

本科生毕业设计(论文)

**面向FPGA的软硬件结合异步状态机接⼝设计与实现**

Design and Implementation of an Asynchronous State Machine Interface for FPGA-Based Hardware-Software Integration

|  |  |
| --- | --- |
| 学 院： | 计算机学院 |
| 专 业： | 人工智能 |
| 班 级： | 07162003 |
| 学生姓名： | 龚思衡 |
| 学 号： | 1120203280 |
| 指导教师： | 陆慧梅 |

2024 年 5 月 15 日

原创性声明

本人郑重声明：所呈交的毕业设计（论文），是本人在指导老师的指导下独立进行研究所取得的成果。除文中已经注明引用的内容外，本文不包含任何其他个人或集体已经发表或撰写过的研究成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。

特此申明。

本人签名： 日 期： 年 月 日

关于使用授权的声明

本人完全了解北京理工大学有关保管、使用毕业设计（论文）的规定，其中包括：①学校有权保管、并向有关部门送交本毕业设计（论文）的原件与复印件；②学校可以采用影印、缩印或其它复制手段复制并保存本毕业设计（论文）；③学校可允许本毕业设计（论文）被查阅或借阅；④学校可以学术交流为目的,复制赠送和交换本毕业设计（论文）；⑤学校可以公布本毕业设计（论文）的全部或部分内容。

本人签名： 日 期： 年 月 日

指导老师签名： 日 期： 年 月 日

**面向FPGA的软硬件结合异步状态机接口设计和实现**

摘　要

在片上系统的发展过程中，异构计算一直是一个需要重点关注的课题。片上系统的异构计算是指在一个芯片上集成多个不同类型的处理器和加速器，以实现高效、灵活和优化的计算。这种异构计算架构可以充分发挥各种处理器和加速器的优势，针对不同的计算任务进行分配和协同工作，以提高系统的性能和效能。设计一个可供CPU调用FPGA的异构计算便利接口，可以减轻软件程序员的开发压力，并提高CPU与FPGA之间的通信效率，提高FPGA与CPU计算资源的利用率。

在本研究中，我们针对FPGA上的异构计算，使用软硬件结合的开发方式，设计并实现了一个供CPU调用FPGA运算功能的异步状态机接口，并对其功能的准确性与效率进行了测试。本研究中开发的异步状态机接口参考了操作系统中系统调用的实现机制，将FPGA内异构计算模块视为一个单独的函数，让CPU软件端能够以调用函数的方式调用FPGA资源辅助运算，实现软件开发者能够便利使用FPGA功能的简易接口。

关键词：FPGA；异步；状态机；函数调用；接口

**Design and Implementation of an Asynchronous State Machine Interface for FPGA-Based Hardware-Software Integration**

Abstract

In the development of System-on-Chip , heterogeneous computing has always been a subject that requires special attention. Heterogeneous computing in SoCs refers to the integration of multiple different types of processors and accelerators on a single chip to achieve efficient, flexible, and optimized computation. This heterogeneous computing architecture can fully leverage the advantages of various processors and accelerators, allocating and collaborating on different computational tasks to enhance the system’s performance and efficiency. Designing a convenient heterogeneous computing interface that allows CPUs to call FPGA can alleviate the development pressure on software programmers and improve the communication efficiency between CPUs and FPGAs, thereby enhancing the utilization of computational resources between FPGAs and CPUs.

In this study, we focus on heterogeneous computing on FPGAs and have designed and implemented an asynchronous state machine interface that allows CPUs to call FPGA computational functions using a hardware-software co-development approach. We have tested the accuracy and efficiency of its functions. The asynchronous state machine interface developed in this study references the implementation mechanism of system calls in operating systems, treating the heterogeneous computing module within the FPGA as a separate function. This allows the CPU software side to call FPGA resources to assist in computation as if calling a function, realizing a simple interface that enables software developers to conveniently utilize FPGA features

Key Words: FPGA; Asynchronous;State machine；Function calls；Interface

目　录

[摘　要 I](#_Toc21006)

[Abstract II](#_Toc12626)

[第1章 绪论 1](#_Toc17805)

[1.1 研究背景 1](#_Toc22838)

[1.2 国内外研究现状 2](#_Toc14810)

[1.3 研究内容及贡献 3](#_Toc24324)

[1.4 全文结构 4](#_Toc3187)

[第2章 相关工作 4](#_Toc5844)

[2.1 FPGA 4](#_Toc17240)

[2.2 软硬件协同设计 6](#_Toc4034)

[2.2.1 传统设计 6](#_Toc21968)

[2.2.2 协同设计 7](#_Toc24114)

[2.3 AXI接口 10](#_Toc24602)

[2.4 DMA 13](#_Toc18726)

[第3章 结构设计 14](#_Toc16874)

[3.1 FPGA与CPU通信 15](#_Toc31570)

[3.1.1传统系统调用 15](#_Toc26028)

[3.1.2 SoC上函数调用 16](#_Toc7641)

[3.2 函数模块 17](#_Toc7891)

[3.2.1 矩阵加法模块 22](#_Toc12752)

[3.2.2 数组乘法模块 23](#_Toc16329)

[3.2.3 全并行排序模块 24](#_Toc26364)

[3.3 多通道DMA 25](#_Toc19516)

[3.3.1 DMA结构 25](#_Toc24936)

[3.3.2 多通道DMA架构 26](#_Toc4815)

[第4章 结构实现 27](#_Toc11462)

[4.1 硬件布线 27](#_Toc4490)

[4.1.1 单通道DMA布线 29](#_Toc21431)

[4.1.2 多通道DMA布线 31](#_Toc25201)

[4.2 函数IP内部布线 33](#_Toc622)

[3.2.1 矩阵加法模块 33](#_Toc6700)

[3.2.2 数组乘法模块 34](#_Toc5958)

[3.2.3 全并行排序模块 34](#_Toc4268)

[4.3 状态机实现 35](#_Toc22859)

[4.4 函数调用实现 37](#_Toc3115)

[第5章 结果与分析 38](#_Toc3012)

[5.1 资源占用量 38](#_Toc18684)

[5.2 仿真波形 38](#_Toc3898)

[5.2.1 数组乘法 39](#_Toc27906)

[5.2.2 矩阵加法 39](#_Toc915)

[5.2.3 全并行排序 40](#_Toc11215)

[5.3 函数调用时间 41](#_Toc15424)

[结　论 42](#_Toc18609)

[参考文献 43](#_Toc7815)

[附　录 44](#_Toc20026)

[致　谢 45](#_Toc880)

1. 绪论

1.1 研究背景

在片上系统（System-on-Chip，SoC）的发展过程中，异构计算一直是一个需要重点关注的课题。片上系统的异构计算是指在一个芯片上集成多个不同类型的处理器和加速器，以实现高效、灵活和优化的计算。这种异构计算架构可以充分发挥各种处理器和加速器的优势，针对不同的计算任务进行分配和协同工作，以提高系统的性能和效能。

在异构计算中，常见的处理器和加速器包括以下几方面：一是中央处理器（CPU），CPU是片上系统中最常见的处理器，负责通用计算任务。它具有较强的单线程性能和复杂的控制逻辑，适用于运行操作系统、控制流程和执行串行任务等。二是图形处理器（GPU），GPU主要用于图形渲染和图像处理任务。它具有高并行计算能力和大规模并行处理单元，适合于同时处理大量数据的并行计算，如图像处理、计算机视觉和科学计算等。三是张量处理器（TPU），TPU是专门用于加速人工智能和机器学习任务的处理器。它具有高度优化的矩阵乘法和向量计算单元，能够高效执行深度神经网络的推断和训练任务。四是数字信号处理器（DSP），DSP主要用于数字信号处理和实时控制任务。它具有高效的乘法和累加运算单元，适合于音频处理、视频编解码和通信信号处理等领域。五是硬件加速器（例如FPGA、ASIC），硬件加速器是定制化的处理器，根据特定的计算需求进行设计和优化。FPGA具有可重构的逻辑和数据通路，适合于快速原型开发和灵活的定制化计算。ASIC是专门设计和定制的应用特定集成电路，具有更高的性能和能效，但开发和生产成本较高。

不同处理器和加速器之间通过高速互联通道进行通信和数据传输。根据具体的计算任务和开发需求，系统可以动态地将任务分配给不同的处理器和加速器，以实现最优的计算效果。其中，集成了FPGA与CPU的SoC可同时具备FPGA的⼤规模并发式处理能⼒以及CPU的通⽤计算能⼒。但其也面临着一些问题，在FPGA程序设计中很难⾼效的使⽤到CPU的计算能⼒造成资源浪费。在需要用到大量浮点数及乘除法运算的步骤中，FPGA上计算元器件数量成为瓶颈。因此，可以通过编写FPGA高效使用CPU计算存储资源的异步接口，来提高对CPU计算资源的利用率。同时，CPU对FPGA方面也存在一定的局限性，如FPGA功能较为固定不够灵活。若采用库函数调用封装的方式，使得CPU对FPGA功能的使用更加多变。

本课题的目标是实现一种基于软硬件的FPGA与CPU通信接口，它在保留了CPU与FPGA通信的优势的前提下，一方面可以提高对CPU计算资源的利用率，节省FPGA资源损耗；另一方面可以增强CPU对FPGA调用的灵活程度，提高计算性能。同时，定义一个具备可移植性的接口规范，便于软件与硬件程序员的使用。

1.2 国内外研究现状

目前，基于FPGA的加速器设计主要关注加速器的性能和其所提高的效率[1]。FPGA程序员可以进行如下操作：( 1 )创建由简单(通常是低位宽)计算单元组成的深度定制流水线，而不是完全成熟的ALU；( 2 )构建高度并行和分布式的控制逻辑和片上存储；( 3 )以显式的方式调度数据流，以尽量减少片外存储器访问，而不使用缓存。这与编程微处理器(即CPUs)和通用图形处理单元(也就是说, GPU)形成鲜明对比，后者的底层硬件架构(指令流水线、存储器层次结构等。)是固定的，软件程序的控制流驱动硬件基于指令的执行。FPGA可以针对特定的应用或应用领域进行重新配置/定制[8]，来利用其大规模的细粒度并行性和片上带宽。与CPU和GPU相比，这往往提供了更高的计算吞吐量，更低的能耗和更可预测的延迟。

FPGA常用的一个领域就是在神经网络的训练中。在卷积神经网络（CNN）的训练阶段[2]，微软的研究人员使用了FPGA（Stratix V D5）来实现高效的加速。目前，包括亚马逊、微软、腾讯和阿里巴巴在内的多家国际知名企业都在逐渐尝试将FPGA集成到他们的数据中心。FPGA和CPU的协同工作已经逐渐成为了现代SoC异构开发的发展趋势。

在SoC FPGA中，CPU和FPGA是紧密结合的[3]，它们被封装在同一个芯片内，并通过总线进行连接。CPU通常有两种类型：硬核和软核。目前，FPGA供应商如Xilinx和Altera所提供的最新SoC开发板，通常采用ARM硬核CPU。SoC FPGA的主要特点是集成度高、功耗低、通信带宽大，这使得它们非常适合用于嵌入式系统。

与SoC FPGA相比，标准FPGA和CPU之间是松散耦合的，这意味着FPGA和CPU不是封装在同一个芯片上，而是通常通过PCIe（外设组件互连快速版）接口来连接。由于标准FPGA没有在芯片中集成微处理器[7]，所以在相同的芯片面积下，标准FPGA可以提供比SoC FPGA中的FPGA逻辑部分更多的硬件资源，从而支持更复杂的应用。在标准FPGA中，CPU和FPGA之间的数据交换通常有两种方式，一种是基于PCIe总线的交换，另一种是基于QPI（快速路径互连）总线的交换。传统的FPGA设计工具主要针对硬件设计专家[4]，而不是软件编程人员。使用传统的寄存器传输级( RTL )方法手动创建和优化加速器架构需要花费大量的精力。人们必须与低级硬件描述语言( HDL )描述和计算机辅助设计( CAD )工具进行斗争，以实现丰富的硬件定制，如定点运算、流水线、存储单元和双缓冲。更糟糕的是，将RTL设计综合为比特流通常需要数小时，甚至数天[5]。这种漫长的编译周期使得FPGA上的设计空间探索( DSE )变得非常昂贵。如何降低FPGA与CPU协同开发的门槛，让软件与硬件程序员能使用友好的接口进行编程是一个需要重点关注的问题。

1.3 研究内容及贡献

本毕设的研究目标是实现一种基于软硬件的FPGA与CPU通信接口，在保留CPU与FPGA通信的优势的前提下，一方面希望提高对CPU计算资源的利用率，节省FPGA资源损耗；另一方面希望增强CPU对FPGA调用的灵活程度，提高计算性能。同时，定义一个具备可移植性的接口规范，便于软件与硬件程序员的使用。

具体工作分为接口设计和性能测试两部分。首先在zedboard上设计并实现CPU与FPGA的异步调用，并验证其正确性，再将其和现有的异构计算机制进行性能测试，对比分析测试结果。

在接口设计方面在CPU与FPGA之间设计一个透明的互相调用接口。在CPU调用FPGA时，对FPGA的调用是一个函数调用的形式，通过类似于系统调用的封装方式，将FPGA中已设计的函数功能模块在CPU内封装为不同的函数调用，在CPU想要使用FPGA内功能时，只需使用对应的函数调用即可。

本研究所实现的软硬件接口需要达到的目标为：可以正常工作并给出期望结果；能够发挥异构计算其特有的性能优势；具有可扩展性并具备可移植性。本研究的对比测试部分需要达到的目标为：设计科学、完善的对比实验并得出可靠结果；分析产生实验结果的原因；分析本研究实现的各个机制对结果的影响。

FPGA与CPU之间通常采用总线传输协议，即AXI、PCIE等。FPGA内的模块通常作为外设模块挂载在总线上，每个模块分配一个物理地址。在CPU中运行的程序直接通过物理地址对FPGA内模块进行读写访问。FPGA要访问内存时，通常采用同步访问方式，要实现较为复杂的状态机控制每一个读写信号，比较麻烦。另外，因为没有Cache因此访问速度慢。

 CPU有内置Cache机制，访问速度快；且有高级语言封装，对于程序员来说不用考虑同步和异步问题，访问方便[6]。FPGA内部通常集成了存储空间和DSP模块（用于浮点计算），但资源较少，无法与CPU的三级存储结构效率和易用性相提并论。

在使用FPGA进行计算的过程中，对于大规模的数据计算，FPGA实现需要消耗的计算资源量较多，受制于其计算元器件数量。而CPU的计算资源得到的利用较少，产生了计算资源的浪费。通过实现FPGA调用CPU的接口，解决了CPU计算资源利用率较少问题，节省了计算资源。同时，CPU对于FPGA调用的灵活性较低，通过实现CPU调用FPGA的函数式封装，提高了调用的灵活性。本课题对于之后SoC相关接口设计有着启发性作用，函数调用的形式使得对FPGA功能模块的调用更加简洁，便利，直观，有助于软件程序设计员的使用。

1.4 全文结构

本文将分为六个章节展开叙述，第一章为绪论，简单介绍研究课题背景及相关研究的国内外现状，简述本课题的研究内容及贡献。第二章为相关工作，将介绍本课题研究时所基于的基本知识以及与其相关的对应设备。第三章为接口结构设计，本章将在宏观上论述接口的结构设计图示，并对其主要功能模块的结构设计进行解释。第四章为结构实现，在本章将从RTL语言的层次对接口结构进行分析，并介绍接口具体逻辑部分的布线情况与逻辑门使用情况。并对接口的主体状态机代码部分进行分析，综合其可行性。第五章为实验结果与分析，在本章将对接口结构实现后的具体资源占用情况进行分析，并测试接口功能，进行多函数调用功能的检验，并对其实验结果进行分析。结论为总结与展望，将对全文内容进行总结，并分析课题接下来研究的展开与可能的改进方向。

