者 CPU 等除外)。

3、维修换货, 需提供淘宝订单编号或合同编号, 联系销售/技术支持安排退回事宜。

售后维修请登录工单系统: https://www.uisrc.com/plugin.php?id=x7ree_service

4、以下情形不属于质保范畴。

A:由于用户使用不当造成板子的损坏: 比如电压过高造成的开发板短路, 自行焊接造成的焊盘脱落、铜线起皮 等

B:用户日常维护不当造成板子的损坏: 比如放置不当导致线路板腐蚀、基板出现裂纹等

5、质保范畴外 (上方第 4 条) 及质保期限以外的产品, 本公司提供有偿维修服务。维修仅收取器件材料成本, 往返运费全部由客户承担。

6、寄回地址, 登录网页获取最新的售后地址: <https://www.uisrc.com/t-1982.html>

3 销售

天猫米联客旗舰店: <https://milianke.tmall.com>

京东米联客旗舰店: <https://milianke.jd.com/>

FPGA|SOC 生态店: <https://milianke.taobao.com>

销售电话: 18921033576

常州溧阳总部: 常州溧阳市中关村吴潭渡路雅创高科制造谷 10-1 幢楼

4 在线视频

<https://www.uisrc.com/video.html>

5 软件下载

<https://www.uisrc.com/f-download.html>