班　级　010911
学　号　01091088

西安电子科技大学

本科毕业设计论文

题　　目　基于 TMS320C6455 的串行 RapidIO 技术及其实现

学　　院　通信工程

专　　业　通信工程

学生姓名　袁斌

导师姓名　杜栓义

Contributers:

HaoChen write the statement page

Olorin183795 bug report

yzg3307 bug report

Author by Xue-Jilong (xuejilong@gmail.com) and Justin-Wong (bigeagle@xdlinux.info)

Typeset by LaTeX 2ε and CTEX and provide for Bachelor Thesis of Xidian University.

摘　要

本文描述了传统互连总线的问题并对新兴的高速互连总线技术进行了对比，总结出了串行 RapidIO 技术是最适合用于嵌入式领域的互连技术。

RapidIO 技术是一种高性能、低引脚数、点对点的基于数据包交换的系统级互连架构。其高带宽、低延时、高效率及高可靠性的优点为高性能的嵌入式系统内部互连通信提供了良好的解决方案。文中研究了 RapidIO 的 3 层协议及关键技术，并以 TI 公司的 DSP TMS320C6455 和 Xilinx 公司的 FPGA XC5VSX50T 为例，阐述了这两大平台在串行 RapidIO 实现方面的细节，设计了 DSP 和 FPGA 间的串行 RapidIO 通信 (基于 4 通道串行 RapidIO 链路) 并对实际传输速率进行了测试分析。分析显示 DSP 中的 NWRITE 事务的最大传输速率 (826.09 MB/s) 较为接近理论传输速率 (1125 MB/s) 而 NREAD 事务的最大传输速率 (62.17 MB/s) 则远远低于理论值。

经分析，DSP 中的 NREAD 事务传输速率远低于理论值的原因主要有如下两个：NREAD 事务中间环节较多，延时十分明显；NREAD 事务的响应包带有数据，相比不接收响应的 NWRITE 事务传输速率要慢很多。

关键词：串行 RapidIO，TMS320C6455，DSP，FPGA

ABSTRACT

This article describes the problems of the traditional interconnect bus and emerging high-speed interconnect bus technology were compared. Summarizing that the Serial RapidIO technology is best suited for embedded interconnect field.

RapidIO technology is a high performance, low pin count, point-based packet switched system level interconnect architecture. Its high bandwidth, low latency, high efficiency and high reliability advantages for high-performance embedded systems interconnect communication provides a good solution. The three layers of RapidIO protocol and its key technology are discussed in this article. Based on TI's DSP TMS320C6455 and Xilinx's FPGA XC5VSX50T, Serial RapidIO implementation aspects in detail were explained in these two platforms. The design of RapidIO communications between DSP and FPGA were implemented by 4x Serial RapidIO Link, and the actual transfer rate were tested on it. Analysis showed that the maximum transfer rate(826.09 MB/s) of NWRITE transaction within DSP is close to the theoretical transfer rate(1125 MB/s) while the maximum transfer rate of NREAD transaction(62.17 MB/s) is far lower than the theoretical value.

By the analysis of the DSP, the reason why the transmission rate of the NREAD transaction is far below the theoretical value mainly comes with the following two: First, more intermediate links of the NREAD transaction are required, which result in high latency; Second, the response of NREAD transaction include lots of data, compared with the NWRITE transaction, which do not receive responses, further reducing the transmission rate.

Keywords: Serial RapidIO, TMS320C6455, DSP, FPGA

目　录

第一章　绪论

1.1　研究背景及意义

在过去的 30 多年间，处理器的主频和性能呈现指数增长，然而处理器总线频率的增长速度却相对缓慢，这导致了由时钟频率表征的 CPU 内核性能和由总线频率表征的 CPU 可用总线带宽之间的差距在不断扩大，传统的总线技术已逐渐成为制约嵌入式高速处理系统快速向前发展的关键因素。

1.1.1　传统总线互连问题

传统总线多以并行总线为主，按功能可分为地址总线、数据总线和控制总线。为了提高总线的传输能力，传统总线多采用增加数据总线的宽度或是提高总线频率的方式来实现。然而，不停地通过增加传统总线的位宽和频率来解决数据传输的瓶颈问题并不是一种明智且可行的办法。对于并行总线而言，数据位宽和总线频率的不断增加虽然在一定程度上满足了人们对高速数据传输的需求，但同时也带来了一些新的问题。过多的引脚使得芯片封装和布局布线变得复杂而艰难则是问题之一。例如，一个主流的处理器上典型的总线一般会有 64 个数据引脚、32 ～ 40 个地址引脚、12 到 13 个检查数据和地址正确性的校验引脚，以及 30 个左右用于传递控制信令的控制引脚，再加上因半导体器件构造所需的约 50 个电源和地引脚，总引脚数可达 200 个以上。这 200 多个引脚为半导体器件的封装和电路板上的走线制造了不小的麻烦。同时，如果使用这类总线通过背板进行多个电路板间的信号传递，会对系统的功能和可行性带来很大的负面影响。[1]

另一个问题是当传统总线的工作频率超过 133 MHz 时，总线上支持的器件很难超过两个。在总线上增加器件会增加容性负载，而容性负载意味着必须填满或排空电荷才能达到希望的信号电平。因此，额外的电容将增加信号的上升和下降时间，从而限制了总线的工作频率。此外，传统总线还存在着时钟与信号的偏移容限的问题。由于传统总线是一组并行信号线的集合，信号的有效时刻取决于时钟信号的有效沿，因此在信号的跳变沿和时钟的跳变沿之间可以容忍的偏移就有一个上限值，在高速率的情况下，走线长度和信号进出器件自身的翻转时间都会影响总线的时钟频率口。[2]

从单分段共享总线互连系统到级联的多分段共享总线互连系统，传统上的总线互连技术已达到了极限性能，互连通信问题已经成为制约嵌入式系统整体性能提高的瓶颈。为了克服传统总线互连方式的种种弊端，低引脚数、低延迟、高带宽、高效率及高可靠性的新兴互连技术应运而生。图1.1表示了总线互连的发展趋势。[3]

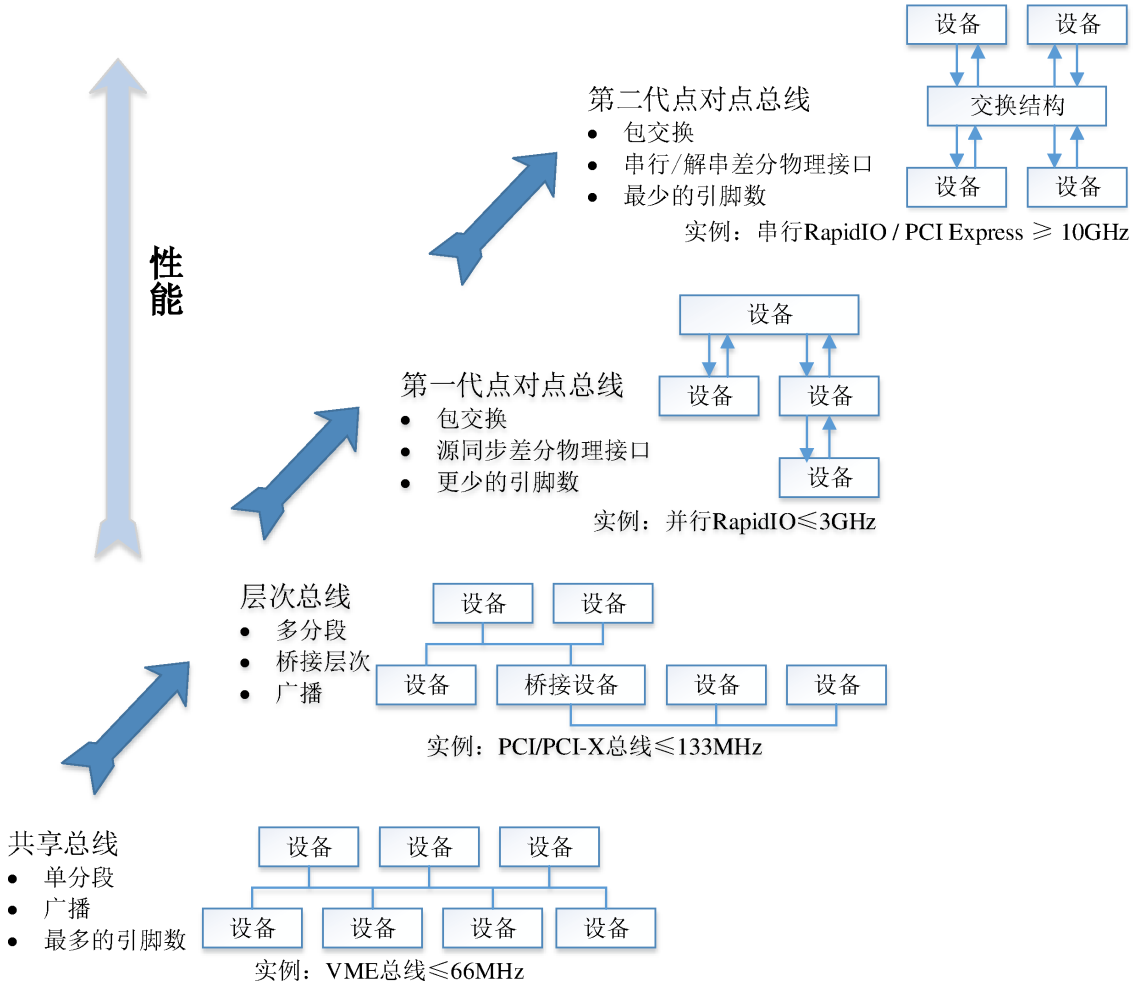


图 1.1 总线互连的发展趋势

1.2 主流互连技术对比

技术和市场的双重动力推动了互连技术及体系结构的重大变革，相继涌现了 PCI Express、RapidIO、InfiniBand 等一系列新兴高性能 I/O 互联技术。这些技术采用点对点交换式总线来设计系统的互连构架，代表了新型总线的发展方向，在应用领域方面既有交叉又各有侧重。[4] 它们的性能对比如表1.1所示。

与 RapidIO 相比，PCI Express 主要的应用仍是计算机，而且为了兼容传统 PCI 技术，使得它在嵌入式设备方面的应用具有一定的局限性，如缺少了提供给复杂嵌入式系统所需的可扩展性、鲁棒性和效率，不支持点对点对等通信等。InfiniBand 则主要进行系统域网络 (SAN) 互联，系统与网络上的操作一般需要通过软件驱动程序并使用消息通道来处理或使用远程直接存储器访问。

另一种通信带宽可达到 10Gbps 的常用互连技术为以太网，它是使用最广泛的局域网互连技术，也被扩展到嵌入式领域，但它的局限性也是显而易见的：

- 不支持硬件纠错，软件协议栈开销较大。
- 打包效率低，有效传输带宽因此而减小。
- 只支持消息传输模式，不支持对对端设备的直接存储器访问 (DMA)。

表 1.1 几种互连总线技术的对比

	串行 RapidIO 2.2	PCI Express 3.0	InfiniBand 1.2
主要应用方向	嵌入式系统芯片到芯片、板到板互连	计算机、通信系统中的外围设备互连	进行系统集群的服务器互连
数据传输带宽	1, 2, 2.5, 4, 5Gbps	8Gbps	2, 4, 8, 10.3125, 13.64, 25Gbps
传输通道数	1x, 2x, 4x, 8x, 16x	1x, 4x, 8x, 16x, 32x	1x, 4x, 12x
拓扑结构	任意结构	树型结构	任意结构
流量控制机制	重试、减速、信用	信用	端点到端点、信用
寻址	8/16 位设备寻址	32/64 位存储器寻址	16 位本地寻址
最大数据载荷	256 字节	4096 字节	4096 字节
对软件的依赖	弱	弱	强
编程模型	DMA,message,GSM	DMA	无标准 API
传输距离	80-100cm	40cm-50cm	～ 17m(铜介质)[5]
打包效率	92%(256 字节)	82%(256 字节)	79%(256 字节)
打包效率	80%(128 字节)	69%(128 字节)	65%(128 字节)

RapidIO 串行总线的小包的传输效率更高、允许更灵活的拓扑结构和多样化的处理部件、更好的系统鲁棒性、更高效率的流控机制、更多级的服务质量和更强的错误管理机制，因而也更适用于高实时性、高可靠性的嵌入式系统的设计。[6]

图1.2鲜明地表示出了各主流互连技术的适用范围，从中可以看出串行 RapidIO(Serial RapidIO) 最适合作为嵌入式系统互连。RapidIO 作为一项开放式标准，其高带宽、低延时、高效率及高可靠性的优点为有着不同厂商和需求的复杂嵌入式系统的内部互连通信提供了良好的解决方案。

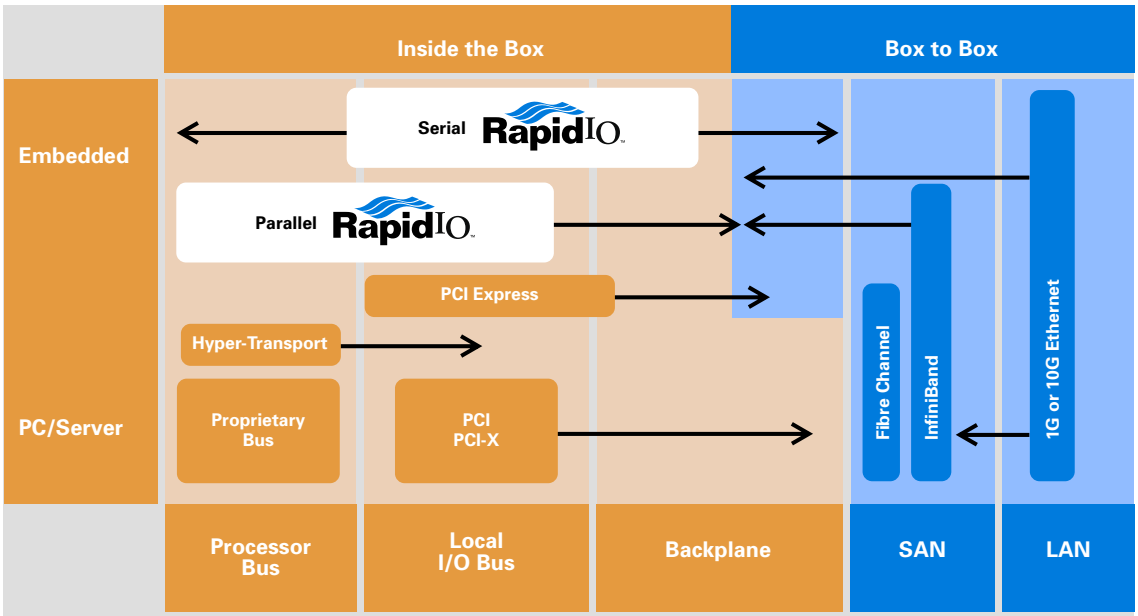


图 1.2 主流总线互连的应用范围[7]

本文正是基于此背景，对众多新兴互联技术进行了比较，选择了 RapidIO 这一在嵌入式互连领域有着出色表现的互连架构进行了全面深入的研究，并根据提供的 DSP 处理器 TMS320C6455 和 FPGA 器件 XC5VSX50T 进行实际上板测试对其出色的数据交换性能进行了验证。

1.3 RapidIO 规范演进历史

RapidIO 技术最初是由 Freescale 和 Mercury 共同研发的一项互连技术，其研发初衷是作为一种符合最流行的集成通信处理器、主处理器和网络数字信号处理器所需求的高性能分组交换技术，致力于为追求高性能的嵌入式互联系统内部提供高可靠、大带宽的高速数据总线。2000 年 2 月，RapidIO 行业协会正式成立后便接管了 RapidIO 的支持开发。2003 年 10 月，国际标准化组织 (ISO) 和国际电工协会 (IEC) 已批准 RapidIO 互连规范为 ISO ／ IEC DIS 18372 标准，成为第一个嵌入式互连国际标准。这使得 RapidIO(ISO) 成为唯一在互连技术方面得到授权的架构体系。RapidIO 的规范发布历史的里程碑如下:

- 1999 年，制定了 RapidIO 1.0 规范，第一次公开发布 RapidIO 规范
- 2001 年 3 月，发布 RapidIO 1.1 规范
- 2002 年 6 月，发布 RapidIO 1.2 规范，完成了具有历史意义的串行物理层协议的制订
- 2003 年 9 月，发布了流量控制扩展规范，为中等速率数据平面应用软件提供了拥塞控制机制
- 2004 年 8 月，发布了多播协议规范，为基于 RapidIO 的处理系统实现数据的分布式处理奠定了基础
- 2005 年 6 月，发布 RapidIO 1.3 规范，RapidIO 技术逐步进入到了成熟应用阶段
- 2007 年 6 月，发布 RapidIO 2.0 规范，串行物理层新增了两种传码率 5.0 Gbaud 和 6.25 Gbaud，新增了三种链路宽度 2x, 8x, 16x
- 2009 年 8 月，发布 RapidIO 2.1 规范
- 2011 年 5 月，发布 RapidIO 2.2 规范

从以上规范可知，RapidIO 已有超过 10 年的历史，仍然生机勃勃，它还在继续为开发人员提供高速、先进的通讯技术。目前 RapidIO 行业协会已经着手于 RapidIO 10xN 规范的制订。