第2章 相关工作

2.1 FPGA

FPGA设备是一种专用集成电路，它具有半定制的特性，能够按照开发者的需求进行功能的定制专有化。实际上，FPGA是一种具有可编程特性的逻辑阵列，这种独特的设计特性，可以有效地解决传统器件中逻辑门数量有限的问题，提供更多的开发资源。FPGA的核心结构的组成部分较多：具备可编程特性的输入输出单元、可以自由配置的逻辑块、数字时钟管理模块、嵌入式块RAM、布线资源、内置专用硬核和底层功能单元。FPGA具有丰富的布线资源、可重复编程性、高集成度和较低的投资成本，这使得FPGA具有其独到的优势，在数字电路设计领域得到了广泛应用。FPGA的设计流程涉及算法设计、代码仿真、设计和上板调试。设计师根据实际生产或研究的需求建立算法框架，设计需要的FPGA功能。使用EDA工具或HDL语言编写设计代码，并通过代码仿真或逻辑行为分析来确保设计方案满足实际开发研究的需求。最终，通过板级调试，将相关工程文件装载到FPGA芯片中，在实际开发板上运行程序，以验证设计的实际效果。[10]

FPGA内部主要以逻辑单元阵列（LCA）为主，这是一种适合研究开发以及工业生产的结构。其内部模块包括可配置逻辑模块（CLB）、输入输出模块（IOB），这些模块通过复杂的内部连线互相串联起来，形成了FPGA的逻辑单元阵列的内部结构。与传统的逻辑电路和门阵列（如PAL、GAL和CPLD）不同，FPGA具有独特的结构，而这一独特的结构使得FPGA在特定领域有着其独有的优势。它使用一种小型查找表，这种查找表多数以16个1位RAM的形式出现，并将这种查找表的结构机制用来实现内部的组合逻辑。在此基础之上，通过使用D触发器将查找表的输出连接到其他逻辑电路或I/O端口。这样，FPGA可以实现基本的逻辑单元模块，这种模块同时具有组合性与时序性，既能完成组合逻辑功能，也能实现时序逻辑功能，这是FPGA的又一优势所在。FPGA的逻辑编程功能是通过向内部静态存储单元加载编程完毕的数据来实现的。这些编程数据决定了逻辑单元的功能以及模块之间或模块与I/O之间的连接方式，定制了开发者需要的FPGA内部结构。同时，FPGA支持无限次的编程，开发者能够在同一块FPGA板上反复迭代其设计的结构，这种特性决定了FPGA适用范围的广泛性。[11]相较于其他类型的芯片设计，FPGA芯片设计对通常开发者设有较高的门槛，需要开发者具备大量的FPGA芯片自身相关知识，以及对应产品的使用知识，在设计过程中，更要遵循严格的设计流程，才能保证芯片的可使用性。身为FPGA设计师，在开发FPGA的过程中需要紧密结合FPGA的原理图来实现大规模的专用芯片设计。通过使用Matlab和C语言的特殊设计算法，可以实现全面的转换，确保设计符合当前主流的芯片设计理念。此外，设计算法应当优先考虑合理性，以优化项目设计效果和芯片运行效率。设计人员首先需要为开发所用的算法设计合适的算法模块，以便完成相关的芯片代码设计。预先设计的代码有助于确保算法的可靠性，并显著优化整体的芯片设计效果。在完成上板调试和仿真测试后，可以缩短整个芯片设计周期，并优化现有硬件结构。例如，在开发非标准硬件接口时，通常会采用这种新的产品设计模式。[13]

FPGA设计的主要挑战在于开发者应熟悉硬件系统的架构和其内部资源的数量，确保设计的模块能够有效地协调内部各元器件之间的配合。在编写程序时，也应注重提高程序的可读性和重复利用率。这种设计要求对设计人员提出了较高的挑战，需要通过参与并开发多个项目的经验积累来达到这些要求。在算法设计阶段，需要开发人员重点考虑算法设计的合理性，并紧密结合项目的需求设计恰当的算法，确保项目的最终实际效果，并根据项目的实际困难情况提出可能的解决方案，提高FPGA的实际运行效率。确定算法后，应合理构建模块以便于后期代码设计。在代码设计阶段，可以使用预先设计好的，或者已经被证实确实有效的代码来加快工作效率和可靠性。之后则需要编写测试平台进行代码的仿真测试和上板调试，完成整个设计过程。与ASIC不同，FPGA的开发周期较短，可以根据设计要求调整硬件结构，在通信协议不成熟的情况下帮助企业迅速推出新产品，满足非标准接口开发的需求。[13]

然而，FPGA其自身也有不小的局限性，它所有的逻辑功能都依赖于硬件来实现，无法进行分支条件跳转等较为复杂的逻辑操作。同时FPGA内部只能执行定点运算，浮点运算则需要编写相应的模块。尽管FPGA在速度上可以与专用芯片相媲美，但其设计的灵活性以及泛用性与通用处理器相比仍有所不足。[12]

2.2 软硬件协同设计

2.2.1 传统设计

传统的系统设计策略通常遵循自顶向下的流程，或者采用硬件模块化的方法，这些方法主要强调硬件模块的优先考虑，即以硬件作为设计的核心，待硬件的总体设计结束之后再在其基础之上进行软件的设计。这种方法直观且符合逻辑，但是却忽视了软件设计对于硬件开发的影响。其核心设计理念流程如图2-1所示。

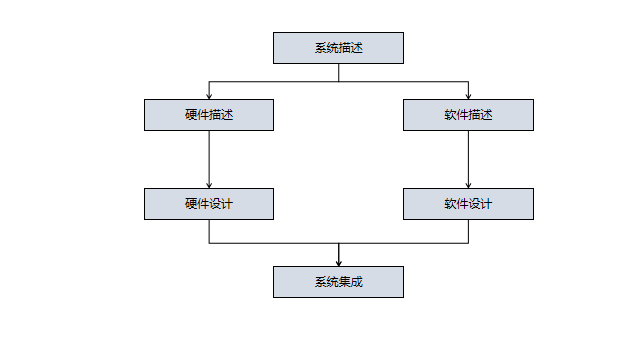


图2-1 传统设计流程

这个设计策略把硬件和软件视为两个分开的部分。在设计的整个过程中，一般先考虑硬件，也就是说，在初步评估了软件的任务需求后，会优先进行硬件的设计。硬件平台建立之后，再在这个平台上进行软件的开发。这种方法在硬件设计阶段往往没有充分考虑到软件的架构和实现细节，因此存在一定的限制。在对系统进行优化和修改时，由于受到设计所考虑的范围的限制，往往只能分别对硬件和软件进行修改，割裂地提升其各自的性能，而无法对整个系统的工作性能进行全面的优化。这样的设计结果通常不能最大化地同时利用硬件和软件资源，使硬件与软件的资源的使用不能紧密结合，难以满足现代复杂系统设计的需求。

2.2.2 协同设计

软硬件协同设计是一种将软件和硬件设计过程紧密结合的工程实践，目的是优化整个系统的性能和效率。在这个过程中，首先需要定义系统的需求和功能，包括硬件和软件的功能、性能、接口等。硬件设计阶段涉及使用硬件描述语言来设计硬件模块，并使用综合工具将这些设计编译成电路。软件设计则包括编写代码和使用编译器将代码转换为机器码。总线设计与连接是将处理器核、外设和内存连接起来的重要步骤。接下来，通过仿真工具验证硬件和软件的正确性，并进行调试以确保系统按预期工作。最后，将机器码下载到FPGA或ASIC中，运行系统并观察结果。这个流程虽然挑战重重，但随着工具和方法的不断改进，软硬件的紧密集成正在逐步成为现实。

软硬件协同设计指的是将软件和硬件的设计过程综合进行考虑，整合为一个统一的并行过程，以此确保软件和硬件的设计能够紧密结合，实现系统设计的高效，精确运作。这种设计方法的核心流程展示在图2-2中。

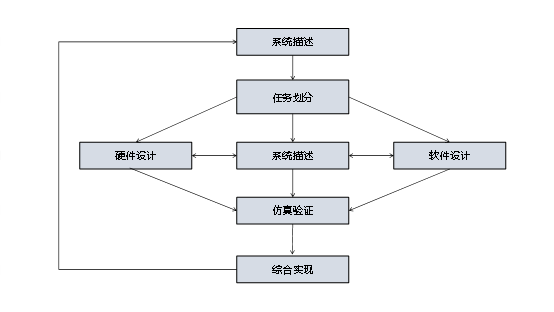


图2-2 软硬件结合设计流程

图示表明，硬件与软件的协同设计的主要优势在于设计阶段的互动性。这种互动不仅贯穿设计的每个阶段，还涉及到各个层面，确保了硬件与软件的协同发展。在功能分配阶段，已经充分考虑到了现有的硬件与软件资源的局限，并根据实际情况对功能进行分配，而在功能设计和仿真评估过程中，硬件与软件相互支撑，互相根据其结果对功能的设计进行修正。这样的设计方法促使硬件与软件的功能模块能在开发初期就实现整合，这有助于尽早识别并解决系统设计中的问题，避免了后期可能需要的重复修改，从而有效地发掘系统的潜力，减少体积，降低成本，并提升系统的整体性能。在硬件与软件协同设计的基础上，我们可以进一步探讨其对系统开发周期的影响。通过在设计初期就实现硬件与软件的紧密结合，开发团队能够更加高效地进行迭代，因为它们可以及时发现潜在的不匹配和问题点。这种早期的问题发现和解决机制，不仅减少了项目延期的风险，还有助于维持预算控制，避免了因后期大规模修改而导致的成本溢出。此外，协同设计还促进了跨学科团队之间的沟通和协作。硬件工程师和软件工程师可以共同工作，共享知识和理念，从而创造出更加创新和高效的解决方案。这种合作环境，不仅提高了团队成员的技能和专业知识，也为项目带来了更多的创新潜力。

软硬件协同设计过程可以大致分为“系统描述、系统设计、仿真验证和综合实现”四个阶段 [14]。

系统描述指的是运用一种或多种表述工具来对目标系统的功能与性能做出详尽的阐释，并构筑其软件与硬件的模型。这个建模过程可以通过EDA工具来辅助实施，亦可由设计人员采用自然语言来手工完成。系统描述的主要目的是为了确保设计团队能够准确理解和实现系统的功能需求和性能指标。通过详细的系统描述，可以建立一个清晰的软硬件模型，这有助于指导整个设计过程，确保最终产品能够满足预定的功能和性能要求。此外，系统描述还能够促进团队成员之间的沟通，提高设计效率，减少误解和错误，从而降低开发成本和时间。

系统设计的过程主要涉及两个关键阶段：功能分配和系统架构确定。在功能分配阶段，决策者需要判定系统中的各项功能是由硬件还是软件来承担。通常情况下，硬件能提供卓越的性能表现，而软件则因其开发和修改的灵活性、成本效益而受到青睐。随着硬件模块的可配置性和可编程性的提升，以及软件功能向硬件和固件的转变，软硬件之间的界限变得模糊，进而协同开发软硬件已具备其实施条件。在分配功能时，需要综合考量市场资源、系统成本和开发周期等多个因素，这使得功能分配成为一个复杂且挑战性的任务。

在系统设计的功能分配阶段完成后，接下来就是系统架构确定阶段。这个阶段的目标是根据功能分配的结果，选择合适的硬件和软件模块，并确定它们之间的通信方式，以及如何将这些模块集成到一起，形成最终的系统架构。在硬件模块的选择中，需要选定能够满足性能要求的硬件组件，如微控制器单元 (MCU)、数字信号处理器 (DSP)、现场可编程门阵列 (FPGA)、存储器和输入/输出接口等。同时，在选择过程中，要考虑到性能、功耗、成本和可靠性等多个因素。除硬件模块的选择之外也要考虑到软件模块的选择即确定系统所需的软件组件，包括操作系统、驱动程序和应用功能模块等。选择软件时，要考虑到开发的复杂度、可维护性和性能等因素。在软硬件结合设计中尤为重要的是接口的设计。接口设计包括设计硬件和软件模块之间的接口，包括数据传输方式、通信协议和总线结构等。接口设计要确保模块间的数据传输和通信高效、可靠的原则。将选定的硬件和软件模块集成到系统中，并进行系统级测试，验证系统是否按照设计要求正常工作。这个阶段是系统设计中至关重要的一步，它确保了系统的功能、性能和可靠性能够满足项目的需求。通过这一阶段的精心规划和执行，可以构建出一个高效、稳定且具有成本效益的系统。

仿真验证是系统设计过程中的一个重要步骤，它通过模拟测试来确保设计的准确性。这个过程的目标是在系统实施之前发现并修正潜在的问题，以减少后期修改的需要。然而，在仿真过程中，由于模拟环境与实际环境存在差异，软硬件的交互可能与现实中的表现不同，这可能影响到系统在实际操作中的可靠性。因此，尽管仿真验证是一个有价值的工具，但它并不能完全替代真实环境下的测试，其结果需要谨慎对待。

综合实施过程涉及软件和硬件系统的详细设计活动。在设计方案通过仿真测试并证明其有效性之后，就可以根据系统设计的规格开始系统的开发和制造。这意味着根据之前的规划来构建硬件和软件，确保它们能够协同工作。完成这些步骤后，系统将接受各种测试以验证其性能和功能。

2.3 AXI接口

AXI（高级可扩展接口）是ARM公司在其AMBA 3.0规范中推出的一种关键总线协议。它专为实现高效能、高数据传输速率和低通信延迟的片上总线系统而设计。AXI协议支持高性能的数据传输操作，包括突发传输和单点传输，这使得它能够有效地处理大量数据。此外，AXI还提供了灵活的交互机制，如分离的读写通道和独立的地址/控制和数据相位，进一步优化了数据流和控制信号的处理。这些特性使AXI成为了设计高速、可靠和可扩展的系统级集成（SoC）设计的理想选择。它广泛应用于需要高带宽和低延迟的应用中，如多媒体处理、网络通信和实时计算任务。在AMBA4.0中将其升级为AXI4协议，具体包括AXI4.0、AXI4.0-lite、ACE4.0、AXI4.0-stream。AXI4.0-Lite是AXI的简化版本，ACE4.0是AXI缓存一致性扩展接口，AXI4.0-Stream是由ARM公司和Xilinx公司一起提出，主要用在FPGA进行以数据为主导的大量数据的传输应用。

AXI以下有三种接口协议，满足不同的使用需求

AXI4：是一种通过地址访问数据的存储器映射协议。这个协议主要用于高速数据传输，例如处理器/FPGA等主设备访问DDR等从设备。相当于原来的AHB接口协议。

AXI-Lite：是AXI4的简化版，单次仅能读写1个数据，类似原来处理器通过EMIF读写FPGA寄存器，主要用于寄存器的配置。相当于原来的APB接口协议。

AXI-stream：与上面两种存储器映射方式不同，这各个协议的数据传输不需要地址，而是在主从设备之间直接连续读写数据，主要用于视频、高速AD、PCIe、DMA接口等需要高速数据传输场景，其机理与FIFO类似。

AXI协议采用基于burst的突发传输机制。这种机制的特点在于定义了读地址通道，读数据通道，写地址通道，写数据通道，写响应通道，这五个独立的传输通道。在AXI接口中，读写地址和数据总线都是分开的，其具体读写架构如图2-3、图2-4所示。

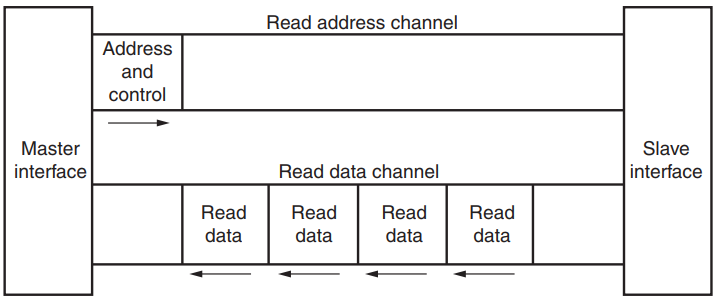


图2-3 AXI协议读架构

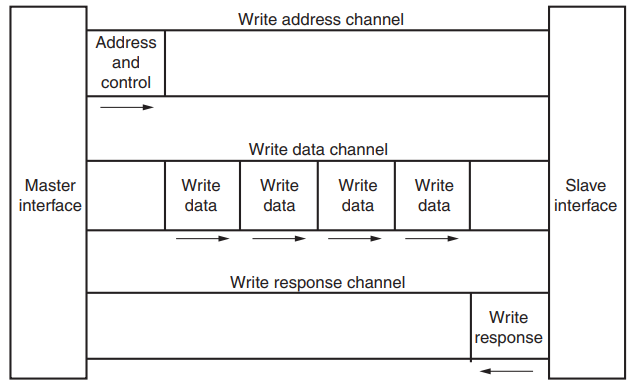


图2-4 AXI协议写架构

AXI协议采用了VALID/READY信号的握手机制来进行数据传输。在这种机制中，发送方通过VALID信号来确认发送地址、控制信号和数据是否具备有效性。而接收方则通过READY信号来向发送方表明它已经准备好接收信息。此外，AXI对于读写操作的控制信号也设定了特定的规则和限制：当VALID信号（AxVALID或xVALID）被激活时，它必须保持激活状态，直到对应的READY信号（AxREADY或xREADY）也被激活。AXI接口发送的VALID信号不应依赖于接收端的READY信号。然而，READY信号可以依赖于VALID信号的状态。写响应应该总是发生在相关写事务的最后一次数据传输完成之后。读取的数据应该总是跟随其对应的地址传输之后。从接口必须在ARVALID和ARREADY信号都接收到有效，即被激活之后，才能激活RVALID信号，以表明相关的数据已经准备好被读取。

AXI协议因其独特的体系结构特点，能够使采用这种协议的系统在获得较高性能的同时，减少SoC面积，降低其功耗。AXI协议本身采用一种单向通道体系结构，在这个结构中，片上的信息流只以单方向传输，这种结构能够降低传输时延。此外，AXI协议还规定了进入和退出低功耗状态时必须遵循的握手过程。这个协议规定了如何通知功能系统进入低功耗模式，应该在何时关闭或中断时钟，何时开启时钟，如何退出低功耗模式等行为逻辑。这使得系统中所有IP模块在进行功耗控制的设计时有可靠的依据，更容易集成在统一的系统中。AXI具有独立的读写数据与读写地址通道。通过将数据与地址通道分开，AXI协议允许对每一个通道单独进行修改与优化。这样可以根据开发的需要控制时序通道，将时钟频率提到最高，并将延时降到最低。AXI协议也为系统接口的开发提供更强的灵活性，AXI协议使得采用其的接口具有对称的主从接口。这种对称接口的特点使得无论在点对点或在多层次系统中，都能十分方便地使用AXI技术。

2.4 DMA

直接存储器访问（DMA）是一种允许系统中的数据在不同内存区域或内存与外设之间直接进行传输的技术，这种技术能够显著提高数据的传输效率。这种传输过程由DMA控制器独立于CPU进行管理，从而释放CPU资源以提高整体系统效率。具体来说，DMA传输机制不依赖CPU的直接介入，也不需要像中断处理那样需要暂停当前任务和恢复任务状态，而是通过硬件直接建立不同内存区域，或者内存与外存之间的数据传输路径。

在硬件设计领域，DMA是一种理想的数据传输解决方案，这种方案使得数据能够按照开发的需求快速写入指定的区域，尤其适用于大量数据的快速传输场景。在嵌入式系统中，DMA常用于实现外设与内存或不同内存区域之间的数据流动。一个标准的DMA系统通常包括：DMA控制器，它是管理传输任务的核心硬件，负责控制传输的行为逻辑；DMA请求线，用于外设向控制器发起传输请求；DMA确认线，用于控制器向外设确认请求接收；DMA通道，每个通道都可以连接到专用的硬件请求或支持软件触发的请求。DMA通道能够直接与特定的硬件请求模块相连，并且支持软件配置的触发机制。

在一个DMA模块中，可以通过软件设置不同请求之间的优先级，共有四个级别：非常高、高、中等和低。当多个请求优先级相同时，硬件会按照预设的顺序（如请求0优先于请求1）来决定优先处理哪个请求。此外，DMA支持不同宽度的数据传输，如字节、半字和全字，以及模拟数据打包和解包的过程。为了保证传输的准确性，源地址和目标地址都需要根据传输宽度进行对齐。

DMA支持一种循环的缓冲器管理机制。这种机制中，在DMA的每个通道中都有DMA半传输、DMA传输完成和DMA传输出错这三个特殊的事件标志。这3个事件标志通过逻辑或成为一个单独的中断请求信号。

直接存储器访问（DMA）是一种高效的数据传输机制，它允许在内存之间、内存与外设之间，或外设与内存之间直接传输数据。在DMA操作中，可以选择多种存储和外设选项作为数据的来源或目的地，包括闪存、静态随机存取存储器（SRAM）、外设的SRAM、高级外设总线（APB1、APB2）和高级高性能总线（AHB）外设。DMA传输的数据量可以达到最大65535（即十六进制的0xFFFF）。

DMA传输的完整过程可以大致划分为四个步骤：DMA请求、DMA响应、数据传输和传输结束。DMA的传输原理如图2-5所示，这个过程可以通过以下步骤描述：初始化，CPU初始化总线控制器，设置工作内存空间，并读取DMA控制器（DMAC）的寄存器信息以确定其状态。请求，I/O设备向DMAC发出DMA请求。DMAC接收到请求后，向CPU发送总线保持信号。授权，CPU完成当前总线周期后，发出总线保持确认信号，授权DMAC进行数据传输。传输，DMAC获得总线控制权后，向I/O设备发送DMA响应信号，开始数据传输。DMAC从源地址读取数据到内部缓存，然后将数据写入目的地址。结束，数据传输完成后，DMAC向CPU发送结束信号，释放总线控制权，使CPU可以继续其它操作。

DMA传输的效率非常高，因为它只需要一个DMA周期来完成，这相当于一个总线读/写周期。这种快速传输能力特别适合于需要高速传输大量数据的外设。

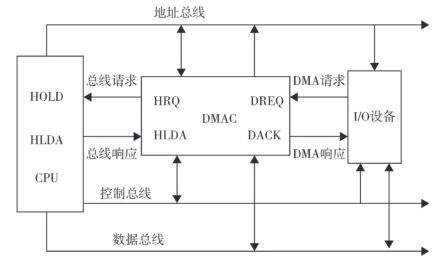


图2-5 DMA传输过程

第3章 结构设计

3.1 FPGA与CPU通信

启发于系统调用，本课题给出了一种近似于系统调用的函数调用通信机制，来定义FPGA与CPU间的通信架构，其具体结构如图3-1。在这种架构中，FPGA被视为CPU的一个外设进行处理，当CPU调用FPGA计算时，可以等效于陷入内核态的状态，将计算移交给FPGA。

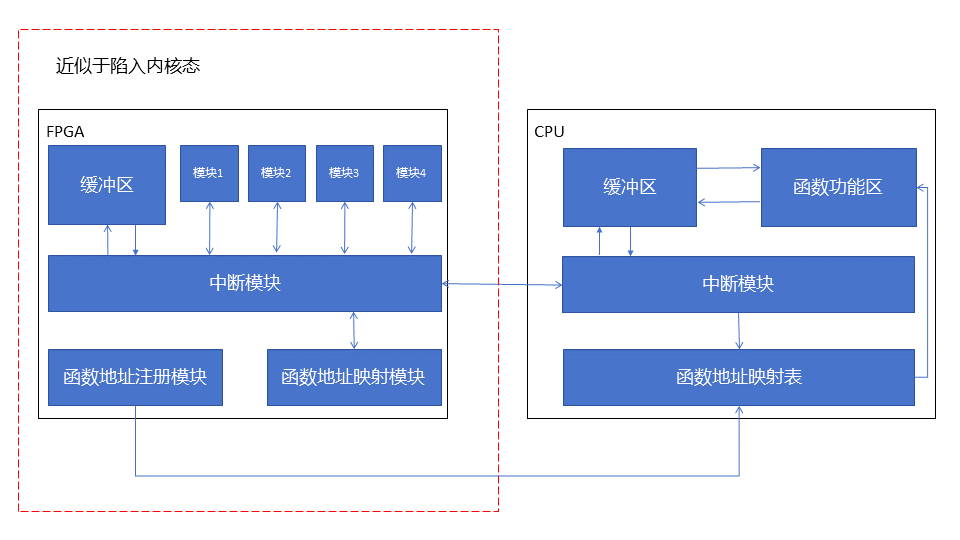


图3-1 总体设计结构

3.1.1传统系统调用

在传统系统调用中，具有内核函数和普通函数两种函数形式。实际上，内核函数与普通函数形式上没有什么区别。但是内核函数是在内核实现的，因此需要满足一些特定的内核编程的要求。系统调用是用户进程进入系统内核的接口。它本身并非内核函数，但它是通过使用内核函数来实现的。进入内核态后，不同的系统调用会分别寻找其相应的内核函数，这些内核函数即它所对应系统调用的“服务例程”。比如系统调用getpid()实际调用的是服务例程sys\_getpid()，也就是说，系统调用getpid()是服务例程sys\_getpid()的“封装例程”。计算机的各种硬件资源数目是有限的，为了确保资源使用的安全性，用户进程是不允许直接对其进行操作的，所有对这些资源的访问都必须由操作系统控制。为此操作系统为用户态运行的进程与其所需的硬件设备之间的交互提供了一组接口，用户态进程只有使用这个接口才能对操作系统内的硬件资源进行使用，这组接口就是所谓的系统调用。

系统调用本质上是一种特殊的函数调用，这个函数调用发生在用户态，它允许用户程序请求操作系统的内核服务。在进行系统调用时，用户程序会提供一个标识符，即系统调用号，这个号码使得内核能够识别并执行相应的系统函数。在LINUX操作系统中，每个系统调用都有一个独属于它自己的编号，即系统调用号。操作系统通过系统调用号来识别不同的系统调用，并为所有的系统调用专门储备了一个用该编号作为索引的系统调用表。这个表是由sys\_call\_table数组构成的，这个数组中包含了指向各个系统调用处理程序的函数指针。这个数组有NR\_syscalls个元素，每个元素对应一个系统调用的服务程序，通过系统调用号作为索引，内核可以找到并执行正确的服务程序。

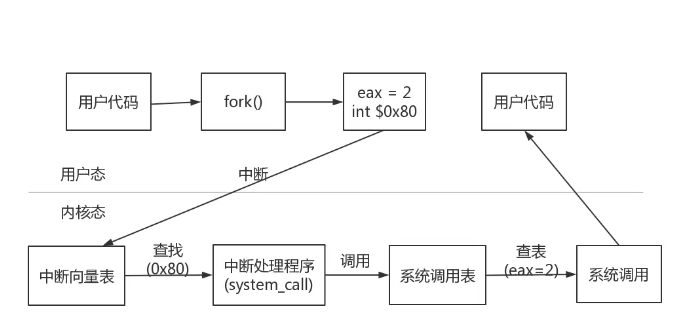


图3-2 系统调用流程

3.1.2 SoC上函数调用

由传统的系统调用方式可以得到SoC上实现函数调用机制的启发，在CPU与FPGA之间以近似系统调用的方式进行函数调用的数据通信。与传统系统调用相似，将数据的发送作为一个单独的函数。正常的函数调用在操作系统中是通过转为系统调用实现的。这里可以采用类似的思想，当需要进行一个函数调用进行数据的计算时，我们对于数据从CPU提取然后发往FPGA也需定义一个专门的函数，这个函数就是我们实现的函数调用之一。在传统系统调用中，参数是通过寄存器来实现的，在这里我们可将需发送的数据写入寄存器中，然后系统（函数）调用从这些寄存器中获得需要的数据信息然后进行其工作。这样我们就得到了一个通用的系统调用，这个调用的功能是从CPU寄存器中取值然后将其发给FPGA特定区域。这里的寄存器，在CPU内可以通过定义内存的特定区域来进行实现，即将内存中取一段特定的地址空间视为发送数据的存储空间，将待发送的数据存在这个地址空间中。

3.2 函数模块

在FPGA内部数据的运算多用逻辑门电路的形式，因而可以得到极快的运算效率，但也会产生一些问题：如何对呈字节流的数据进行处理，如何将数据存入一定的区域然后以特定的时序进行输出。而采用先入先出队列（FIFO）就能够很好的解决这个问题。在此基础之上给出函数模块的设计架构，如图3-3所示。

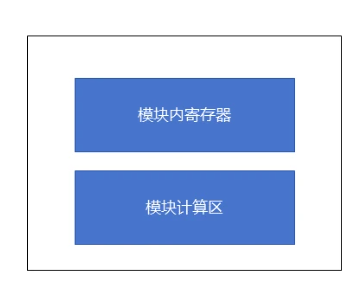


图3-3 函数模块设计结构

FIFO即First Input First Output，它是一种广泛运用于各种芯片设计中的存储器结构。FIFO这种结构由存储单元阵列或队列构成，第一个被写入其中的数据也是第一个从这个结构中读出的数据，这种独特的结构很好的保证了数据传输处理的有序性。同时，通过对FIFO进行设计，可以满足不同的芯片设计需求。FIFO有着调整数据输入输出速率的作用，当数据的输入速率与数据的输出速率不同时，FIFO可以起着作为缓冲的临时存储区的作用，使得数据流的输入输出速率得到匹配。CPU可将数据先写入FIFO中，然后做其他工作，而设备也能很方便地从FIFO中异步读取数据。FIFO也可用于不同时钟域之间，为不同时钟域的输入输出提供同步。在实际的使用中，数据会面临不得不从一个时钟域进入另一个时钟域的情况，此时FIFO则不仅承担着做为临时数据存储单元的作用，也起着数据同步的作用。此外，FIFO的位宽是可以调整的，输入数据路径与输出数据路径之间数据位宽不匹配时，FIFO可以调整其位宽，用于数据位宽调整电路。

在xilinx官方文档中，其FIFO IP核提供了两种对外的接口：Native接口和AXI4接口。Native接口为FIFO generator的默认接口。它具有FIFO的基本功能，其信号主要包括读写使能，输入输出，空满等，具体接口结构如图3-4所示。AXI4接口为Native接口的再封装，可以实现AXI4，AXI3，AXI4-Lite和AXI4-Streaming等协议。

因xilinx官方FIFO的这种特性，我们可以在函数模块中使用其存放数据，使其做到以下几方面的功能：缓存连续数据流，这种功能确保数据在传输到FPGA和存储设备时不会遗失；集中处理数据，这种功能可以减少对总线的操作次数，从而降低CPU的工作强度；系统的DMA功能能够加快数据传输的速度。这一机制非常关键，因为如果不通过DMA来传输数据，不仅无法满足传输效率的要求，还会导致CPU负荷过重，进而无法有效地存储数据。