RapidIO 10xN 规范将向后兼容目前应用 RapidIO 1.x 和 RapidIO 2.x 系统的产品。RapidIO 10xN 的初始规范将在位宽为 16x 的串行通道上支持每通道 10 Gbaud 的速率，而在后续规范中将使串行通道速率扩展至 25 Gbps 或者更高。相应的，为了达到更高的传输效率，物理层中的编码方案也将从 8B/10B（编码开销为 25%）转移至编码开销小于 5% 的其他编码方法。到目前为止，RapidIO 已经成为电信、通讯以及嵌入式系统内的芯片与芯片之间、板与板之间的互连技术的生力军。[2,8]

第二章 RapidIO 互连架构的研究

2.1 RapidIO 概览

RapidIO 是一种高性能、低引脚数、基于数据包交换的系统级互连架构。这一互连架构作为一种开放式标准，满足了高性能嵌入式行业在系统内部互连中对高可靠性、大带宽以及更快的总线速度的需要；满足了嵌入式基础设施在应用方面的广泛需要，典型的应用包括连接多处理器、存储器、网络设备上的存储器映射 I/O 器件、存储子系统和通用计算平台。RapidIO 互连主要用作系统内部互连的接口，支持芯片到芯片和板到板之间的互连通信，实际数据传输带宽可从 1 Gbps 到 80 Gbps。另外，RapidIO 标准路线图紧跟影响嵌入式系统设计的变化，确保了 RapidIO 技术将满足未来的系统需求，其在高速嵌入式互连领域将会有广阔的发展前景。

RapidIO 互连包括两类技术：

- 面向高性能微处理器及系统互连的并行接口
- 面向串行背板、DSP 和相关串行控制平面应用的串行接口

串行和并行 RapidIO 具有相同的编程模型、事务处理和寻址机制。RapidIO 支持的编程模型包括基本存储器映射 I/O 事务、基于端口的消息传递和基于硬件一致性的全局共享分布式存储器。RapidIO 提供很好的错误检测机制，还提供定义良好的架构以纠正和报告传输错误。RapidIO 互连架构被定义为分层结构，在保证后向兼容性的同时提供了可扩展性和未来增强的可能。[7]

RapidIO 互连架构主要由以下几大部分组成：规范体系、包和控制符号、包格式、事务格式与类型、消息传递和流量控制。

2.1.1 RapidIO 规范体系

RapidIO 采用三层分级体系结构，分别为逻辑层规范、传输层规范和物理层规范。逻辑层规范位于最高层，定义全部协议和包的格式，为端点器件发起和完成事务提供必要信息；传输层规范在中间层，定义 RapidIO 地址空间和端点器件间包传输所需的路由信息；底层物理层规范描述了器件级接口信息，如包传输机制、流量控制、电气特性和低级错误管理等。分级的层次划分保证了任意层增加新的事务类型无需改变其他层规范，提供了设计的灵活性，支持更好的前后兼容性。[1] 图2.1是 RapidIO 规范层次结构的示意图。

2.2 RapidIO 协议概览

RapidIO 协议的基本组成元素包括包和控制符号。包是系统中端点器件间的基本通信单元，提供终端节点设备间进行逻辑事务处理的接口。控制符号用于管理 RapidIO 物理层互连的事务流，也用于包确认、流量控制信息和维护功能。

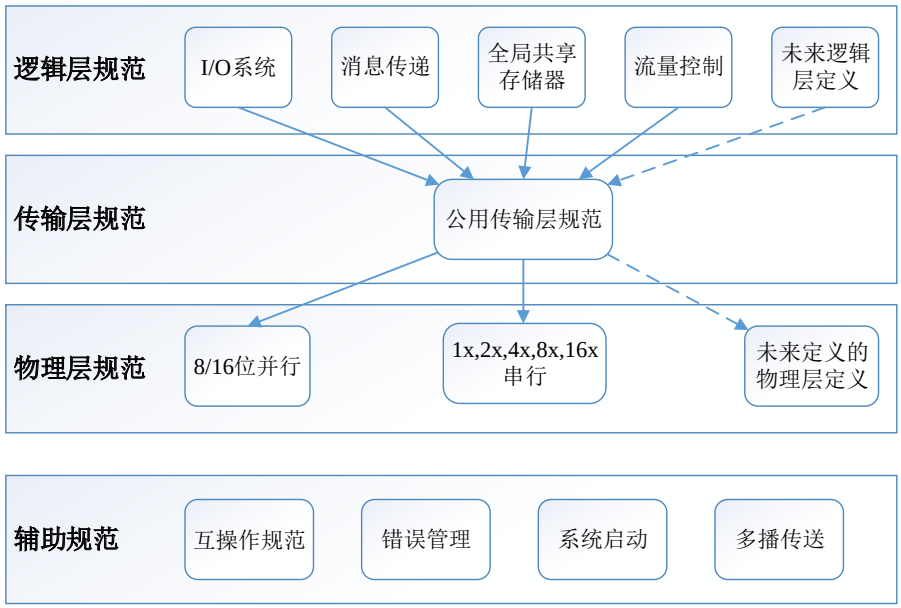


图 2.1 RapidIO 规范层次结构

2.2.1 RapidIO 操作概述

RapidIO 操作是基于请求和响应事务的。发起器件或主控器件产生一个请求事务，该事务被发送至目标器件。目标器件于是产生一个响应事务返回至发起器件来完成该次操作。RapidIO 事务被封装在包中，而包则包含确保将事务可靠传送至目标端点的所有必需的位字段。通常不会将 RapidIO 端点相互直接连在一起，而是通过介于其间的交换结构（fabric）连接。名词“交换结构”指的是提供系统互连的单个或多个交换器件的集合。图2.2 显示了事务是如何在 RapidIO 系统中传送的。

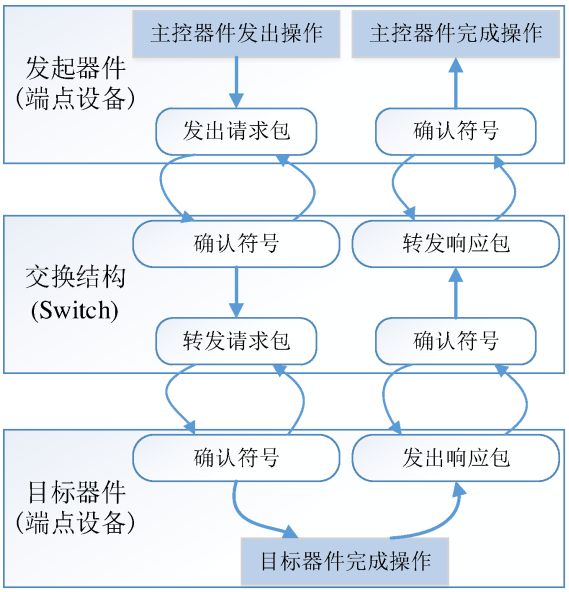


图 2.2 RapidIO 传输操作框图

在图2.2 所示的示例中，系统中的发起器件通过产生一个请求事务开始一次操作。该请求包被传送至交换结构器件，通常是一个交换机，交换结构器件发出控制符号确认收到了该请求包，然后交换结构将该包转发至目标器件。这就完成了此次操作的请求阶段。目标器件完成要求的操作后产生响应事务，通过交换结构将承载该事务的响应包传送回发起器件，传送时使用控制符号对每一跳（hop）进行确认。一旦响应包到达发起器件并得到确认，就可认为此次操作已经完成。[7]

2.2.2　RapidIO 包格式

RapidIO 包由代表 3 级规范体系结构的多个字段组成。图2.3 显示了典型的请求和响应包的格式。这些包的格式属于并行物理层包格式，串行物理层包的格式与此稍有不同。某些字段是依赖于具体的上下文的，并不会在所有的包中出现。

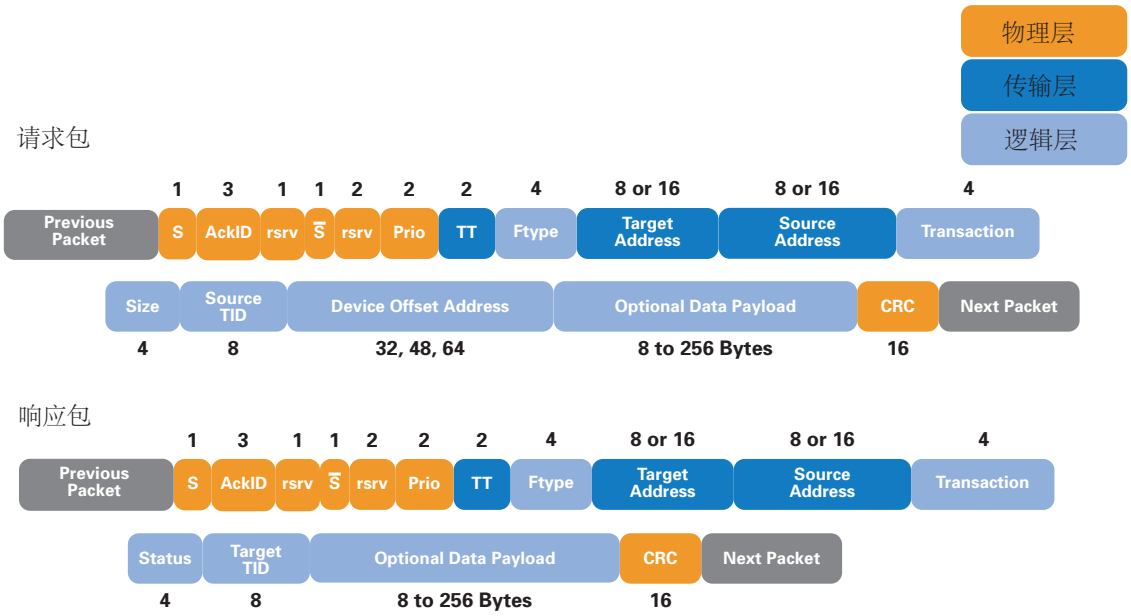


图 2.3 由不同规范等级的字段组成的 RapidIO 包格式

请求包以物理层字段开始，S 位指示这是一个包还是一个控制符号。AckID 表明交换结构器件将用控制符号来确认哪一个包。PRIO 字段指示用于流量控制的包优先级。TT、目标地址（Target Address）和源地址（Source Address）字段指示传输地址的机制类型、包应被递送到的器件的地址和产生包的器件的地址。Ftype 和事务（Transaction）指示正被请求的事务。长度（Size）字段等于编码后事务的长度。RapidIO 事务数据的有效载荷（Payload）长度从 1 到 256 字节不等。SrcTID（源事务 ID）指示事务 ID。RapidIO 器件在两个端点器件间最多允许有 256 个未完成的事务。对于存储器映射事务，跟随在 SrcTID 后面的是器件偏移地址（Device Offset Address）字段。写事务由数据的有效载荷作为结束。所有包以 16 位循环冗余校验码（CRC）结束。响应包与请求包类似。状态（Status）字段指示是否成功完成了事务。目标 TID（目标事务 ID）字段的值与请求包中源事务 ID 字段的值相等。[1,9]

2.3 事务格式与类型

对软件完全透明的互连应当有一套丰富的事务类型。RapidIO 事务由两个字段描述：包格式类型 Ftype 字段与事务字段。为了减轻解析事务的负担，按格式对事务分类，如表2.1 所示。[7]

表 2.1 RapidIO 丰富的事务格式集

功能	事务类型
I/O 非一致功能	NREAITED（读非共享存储器）
	NWRITE、NWRITE_R、SWRITE（写非共享存储器）
	原子（ATOMIC）（读 - 修改 - 写至非共享存储器）
基于端口的功能	门铃（DOORBELL）（产生中断）
	消息（MESSAGE）（对端口写）
系统支持功能	维护（MAINTENANCE）（读写配置、控制、状态寄存器）
用户定义功能	对专用事务开放
高速缓存一致性功能	读（READ）（读全局共享高速缓存器）
	READ_TO_OWN（写全局共享高速缓存器）
	抛弃（CASTOUT）（交出全局共享高速缓存器拥有权）
	IKILL（指令缓冲失效）
	DKILL（数据缓冲失效）
	刷新（FLUSH）（返回全局共享高速缓存器至存储器）
	IO_READ（读非缓冲全局共享高速缓存器的副本）
操作系统支持功能	TLBIE（TLB 失效）
	TLBSYNC（TLB 强迫完成失效）

2.4 消息传递

当数据必须被系统中的多个处理器件共享时，必须由协议维护和管理多个器件对共享数据的临时占用。许多嵌入式系统用软件机制实现该协议。如果存储器空间可被多个器件访问，可以使用锁或信号量来保证器件间正确的访问次序。在其他情况下，处理部件可能只有访问本地存储器空间的权利。在这些“非共享”的系统中，需要一种机制把数据从一个处理单元传递到另一个单元。使用对软件可见的消息传递（Message Passing）信箱（mailbox）可以实现这种机制。

RapidIO 提供了一种有用的消息传递机制。RapidIO 消息传递协议描述了支持信箱和门铃通信的事务。RapidIO 信箱是一个端口，器件间可通过它发送消息。接收器件在消息到达后对其进行处理。RapidIO 消息的长度从 0 到 4096 字节不等。一个接收器件有 1 到 4 个可寻址消息队列来捕获呼入的消息。

RapidIO 门铃（doorbell）是一种基于端口的轻量级事务，可用于带内（in-band）中断。门铃消息包括一个由软件定义的 16 位字段，该字段可用来在两个器件间传达多种不同意图的消息。[7]

2.5 流量控制

流量控制是所有互连技术的重要内容，流量控制的目的是使器件完成系统中的事务，避免被其他事务阻塞。基于总线的互连技术使用仲裁算法来确保器件进行恰当的转发操作，确保高优先级的事务优先于低优先级的事务得到转发。采用交换的互连技术，事务从系统的不同位置进入，从而无法使用集中式的仲裁机制。因此 RapidIO 使用若干补充机制来获得系统中平稳的数据流并避免系统死锁。[1]

2.5.1 链路级流量控制

由于事务流与物理互连和系统划分密切相关，为了在流量控制方面尽可能限制开销和复杂度，规定流量控制为 RapidIO 物理层规范的内容。每个 RapidIO 包都有一个事务优先级，每个事务的优先级都与一个事务请求流相关。目前定义的事务请求流有三个优先级。高优先级的事务请求流先于低优先级的事务请求得到传送。在同一事务请求流内，执行事务排序规则，不同流之间没有排序要求。

RapidIO 在链路级定义了三种流量控制机制：重传、减速和基于信用的流量控制。重传机制是最简单的机制，不仅用于流量控制，还用于硬件错误恢复。接收方在因资源缺乏或接收到损坏的包而不能接收包时，就发出一个重传控制符号作为响应，发送方将重传该包。减速机制使用空闲控制符号，发送包时可以在包中插入空闲控制符号，以便器件在包间插入等待状态。接收器件也可以通过向发送器件发送一个减速控制符号，来请求发送器件通过插入空闲控制符号来减慢流速。基于信用的流量控制机制可供带有事务缓冲区的器件，尤其是交换结构使用。该策略使用特定的控制符号指明接收方每种事务流对应的缓冲区现状，发送方只有在得知接收方有可用的缓冲空间时才发送包。

2.5.2 端到端的流量控制

除了链路级流量控制机制，RapidIO 还定义了一套端到端的流量控制机制。链路级机制管理毗连器件间的信息流量。在更为复杂的系统中，来自多个源、但发往同一个或多个目标的复合流量可能严重降低整个系统的性能，这种状况可能在数个周期内持续发生。端到端的流量控制机制使用由交换或端点器件产生的特殊拥塞控制包来控制流量。通过交换结构将拥塞控制包传回引起拥塞的源器件，持续关闭拥塞流量一段时间。该机制通过限制源头流量来达到降低系统流量拥塞的效果。链路级流量控制机制属于物理层规范，用控制符号进行流量控制，与之不同的是，端到端流量控制机制属于逻辑层事务，用包进行流量控制。尽管现有的交换结构器件不会产生或响应新包，它们仍可传递这种新的流量控制包。

第三章　RapidIO 规范各层次的研究

如前文所述，RapidIO 规范[1] 由逻辑层规范、传输层规范和物理层规范构成。本章对各层规范进行简略的分析。

3.1　RapidIO 逻辑层规范

逻辑层规范位于最高层，定义全部协议和包的格式，为端点器件发起和完成事务提供必要信息。整个逻辑层由五大规范构成，分别是：I/O 逻辑操作规范、消息传递规范、全局共享存储器规范、流量控制扩展规范以及数据流规范。限于本次测试平台限制，这里主要介绍 I/O 逻辑操作和消息传递规范。

3.1.1　I/O 逻辑操作规范

I/O 操作可以通过使用请求和响应事务对来完成。在 RapidIO 体系结构中定义了 6 种基本的 I/O 操作，本文主要分析常用的“读”和“写”操作。表3.1给出了 4 种基本的 I/O 操作、用来执行相应操作的事务和对操作的描述。

表 3.1 I/O 逻辑操作

操作	使用的事务	描述
读	NREAD, RESPONSE	从系统内存中不一致读
写	NWRITE	向系统内存中不一致写
有响应写	NWRITE_R, RESPONSE	向系统内存中不一致写，操作结束前等待一个响应
流写	SWRITE	面向大数据量 DMA 传输优化的不一致写

3.1.1.1　读操作

如图3.1所示，读操作由 NREAD 和 RESPONSE 事务组成。NREAD 事务用来请求得到另一器件上某内存区域中的内容，请求读取的内容长度在 1 到 256 字节之间。请求者通过 RapidIO 互连结构向目标发送 NREAD 事务，目标取出所要读取的数据并将其作为 RESPONSE 事务返回给请求者。