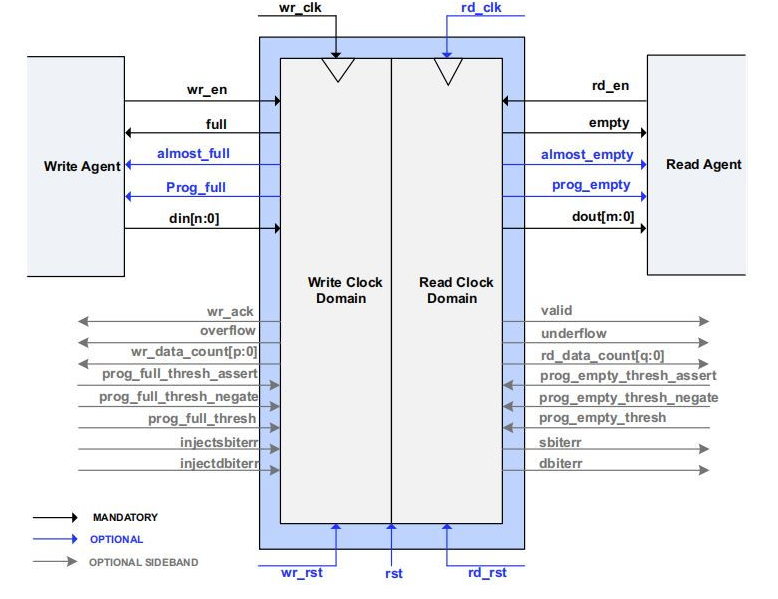


图3-4 FIFO结构

在如上文使用FIFO定义了函数模块中数据缓存的部分后，我们即按照图3-3的结构实现了函数模块设计的初步框架。

函数模块的初步结构已在上文定义，但是其具体结构以及通信逻辑我们仍需进行设计。图3-4的FIFO的具体结构则给出了设计的启发。FIFO IP可以采用不同的资源形式达到不同的功能：Block RAM（BRAM）资源，这是FPGA底层内嵌的存储资源，不仅可以实现FIFO，同时还可以实现RAM、ROM；（Distributed RAM）分布式RAM，本质是底层逻辑资源LUT；（Shift Register）移位寄存器，本质仍是底层逻辑资源LUT；（Built-in FIFO）嵌入式FIFO，相当于BRAM资源外部封装了FIFO相关的接口。如图3-5所示，xilinx官方将不同资源类型的FIFO支持的功能特征给出了详细说明。比如图3-5中的Built-in FIF资源组成的异步FIFO，可以实2、3、4、5，但是无法实现功能1----不同的读、写数据位宽。

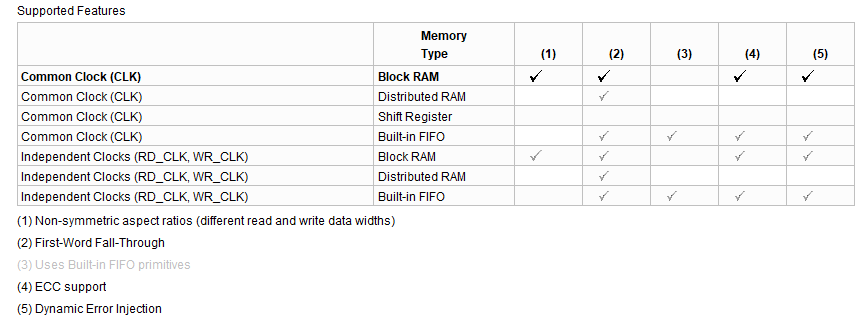


图3-5 FIFO功能特征图

故而我们可以在FIFO以及运算模块外再进行一层封装，将其视为一个带计算功能的FIFO来进行改造，这个封装和FIFO相似也具有读接口与写接口，并具有类似的一系列控制信号来辅助其实现功能。FIFO的读写既可以同步也可以异步，想要实现异步接口我们也可以参考其架构进行设计。很自然地，我们可以得到如图3-6所示的初步设计。

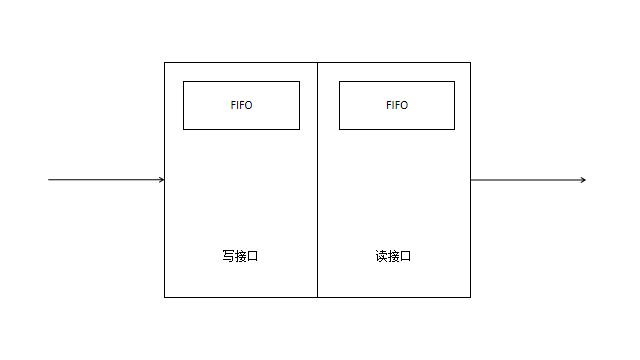


图3-6 函数模块初步结构

但是考虑到FPGA内部资源的有限程度以及模块的去复杂化，我们可以将结构进行简化，即将读写接口处的两个FIFO合为一个，这样可以一定程度上减少时序的复杂性，并节约资源空间。进而我们可以得到如图3-7所示的宏观设计结构。

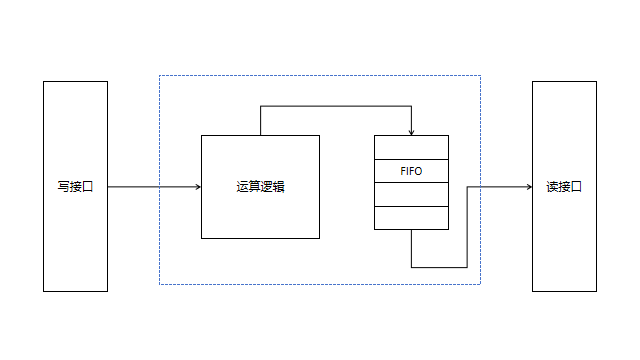


图3-7 函数模块结构

在图3-7函数模块的设计中，蓝色虚线所框选部分为图3-3函数模块设计的映射部分，而读写接口则与FIFO IP核采用类似的接口，以便于后续总体系统设计。

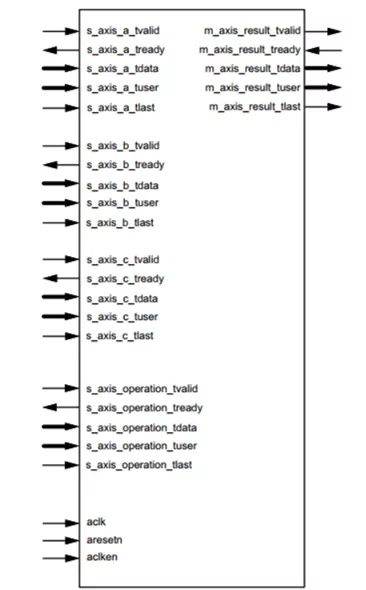


图3-8 浮点数IP结构

写接口在接收到一串数据流后，按照其自身行为逻辑将这串数据流进行解析，然后送往对应的运算逻辑中，运算逻辑将运算完后的数据送入一个近似于寄存器的FIFO中，然后读接口根据其自身逻辑从FIFO中取出数据转化为一串数据流的形式送入其他模块。至此就完成了函数模块外部封装及通信机制的设计，接下来将逐一描述函数模块内部运算逻辑的具体设计部分。

3.2.1 矩阵加法模块

在C语言的程序中，我们对于数据有float,double这样的类型定义，但在Verilog语言中则没有这样的数据类型，Verilog语言中的数据都由进制数字来进行表示，如1b`1,8d`10这样的表示形式。

若在FPGA内想要进行小数运算，可以采用xilinx官方提供的Floating-point IP核，该IP的具体结构如图3-8所示。这个IP核内部包含了基本的整数和浮点数的互换操作，以及对应的浮点数运算操作。但在小数运算前需要将其转化为对应的浮点数形式。

在寄存器中小数有定点数与浮点数两种形式，定点与浮点即表示小数点的位置是否固定。定点数的小数点位置固定，对于计算器件来说计算压力较小，但其数值的动态范围没有浮点数大。xilinx官方默认Floating-point IP核内采用浮点数的形式。

在该IP中，可以有至多三个端口输入数据，同时可对操作符进行输入来控制该IP具体行为逻辑。考虑到时序的复杂性，在矩阵加法模块中不采用其操作符输入端口，而仅适用其两个数据输入端来进行数据的运算。与FIFO IP相似，该IP也具有一系列控制信号来控制数据的传输，其中valid信号表示数据是否有效，last信号则表示数据是否终止，ready信号则表示该端口能否接受数据的输入或输出，user则用于标识数据的所属，与运算对应的，其具有一个结果输出端口，其信号具体含义同数据输入端口相似。

3.2.2 数组乘法模块

与矩阵加法模块相似，在数组乘法模块中也使用了Floating-point IP核，其同样具有两个数据输入端口，并省去了操作符输入端口。但考虑到加法与乘法的时序不同，故两个模块的实际运行时序有所区别，考虑到模块结构设计的规范性，通用性，可移植性。这里采用一种新的解决方法，即将输入的两个数据在写接口中同时输入。具体输入行为如图3-9所示。

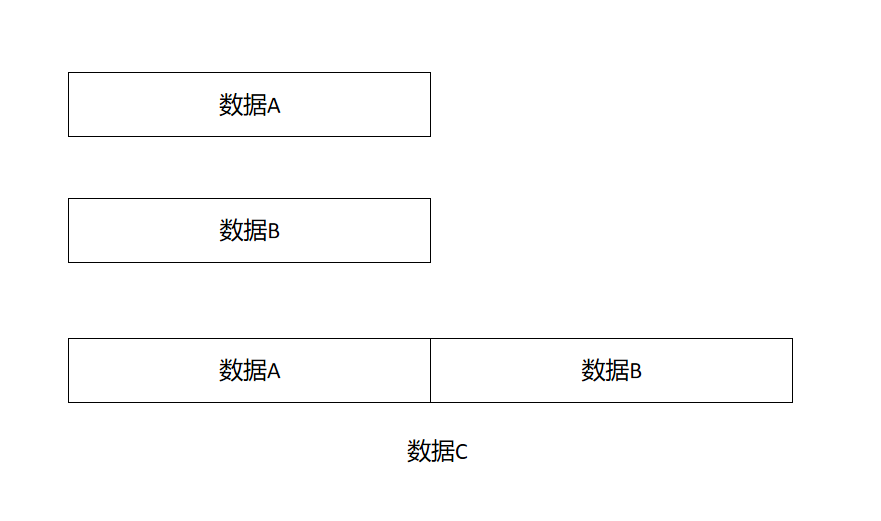


图3-9 数据处理

对于待运算的数据A与数据B，将其进行位扩展后合并为位宽等于A的位宽加上B的位宽的数据C，然后将数据C传入写接口，这样只需在写接口中对于传入数据进行解析即可，减少了时序的复杂程度以及结构处理不同函数功能的复杂程度，对结构进行了设计上的简化。传入的数据经过Floating-point IP运算后位宽正常，将其送入读接口进行输出即可。

3.2.3 全并行排序模块

与数组乘法模块和矩阵加法模块相似，全并行排序也只需对其运算逻辑部分进行相应修改。这里采用并行全比较算法来编写相应运算逻辑。传统的排序算法是对待排序数据两两之间分别进行比较，这种比较方式非常费时，并行全比较算法则是将待排序数据中的数据同时进行比较，这种方式非常迅速，但会消耗大量的比较器资源。其具体流程如下：第一个时钟周期，将其中一个数据和其他数据同时进行比较，得到每个数据的比较结果。第二个时钟周期，将每个数据和其他数据比较后的结果进行累加。第三个时钟周期，将每个数据根据自己的比较得分赋值给新的数组。根据上述流程，并行全比较排序可以在三个时钟周期内就完成排序任务。在采用图3-7函数模块总体结构的基础上只需采用与图3-9相似的数据处理操作，将多个待排序数据合并后输入写接口即可完成全并行排序模块的设计。

3.3 多通道DMA

xilinx公司在ZYNQ中提供了两种不同的DMA，一种是集成在PS中的硬核DMA，另一种是在PL中使用的软核AXI DMA IP。本节的系统结构设计采用AXI DMA IP来实现。AXI DMA IP提供了内存和AXI4-Stream接口的目标外设之间的高带宽直接内存访问，可以将CPU从数据传输任务中解放出来。

3.3.1 DMA结构

AXI DMA IP具有三种模式：Direct Register Mode (Simple DMA)，Scatter/Gather Mode，Cyclic DMA Mode，其具体结构框图如图3-10。

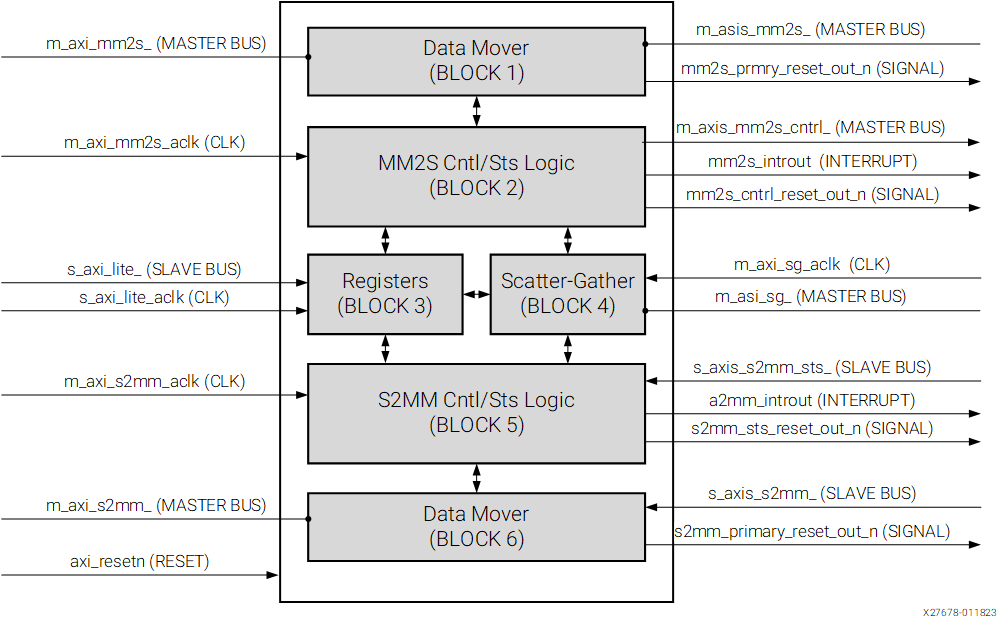


图3-10 DMA结构

由图3-10可知，DMA结构通过s\_axi\_lite接口访问和初始化模块，读取模块状态，并控制和管理模块内寄存器。通过MM2S接口完成MemoryMap to Stream的转换，即将存储器映射转换到AXI4-Stream接口的转换。与之类似的，S2MM接口则负责完成Stream to MemoryMap的转换，即AXI4-Stream接口转换到存储器映射的转换。在这三种模式中，m\_axis\_mm2s\_cntrl，m\_axi\_sg，s\_axis\_s2mm\_sts等一系列接口，只会出现在S/G模式下，也就是说在简单DMA模式下，可以省去部分复杂的数据流传输控制信息。

在单通道的DMA中，我们考虑使用Simple DMA模式。这种模式允许在MM2S和S2MM通道上进行基本的DMA传输，并且只需占用少量的FPGA资源。启动DMA传输的过程包括设置DMACR、源地址、目的地址和长度寄存器。传输结束后，如果启用了中断，DMASR寄存器的相关通道位将被激活，产生中断信号。

对于MM2S通道，启动流程如下：将MM2S\_DMACR的运行/停止位设为1以启动通道。DMASR的停止位应该是低电平，表示通道正在运行。通过将MM2S\_DMACR的IOC\_IrqEn和Err\_IrqEn位设为1，可以启用中断。在Direct Register模式下，不使用延迟中断、延迟计数和阈值计数。向MM2S\_SA寄存器写入有效的源地址。如果AXI DMA的地址空间超过32位，还需要设置MM2S\_SA MSB寄存器。如果没有启用数据重新对齐，地址必须对齐，否则会导致不确定的结果。地址对齐取决于数据流的宽度。

S2MM通道的启动流程与MM2S通道相似：首先启动S2MM通道。如果需要，可以启用中断。向S2MM\_SA寄存器写入有效的源地址，如果没有启用DRE，需要确保地址对齐。向LENGTH寄存器写入传输的字节数，该值不能为零，它决定了从存储器映射到Stream流的数据量。S2MM\_LENGTH寄存器应该最后写入。

3.3.2 多通道DMA架构

在3.1.1节中，单通道DMA采用的Simple DMA模式，倘若想要使用多通道DMA，则需要启动S/G模式，来配置管理通道的数据流。

在S/G模式中，AXI DMA的操作依赖于一个内存中的数据结构，用于记录DMA操作的列表。这些操作指令被安排成一个称为描述符链的序列。每个描述符包含一个指针，指向链中下一个待处理的描述符，而链的最后一个描述符则指回链的起始。

S/G模式支持使用多个描述符来描述单个数据包。这种方式的一个常见应用是允许头部数据从内存的一个位置存取，而有效载荷数据则从另一个位置存取，从而提升数据处理的效率。数据包的开始和结束由帧起始位(TXSOF)和帧结束位(TXEOF)标记，分别位于描述符链的首尾。当DMA遇到设置了TXSOF的描述符时，会触发数据包的开始，然后继续处理链中的后续描述符，直到遇到设置了TXEOF的描述符为止。在接收(S2MM)通道中，AXI DMA使用RXSOF标记来标识数据包的开始，并将其与相应的数据缓冲区关联。如果接收到的数据包大小超出了描述符指定的范围，则会使用下一个描述符的缓冲区来存储剩余的数据。这个过程会持续进行，直到数据包完全接收。当数据包接收结束时，正在处理的描述符会被标记为RXEOF=1，告知软件该缓冲区包含了数据包的末尾。每个描述符都有一个状态字段，记录了实际传输的字节数。软件可以通过从RXSOF标记的描述符开始，遍历到RXEOF标记的描述符，来计算接收到的数据包的总字节数。Scatter Gather模式通过获取额外的描述符并存储它们来提高DMA的性能。在S/G模式下，配置DMA操作首先需要设置控制寄存器和描述符指针。基本上，这涉及将传输参数（称为BD，即Buffer Descriptor）存储到内存中，并通过SG接口加载和更新BD的状态，以执行对特定内存位置的数据读写操作。

S/G模式下的DMA操作与Simple DMA模式有所不同。MM2S通道的DMA操作通过以下步骤建立和启动：首先，将起始描述符的地址写入当前描述符寄存器。如果AXI DMA配置了大于32位的地址空间，则还需要对当前描述符的MSB 32位进行设置。然后，设置运行/停止位为1 (MM2S\_DMACR.RS=1)以启动MM2S通道。确保停止位(DMASR.Halted)未被激活，以表示MM2S通道正在运行。可以通过设置MM2S\_DMACR.IOC\_IrqEn和MM2S\_DMACR.Err\_IrqEn为1来启用中断。最后，向尾部描述符寄存器写入有效地址，如果AXI DMA配置了大于32位的地址空间，则还需要对尾部描述符的MSB 32位进行设置。写入尾描述符寄存器会触发DMA开始从内存中获取描述符，并在多通道配置下，当数据包到达S2MM通道时，开始处理描述符，将数据从内存中读取并输出到MM2S流通道。

S2MM通道的DMA操作也通过类似的步骤建立和启动：将起始描述符的地址写入当前描述符寄存器，并对MSB 32位进行设置（如果适用）。设置运行/停止位为1 (S2MM\_DMACR.RS=1)以启动S2MM通道，并确保停止位(DMASR.Halted)未被激活。启用中断，并向尾部描述符寄存器写入有效地址。这将触发DMA开始从内存中获取描述符，并处理接收到的数据，将其写入内存。

第4章 结构实现

4.1 硬件布线

在第三章中，我们介绍了几个主要模块的结构设计以及整体的结构框架。其中所有的模块间的通讯都要采用AXI协议，进而我们需要对整个接口系统的AXI总线进行管理以及布线。自然地，我们可以采用xilinx官方提供的AXI interconnect IP来实现总线通信，管理，以及主从机之间的交互。AXI interconnect用于对AXI总线进行管理。这个IP支持多个主机采用AXI总线访问单个从机，或者一个主机通过AXI总线访问多个从机。采用这个IP可以真正实现总线通信。该IP核最多可以支持16个主设备，16个从设备，如果需要更多的主机或者从机接口，可以在一个设备处使用多个该IP进行布线。

AXI interconnect IP多用于Block Design设计。在vivado的Block Design中通常会简化IP的用法，但是对于连线较为复杂的系统其可以完成自动布线，对于AXI协议这种信号较多的通讯协议，在Block Design中也能很方便的进行结构设计与布线。AXI interconnect IP有着多种不同的使用方式：单个主机访问多个从机（1-N），多个主机访问单个从机（N-1），多个主机访问多个从机（M-N），这里我们主要使用的是其1-N与N-1的通信模式，其具体通信结构如图4-1，4-2。

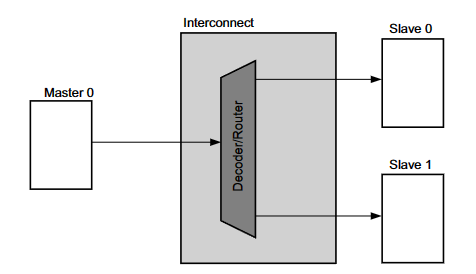


图4-1 单主机对多从机（1-N）

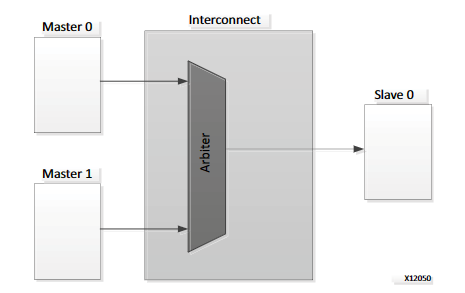


图4-2 多主机对单从机（N-1）

单个主机访问多个从机的原理是采用为从机的AXI总线分配不同的地址，这样主机通过访问不同的地址分区来访问不同的从机。多个主机读写一个从机是通过主机的ID号不同来实现的，每个主机具有不同的ID号，内部通过读写的时候地址会把这个主机号加上，代表哪个主机进行访问，传回的数据就传到相应的主机。

4.1.1 单通道DMA布线

在IP的指导手册中有图4-3这样的一个控制AXI DMA IP的设计系统。我们可以以这个系统为参考，来进行我们的单通道DMA布线设计。

在图中我们可以看到，对于处理器（microblaze），只需要使用少量的控制指令，即可完成高速的多数据传输。处理器（microblaze）通过interconnect互联模块连接到AXI4-Lite接口，进行IP的寄存器配置。在AXI DMA IP的完成数据传输时，通过MM2S\_IntrOut，S2MM\_IntrOut指示数据传输完成，并发送给处理器进行进一步操作。在图中的系统AXI DMA IP使能了S/G模式，该存储器映射接口通过互联模块连接到DDR控制端口，ctrStrm（控制Stream）、StatusStrm（判断Stream状态）和SG R/W是在SG模式下使用的三个端口。剩下的MM2S和S2MM进行数据的交互和传输。但是在单通道的DMA中，S/G模式实际上并不是必须的设置，我们可以通过自己编写的接口处理来完成一部分需要的S/G模式的功能，进而我们可以在Block Design中得到如图4-4所示的设计。这个设计中，我们将第三章叙述的函数模块加入其中，并对各个官方提供的模块进行串联。

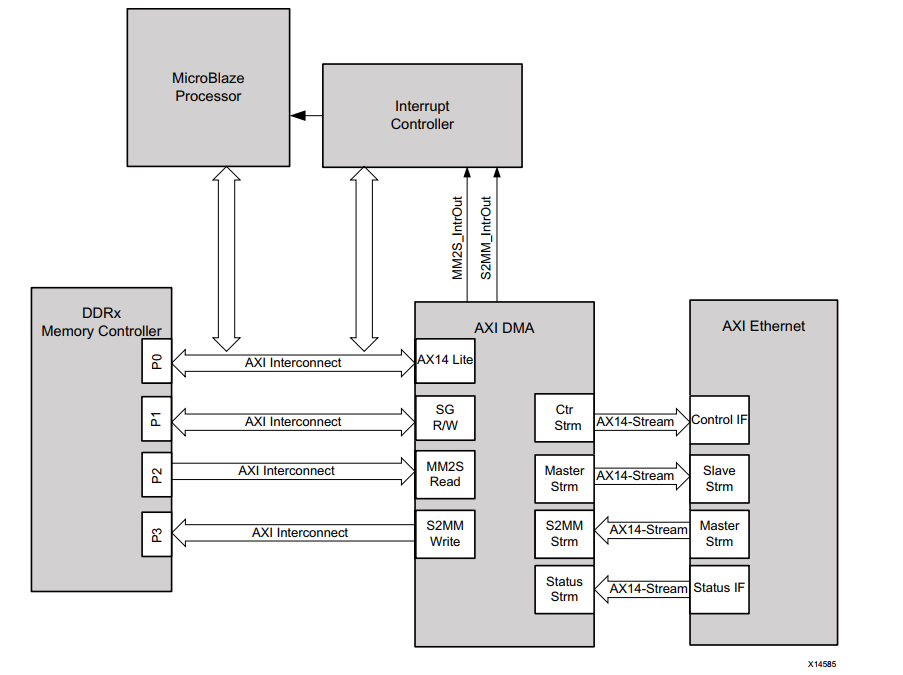


图4-3 AXI DMA设计系统

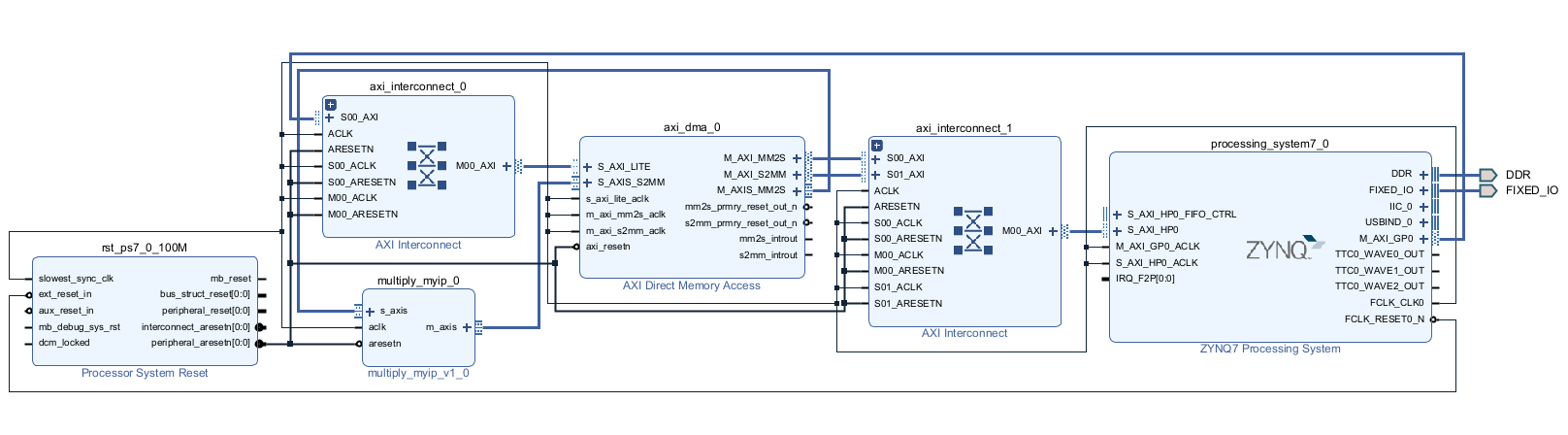


图4-4 单通道DMA Block Design设计

在这个系统的设计中，我们使用了第3章所叙述的系统架构，以及xilinx官方提供的模块，并在我们架构的基础上对其进行定义，然后通过AXI interconnect IP对主机与从机进行了串联管理。此外，还添加了我们使用的处理器内核以及与之相对应的时钟。各IP间的连线以及处理器的引脚则通过Block Design内置的自动布线功能完成，省去了复杂的连线处理过程。在这个设计中值得注意的一点是处理器内核的配置，本课题设计采用的开发板是Zynq-7000系列开发版中的Zedboard，其具有双核Cortex-A9 MPcore以及最新的28nm 7系列可编程逻辑的紧密集成。在处理器的配置上我们自然要与其相对应，图4-5给出了处理器PL的相应配置信息。

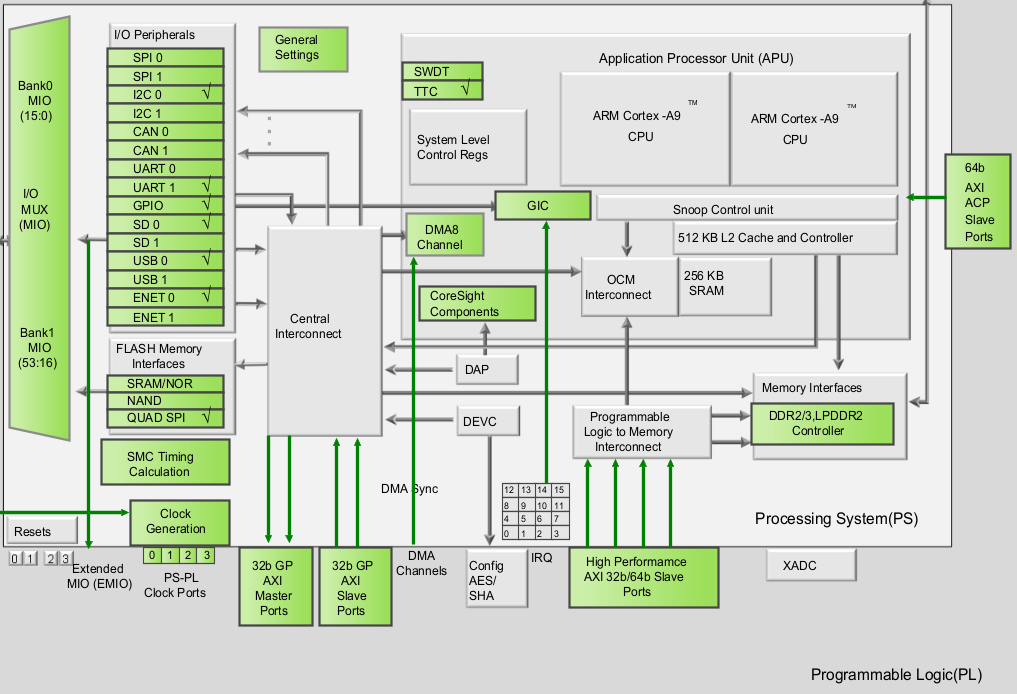


图4-5 PL配置

如图所示，在xilinx官方提供的Zedborad开发版的配置中，我们选择了我们需要的I/O端口来进行设置。并设置了其所需的中断处理端口。系统设计中其他IP的端口设置也与之类似。遵循Block Design的设计原则，我们对各IP的接口进行了对应的简化，只保留需要的接口进行连线。