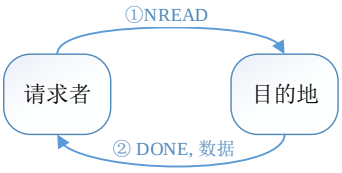


图 3.1 读操作

[1]这里以最近发布的 RapidIO 2.2 规范进行分析

3.1.1.2 写操作

如图3.2所示，写与流写操作由 NWRITE 和 SWRITE 事务组成。

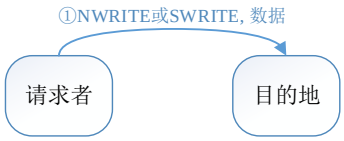


图 3.2 写与流写操作

NWRITE 事务允许的数据载荷长度（通过填充和对齐后）为双字、整字、半字和单字节的整数倍，SWRITE 事务是 NWRITE 事务的轻微变化，与 NWRITE 不同的是：其数据载荷长度只允许双字的整数倍；简化了头部开销，主要用来迁移 DMA 类操作的大量数据。SWRITE 操作的头部和循环冗余校验码开销仅为 10 字节，相对于 256 字节的有效数据载荷来说，开销不到数据载荷的 5%，所以效率高于 95%。NWRITE 和 SWRITE 事务均不要求接收端响应，所以事务在目标完成操作时并不会有响应给发送者。

如图3.3所示，有响应的写操作由 NWRITE_R 和 RESPONSE 事务组成。与“写操作”不同的是，“有响应的写操作”在目标完成操作时会发送响应以通知发送者。

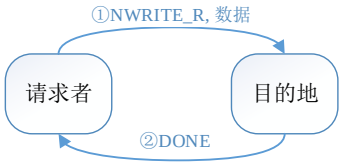


图 3.3 有响应的写操作

3.1.1.3 请求类事务

请求类事务主要包括 NREAD 事务和 ATOMIC 事务，ftype 指示事务的格式类型，请求类事务包对应的 ftype 字段值为 2，因此也称为第 2 类包。图3.4 给出了请求类包的位流格式。事务的具体类型由 transaction 字段决定，ftype 和 transaction 的组合唯一地标示了事务的格式。rdsize 字段结合 address 字段、wdptr 字段和 xamsbs 字段指出了所要读取数据的位置、大小和对齐方式。

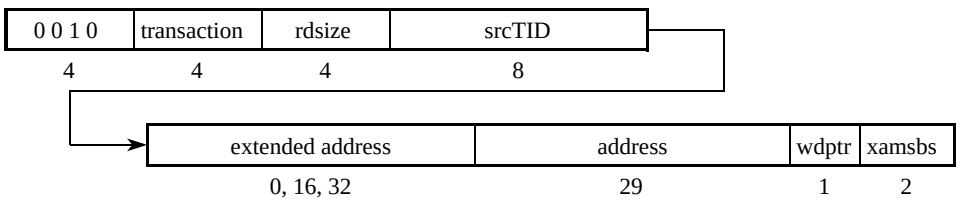


图 3.4 第 2 类（请求类）包位流格式

3.1.1.4　响应类事务

当 RapidIO 端点接收到由另一个 RapidIO 端点发起的请求时，该端点就会发送一个响应事务。响应事务总是以与请求事务相同的方式被发送和路由。响应类包对应的 ftype 字段值为 13(0b1101)，因此也称为第 13 类包。从广义上说，第 12、13、14 和 15 类包是响应类事务。通常第 12 和 14 类包是保留的，第 15 类包由具体应用定义，因此第 13 类包是主要的响应事务。

图3.5 给出了第 13 类（响应类）包的位流格式。transaction 字段表明了正在

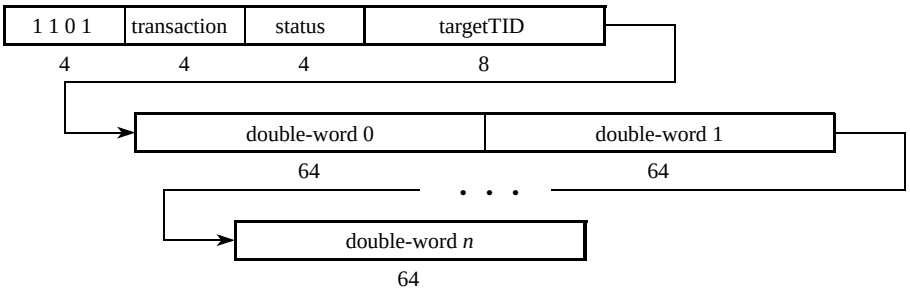


图 3.5 第 13 类（响应类）包位流格式

发送的响应事务的类型，规范定义了两种响应事务类型，有数据载荷类和无数据载荷类，其它的类编码是保留的。目标事务 ID 与响应包正在响应的请求事务 ID 相同，请求器件会根据该 ID 来匹配响应和请求。传给请求处理单元的响应事务 (status 字段中的子字段定义) 可能有两种：

1. DONE 响应向请求者表明所请求事务已经完成，它也为上面描述的读事务返回数据。
2. ERROR 响应意味着事务的目标遇到了不可修复的错误而不能完成事务。

3.1.2　消息传递规范

RapidIO 消息传递规范定义了两种不同的包格式用于消息传递操作，分别是第 10 类和第 11 类包格式。第 10 类包格式对应为门铃 (Doorbell) 操作，第 11 类包格式对应为数据消息 (Date Message) 操作。

3.1.2.1　门铃操作

如图3.6 所示，门铃操作由门铃 (DOORBELL) 事务和响应 (RESPONSE) 事务（通常是 DONE 响应）组成。处理单元用该操作将非常短的消息通过互连结构发

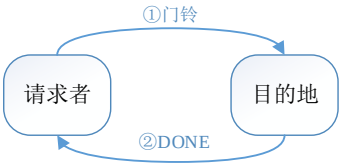


图 3.6 门铃操作

送到另一个处理单元。门铃事务包含用于保持事务信息的信息字段，但没有数据载荷，很适合用于发送处理器间的中断。它的信息字段是由软件定义的，可以用于任何目的。

收到门铃事务的处理单元将包放进处理单元中的门铃消息队列，该队列可以在硬件或本地存储器中实现。本地处理器通过读取门铃消息队列判断发送门铃消息的处理单元和信息字段，并根据该信息决定相应的操作。图3.7 给出了第 10 类(门铃类）包的所有字段。字段值 0b1010 表明包格式为第 10 类，rsrv 为 8 位保留字段，srcTID 为源事务 ID。

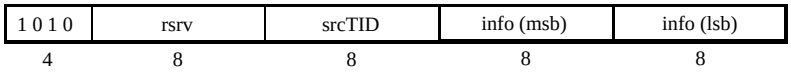


图 3.7 第 10 类（门铃类）包位流格式

3.1.2.2 数据消息操作

如图3.8 所示，数据消息操作由消息 (MESSAGE) 和响应事务（通常是 DONE 响应）组成，一般用于处理单元支持消息传递的硬件向另一个处理单元发送数据消息。完成一次数据消息操作最多需要 16 个单独的消息事务，消息事务数据载荷的大小是双字长度的整数倍。

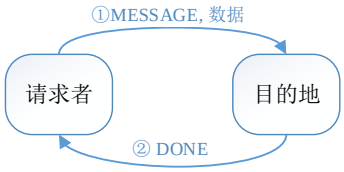


图 3.8 数据消息操作

图3.9 给出了第 11 类（消息类）包的位流格式，字段值 0b1011 表明包格式为第 11 类。处理单元的消息传递硬件负责接收数据消息操作，为了将消息包数据放

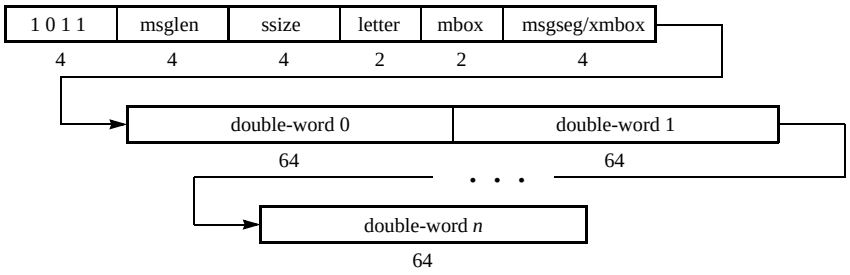


图 3.9 第 11 类（消息类）包位流格式

入本地存储器，处理单元的消息传递硬件会检查多个包字段（见图3.9)。

- 消息长度 (msglen) 字段——指定组成数据消息操作的事务数量。

- 消息分段 (msgseg) 字段——标识这个事务中包含了数据消息操作的哪一部分。消息长度和消息分段字段允许无序地发
- 送和接收数据消息的各个包，最多可规定 16 个消息分段。
- 信箱 (mbox) 字段——指定哪个信箱是数据消息的目标，最多可支持 4 个信箱。
- 信件 (letter) 字段——允许接收来自相同源、发往相同信箱的多个并发数据消息操作。最多可指定 4 个信件。
- 标准大小 (ssize) 字段——指定除数据消息的最后一个事务（可能）外所有事务的数据大小。

知道了以上这些信息，接收处理单元的消息传递硬件就可以计算出应该把事务数据存放在本地存储器的什么位置。

3.1.2.3 响应类包格式

所有对请求包的响应包均由第 13 类包产生，它已在节3.1.1.4中讨论过，它们在消息操作中的使用与在 I/O 操作中的使用类似。图3.10 给出了第 13 类（消息类）包的位流格式，字段值 0b1101 表明包格式为第 13 类。其中 target_info 字段

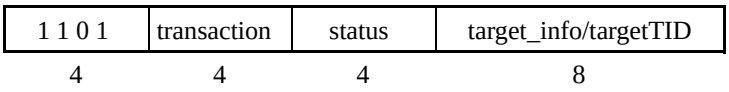


图 3.10 第 13 类（消息类）包位流格式

含 3 个子字段，如图3.11所示 在消息传递规范中的第 13 类包（响应类包）格式返

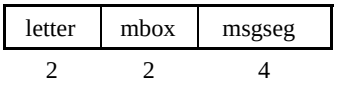


图 3.11 消息响应中的 target_info 字段

回状态信息和请求者的事务 ID、消息分段和信箱信息。请求处理单元可能接收到如下 3 种响应事务:

1. DONE 响应向请求者表明所请求事务已经完成，完成的响应可能包括 target_info, 该字段用来向操作的发起者返回状态信息。
2. RETRY 响应意味着请求事务不可接收，需要发送者重传所有需要重传的事务。
3. ERROR 响应意味着事务的目标遇到了不可修复的错误而不能完成事务。

3.2 RapidIO 传输层规范

传输层规范在中间层，定义 RapidIO 地址空间和端点器件间包传输所需的路由信息。

3.2.1 公用传输层规范概览

RapidIO 协议规范对公用传输层分别从功能、物理和性能特性三方面进行了定位。[10] 功能特性有：

- 其须确保在相同的或者兼容的包格式中支持从非常小到非常大不同的系统
- 由于只有一种类型的传输层协议规范，因此需要该规范上下兼容不同的逻辑层和物理层
- 传输层规范必须是灵活的，这样才能充分适应将来的应用
- 假定包可以直接从一个唯一的源传输到另一个唯一的目的地

物理特性：

- 传输层的定义不依赖于互连结构设备间的物理接口位宽
- RapidIO 不需要使用具体地理位置寻址，器件 ID 不依赖于其在地址映射中的位置进行标示

性能特性：

- 包头必须尽可能的小以最小化控制开销，同时还须快速、高效地组装和分拆
- 支持在互连结构中通过对传输层字段的解析完成点对多点的多播操作
- 可在 RapidIO 系统中实现某些器件的特定操作对带宽和时延较强的约束

3.2.2 系统拓扑结构及包的路由

RapidIO 网络主要由两种器件，终端器件（End Point）和交换器件（Switch）组成。终端器件是数据包的源或目的地。RapidIO 架构假设在系统中可能有成百上千个器件，并使用器件 ID 唯一地识别组成一个 RapidIO 系统的所有器件。器件间的通信通过发送包含源和目的器件 ID 的包进行。RapidIO 定义了两种器件寻址模式，小系统模式提供 8 位器件 ID，系统最多支持 256 个单独器件；大系统模式提供 16 位器件 ID，系统最多支持 65536 个单独器件。在使用器件 ID 作为系统级寻址方案的基础上，RapidIO 几乎可以支持任何系统拓扑结构。

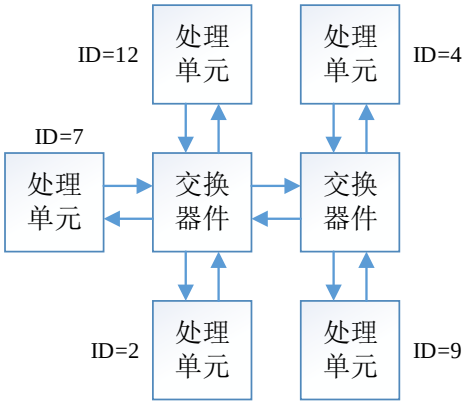


图 3.12 基于交换的小型系统

通常，RapidIO 系统多采用基于交换机 (Switch) 的拓扑结构 (如图3.12所示)，端点设备间不是直接连接而是通过交换机互连。为充分利用系统可用带宽，

RapidIO 采用源路由方法路由事务，包头中包含有源设备指定的目的器件 ID，交换机中则包含了一张基于 ID 和端口号的路由表。包在系统中传输时，交换机提取出包的传输层字段中的目的器件 ID，根据目的器件 ID 查询交换机自身的路由表以得到目的 ID 所对应的端口号，最后将数据包传送到相应的端口。通过查找路由表确定输出路径进行路由而确保包的正确传输。这种路由方法使得系统中的多个设备可以并行通讯，有效地提高了系统效率和可靠性。

与以太网类似，RapidIO 也支持广播或组播，每个终端器件除了独有的器件 ID 外，还可配置广播或组播 ID。交换器件根据包的目地器件 ID 进行包的转发，交换器件本身没有器件 ID。RapidIO 的互连拓扑结构非常灵活，除了通过交换器件外，两个终端器件也可直接互连。

3.2.3　传输层包格式描述

RapidIO 逻辑层规范的包格式中加入了 3 个字段以支持系统级寻址。传输格式的设计初衷是能够不依赖于交换结构，这样系统互连就可以采用特定应用所需要的任何技术。因此，所有的传输字段和它们与逻辑包的关系都以位流的形式描述，图3.13表明了位流的传输头部定义。阴影部分是与传输位有关的逻辑包定义

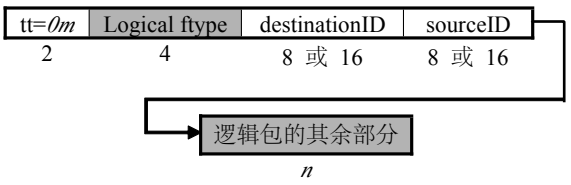


图 3.13 目的 -源传输位流

位，无阴影字段（tt=0b00——8 位器件 ID 字段，tt=0b01——16 位器件 ID 字段，目的器件 ID 和源器件 ID）是传输层字段，这些字段由 RapidIO 公用传输规范加入到逻辑包中。

3.3　RapidIO 物理层规范

RapidIO 物理层规范目前主要有两种：并行物理层规范和串行物理层规范。物理层最初定义的是并行传输规范，之后又定义了串行传输规范。

3.3.1　RapidIO 并行物理层规范

并行物理层规范中的物理接口是使用低压差分信号的 8/16 位链路协议端点规范（8/16 LP-LVDS）。该规范详细规定了使用 IEEE 标准 LVDS 信号技术时同时发送或接收 8/16 位数据、时钟和帧（FRAME）信号的方法。由于所有信号都由同一对差分线传送，实现 8/16 位数据通道实际需要的引脚数要多于该接口预计的引脚数。8 位接口需要 40 个信号引脚；16 位接口需要 76 个信号引脚。并行总线虽然在峰值速率上要高丁串行总线，但是由于其管脚多、开销相对大等因素使得并

行总线相对于串行总线而吉并无多少优势。所以只是在虽初有支持这种总线的芯片出现，目前几乎所有的物理层均采用了串行方式。[2] 因此，本论文主要研究串行 RapidIO，并以串行方式进行设计。

3.3.2 RapidIO 串行物理层规范

串行物理层规范定义了器件间的全双工串行链路，采用符合 IEEE 802.3 规范的万兆位以太网附加单元接口（10-Gigabit Ethernet Attachment Unit Interface, 也称为 XAUI)，链路由一个或多个通道构成，每个通道在每个方向上均使用单向差分信号。通道的概念用于描述串行 RapidIO 链路的宽度，通道定义为每个方向上的单向差分对。目前串行 RapidIO 规定了五种链路宽度：1x, 2x, 4x, 8x, 16x. RapidIO 串行物理层支持 RapidIO 器件间的包传送，包括包和控制符号的传递、流量控制、错误管理以及其他一些功能。[?]