4.1.2 多通道DMA布线

与单通道DMA不同，使用多通道DMA时必须对连线模块即AXI interconnect IP进行扩充，使其端口数目符合我们需要增加的模块数。除此之外，在DMA IP中我们也得使用S/G模式来实现对于不同输入通道的管理，进而，我们需要增加AXI4-Stream interconnect模块来实现对于数据流的处理。图4-6显示了AXI4-Stream interconnect模块的结构。

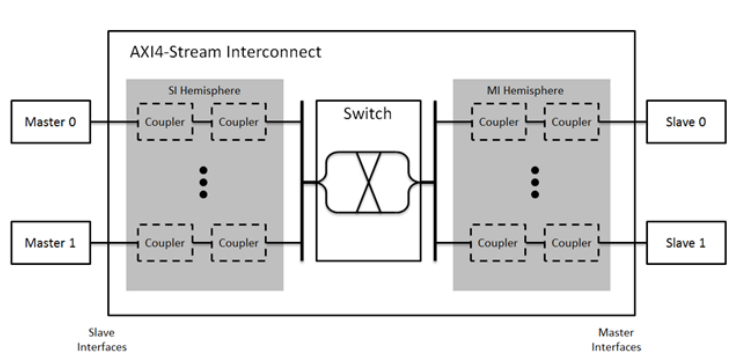


图4-6 AXI4-Stream interconnect模块

在AXI4-Stream互连体系中，AXI4-Stream交换机核心负责在从接口(SI)与主接口(MI)之间进行通信的路由。每条连接到交换机的SI或MI路径上，都可以部署一系列可选的AXI4-Stream基础设施核心（即耦合器），以实现多样的转换和缓冲操作。这些耦合器包括AXI4-Stream寄存器切片、数据FIFO、时钟转换器、数据宽度转换器以及协议转换器等。AXI4-Stream互连核心能够支持最多16个SI和16个MI。每个SI都与一个AXI4-Stream主设备相连，并接收来自该主设备的流式传输。每个MI则连接到一个从设备，并向该从设备发送流式传输。交换机核心位于中央，负责SI与MI之间传输的路由。在SI到交换机以及交换机到MI的路径上，可以配置一个或多个AXI4-Stream基础设施核心来完成各种转换和缓冲任务。交换机核心将AXI4-Stream互连核心分隔为与SI相关的功能单元（SI半球）和与MI相关的单元（MI半球）。

在引入AXI4-Stream interconnect模块后，我们的Block Design的布线方式也要随之进行对应的修改，图4-7给出了多通道DMA的Block Design。

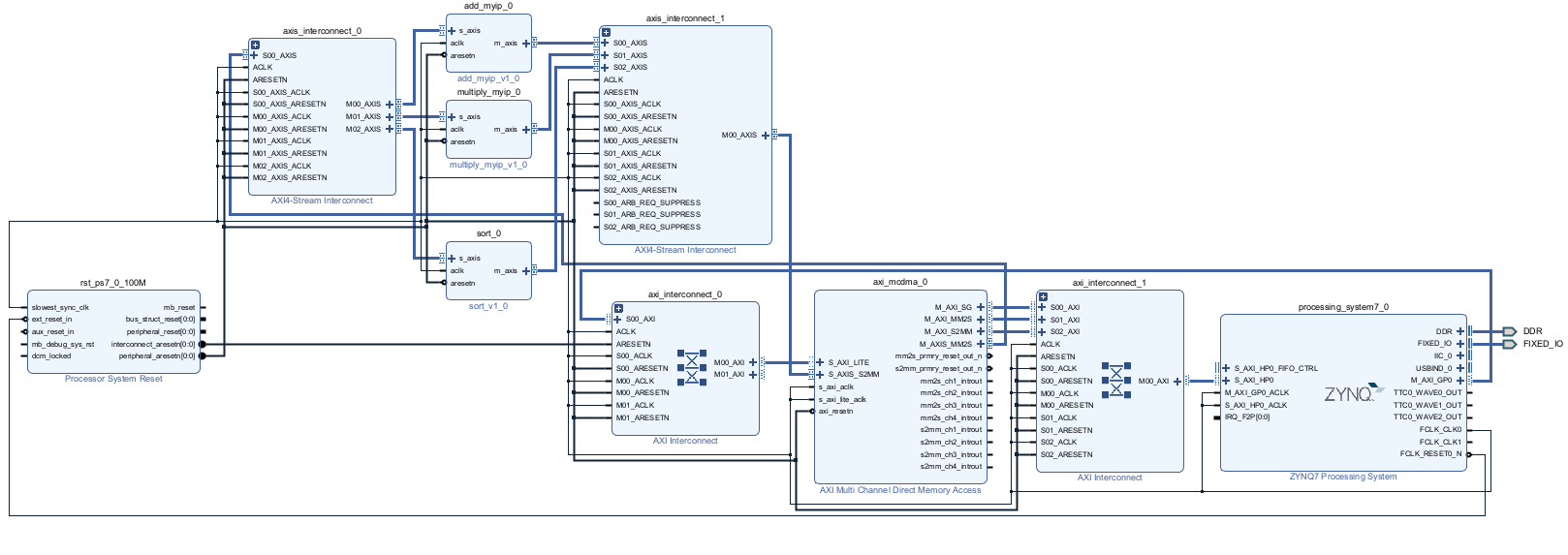


图4-7 多通道DMA Block Design设计

除端口数目及连接IP的修改，以及DMA IP SG模式的使用外，其余IP的配置与单通道DMA配置相似。

4.2 函数IP内部布线

在4.1节中，我们给出了使用各个IP模块后的Block Design设计，但这只是整体系统的布线，对于我们自定义函数IP内部的布线结构以及封装后IP的接口含义我们仍需进行阐述。

3.2.1 矩阵加法模块

矩阵加法模块的IP布线如图4-8所示。在参考FIFO IP的基础上完成了矩阵加法具体设计。其中FIFO generator模块被内置在add\_S\_AXIS设计，即写模块中。其读写模块都具有输入输出数据端口，且位宽符合结构设计。同时读写模块都具有valid信号，ready信号以便与其他模块IP连线通信交互，整个IP采用一个共同的时钟aclk，其控制由整体系统的时钟控制，并给出相对应的使能控制信号以控制IP的读、写、运算时序。同时具备rb信号以表示输入数据的一字节长度数据是否输入或输出IP。运算模块即Floating-point IP与FIFO generator模块相似，均内置在写接口中。

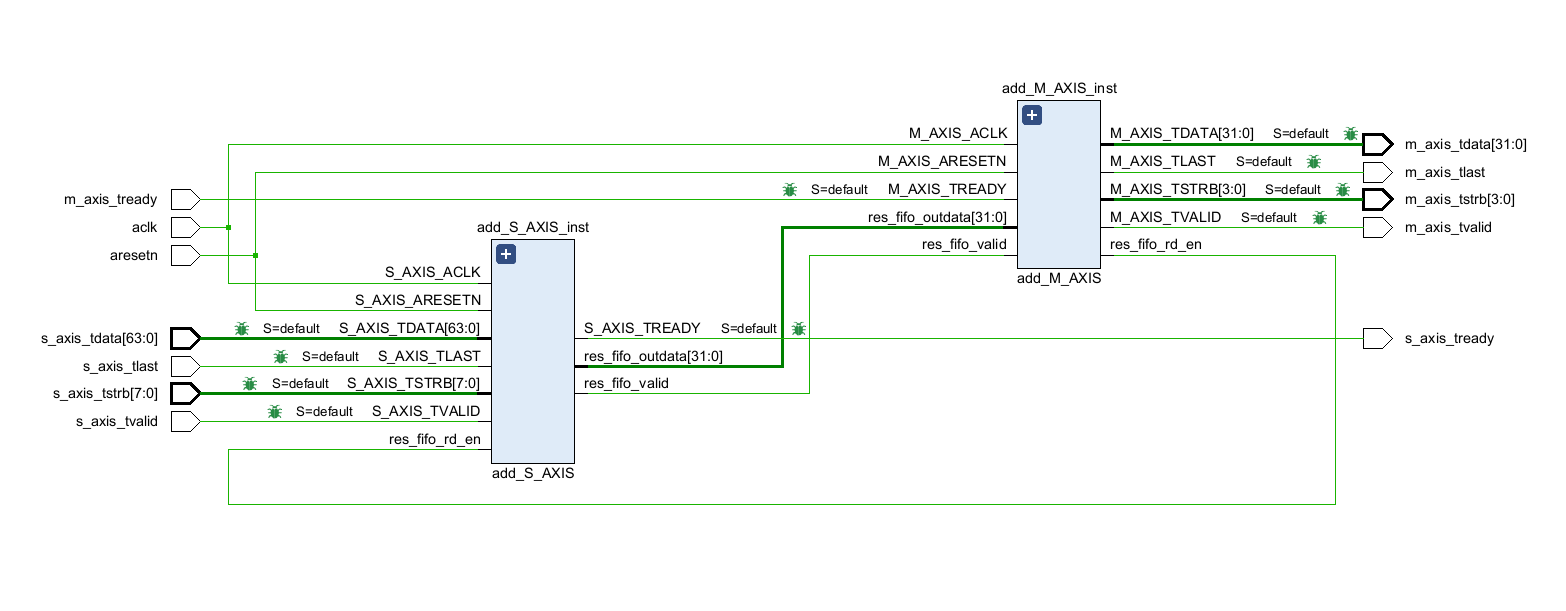


图4-8 矩阵加法模块布线

3.2.2 数组乘法模块

与矩阵加法模块类似，数组乘法模块也采用与之相同的布线方式，其具体布线如图4-9所示。具体接口与信号含义与矩阵加法相同，且其也采用系统时钟控制整个IP的方式。

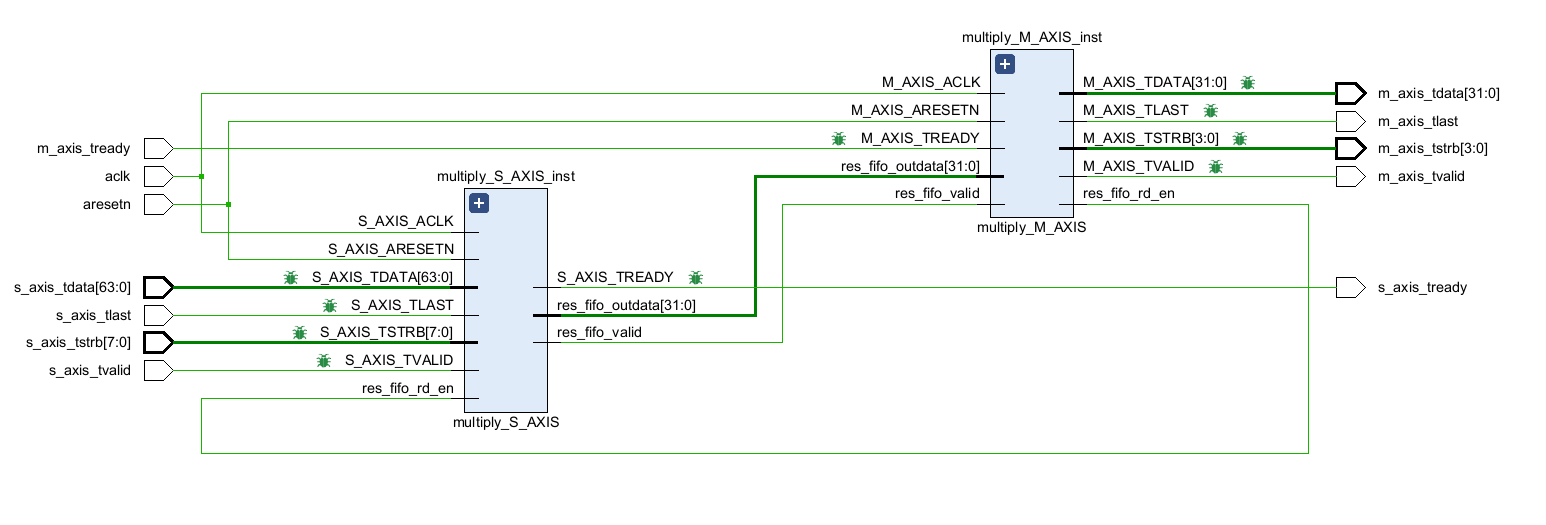


图4-9 数组乘法模块布线

3.2.3 全并行排序模块

排序模块布线如图4-10所示。其结构也与上述两模块相似，最大的区别在于读模块的位宽有了改变。

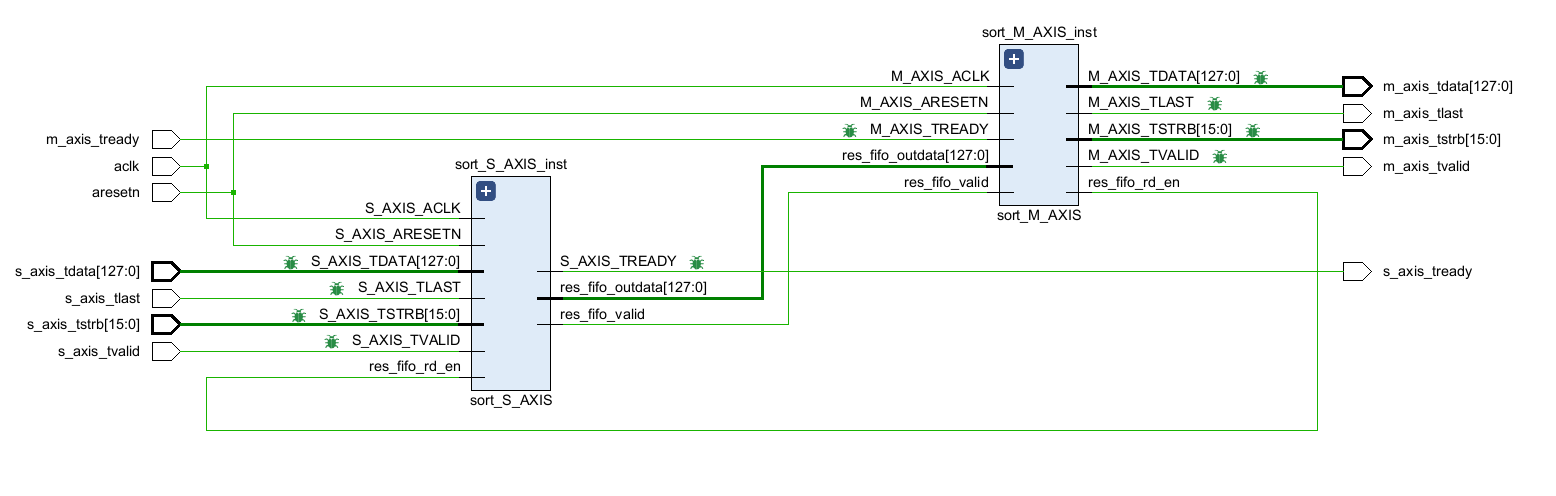


图4-10 排序模块布线

4.3 状态机实现

在实现整个系统的结构设计，Block Design以及实际布线之后，我们需要实现整个系统运行时的具体流程，而整个系统运行的流程由如图4-11的状态机的形式给出。

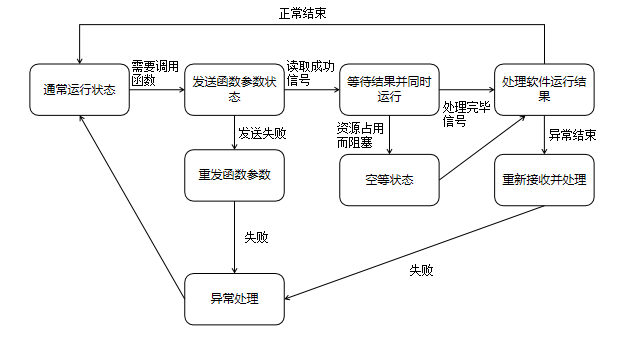


图4-11 系统运行时状态机

按流程图所示，系统中的CPU一般处于通常运行状态，当需要发送函数调用请求时进入发送函数参数状态，将所需的参数压入相应的参数栈中，如果接收到读取成功信号就等待结果，此时CPU可以处理其他事务，如果失败则重发对应参数。而在等待结果状态时若遇FPGA资源占用情况则陷入空等状态。当收到数据处理完毕信号时，进入结果处理状态，若异常则重新接收。同时，整个系统具有异常处理状态来处理一些特殊异常。

在定义整个系统的运行时状态机后，自然地，我们需要细化函数模块处理数据流通的状态机。在第三章设计中，我们对函数模块的定义是其读写接口分别有不同的处理状态机来分别接收和发送数据。考虑到数据的输入情况，我们很自然地可以设计出如图4-12的状态机。

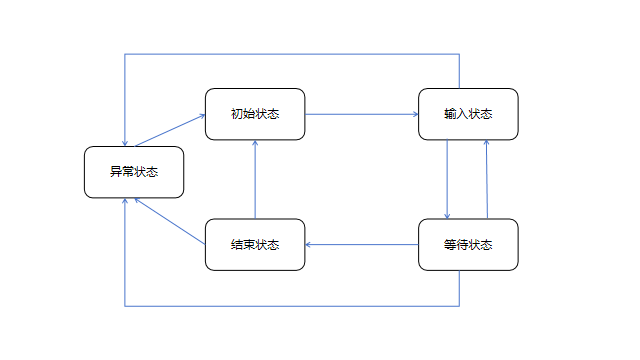


图4-12 写接口状态机

接口通常处于初始状态，当得到数据输入时进入输入状态，然后将数据送往运算模块，同时陷入等待状态，待运算模块将数据输入FIFO时继续返回输入状态将数据送往运算逻辑。待数据输入完后，再从等待状态进入结束状态，然后随着复位信号返回初始状态。每个状态都可能发生异常，因而设计异常状态来处理异常情况。如此，便完成了函数模块写接口的状态机实现。

与写模块相比，读模块的任务则相对简单，只需从FIFO中取数据然后输出即可，因而可以得到与图4-12所示类似的读接口状态机。

4.4 函数调用实现

与硬件设计不同，函数调用的实现是在软件端即vitis端实现的。函数是一组一起执行一个任务的语句，每个c程序都必须有一个main函数，程序员可以把代码划分到不同的函数当中去，在逻辑上，划分通常是根据每个函数执行一个特定的任务来进行的。与之类似的，我们可以将每个模块的功能写作一个单独的函数然后将其封装入一个统一的头文件中，当想要使用FPGA外设的定义好的函数调用功能时，我们只需引用相应的头文件即可得到对应函数调用的声明。其具体代码实现如图4-13所示。

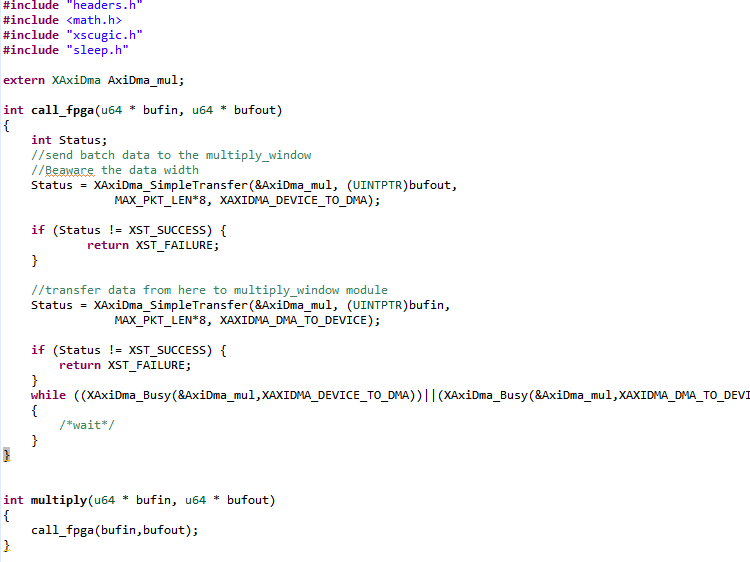


图4-13 函数调用示例

至此，我们便完成了整个接口系统的硬件端以及软件端的实现。

第5章 结果与分析

5.1 资源占用量

在将硬件布线后，我们能通过vivado自带的综合分析功能来得到其估计的资源占用量，表5-1,5-2分别给出了单通道DMA架构和多通道DMA架构的估计资源占用量。

表5-1 单通道DMA架构资源占用

|  |  |  |  |
| --- | --- | --- | --- |
| Resource | Utilization | Available | Utilization % |
| LUT | 4976 | 53200 | 9.35 |
| LUTRAM | 379 | 17400 | 2.18 |
| FF | 7749 | 106400 | 7.28 |
| BRAM | 4 | 140 | 2.86 |
| DSP | 2 | 220 | 0.91 |
| BUFG | 2 | 32 | 6.25 |

表5-2 多通道DMA架构资源占用

|  |  |  |  |
| --- | --- | --- | --- |
| Resource | Utilization | Available | Utilization % |
| LUT | 11284 | 53200 | 21.21 |
| LUTRAM | 525 | 17400 | 3.02 |
| FF | 16385 | 106400 | 15.40 |
| BRAM | 15 | 140 | 10.71 |
| DSP | 13 | 220 | 5.91 |
| BUFG | 3 | 32 | 9.38 |

如表所示，两种架构对于板上资源的估计占用量最高都不超过30%，可以认为该架构的设计具有很大的可扩展性，所定义函数模块数量可以达到一个较大的数值而不消耗完板上资源。同时较少的资源占用量也说明了所设计架构的优势，并具有很大的修改空间来修改相应接口架构使其功能更加多样。

5.2 仿真波形

在本节中，我们将分别对函数模块内部的运算逻辑进行时序仿真，并对仿真得到的波形图进行分析，验证其计算行为的正确性。

5.2.1 数组乘法

在本次仿真中，我们将乘数a，b均使用32位浮点数的形式表示，其十进制值均为100，仿真得到的波形应为输入数据a，b均为42c80000，输出数据为461c4000。其具体波形如图5-1所示。

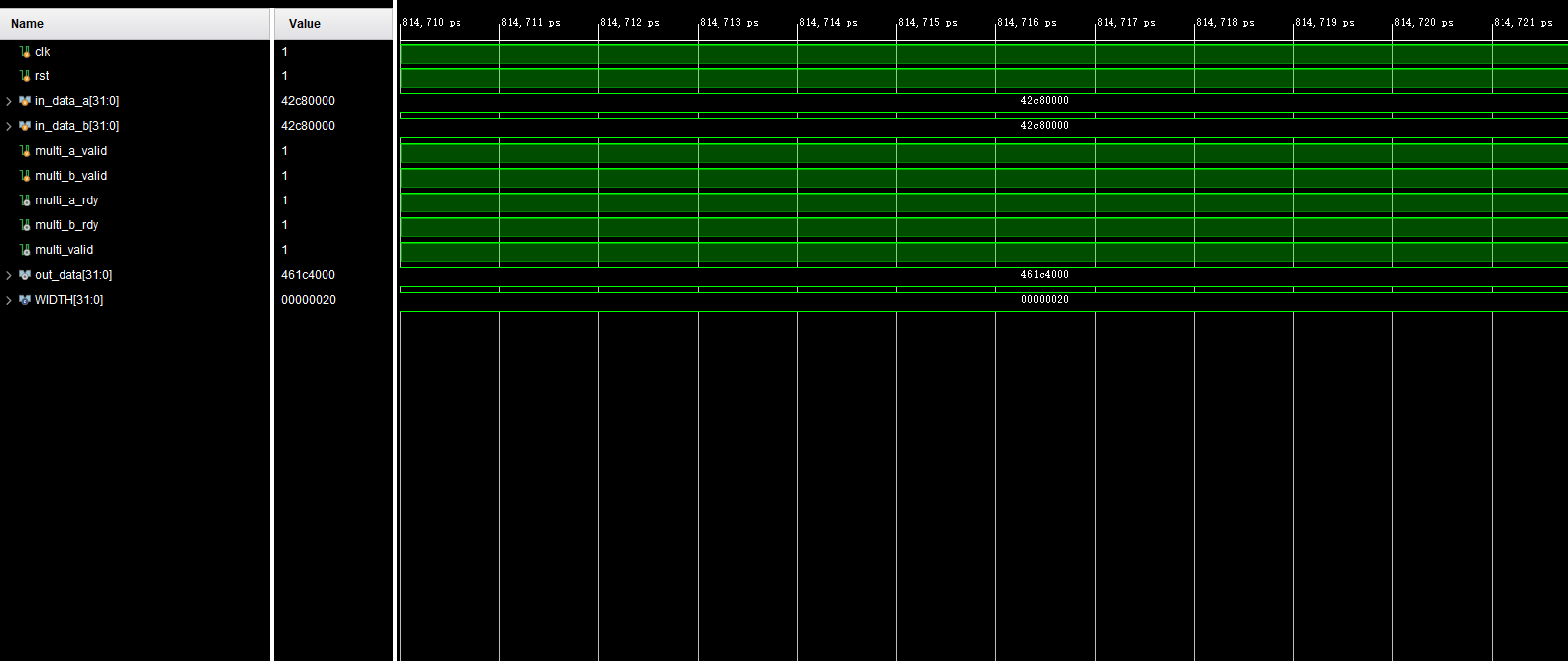


图5-1 数组乘法仿真

由图5-1可知，在将a，b输入数据的valid信号拉高后，模块正常运行，且a，b与输出的值的波形结果符合预期，数组乘法模块具有正确性。

5.2.2 矩阵加法

在本节仿真中，和5.2.1节类似，将输入数据a，b均取32位浮点数表示，其十进制值为100，仿真得到的波形应为输入数据a，b均为42c80000，输出数据为43480000，仿真得到的具体波形如图5-2所示。

由图5-2可知，在将a，b输入数据的valid信号拉高后，模块正常运行，且a，b与输出的值的波形结果符合预期，矩阵加法模块也具有正确性。

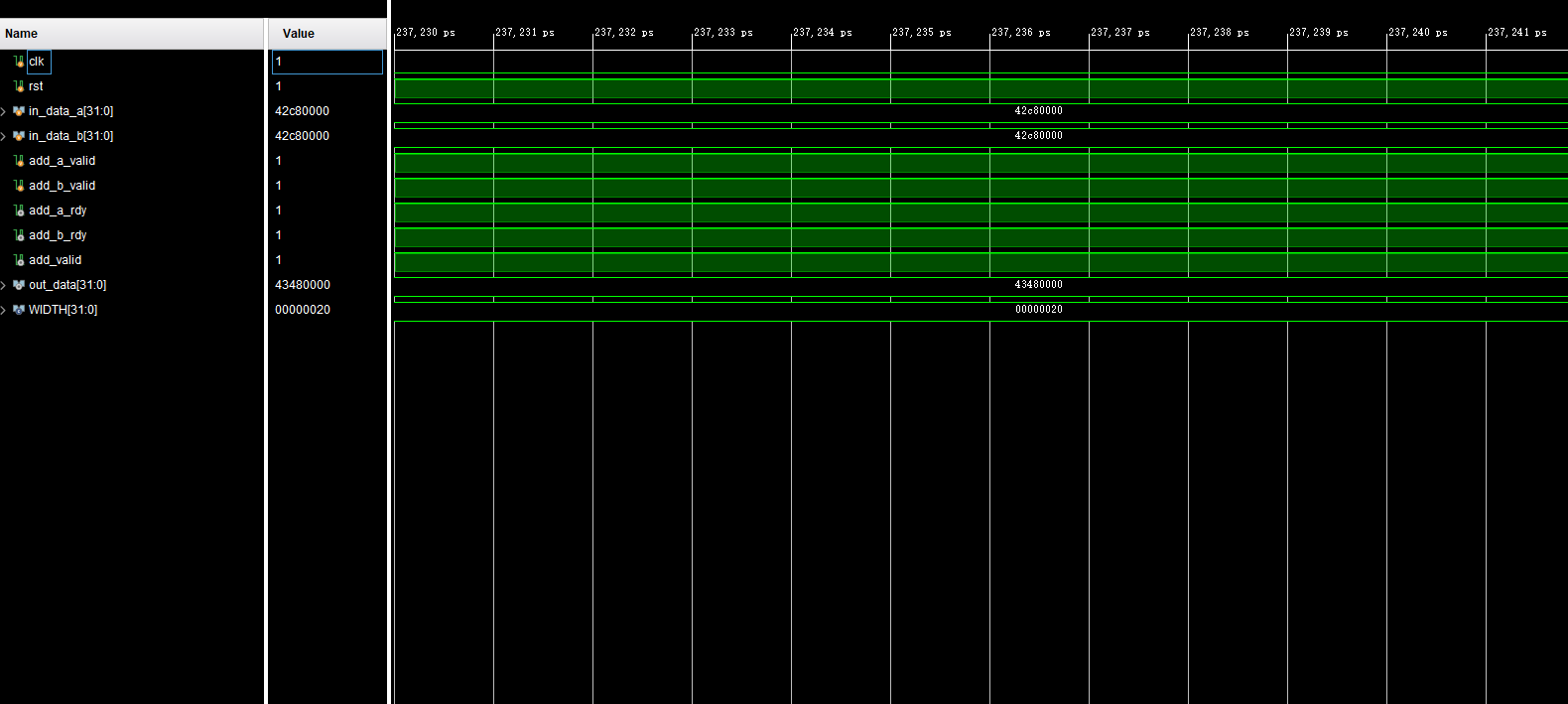


图5-2 矩阵加法仿真

5.2.3 全并行排序

本节仿真与前面两节稍有区别，输入的排序数据为4个输入，而输出的排序后数值则为采用类似图3-9的方式，将排序后数据拼接后一起输出，输入数据均为32位int型整数，输出数值应为128位数据流。排序后数据应为降序。这里取输入数据十进制值分别为40,20,30,10，输入数据波形应为00000028,00000014,0000001e，0000000a，输出数据流应为00000028\_0000001e\_00000014\_0000000a，其仿真波形如图5-3所示。