RapidIO 串行物理层的特征如下所述:

- 采用 8B/10B 的编码机制和数据恢复同步技术将时钟嵌入到数据中
- 支持在每个方向上使用 1 个串行差分对作为 1 个通道到最高 16 个并列的串行差分对作为 16 个通道
- 支持每通道 1.25, 2.5, 3.125, 5 和 6.25 Gbaud 的传送速率（有效数据传输速率：1, 2, 2.5, 4 和 5 Gbps）
- 允许在串行 RapidIO 端口和并行 RapidIO 端口之间进行包传输而无需对包做额外的包处理
- 使用与并行 RapidIO 物理层相似的重传和错误恢复协议
- 使用专用的 8B/10B 码 (称为 K 码) 来管理链路，管理内容包括流量控制、包定界和错误报告
- 支持将物理层的带宽划分到 9 个支持独立流量控制的虚拟通道（可选特性）

3.3.2.1 包格式

如前面所述，RapidIO 使用包来发送系统内的数据和控制信息。在串行物理层中，有 24 位是为物理层部分定义的。这 24 位中 8 位是头信息，16 位是循环冗余校验码 (CRC)。图3.14展示了一个串行 RapidIO 的包格式并如何在包开始处附加物理层 ackID, VC, CRF, prio 字段以及如何在包尾附加 16 位循环冗余校验码字段。

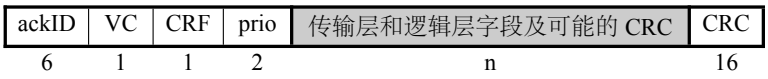


图 3.14 串行 RapidIO 包格式

无阴影的部分是由物理层加入的，有阴影的部分是传递给物理层的逻辑层和传输层位字段。物理层字段在传输层和逻辑层字段的两端。表3.2 显示了串行物理层附加到包中的字段含义。[11]

表 3.2 串行 RapidIO 包字段定义

字段	描述
ackID	ackID 是用于链路级包确认的包标识符。短控制符为 5 位，长控制符为 6 位
VC	VC 比特位规定了 PRIO 和 CRF 字段的用法
prio	PRIO 规定了包的优先级，或者根据 VC 字段的值包含虚拟通道 ID 中最重要的位
CRF	根据 VC 字段的值，CRF 在虚拟通道 0 的相同优先级的流之间区分或者包含有虚拟通道 ID 中最不重要的位
CRC	循环冗余校验码用于检测包中的发送错误。该字段通常加在包尾。

3.3.3 PCS 层与 PMA 层

串行 RapidIO 规范使用物理编码子层 (PCS) 和物理媒介附属子层 (PMA) 功能将包在发送方转化成串行比特流，并在接收方提取出该比特流（术语 PCS 和 PMA 出自 IEEE 802.3）。

3.3.3.1 物理编码子层 (PCS)

物理编码子层 (PCS) 的功能时负责产生空闲序列、通道分段、发送编码、解码、通道对齐和在接收时将分段合并的操作。PCS 使用 8B/10B 编码技术在整个链路上传输数据。此外 PCS 层还提供了一种机制，用于自动决定端口的工作模式和检测链路状态的方法。该层容许在发送者和接收者之间存在时钟差异而无需流量控制。下面将 PCS 分成发送和接收两个方面分别对其功能进行描述。

PCS 层执行以下发送功能:

- 在需要时将待发送的数据包和定界控制数据组成字符流。
- 在可用的通道上对需要发送的字符流进行分段。
- 当没有待发送的包和定界控制符号时，产生空闲序列并将其插入到每个通道的发送字符流中。
- 将每个通道的字符流独立编码为 10 位并行码组。
- 将编码得到的 10 位并行码组传递给 PMA 层。

PCS 层执行以下接收功能:

- 将接收到的 10 位并行码组解码成各通道独立的字符。
- 将从无效码组解码得到的字符标记为无效。
- 如果链路使用的通道多于一个，对齐字符流来消除通道间的偏移并将每个通道上的字符流重新组装为一个单独的字符流。
- 将解码字符流得到的包和控制符号递送到更高层。

3.3.3.2 物理媒介附属子层 (PMA)

物理媒介附属子层 (PMA) 的功能是逐通道将 10 位并行码组串行化为串行比特流，或将串行比特流转换为 10 位并行码组。在接收数据时，PMA 层的功能是

将接收的比特流逐通道分别对齐到 10 位码组边界。接着该层向 PCS 层的每个通道提供一个连续的 10 位码组流。10 位码组对 PCS 层以上的各层是不可见的。

3.3.4 通道数据流编解码发送顺序

如图3.15所示，该图给出了一个字符经过编码、并串转换、发送、串并转换和解码的完整过程。图的左边显示的发送过程是使用 8B/10B 编码对字符流编码和 10 位并串转换过程。右边显示的是接收者对接收到的码组进行串并转换和 8B/10B 解码的过程。点划线是产生 10 位码组的 PCS 层和串行化码组的 PMA 层的功能分界线。接收方还可使用包含逗点分隔符序列的特殊字符来建立 10 位码组的边界同步。[?]

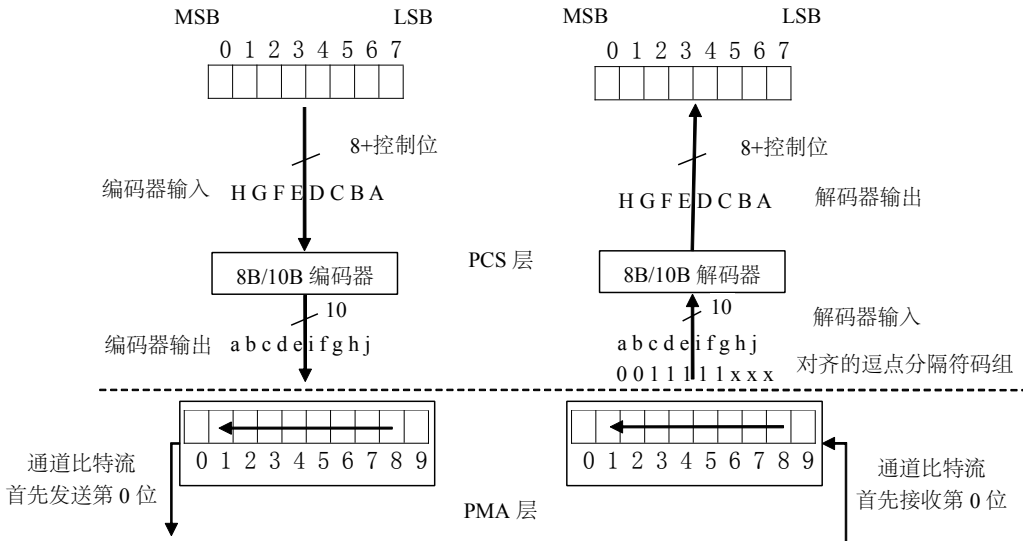


图 3.15 通道编码、串行化、并行化和解码的过程

3.3.5 错误管理与恢复

RapidIO 技术的一个主要应用目标是帮助实现高度可靠的电子设备，如电信级电信设备和企业存储系统。在这些系统中，尽可能合理地进行每项操作以避免错误并在发生错误时检测错误和尝试从错误中恢复是极其重要的。串行 RapidIO 的错误恢复是通过物理层规范实现的，不需要软件或高级系统干预，这样系统能够在最短的中断时间内从错误中恢复过来。所有的串行 RapidIO 系统都能够基于协议对数据包字段进行有效的检查。对协议不能进行保护的比特错误则会通过 CRC 校验来保护。对于较长的数据包，CRC 能够插入到中间对其头部和开始字段进行错误检测，不必等整个数据包传输完成后再检查。8B/10B 编码为链路的比特流传输提供了进一步的保护。[12]

RapidIO 具有丰富的维护和错误管理功能，这些功能支持最大的系统发现、配置、错误管理检测和恢复机制。RapidIO 使用维护包事务访问 RapidIO 能力寄存器 (CAR)、状态寄存器 (CSR) 和数据结构等来支持维护操作。RapidIO 错误管理机制由物理层规范处理，控制符号是硬件错误恢复的核心。

第四章　TMS320C6455 的串行 RapidIO 实现

4.1　TMS320C6455 简介

C6455 是目前单片处理能力最强的新型高性能定点 DSP，它是 TI 公司基于第 3 代先进 VeloviTI VLIW(超长指令字) 结构开发出来的新产品。最高主频为 1.2 GHz，16 位定点处理能力为 9600MMA C/ s。C6455 建立在增强型 C64x+ DSP 内核基础之上，代码尺寸平均缩短了 20% ～ 30%，周期效率提高了 20%。C6455 不仅是内核的增强和运算速度的提升，相比以前的芯片，集成了丰富的外围接口，如千兆以太网控制器，66 MHz PCI 总线接口，最重要的是增加了新的外设接口 SRIO，全双工工作时，4 个端口峰值速率每秒高达 25 Gb，解决了 DSP 高速数据传输的瓶颈，降低了开发多处理器系统的难度。[13]

本次对 TMS320C6455 芯片的编程开发采用 TI 公司提供的 Code Composer Studio Version 5.2 集成开发环境，该软件对用户友好，调试开发也比较便利。

4.1.1　TMS320C6455 所支持的串行 RapidIO 特性

基于 TMS320C6455 的串行 RapidIO 实现方面最具参考性的用户手册为 *TMS320C645X DSP Serial RapidIO(SRIO) User's Guide*[14]。TMS320C6455 所支持的 SRIO 特性如下:

- 遵循 RapidIO 互连规范 V1.2 勘误 1.2
- 遵循物理层 1x/4x 链路协议串行规范 V1.2
- 4x 串行 RapidIO 自动调整到 1x 端口，可以选择对 4 个端口进行操作
- 将时钟恢复集成到 TI 的 SERDES 宏中
- 使用包含 CRC 的硬件错误管理
- 支持交流耦合的差分电流型信号
- 支持的速率有 1.25, 2.5, 和 3.125 Gbps
- 可对不用的端口进行掉电处理
- 支持读、写、有响应的写、流写、对外部的原子操作和维护操作
- 可对 CPU 产生中断 (门铃包和内部调度)
- 支持 8 位和 16 位器件 ID
- 支持接收 34 位地址
- 支持产生 34、50 和 66 位地址
- 支持以下数据大小：字节、半字、字和双字
- 大端数据传输模式
- 直接 IO 传输
- 消息传递
- 数据载荷最大到 256 字节

- 一次消息可以包含最多 16 个包
- 可用于时钟域切换的弹性存储机制 (FIFO 块)
- 遵循短距离和长距离传输两种标准
- 支持错误管理扩展
- 支持拥塞控制扩展
- 支持一个多播 ID

不支持的特性有:

- 全局共享存储器
- 8/16 位并行链路差分电压信号标准兼容
- RapidIO 原子操作的目的地支持
- 1x 端口的混合频率支持（所有端口必须在相同频率）
- 内部 L2 存储器和寄存器的目标原子操作

4.2 TMS320C6455 的串行 RapidIO 物理层实现

串行 RapidIO 基于现在已广泛用于背板互连的 SerDes（Serialize Deserialize）技术，它采用差分交流耦合信号。差分交流耦合信号具有抗干扰强、速率高、传输距离较远等优点。差分交流耦合信号的质量不是由传统的时序参数来衡量，而是通过眼图来衡量，眼图中的“眼睛”张得越开则信号质量越好。图4.1 是一个典型的串行 RapidIO 信号的眼图。[15]

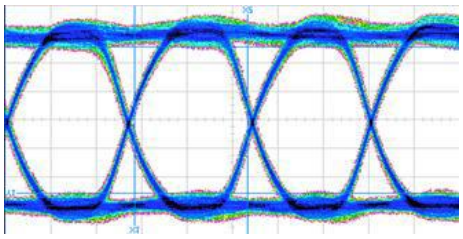
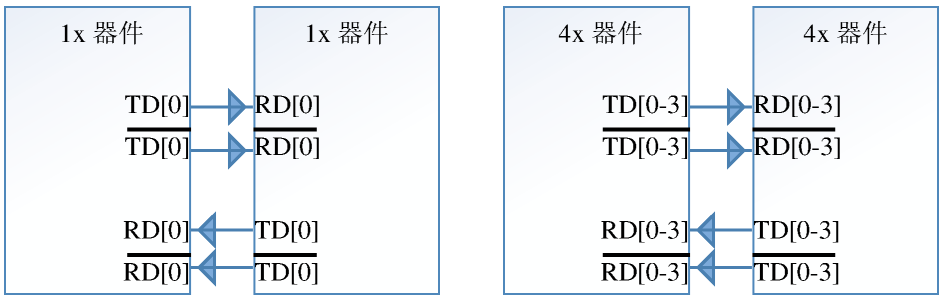


图 4.1 串行 RapidIO 信号眼图

差分信号的强弱由一对信号线的电压差值表示，串行 RapidIO 规定信号峰 -峰值的范围是 200mV － 2000mV。信号幅度越大，则传输距离越远，串行 RapidIO 按信号传输距离定义两种传输指标:

- 短距离传输 (Short Run)，<=50 厘米，主要用于板内互连，推荐的发送端信号峰－峰值为 500mV － 1000mV
- 长距离传输 (Long Run)，>50 厘米，主要用于板间或背板互连，推荐的发送端信号峰－峰值为 800mV － 1600mV

在 TMS320C6455 中，串行 RapidIO 差分接收器允许的最小差分信号峰 -峰值为 175mv。为了支持全双工传输，串行 RapidIO 收发信号是独立的，所以每一个串行 RapidIO 端口由 4 根信号线组成。TMS320C6455 DSP 上集成了标准的 1x/4x 串行 RapidIO 接口，与其它器件的 SRIO 端口互连时可采取的方案如图4.2所示。



(a) SRIO 1x 器件对 1x 器件接口框图　　(b) SRIO 4x 器件对 4x 器件接口框图

图 4.2 SRIO 1x/4x 器件互连接口框图

4.2.1　串行 RapidIO 中的数据流

数据流以串行 RapidIO 方式在 TMS320C6455 中传输时的相关外围器件如图4.3 所示。

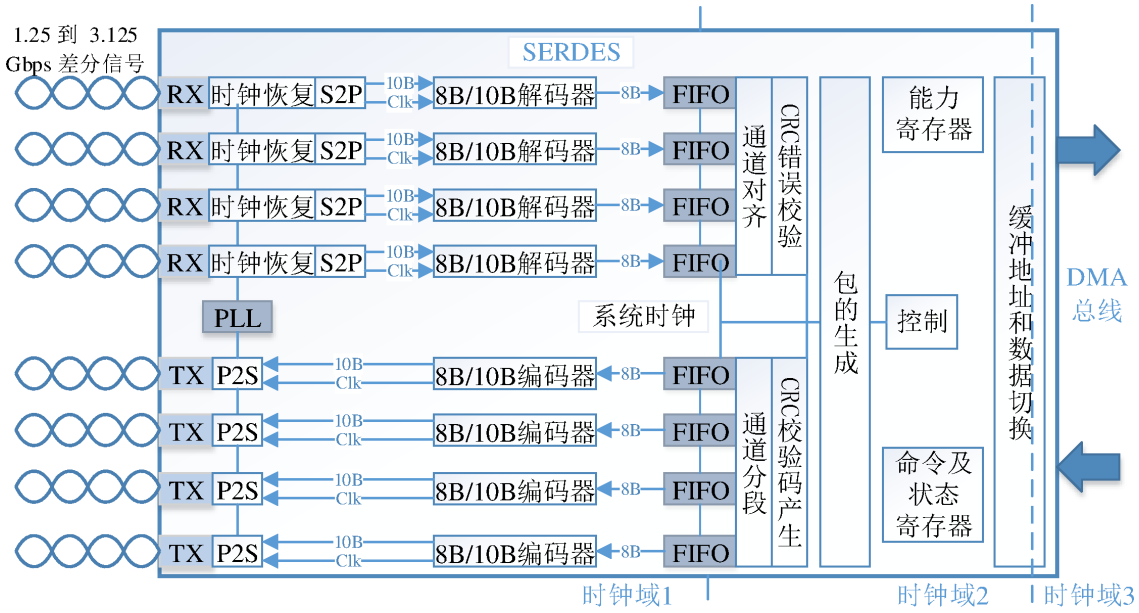


图 4.3 TMS320C6455 1x/4x 串行 RapidIO 外围器件框图

其中框图中各模块功能如下：

- RX: 接收器模块——用以接收从器件引脚进入的高速数据流。
- 时钟恢复: 从数据流中提取时钟信号。需要低频参考时钟和 PLL 模块以达到数据速率。
- S2P: 串行到并行转换模块——将接收到的串行数据转换为 10 位并行数据。
- 8B/10B 译码器: 将接收到的 10 位码组解码为原来的 8 位码组。
- FIFO: “先进先出”模块——提供一种用于在恢复的时钟域和公共的系统时钟域之间切换的弹性存储机制。深度为 8 个字。
- 通道对齐: 消除 4 通道串行链路通道间偏移并以纵列输出。仅在 4x 下有意义。

- CRC 错误校验: 负责对接收的数据进行运行时计数并在 1x 或者 4x 模式下计算出相应的 CRC，再将计算出的 CRC 与接收包包尾部分的 CRC 进行比较。
- TX: 发送器模块——将数字比特转换成差分交流耦合信号在信号线上发送出去。
- P2S: 并串转换模块——将并行 10 位码组与时钟信号转换为串行数据流。
- 8B/10B 编码器: 将 8 位码组转换为 10 位码组。
- 通道分段: 4 个通道上同时发送数据（4x 链路上使用），它将字符流逐字符分散到多个通道上。
- CRC generation: CRC 检验码产生模块。

发送/接收整个过程流图可用图4.4加以说明:

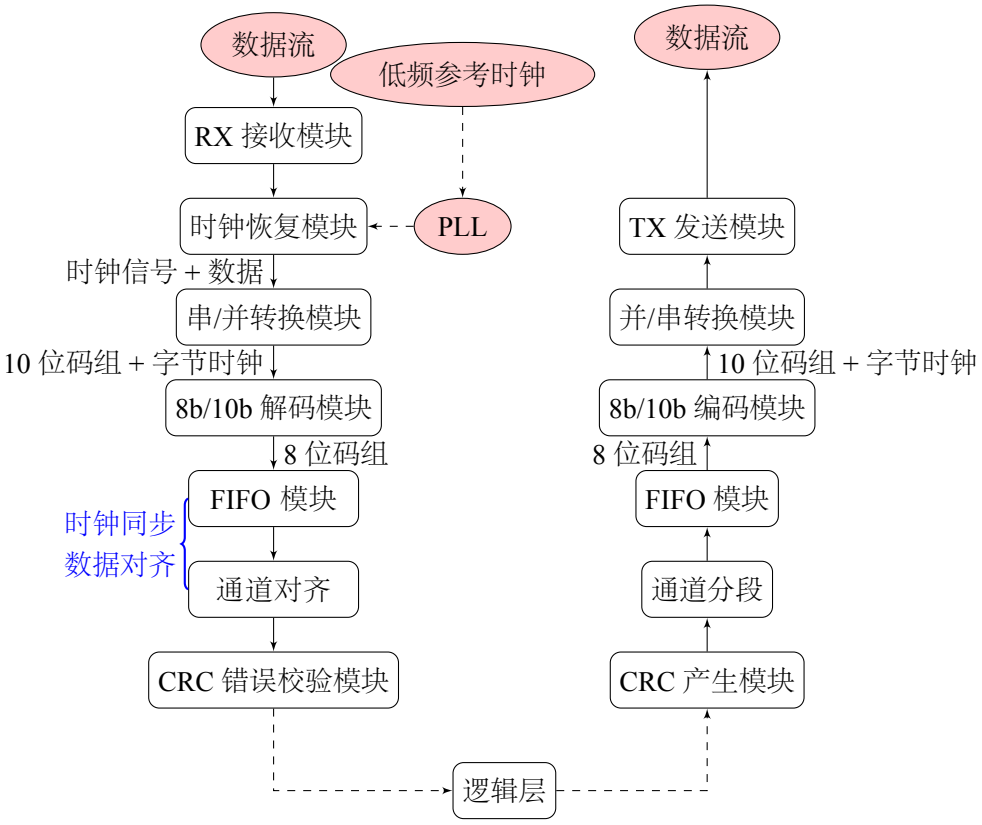


图 4.4 SRIO 数据流接收/发送处理流程

发送时，逻辑层和传输层将组好的包经过 CRC 编码后被送到物理层的 FIFO 中，“8b/10b 编码”模块将每 8 bits 数据编码成 10 bits 数据，“并/串转换”单元将 10 bits 并行数据转换成串行比特流 s，发送模块将数字 bit 转换成差分交流耦合信号在信号线上发送出去。这里的 8b/10 编码的主要作用是:

- 保证信号有足够的跳变，以便于接收方恢复时钟。串行 RapidIO 没有专门的时钟信号线，接收端靠数据信号的跳变恢复时钟。所以需要把信号跳变少的 8 bits 数据 (如全 0 或全 1) 编码成有一定跳变的 10 bits 数据。
- 使得总体数据中 0 和 1 的个数均衡，以消除直流分量，保证交流耦合特性。
- 可扩大符号空间，以承载带内控制符号。10 bits 能表示 1024 个符号，其中

256 个表示有效的 8 bits 数据，剩下的符号中的几十个被用作控制符号。控制符号可被用作包分隔符，响应标志，或用于链路初始化，链路控制等功能。

- 能实现一定的检错功能。1024 个符号中，除了 256 个有效数据符号和几十个控制符号外，其它符号都是非法的，接收方收到非法符号则表示链路传输出错。

接收的过程则正好相反，首先接收方需要根据数据信号的跳变恢复出时钟，用这个时钟采样串行信号，将串行信号转换为 10 bits 的并行信号，再按 8b/10b 编码规则解码得到 8 bits 数据，最后做 CRC 校验并送上层处理。

数据被正确的接收时，接收端会发送一个 ACK 响应包给发送端；如果数据不正确（CRC 错或非法的 10 bits 符号），则会送 NACK 包，要求发送方重传。这种重传纠错的功能由物理层完成，而物理层功能往往由硬件实现，所以不需要软件干预。

4.3　TMS320C6455 的串行 RapidIO 功能操作

4.3.1　SRIO 组件框图

图4.5展示了 TMS320C6455 中串行 RapidIO 相关器件的组件框图。

各组件功能如下所示：

- 装载/存储单元 (LSU): 控制直接 I/O(direct I/O) 包和维护 (maintenance) 包的发送。
- 存储器访问单元 (MAU): 控制直接 I/O(direct I/O) 包的接收。
- TXU: 控制消息包的发送。
- RXU: 控制消息包的接收。

以上四个单元使用内部 DMA 与内部存储器进行通信，使用缓存 (buffers) 和接收/发送 (receive/transmit) 端口与外部设备进行通信。端口所允许的操作由 SERDES 宏支持。

4.3.2　SERDES 宏

SRIO 允许可扩展的非专有接口，这为用户提供了诸多优点。利用 TI 提供的 SERDES 宏，相关外围器件的适应性和带宽的可扩展性变得非常强。由于接口采用串行方式，这意味着它不仅可以在单板上使用，还可以扩展至背板互连中。ASIC 或 DSP 上这些宏的集成可以有效减少板上离散组件的数目，总线驱动芯片也不再需要使用。TI 的 SERDES 宏是一个非常完备的宏，它包含发送器 (TX)、接收器 (RX)、锁相环 (PLL)、时钟恢复 (clock recovery)、串并转换 (S2P) 和并串转换 (P2S) 模块。各模块功能已在节给出。

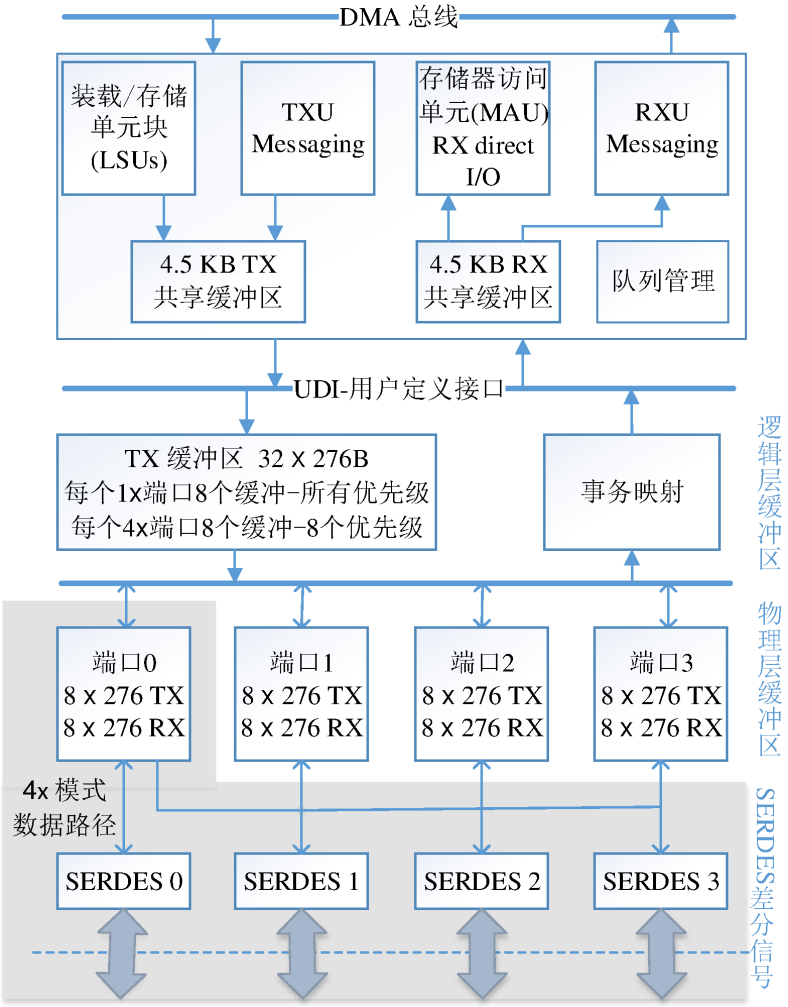


图 4.5 串行 RapidIO 组件框图

4.4 TMS320C6455 的 SRIO 基本读写和门铃操作

根据包的格式的不同, 将事务划分成很多类型, 其中最重要的类型有 3 种: NREAD(基本读操作)、NWRITE(基本写操作)、DOORBELL(门铃操作)。通过这 3 种类型的组合就可以完成所有的存储器读写操作。在介绍读写操作之前, 先介绍一下与 SRIO 有关的 DMA 操作。

在 C6455 上, SRIO 数据传输和 DMA 传输是结合的。此 DMA 与 EDMA 方式是独立的, 当进行 SRIO 传输时,DMA 以自动方式启动。对于发送方来说, DMA 将数据从 L2 SRAM 搬移到 SRIO 端口, 对于接收方来说, DMA 将数据从 SRIO 端口搬移到 L2 SRAM 内存。因此, 在进行传输时, 读写地址是直接显示在包里的, 而且此地址就是被读写的 DSP 的地址。换句话说, DSP 可以对另一片 DSP 的 L2 SRAM 直接进行读写操作。图4.6 就是自动 DMA 的传输操作。[13]

在 SRIO 部分的初始化之前是 DSP 芯片的初始化，用以设置系统时钟等。可用函数`int init_6455()` 实现。SRIO 使用之前同样也要对其进行初始化，其基本读写与门铃操作的流程可以用图4.7来说明。

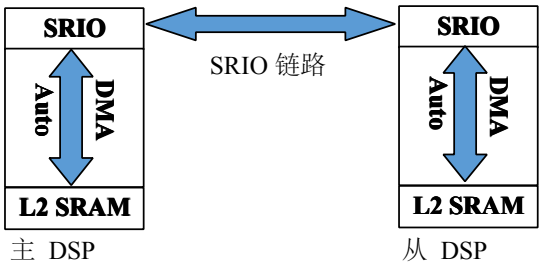


图 4.6 SRIO 传输时的自动 DMA 操作

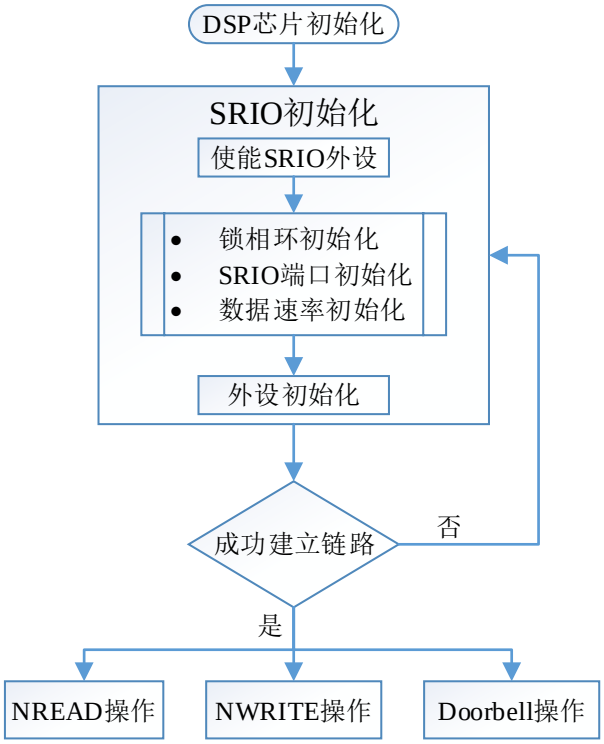


图 4.7 SRIO 基本读写与门铃操作流程

SRIO 初始化、读、写可以用 3 个函数实现：

- SRIO 初始化：`void SRIO_init()`.
- 读函数：`void srio_read(unsigned int src, unsigned int dst, unsigned int len)`.
- 写函数：`void srio_write(unsigned int src, unsigned int dst, unsigned int len)`.

在上述函数中，src 为源地址即发送方 DSP 器件 ID，dst 为目的地址即接收方器件 ID，len 为发送数据的字节数。

SRIO 初始化函数`void SRIO_init()` 主要包含下列操作：

- 使能 SRIO 外设
- 锁相环 (PLL), SRIO 端口 (Ports), 数据速率 (Data Rate) 初始化
- 外设初始化 (设置设备 ID、配置并使能 TX 和 RX 模块)

“使能 SRIO 外设”又可进一步分为如下步骤:

- SRIO 全局使能
- 使能 SRIO 外设存储器映射控制寄存器
- 使能 SRIO 端口 0
- 使能 LSU 模块
- 使能 MAU 模块
- 使能 TXU 模块
- 使能 RXU 模块
- 使能 SRIO 端口 1, 2, 3

附录 A 有 TMS320C6455 DSP 串行 RapidIO 的详细测试程序及注释说明。

第五章　XC5VSX50T 的串行 RapidIO 实现

5.1　Xilinx XC5VSX50T FPGAd 简介

本次测试平台中 FPGA 的型号为 Xilinx Virtex-5 系列中的 XC5VSX50T，该型号的 FPGA 侧重于数字信号处理任务，其可配置逻辑单元 (CLB) 为 120 x 34，最大分布式 RAM 780Kb，288 个 DSP48E Slice，6 个时钟管理模块 (Clock Management Technology，CMT)，12 个 RocketIO 低功耗吉比特收发器 (Gigabit Transceiver with Low Power，GTP)，总 IObank 15 个,，最大用户 IO 数 480 个。高速连接资源主要包括千兆以太网接口、串行 RapidIO 等。[16] 采用 XC5VSX50T 进行 RapidIO 接口设计时，可直接调用 Xilinx 公司的 RapidIO IP 核，大大降低了设计的难度。本次对 XC5VSX50T 的开发采用 Xilinx 公司提供的 ISE Design Suite 14.2，RTL 仿真采用的是 Mentor Graphics 公司的 Modelsim SE 10.1b。

5.2　XC5VSX50T 串行 RapidIO 端点解决方案

RapidIO 器件的实现可以广义地分为端点和交换机。端点向 RapidIO 网络发送事务，从 RapidIO 网络接收事务。交换机根据路由表负责数据包向各个端点的转发，交换机和端点都支持访问系统结构寄存器的维护事务。[17]

为了在通过逻辑 (I/O) 和传输层 IP 上的目标接口和源接口收发用户数据时支持完全兼容的最大载荷操作，Xilinx 厂商根据 RapidIO v1.x 和 v2.x 规范设计了其端点 IP 解决方案。本次测试采用的 IP 核版本为 LogiCORE IP Serial RapidIO v5.6。图5.1所示是 Xilinx 的一套完整的串行 RapidIO 端点 IP 方案，它由物理层核、逻辑和传输层核、缓冲层核和参考设计 4 个部分组成。其中参考设计含时钟、复位及配置部分。

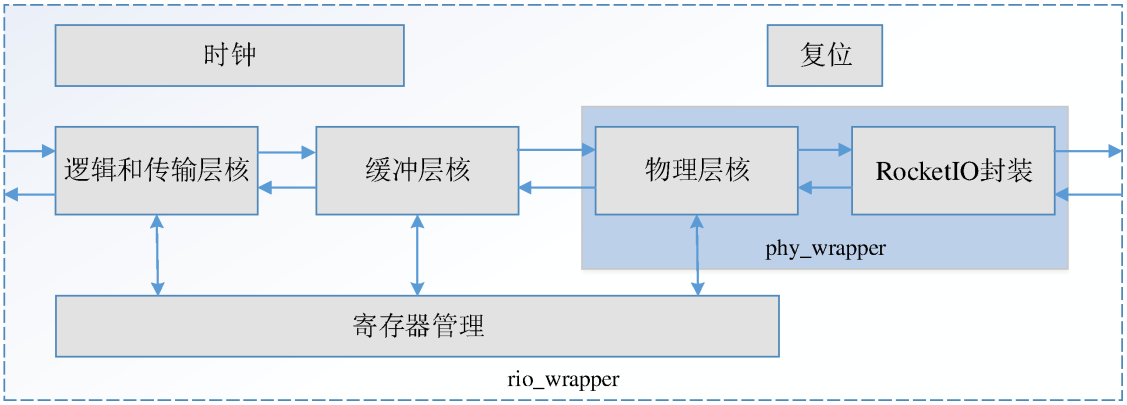


图 5.1 Xilinx 串行 RapidIO 端点解决方案

Xilinx 串行 RapidIO 端点解决方案采用分层的方法设计，可允许用户根据自己的实际需要来集成其中的部分设计功能。例如phy_wrapper 仅集成了串行

RapidIO 物理层核 (PHY core) 和串行发送器，这对于那些只需要实现物理层核功能的用户来是非常有用的。而`rio_wrapper`除集成了`phy_wrapper`之外还包含了逻辑和传输层核、缓冲层核、寄存器管理器参考设计、参考时钟模块和复位模块，这种情况适用于那些需要在设计中集成所有的 RapidIO 端点功能的用户。[18]

5.3 端点 IP 核架构

Xilinx 的串行 RapidIO 解决方案可以描述为由三个核，这三个核使用 RapidIO 的封装模块结合成一个单一的解决方案。`rio_wrapper`封装模块 (rio_wrapper.v) 将串行 RapidIO 端点的各个组件封装在一起。每条数据通道接口使用本地链路协议来提供连续的端到端的数据流。图5.1中给出了各个块在封装模块中的结合框图和各个块之间的数据交互概览。

5.3.1 逻辑和传输层核

逻辑层按照 RapidIO Logical I/O and Common Transport Specification v 1.3 来设计。它被分成若干个模块以完成对发送包和接收包的组包及解包工作。RapidIO 逻辑层 (I/O) 和传输层 (LOGIO) 有如下三种接口：用户接口 (User Interface)、链路接口 (Link Interface) 和维护接口 (Maintenance Interface)。图5.2为 Xilinx RapidIO 逻辑层接口框图。