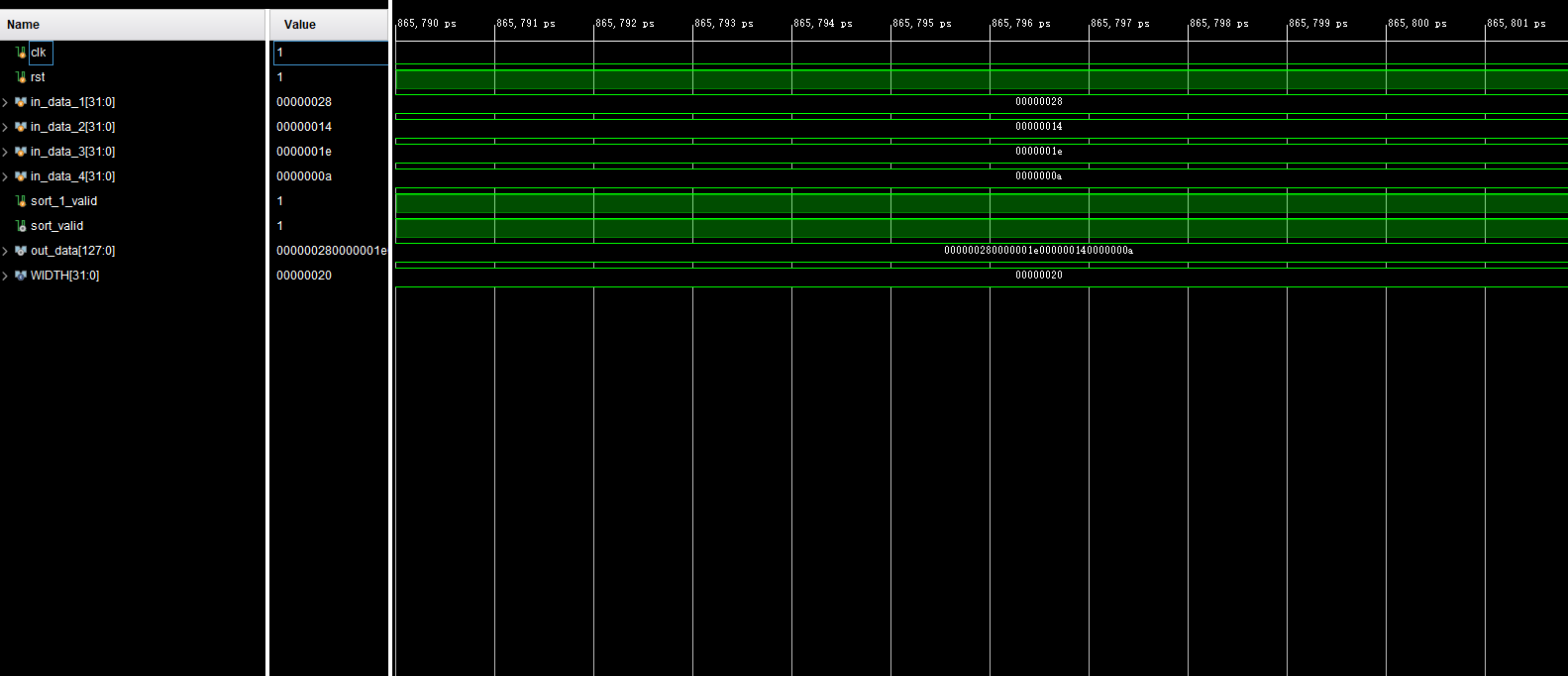


图5-3 并行排序仿真

由图5-2可知，在将待排序输入数据的valid信号拉高后，模块正常运行，且待排序输入值与输出的值的波形结果符合预期，并行排序模块的正确性得到验证。

5.3 函数调用时间

在本节中，我们将所设计系统软硬件程序编写后，将其烧录到Zedboard开发板上并对其函数调用时间进行测量。本实验主要以软硬件实现的对比为主，以此讨论硬件实现对于相应功能的加速效率。对于函数时间的测量则采用xilinx官方库内置的板级时间获取函数XTime\_GetTime()来获得其函数调用开始时的时间以及函数调用结束后的时间，然后按照计算精度精确到ns来测量其运行时间。三个函数模块分别测量的结果如图5-4，图5-5，图5-6所示。在对比实验中，对于每个函数调用将分别测量其软件实现以及硬件实现各100次。

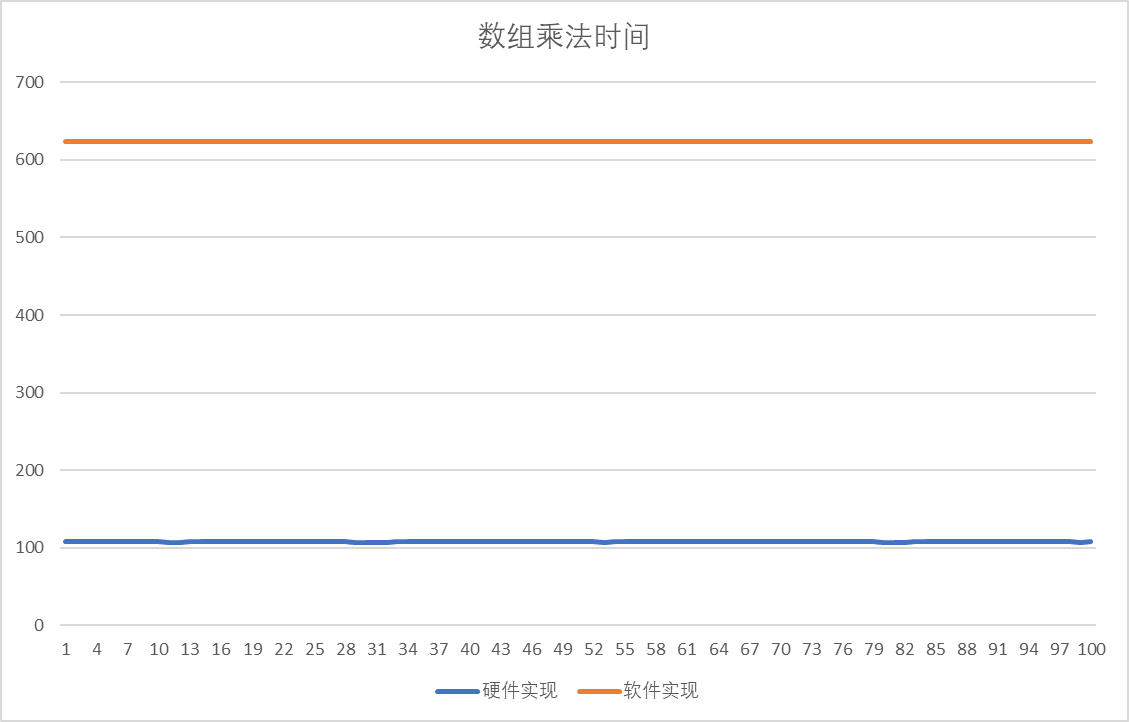


图5-4 数组乘法时间对比

由上述对比曲线图可知，在Zedboard开发板的体系结构中，我们实现的硬件实现的函数调用的运行时间均高于软件实现。本次实验证明了我们设计并实现的函数调用结构确实能够加速CPU端的特定功能，并且能够为软件程序员提供便利接口。

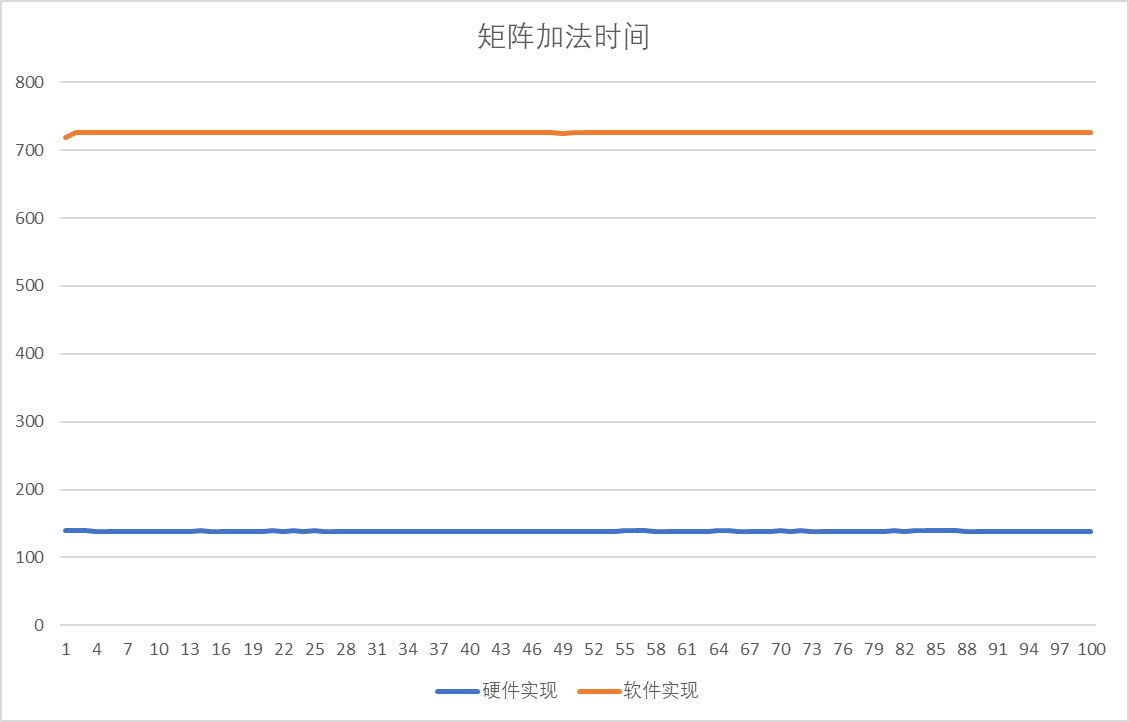


图5-5 矩阵加法时间对比

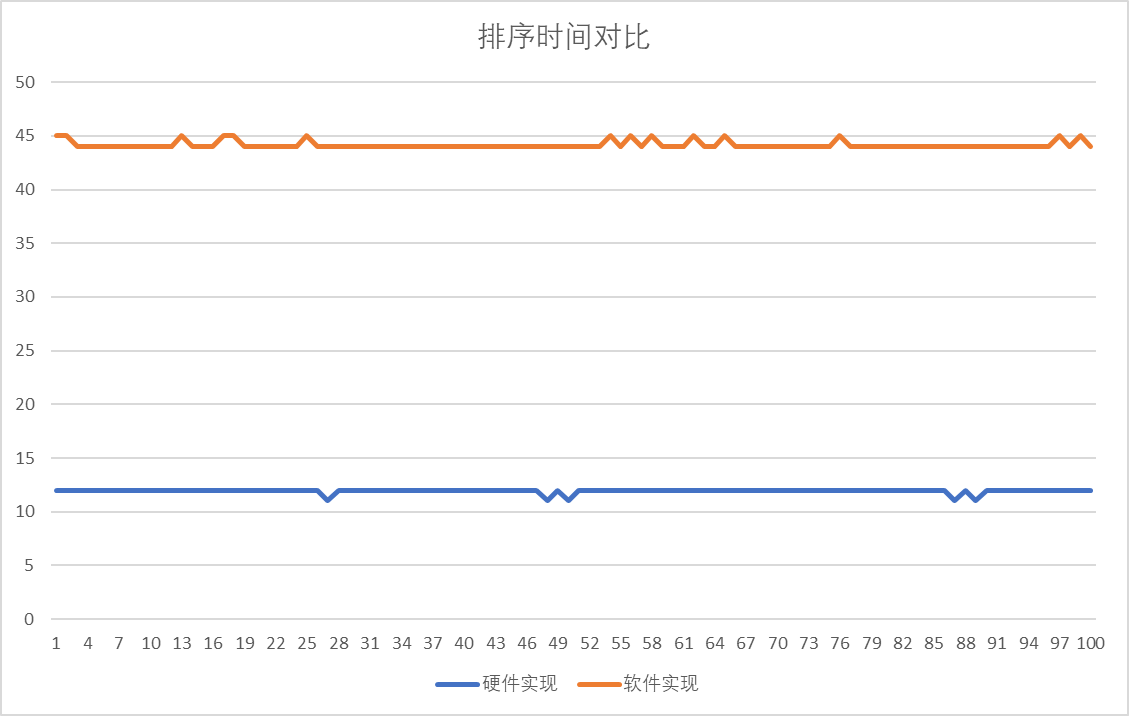


图5-6 排序时间对比

结　论

在本研究中，我们针对基于FPGA的SoC开发进行了调研，学习了FPGA开发的相关知识。此外，在FPGA的学习过程中，根据xilinx官方提供的例程，逐步熟悉了官方IP核的使用方法以及其内部结构，并在自己深入理解其工作原理的基础上对其进行了符合课题研究的修改。根据现有FPGA设计的开发流程，分析了FPGA自身的优势，得出开发基于FPGA的异步状态机接口的可能性和意义。

最终，我们结合vivado+vitis开发套件，设计了一个CPU与FPGA之间进行异步通信的函数调用接口，并在实际ZYNQ-7000系列开发板上，测试其运行的准确性和效率。

在理论上，我们设计的函数调用接口模块具备可扩展性与高效性能。不仅可以加速CPU内特定功能的计算速度，更为软件开发者提供了一个使用已定义好FPGA功能的简易接口。这种接口能够充分发挥CPU与FPGA内计算资源的优势，在合理利用CPU与FPGA内资源的基础上进行特定功能的计算。在实际使用场景中，根据本接口的定义，软件程序员可以更轻便的将FPGA作为外设使用，提高了软件的开发效率。

在未来可能的工作方面，本接口有更进一步扩展的空间。除了提供CPU调用FPGA内功能的接口外，还能提供FPGA请求CPU内资源辅助运算的接口，实现将FPGA作为协处理器与CPU一同辅助开发。而函数调用的功能，也能进一步优化为操作系统中的系统调用，使得接口具有可移植性。

参考文献

[1]吴艳霞,梁楷,刘颖,等. 深度学习FPGA加速器的进展与趋势[J]. 计算机学报,2019,42(11):2461-2480. DOI:10.11897/SP.J.1016.2019.02461.

[2]Lacey G, Taylor G W, Areibi S. Deep learning on fpgas: Past, present, and future[J]. arXiv preprint arXiv:1602.04283, 2016.

[3]Wang T, Wang C, Zhou X, et al. A survey of FPGA based deep learning accelerators: Challenges and opportunities[J]. arXiv preprint arXiv:1901.04988, 2018.

[4]刘焰强, 戚正伟, 管海兵. FPGA加速系统开发工具设计:综述与实践. 软件学报, 2020, 31(10): 3087-3099. http://www.jos.org.cn/1000-9825/6065.htm

[5]Lai Y H, Ustun E, Xiang S, et al. Programming and synthesis for software-defined FPGA acceleration: status and future prospects[J]. ACM Transactions on Reconfigurable Technology and Systems (TRETS), 2021, 14(4): 1-39.

[6]Zhou Z, Bi Y, Wan J, et al. Userspace Bypass: Accelerating Syscall-intensive Applications[C]//17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23). 2023: 33-49.

[7]Rehan Hameed,Wajahat Qadeer,Megan Wachs,等.Understanding Sources of Inefficiency in General-Purpose Chips[C].2010.

[8]Doug DiSabello,Tom VanCourt,Josh Model,等.Achieving High Performance with FPGA-Based Computing[J].Computer.2007,40(3).

[10]赵志强;FPGA芯片设计及其应用[J];电子技术与软件工程;2018年21期

[11]冯建文等编著．基于FPGA的数字电路实验指导书．西安电子科技大学出版社．2016.04．第4页

[12]李洪涛等编著．数字信号处理系统设计．国防工业出版社．2017.01．第180页

[13]田雨晨.FPGA芯片设计及其应用[J].数字通信世界,2019,(4):228. DOI:10.3969/J.ISSN.1672-7274.2019.04.187.

[14]潘新祥,胡习霜,韩立宏.软硬件协同设计分析[J].指挥控制与仿真,2008(03):117-119.

附　录

附录正文样式与文章正文相同

致　谢

值此论文完成之际，首先向我的导师……