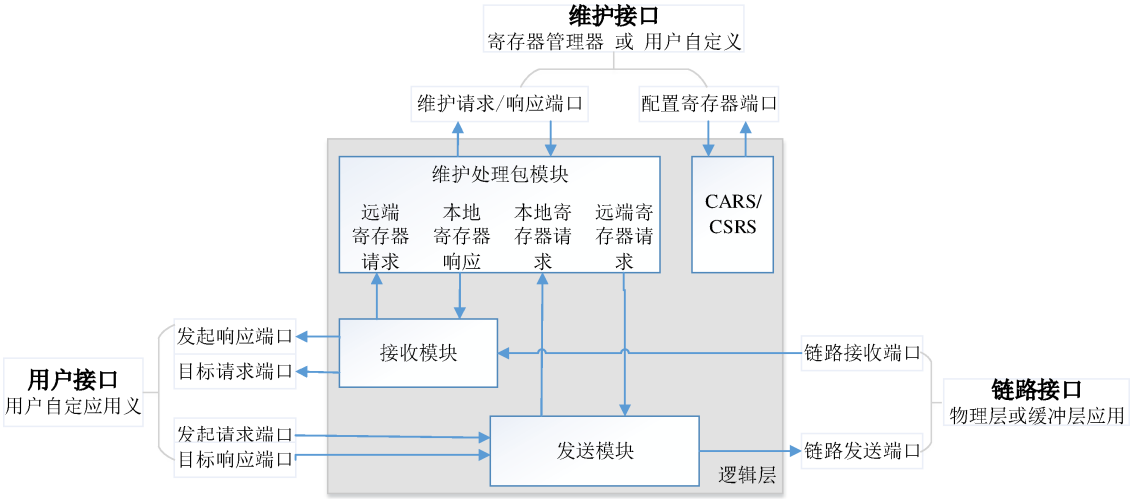


图 5.2 Xilinx RapidIO 逻辑层接口框图

用户接口包含四个端口 (Initiator Request, Initiator Response, Target Request and Target Response)，这四个端口用来生成数据包，或者处理远程端点的请求包。用户也可以通过这些端口初始化本地配置的读写访问，配置相应的 RapidIO 端点器件的配置寄存器。链路接口包含两个端口——接收 (Receive) 和发送 (Transmit) 端口，它与 RapidIO 物理层或缓存应用相连，并被设计为对封装模块外部不可见。维护接口则包含两个端口——维护请求/响应端口 (Maintenance Request/Response)

以及配置寄存器端口 (Configuration Register)，它们用以控制逻辑层中的配置寄存器的读写，以及用户自定义或者与相应物理层相连的所有配置寄存器。

面向 RapidIO 互连网络的远程端点响应数据或请求数据可以用作“发起请求 (Initiator Request)”和“目标响应 (Target Response)”的输入。逻辑层负责将这些事务组装成包，然后由链路发送接口发送至遵循 RapidIO 的物理层器件。链路接收端口接收从 RapidIO 互连网络进来的数据或是包的请求。进来的事务被传送至“发起响应 (Initiator Response)”、“目标请求 (Target Request)”和“维护请求 (Maintenance Request)”端口中的一个。逻辑层将包头部的信息提取出来并将包头进行独立传送。如果包中包含数据，逻辑层则在发送之前将数据进行 64 比特位对齐。[19]

5.3.2　缓冲层核

由 SRIO 物理层产生的缓冲层设计为用于发送和接收数据包的缓冲。为确保可靠地数据包传送和流量控制操作，缓冲块是非常有必要的。Xilinx 提供了可配置的缓冲层解决方案，用户可在系统性能和资源需求之间进行权衡。缓冲层的所有接口在内部连接到 RapidIO 的封装模块，而对封装层之外是不可见的。

缓冲层核被分成三个主要组件：发送块、接收块和管理块。

5.3.3　串行物理层核

串行物理层核 (也称 PHY) 主要用于处理链路训练、初始化和协议，除此之外还包括 CRC 和确认标识符插入到传出的数据包。物理层核还用于多千兆收发器（MGTs）的接口。这些收发器在核中作为一个外部实例以减轻客户使用模型的负担。然而，PHY 与 MGTs 的连接被两个封装模块 (PHY wrapper 和 rio_wrapper) 抽象了。因此，进入 MGTs 和缓冲区的接口对 rio_wrapper 的外部是不可见的。

串行物理层核被分成如图5.3所示的几个主要组件。

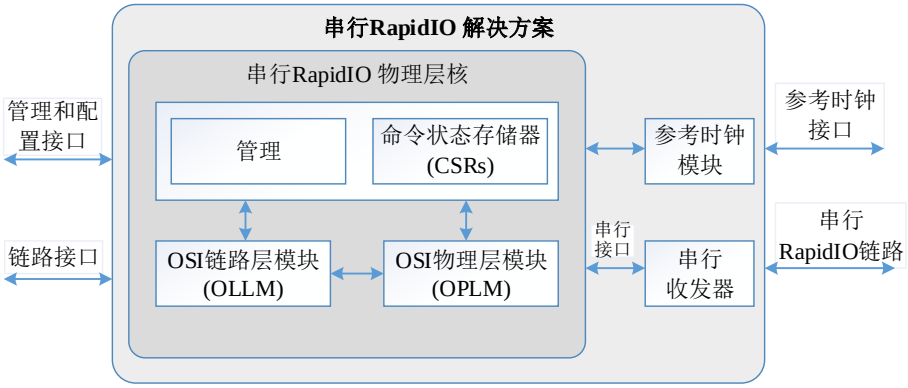


图 5.3 Xilinx RapidIO 串行物理层块框图

串行 RapidIO 物理层核使用两个链路接口连接至缓冲器，一个用来发送，另一个用来接收。缓冲器在串行 RapidIO 物理层内核内部连接至链接层 (OLLM)。

OLLM 负责 CRC 校验码的产生和校验、符号的产生和解码、包交换协议握手和缓冲管理。OLLM 模块进一步被分成两个主要模块 (发送模块和接受模块) 来负责发送方和接收方的数据。

当包和控制符号形成后，它们便移至 OPLM 模块内，OPLM 模块又可进一步划分为两个子模块：PCS 层和 PMA 层。OPLM 负责串行/解串、链路初始化和链路训练。OPLM 核心实现 PCS 层逻辑，串行收发器则实现 PMA 层的功能。串行收发器可在核外部使用以增强灵活性。

5.4 串行 RapidIO IP 核的生成和使用

串行 RapidIO IP 核可使用 Xilinx 内核生成器生成并对其参数进行配置。测试中选择了链路宽度为 4 通道 (4x)，通道波特率分别为 3.125 Gbaud 的配置以生成相应的 bit 文件供 FPGA 烧写使用，将 bit 文件烧写到 FPGA 之后等待片刻即可同 DSP 进行串行 RapidIO 通信。

图5.4为 Modelsim 中的仿真波形。从上到下依次为系统时钟、复位信号、各发送/接收通道数字比特波形。

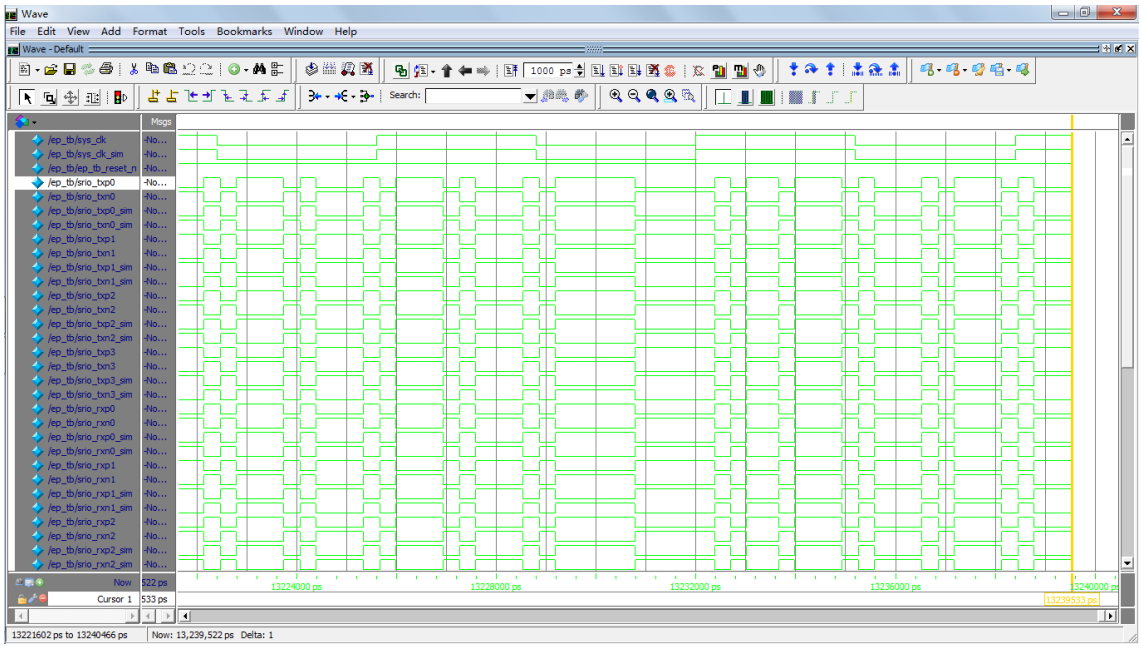


图 5.4 串行 RapidIO 在 XC5VSX50T(FPGA) 中的仿真波形

第六章　串行 RapidIO 传输性能的实测分析

6.1　基于 TMS320C6455 和 XC5VSX50T 的测试平台

通常，RapidIO 系统多采用基于交换机 (Switch) 的拓扑结构，限于实验器材限制，本次测试采用的拓扑结构为 TMS320C6455(DSP) 和 XC5VSX50T(FPGA) 之间通过 4 通道串行 RapidIO 链路直接互连通信。两个调试接口分别为 DSP 的 JTAG 仿真器接口与 FPGA 的 JTAG 下载接口，如图6.1 所示。

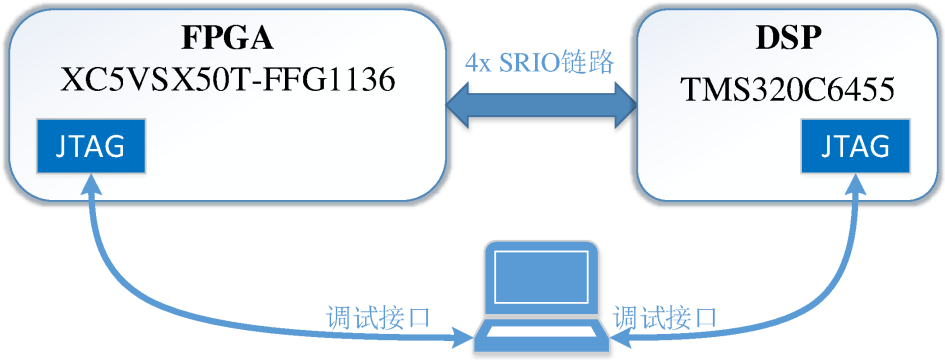


图 6.1 基于 TMS320C6455 和 XC5VSX50T 的测试平台示意图

测试中 TMS320C6455 DSP 做主设备，XC5VSX50T FPGA 中的 RapidIO 接口做从设备，DSP 通过 RapidIO 接口对 FPGA 进行读、写访问。

6.2　串行 RapidIO 的理论传输性能计算

为了对实际测试的传输性能进行对比分析，首先从串行 RapidIO 协议入手计算其理论最大传输速率，然后将实测的传输速率与理论传输速率进行对比分析。以串行 RapidIO 协议 3.125Gbps 在 4x 链路时的理论最大传输速率计算为例：

$$3.125\,Gbps \times 4 = 12.5\,Gbps,\ 12.5\,Gbps \times \frac{8}{10} = 10\,Gbps(\frac{8}{10}\text{为 8B/10B 编解码的效率})$$

$$10\,Gbps \div 8 = 1.25\,GByte/s,\ 1.25\,GB/s \times 90\% = 1.125\,GB/s$$

90% 为数据载荷占包总长度的百分比，计算方法为：典型串行 RapidIO 包协议开销为 28 Bytes，最大有效载荷为 256 Bytes，$\frac{256}{256+28} \approx 90\%$ 把以上计算得出的值作为理论上的最大传输速率。

按照以上方法，在链路传输速率为 2.5 Gbps 和 1.25 Gbps 时，4 通道串行 RapidIO 链路宽度下理论上有效数据最大传输速率分别为 900 MB/s, 450 MB/s。

6.3 串行 RapidIO 传输性能测试分析

为了充分验证串行 RapidIO 的高数据带宽，本次实验测试 TMS320C6455 和 XC5VSX50T 通过串行 RapidIO 协议在 3.125Gbps 4x 模式下实际达到的数据传输性能。TMS320C6455(DSP) 工作频率为 1 GHz，设置其通道模式为 1x/4p(等价于 4x 模式)，链路速率为 3.125 Gbps。FGPA 部分配置为 4x 模式，链路速率为 3.125 Gbps，其参考时钟频率为 156.25 MHz。

基于以上平台，主要完成串行 RapidIO 典型的 Direct IO 模式性能测试，传输方式为 NWRITE 和 NREAD。测试时，采用 C6000 编译手册中提供的`clock()`函数测定传输所用时钟周期数，进而可根据 CPU 工作频率来计算数据传输速率。[20] 传输的数据总量与传输时间之比定义为测定的实际传输速率。由此可得实际传输速率的计算公式为：

$$传输速率(MB/s) = \frac{一次事务传输的字节数(byte) \times \text{CPU 工作频率}(Hz)}{一次事务传输所用时钟周期数 \times 2^{20}}$$

实际测试中传输的字节数设置为 14 组，字节数大小分别为 8, 16, 32, 64, 128, 192, 256, 384, 512, 768, 1024, 2048, 3072, 4096。每组连续测试 20 次取其平均值。

图6.2为本次实际测试中的调试截图。

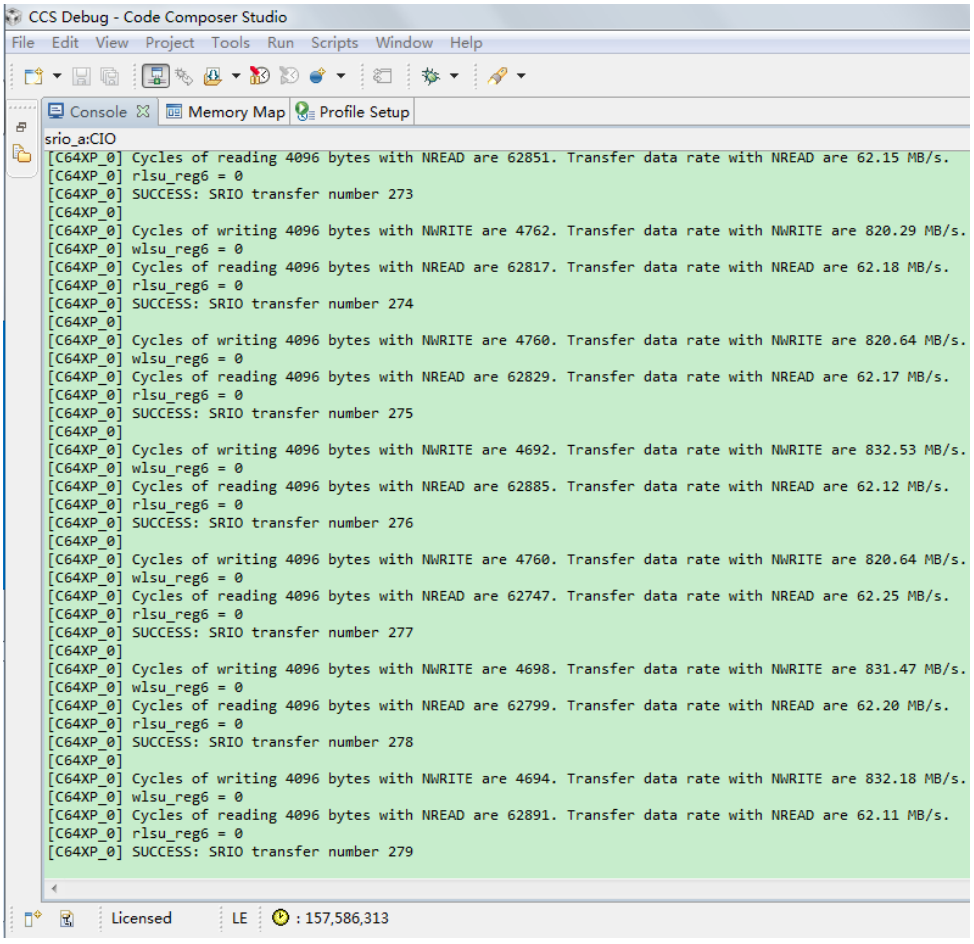


图 6.2 串行 RapidIO 实际传输速率测试

6.3.1　NWRITE 事务传输测试

表6.1详细记录了 NWRITE 事务在 3.125 Gbps 传输速率下传输不同数据载荷大小所用时钟周期数及相应有效数据传输速率。表格中数据已经过统计平均处理。

表 6.1 实际数据传输速率 (MB/s)——NWRITE 事务

数据载荷 (Byte)	所用时钟周期数	实际传输速率 (MB/s)
8	728.1	10.48
16	702.4	21.80
32	732.1	41.68
64	732.5	83.32
128	737.7	165.48
192	816.8	224.17
256	834.6	292.52
384	895.1	409.32
512	895.2	545.70
768	1047.5	699.65
1024	1287.4	758.56
2048	2477.7	788.27
3072	3606.9	812.24
4096	4729.1	826.09

图6.3为测试平台上串行 RapidIO NWRITE 事务在 3.125 Gbps 通道传输速率下 4 通道传输时实测的传输速率曲线图。横坐标显示了一次事务传输的有效数据载荷的字节数 (Byte)，纵坐标为实测的不同条件下 NWRITE 事务的实际传输速率 (MByte/s)。

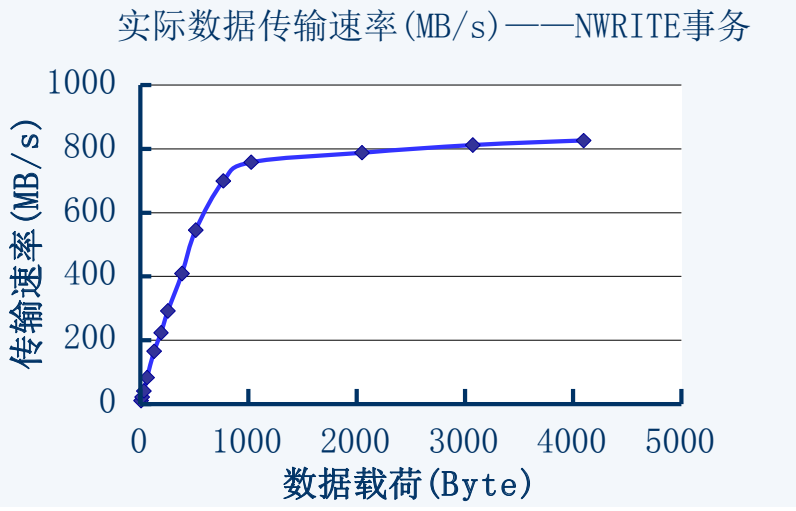


图 6.3 实际数据传输速率 (MB/s)——NWRITE 事务

由表6.1和图6.3可知：

1. TMS320C6455(DSP) 在 3.125 Gbps 传输速率下通过 NWRITE 事务单向传输时，最大的实际数据传输速率为 826.09MB/s。接近理论速度值的 73.4%
2. 数据载荷在 8-256 Bytes 时，NWRITE 事务的传输速率随数据量的增大而增长。
3. 一次 NWRITE 事务传输数据量为一个串行 RapidIO 包的最大数据载荷 (256 Bytes) 时，3.125 Gbps 传输速率下的数据传输速率约为 292.52 MB/s。
4. 一次 NWRITE 事务传输数据达 1024 字节时，速率接近最大传输速率。

6.3.2 NREAD 事务传输测试

表6.2详细记录了 NREAD 事务在 3.125 Gbps 传输速率下传输不同数据载荷大小所用时钟周期数及相应有效数据传输速率。表格中数据已经过统计平均处理。

表 6.2 实际数据传输速率 (MB/s)——NREAD 事务

数据载荷 (Byte)	所用时钟周期数	实际传输速率 (MB/s)
8	2679.9	2.85
16	2736.3	5.58
32	2897.5	10.53
64	3140.2	19.44
128	3532.7	34.56
192	4087.1	44.80
256	5303.9	46.03
384	7478.3	48.97
512	8457.8	57.73
768	12319.6	59.45
1024	16195.5	60.30
2048	31751.8	61.51
3072	47307.9	61.93
4096	62825.4	62.17

图6.4为测试平台上串行 RapidIO NREAD 事务在 3.125 Gbps 通道传输速率下 4 通道传输时实测的传输速率曲线图。横坐标显示了一次事务传输的有效数据载荷的字节数 (Byte)，纵坐标为实测的不同条件下 NWRITE 事务的实际传输速率 (MByte/s)。

由表6.2和图6.4可知:

1. TMS320C6455(DSP) 在 3.125 Gbps 传输速率下通过 NREAD 事务单向传输时，最大的实际数据传输速率为 62.17 MB/s。仅相当于理论速度值的 5.5%
2. 数据载荷在 8-256 Bytes 时，NREAD 事务的传输速率随数据量的增大而增长。
3. 一次 NREAD 事务传输数据量为一个串行 RapidIO 包的最大数据载荷 (256 Bytes) 时，3.125 Gbps 传输速率下的数据传输速率约为 46.03 MB/s。

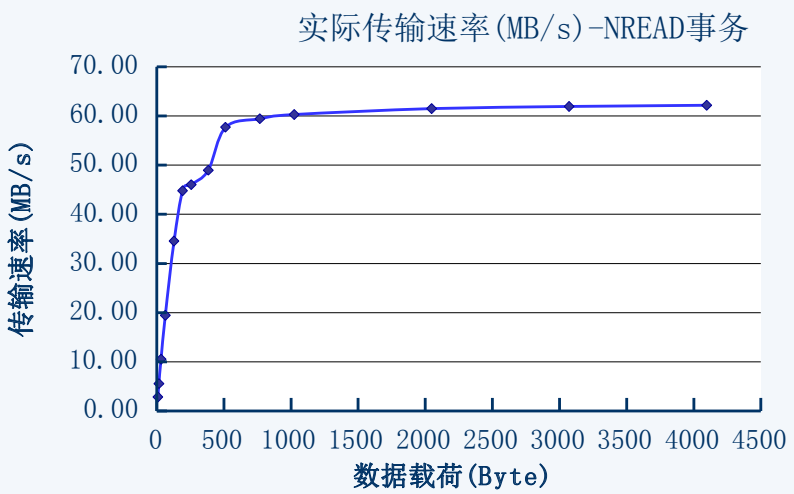


图 6.4 实际数据传输速率 (MB/s)——NREAD 事务

4. 一次 NREAD 事务传输数据量大于 256 Bytes 时，传输速率缓慢上升至最大传输速率 62.17 MB/s。
5. 利用 TMS320C6455 的 RapidIO 接口同 XC5VSX50T(FPGA) 实现 4x NREAD 事务传输时，NREAD 传输的速率只有理论值的 7.5%。因此在使用 DSP 进行串行 RapidIO 数据传输时，尽量使用 NWRITE 事务而不要采用 NREAD 事务。

6.3.3　影响传输效率的因素分析

通过以上的分析，得出无论是 NWRITE 事务还是 NREAD 事务。实际最大传输速率和理论传输速率都有一定的差距，尤其是 NREAD 事务。在实际传输过程存在以下影响传输效率的因素：[21]

1. 发送和接收端的 DSP 和 FPGA 处理数据的时间，包括 DSP 和 FPGA 内部数据的预读取、DMA 传输、串行 RapidIO 包的拆分组合等。
2. 用于协议传输控制的 RapidIO 控制符号、特殊码组字符等的传输。RapidIO 协议传输时，通道中需要传输大量的控制符号和特殊码组字符等，用于链路维护、包定界、包确认、错误报信和错误恢复等。
3. NREAD 事务的响应包带有数据，相比不接收响应的 NWRITE 事务传输速率要慢很多。
4. 测试方法中`clock()`计数的测量误差。
5. TMS320C6455 DSP RapidIO 接口对传输速率的影响。根据 TMS320C6455/54 Digital Signal Processor Silicon Revisions 2.0，1.1 Silicon Errata(SPRZ234i) 说明，在 4x 3.125 Gbps NWRITE 传输时，接口对传输速率有影响。

6.4 传输实测结论

本章对 TMS320C6455 内置 RapidIO 接口的 NWRITE 和 NREAD 事务的实际传输速率进行了测试，对比总结了测试结果，对可能影响传输效率的因素做了分析。实测结果显示，TMS320C6455(DSP) 和 XC5VSX50T(FPGA) 间通过 4 通道串行 RapidIO 链路直接互连通信时，以 NWRITE 事务传输时，最大传输速率约为理论计算带宽的 92%；NREAD 事务传输的最大速率只有理论值的 7.5%。

经过进一步查阅 TI 公司的相关文档，发现在 TMS320C6672 的勘误手册[22]有关串行 RapidIO 的死锁预防中指出：应尽量使用 NWRITE 事务 (PUSH) 而不是 NERAD 事务 (PULL)，结合 DSP 硬件来分析也就不难理解为何 NREAD 事务的传输速度远远低于理论值，而 NWRITE 事务可以相对高效地传输数据。

实现高速数据传输一直是通信界的努力方向，我们很高兴看到在嵌入式互连领域能有一个开放式的国际互连标准，这大大减少了为了处理各种不同私有协议之间交互的开销，加速了系统的开发和减少了硬件成本。但是通过实际测试我们发现实际传输速率离理论值还有很大一段距离，具体的改进将很大程度上依赖于相关工程技术人员。

附录 A　TMS320C6455 DSP 串行 RapidIO 测试程序

srio–c6455.h

```
#ifndef macro_h
#define macro_h

//#define DEBUG

/*  CPU registers                   */
extern cregister volatile unsigned int CSR;  /* Control Status Register  */
extern cregister volatile unsigned int IER;
extern cregister volatile unsigned int ISTP;

/*  GPIO registers                  */
#define  GP_BINTEN          *(volatile unsigned int*)(0x02B00008)
#define  GP_DIR             *(volatile unsigned int*)(0x02B00010)
#define  GP_OUT_DATA        *(volatile unsigned int*)(0x02B00014)
#define  GP_SET_DATA        *(volatile unsigned int*)(0x02B00018)
#define  GP_CLR_DATA        *(volatile unsigned int*)(0x02B0001C)
#define  GP_IN_DATA         *(volatile unsigned int*)(0x02B00020)
#define  GP_SET_RIS_TRIG    *(volatile unsigned int*)(0x02B00024)


/*  interrupt registers             */
#define INTMUX1 *(volatile unsigned int*)(0x01800104)

/* Device State Control Registers*/
#define PERLOCK *(volatile unsigned int*)(0x02AC0004)
#define PERCFG0 *(volatile unsigned int*)(0x02AC0008)
#define PERCFG1 *(volatile unsigned int*)(0x02AC002C)

/* Base address of the EMIFA CE4 */
#define EMIFA_CE4_BASE_ADDR    (volatile long*)(0xC0000000u)
/* Base address of the EMIFA configration register */
#define EMIFA_CONFIGREG_STAT    *(volatile unsigned int*)(0x70000004u)
#define EMIFA_CONFIGREG_CE4CFG  *(volatile unsigned int*)(0x70000088u)



/* EDMA3 registers */
#define PING 0
#define PONG 1
#define EDMA3CC_BASE_ADDR   0x02A00000
#define EDMA3CC_DCHMAP48    *(volatile unsigned int*)(EDMA3CC_BASE_ADDR+0x01C0
    )
#define EDMA3CC_EESRH       *(volatile unsigned int*)(EDMA3CC_BASE_ADDR+0x1034
    )
```

```
#define EDMA3CC_IESR        *(volatile unsigned int*)(EDMA3CC_BASE_ADDR+0x1060)
#define EDMA3CC_IESRH       *(volatile unsigned int*)(EDMA3CC_BASE_ADDR+0x1064)
#define EDMA3CC_ICR         *(volatile unsigned int*)(EDMA3CC_BASE_ADDR+0x1070)
#define EDMA3CC_ICRH        *(volatile unsigned int*)(EDMA3CC_BASE_ADDR+0x1074)
#define EDMA3CC_IEVAL       *(volatile unsigned int*)(EDMA3CC_BASE_ADDR+0x1078)
#define EDMA3CC_DMAQNUM6    *(volatile unsigned int*)(EDMA3CC_BASE_ADDR+0x0258)
#define EDMA3CC_EMCR        *(volatile unsigned int*)(EDMA3CC_BASE_ADDR+0x0308)
#define EDMA3CC_EMCRH       *(volatile unsigned int*)(EDMA3CC_BASE_ADDR+0x030C)
#define EDMA3CC_IPR         *(volatile unsigned int*)(EDMA3CC_BASE_ADDR+0x1068)
struct EDMA3CC_PaRAM
{
    unsigned int OPT;
    volatile long *SRC;
    unsigned int BCNT_ACNT;
    volatile long *DST;
    unsigned int DSTBIDX_SRCBIDX;
    unsigned int BCNTRLD_LINK;
    unsigned int DSTCIDX_SRCCIDX;
    unsigned int RSVD_CCNT;
}*pEDMA3CC_PaRAM;
#define EDMA3CC_PaRAM1  (struct EDMA3CC_PaRAM*)(EDMA3CC_BASE_ADDR+0x4020)
#define EDMA3CC_PaRAM2  (struct EDMA3CC_PaRAM*)(EDMA3CC_BASE_ADDR+0x4040)
#define EDMA3CC_PaRAM3  (struct EDMA3CC_PaRAM*)(EDMA3CC_BASE_ADDR+0x4060)
#define EDMA3TC0_BASE_ADDR  0x02A20000
#define EDMA3TC0_RDRATE *(volatile unsigned int*)(EDMA3TC0_BASE_ADDR+0x0140)

//PLL registers
#define PLLCTL_1    *(volatile unsigned int*)(0x029A0100)   // PLL1 control register
#define PLLM_1      *(volatile unsigned int*)(0x029A0110)   // PLL1 multiplier control register
#define PREDIV_1    *(volatile unsigned int*)(0x029A0114)   // PLL1 pre-divider control register
#define PLLCMD_1    *(volatile unsigned int*)(0x029A0138)   // PLL1 controller command register
#define PLLSTAT_1   *(volatile unsigned int*)(0x029A013C)   // PLL1 controller status register
#define DCHANGE_1   *(volatile unsigned int*)(0x029A0144)   // PLL1 PLLDIV ratio change status register
#define SYSTAT_1    *(volatile unsigned int*)(0x029A0150)   // PLL1 SYSCLK status register
#define PLLDIV4_1   *(volatile unsigned int*)(0x029A0160)   // PLL1 controller divider 4 register
```

```
#define PLLDIV5_1   *(volatile unsigned int*)(0x029A0164)   // PLL1 controller
    divider 5 register
79
#define PLLDIV1_2   *(volatile unsigned int*)(0x029C0118)   // PLL2 controller
    divider 1 register
81  #define PLLCMD_2    *(volatile unsigned int*)(0x029C0138)   // PLL2 controller
    command register
    #define PLLSTAT_2   *(volatile unsigned int*)(0x029C013C)   // PLL2 controller
    status register
83  #define DCHANGE_2   *(volatile unsigned int*)(0x029C0144)   // PLL2 PLLDIV
    ratio change status register
    #define SYSTAT_2    *(volatile unsigned int*)(0x029C0150)   // PLL2 SYSCLK
    status register
85


87  /* SRIO */
    #define SRIO_BUF_SIZE   4096
89  volatile unsigned long SRIO_WRITEBUF[SRIO_BUF_SIZE];
    #pragma DATA_SECTION (SRIO_WRITEBUF, "SRIO_LOCALBUF");
91  #pragma DATA_ALIGN(SRIO_WRITEBUF, 8);
    volatile unsigned long SRIO_READBUF[SRIO_BUF_SIZE];
93  #pragma DATA_SECTION (SRIO_READBUF, "SRIO_LOCALBUF");
    #pragma DATA_ALIGN(SRIO_READBUF, 8);
95

    /* data transfer speed test */
97  #define SRIO_SPEED_TEST 1
    #define NREAD_FUNC_OVERHEAD 0
99  #define NREAD_SPEED_TEST    1
    #define NREAD_OVERHEAD 149
101 #define NWRITE_FUNC_OVERHEAD 0
    #define NWRITE_SPEED_TEST   1
103 #define NWRITE_OVERHEAD 148
    //#define srio_3.125G_125M
105 //#define srio_3.125G_156M
    //#define srio_2.5G
107 //#define srio_1.25G
    #define SRIO_TESTTIMES  280
109 #define SRIO_TESTLEN    4096
    #define SRIO_TARGETID   0x00AB
111 #define SRIO_TARGETBUF_ADDR  0x00000000


113


115 #define SRIO_REG_BASEADDR   0x02D00000

117 #define SRIO_PCR    *(volatile unsigned int*)(SRIO_REG_BASEADDR + 0x0004)   //
    Peripheral Control Register
    #define SRIO_PER_SET_CNTL   *(volatile unsigned int*)(SRIO_REG_BASEADDR + 0
    x0020)   // Peripheral Settings Control Register
119 #define SRIO_GBL_EN *(volatile unsigned int*)(SRIO_REG_BASEADDR + 0x0030)   //
    Peripheral Global Enable Register
```

```
#define SRIO_GBL_EN_STAT    *(volatile unsigned int*)(SRIO_REG_BASEADDR + 0x0034)   // Peripheral Global Enable Status Register
#define SRIO_BLK0_EN    *(volatile unsigned int*)(SRIO_REG_BASEADDR + 0x0038)       // Block 0 Enable Register
#define SRIO_BLK1_EN    *(volatile unsigned int*)(SRIO_REG_BASEADDR + 0x0040)       // Block 1 Enable Register
#define SRIO_BLK2_EN    *(volatile unsigned int*)(SRIO_REG_BASEADDR + 0x0048)       // Block 2 Enable Register
#define SRIO_BLK3_EN    *(volatile unsigned int*)(SRIO_REG_BASEADDR + 0x0050)       // Block 3 Enable Register
#define SRIO_BLK4_EN    *(volatile unsigned int*)(SRIO_REG_BASEADDR + 0x0058)       // Block 4 Enable Register
#define SRIO_BLK5_EN    *(volatile unsigned int*)(SRIO_REG_BASEADDR + 0x0060)       // Block 5 Enable Register
#define SRIO_BLK6_EN    *(volatile unsigned int*)(SRIO_REG_BASEADDR + 0x0068)       // Block 6 Enable Register
#define SRIO_BLK7_EN    *(volatile unsigned int*)(SRIO_REG_BASEADDR + 0x0070)       // Block 7 Enable Register
#define SRIO_BLK8_EN    *(volatile unsigned int*)(SRIO_REG_BASEADDR + 0x0078)       // Block 8 Enable Register
#define SRIO_DEVICEID_REG1  *(volatile unsigned int*)(SRIO_REG_BASEADDR + 0x0080)   // RapidIO DEVICEID1 Register
#define SRIO_DEVICEID_REG2  *(volatile unsigned int*)(SRIO_REG_BASEADDR + 0x0084)   // RapidIO DEVICEID2 Register
#define SRIO_SERDES_CFGRX0_CNTL *(volatile unsigned int*)(SRIO_REG_BASEADDR + 0x0100)   // SERDES Receive Channel Configuration Register 0
#define SRIO_SERDES_CFGRX1_CNTL *(volatile unsigned int*)(SRIO_REG_BASEADDR + 0x0104)   // SERDES Receive Channel Configuration Register 1
#define SRIO_SERDES_CFGRX2_CNTL *(volatile unsigned int*)(SRIO_REG_BASEADDR + 0x0108)   // SERDES Receive Channel Configuration Register 2
#define SRIO_SERDES_CFGRX3_CNTL *(volatile unsigned int*)(SRIO_REG_BASEADDR + 0x010C)   // SERDES Receive Channel Configuration Register 3
#define SRIO_SERDES_CFGTX0_CNTL *(volatile unsigned int*)(SRIO_REG_BASEADDR + 0x0110)   // SERDES Transmit Channel Configuration Register 0
#define SRIO_SERDES_CFGTX1_CNTL *(volatile unsigned int*)(SRIO_REG_BASEADDR + 0x0114)   // SERDES Transmit Channel Configuration Register 1
#define SRIO_SERDES_CFGTX2_CNTL *(volatile unsigned int*)(SRIO_REG_BASEADDR + 0x0118)   // SERDES Transmit Channel Configuration Register 2
#define SRIO_SERDES_CFGTX3_CNTL *(volatile unsigned int*)(SRIO_REG_BASEADDR + 0x011C)   // SERDES Transmit Channel Configuration Register 3
#define SRIO_SERDES_CFG0_CNTL   *(volatile unsigned int*)(SRIO_REG_BASEADDR + 0x0120)   // SERDES Macro Configuration Register 0
#define SRIO_SERDES_CFG1_CNTL   *(volatile unsigned int*)(SRIO_REG_BASEADDR + 0x0124)   // SERDES Macro Configuration Register 1
#define SRIO_SERDES_CFG2_CNTL   *(volatile unsigned int*)(SRIO_REG_BASEADDR + 0x0128)   // SERDES Macro Configuration Register 2
#define SRIO_SERDES_CFG3_CNTL   *(volatile unsigned int*)(SRIO_REG_BASEADDR + 0x012C)   // SERDES Macro Configuration Register 3
#define SRIO_LSU1_REG0  *(volatile unsigned int*)(SRIO_REG_BASEADDR + 0x0400)       // LSU1 Control Register 0
#define SRIO_LSU1_REG1  *(volatile unsigned int*)(SRIO_REG_BASEADDR + 0x0404)       // LSU1 Control Register 1
```

```
#define SRIO_LSU1_REG2  *(volatile unsigned int*)(SRIO_REG_BASEADDR + 0x0408)
       // LSU1 Control Register 2
#define SRIO_LSU1_REG3  *(volatile unsigned int*)(SRIO_REG_BASEADDR + 0x040C)
       // LSU1 Control Register 3
#define SRIO_LSU1_REG4  *(volatile unsigned int*)(SRIO_REG_BASEADDR + 0x0410)
       // LSU1 Control Register 4
#define SRIO_LSU1_REG5  *(volatile unsigned int*)(SRIO_REG_BASEADDR + 0x0414)
       // LSU1 Control Register 5
#define SRIO_LSU1_REG6  *(volatile unsigned int*)(SRIO_REG_BASEADDR + 0x0418)
       // LSU1 Control Register 6

#define SRIO_SP_IP_MODE *(volatile unsigned int*)(SRIO_REG_BASEADDR + 0x12004)
      // Port IP Mode CSR
#define SRIO_DEV_ID     *(volatile unsigned int*)(SRIO_REG_BASEADDR + 0x1000)
       // Device Identity CAR
#define SRIO_DEV_INFO   *(volatile unsigned int*)(SRIO_REG_BASEADDR + 0x1004)
       // Device Information CAR
#define SRIO_ASBLY_ID   *(volatile unsigned int*)(SRIO_REG_BASEADDR + 0x1008)
       // Assembly Identity CAR
#define SRIO_ASBLY_INFO *(volatile unsigned int*)(SRIO_REG_BASEADDR + 0x100C)
       // Assembly Information CAR
#define SRIO_PE_FEAT    *(volatile unsigned int*)(SRIO_REG_BASEADDR + 0x1010)
       // Processing Element Features CAR
#define SRIO_SRC_OP     *(volatile unsigned int*)(SRIO_REG_BASEADDR + 0x1018)
       // Source Operations CAR
#define SRIO_DEST_OP    *(volatile unsigned int*)(SRIO_REG_BASEADDR + 0x101C)
       // Destination Operations CAR
#define SRIO_PE_LL_CTL  *(volatile unsigned int*)(SRIO_REG_BASEADDR + 0x104C)
       // Processing Element Logical Layer Control CSR
#define SRIO_LCL_CFG_HBAR   *(volatile unsigned int*)(SRIO_REG_BASEADDR + 0
    x1058)   // Local Configuration Space Base Address 0 CSR
#define SRIO_LCL_CFG_BAR    *(volatile unsigned int*)(SRIO_REG_BASEADDR + 0
    x105C)   // Local Configuration Space Base Address 1 CSR
#define SRIO_BASE_ID    *(volatile unsigned int*)(SRIO_REG_BASEADDR + 0x1060)
       // Base Device ID CSR
#define SRIO_HOST_BASE_ID_LOCK  *(volatile unsigned int*)(SRIO_REG_BASEADDR +
    0x1068)   // Host Base Device ID Lock CSR
#define SRIO_COMP_TAG   *(volatile unsigned int*)(SRIO_REG_BASEADDR + 0x106C)
       // Component Tag CSR

#define SRIO_SP_IP_DISCOVERY_TIMER  *(volatile unsigned int*)(
    SRIO_REG_BASEADDR + 0x12000)  // Port IP Discovery Timer in 4x mode
#define SRIO_IP_PRESCAL *(volatile unsigned int*)(SRIO_REG_BASEADDR + 0x12008)
      // Port IP Prescaler Register
#define SRIO_SP0_SILENCE_TIMER      *(volatile unsigned int*)(
    SRIO_REG_BASEADDR + 0x14008)  // Port 0 Silence Timer Register
#define SRIO_SP_LT_CTL  *(volatile unsigned int*)(SRIO_REG_BASEADDR + 0x1120)
       // Port Link Time-Out Control CSR
#define SRIO_SP_RT_CTL  *(volatile unsigned int*)(SRIO_REG_BASEADDR + 0x1124)
       // Port Response Time-Out Control CSR
#define SRIO_SP_GEN_CTL *(volatile unsigned int*)(SRIO_REG_BASEADDR + 0x113C)
       // Port General Control CSR
```

```
173
#define SRIO_SP0_CTL    *(volatile unsigned int*)(SRIO_REG_BASEADDR + 0x115C)
      // Port 0 Control CSR
175 #define SRIO_ERR_DET    *(volatile unsigned int*)(SRIO_REG_BASEADDR + 0x2008)
      // Logical/Transport Layer Error Detect CSR
#define SRIO_ERR_EN *(volatile unsigned int*)(SRIO_REG_BASEADDR + 0x200C)   //
     Logical/Transport Layer Error Enable CSR
177 #define SRIO_H_ADDR_CAPT    *(volatile unsigned int*)(SRIO_REG_BASEADDR + 0
    x2010)   // Logical/Transport Layer High Address Capture CSR
#define SRIO_ADDR_CAPT  *(volatile unsigned int*)(SRIO_REG_BASEADDR + 0x2014)
      // Logical/Transport Layer Address Capture CSR
179 #define SRIO_ID_CAPT    *(volatile unsigned int*)(SRIO_REG_BASEADDR + 0x2018)
      // Logical/Transport Layer Device ID Capture CSR
#define SRIO_CTRL_CAPT      *(volatile unsigned int*)(SRIO_REG_BASEADDR + 0
    x201C)   // Logical/Transport Layer Control Capture CSR
181
#define SRIO_SP_IP_PW_IN_CAPT0  *(volatile unsigned int*)(SRIO_REG_BASEADDR +
    0x12010)  // Port-Write-In Capture CSR Register 0
183 #define SRIO_P0_ERR_STAT        *(volatile unsigned int*)(SRIO_REG_BASEADDR +
    0x1158)   // Port Error and Status CSR


185


187 #endif
```

srio–c6455.c

```
#include "macro.h"
2 #include <stdio.h>
#include <time.h>
4
unsigned int time_start, time_stop, nread_overhead, nwrite_overhead,
    time_overhead, cycles;
6
int init_6455();
8 void SRIO_init();
void srio_read();
10 void srio_write();

12 main() {
    int i, j, k, n;
14  unsigned int srio_testlen;
    unsigned int srio_byte_len[] =
        {8,16,32,64,128,192,256,384,512,768,1024,2048,3072,4096};
16  int* pBUF = (int*) SRIO_WRITEBUF;

18  init_6455(); //initialize C6455

20  memset((void *) SRIO_WRITEBUF, 0, SRIO_BUF_SIZE); // Clear memory
    memset((void *) SRIO_READBUF, 0, SRIO_BUF_SIZE); // Clear memory
22
    i = SRIO_TESTTIMES;
```

```
    printf("Run %d times to analyze the SRIO transfer...... \nplease wait
        ......\n", i);

    // run 1000 times  to analyze the SRIO transfer
    for (i = 0; i < SRIO_TESTTIMES; i++) {
        for (j = 0, pBUF = (int*) SRIO_WRITEBUF; j < SRIO_TESTLEN;
                j++, *(pBUF++) = ((j + i) + ((j + i) << 16)))
            ;
        srio_testlen = srio_byte_len[i/20];
        srio_write(SRIO_WRITEBUF, SRIO_TARGETBUF_ADDR, srio_testlen);

        for (k = 1; k < 1000000; k++)
            ;

        srio_read(SRIO_TARGETBUF_ADDR, SRIO_READBUF, srio_testlen);
        for (k = 1; k < 1000000; k++)
            ;

        if (memcmp(SRIO_WRITEBUF, SRIO_READBUF, SRIO_TESTLEN) == 0)
            printf("SUCCESS: SRIO transfer number %d\n", i);
        else
            printf("FAIL: SRIO transfers number %d\n", i);
    }

}

int init_6455() //initialize C6455
{
    int i;
    int PLLM_val = 20; /* CLKIN1 = 50MHz */
    int PREDIV_val = 1;
    int PLLDIV4_val = 8;
    int PLLDIV5_val = 4;

    CSR &= ~(0x1); //turn off interruput

    /* In PLLCTL, write PLLENSRC = 0 (enable PLLEN bit).*/
    PLLCTL_1 &= ~(0x00000020);
    /* In PLLCTL, write PLLEN = 0 (bypass mode).*/
    PLLCTL_1 &= ~(0x00000001);
    /* Wait 4 cycles of the slowest of PLLOUT or reference clock source (CLKIN
        ).*/
    for (i = 0; i < 100; i++)
        ;
    /*In PLLCTL, write PLLRST = 1 (PLL is reset).*/
    PLLCTL_1 |= 0x00000008;
    /*If necessary, program PREDIV and PLLM.*/
    PLLM_1 = PLLM_val - 1;
    PREDIV_1 = (PREDIV_val - 1) | 0x8000; /* set PLLDIV0 */

    /*If necessary, program PLLDIV1n. Note that you must apply the GO
        operation
```

```
     to change these dividers to new ratios.*/

    /* Check that the GOSTAT bit in PLLSTAT is cleared to show that no GO
     operation is currently in progress.*/
    while ((PLLSTAT_1) & 0x00000001)
        ;

    /* Program the RATIO field in PLLDIVn to the desired new divide-down rate.
     If the RATIO field changed, the PLL controller will flag the change
     in the corresponding bit of DCHANGE.*/
    PLLDIV4_1 = (PLLDIV4_val - 1) | 0x8000; /* set PLLDIV4 */
    PLLDIV5_1 = (PLLDIV5_val - 1) | 0x8000; /* set PLLDIV5 */

    /* Set the GOSET bit in PLLCMD to initiate the GO operation to change
     the divide values and align the SYSCLKs as programmed.*/
    PLLCMD_1 |= 0x00000001;

    /* Read the GOSTAT bit in PLLSTAT to make sure the bit returns to 0
     to indicate that the GO operation has completed.*/
    while ((PLLSTAT_1) & 0x00000001)
        ;

    /* Wait for PLL to properly reset.(128 CLKIN1 cycles).*/
    for (i = 0; i < 1000; i++)
        ;

    /* In PLLCTL, write PLLRST = 0 to bring PLL out of reset.*/PLLCTL_1 &=
            ~(0x00000008);

    /* Wait for PLL to lock (2000 CLKIN1 cycles). */
    for (i = 0; i < 4000; i++)
        ;

    /* In PLLCTL, write PLLEN = 1 to enable PLL mode. */PLLCTL_1 |=
            (0x00000001);

    PERLOCK = 0x0f0a0b00; /* Unlock PERCFG0 through PERLOCK */
    PERCFG0 = 0xC0015555; /* Enable SRIO EMAC, Timers, McBSPs, I2C, GPIO in
        PERCFG0 */
    PERCFG1 = 0x3; /* Enable DDR and EMIFA in PERCFG1 */
    GP_DIR &= 0xfffffffe;
    GP_OUT_DATA |= 0x00000001; //FPGA link_reset

    SRIO_init();

    CSR |= 0x1; //enable intertuput

    return (1);
}

void SRIO_init() //initialize C6455 SRIO
{
```

```
124     int rdata = 0;

126     /* Enabling the SRIO Peripheral */
        SRIO_GBL_EN = 0x00000001; //Glb enable srio
128     SRIO_BLK0_EN = 0x00000001; //MMR_EN
        SRIO_BLK5_EN = 0x00000001; //PORT0_EN
130     SRIO_BLK1_EN = 0x00000001; //LSU_EN
        SRIO_BLK2_EN = 0x00000001; //MAU_EN
132     SRIO_BLK3_EN = 0x00000001; //TXU_EN
        SRIO_BLK4_EN = 0x00000001; //RXU_EN
134     SRIO_BLK6_EN = 0x00000001; //PORT1_EN
        SRIO_BLK7_EN = 0x00000001; //PORT2_EN
136     SRIO_BLK8_EN = 0x00000001; //PORT3_EN

138     /* PLL, Ports, and Data Rate Initializations */
        //full sample rate at 3.125Gbps
140     //SERDES reference clock (RIOCLK) 156.25 MHz
        //MPY = 10      156.25 MHz = ((3.125 Gbps)*(0.5))/MPY
142     SRIO_PER_SET_CNTL = 0x0400026F; //peripheral settings control register
        SRIO_SERDES_CFG0_CNTL = 0x0000000B;
144     SRIO_SERDES_CFG1_CNTL = 0x00000000; //SRIO_SERDES_CFG1_CNTL not used
        SRIO_SERDES_CFG2_CNTL = 0x00000000; //SRIO_SERDES_CFG2_CNTL not used
146     SRIO_SERDES_CFG3_CNTL = 0x00000000; //SRIO_SERDES_CFG3_CNTL not used

148     //enable rx, rate=00 full rate; EQ=0001 adaptive;
        //align=01 Comma alignment enabled;
150     SRIO_SERDES_CFGRX0_CNTL = 0x00081101;
        SRIO_SERDES_CFGRX1_CNTL = 0x00081101;
152     SRIO_SERDES_CFGRX2_CNTL = 0x00081101;
        SRIO_SERDES_CFGRX3_CNTL = 0x00081101;
154
        // enable tx, rate=00 full rate; swing=100:750mv;
156     SRIO_SERDES_CFGTX0_CNTL = 0x00010801;
        SRIO_SERDES_CFGTX1_CNTL = 0x00010801;
158     SRIO_SERDES_CFGTX2_CNTL = 0x00010801;
        SRIO_SERDES_CFGTX3_CNTL = 0x00010801;
160
        SRIO_SP_IP_MODE = 0x0400003F; // Port IP Mode CSR:mltc/rst/pw enable,
            clear
162
        /*
164      * Peripheral Initializations
         */
166     //Set Device ID Registers
        SRIO_DEVICEID_REG1 = 0; // id-16b=0000, id-08b=00
168     SRIO_DEVICEID_REG2 = 0; // id-16b=0000, id-08b=00

170     SRIO_DEV_ID = 0x00000030; // id=0000, ti=0x0030
        SRIO_DEV_INFO = 0x00000000; // 0
172     SRIO_ASBLY_ID = 0x00000030; // ti=0x0030
        SRIO_ASBLY_INFO = 0x00000100; // 0x0000, next ext=0x0100
174     SRIO_PE_FEAT = 0x20000009; // proc, bu ext, 8-bit ID, 34-bit addr
```

```
    SRIO_SRC_OP = 0x0000FDF4; // all
    SRIO_DEST_OP = 0x0000FC04; // all except atomic
    SRIO_PE_LL_CTL = 0x00000001; // 34-bit addr
    SRIO_LCL_CFG_HBAR = 0x00000000; // 0
    SRIO_LCL_CFG_BAR = 0x00000000; // 0
    SRIO_BASE_ID = 0x00000000; // 16b-id=0000, 08b-id=00
    SRIO_HOST_BASE_ID_LOCK = 0x00000000; // id=0000, lock
    SRIO_COMP_TAG = 0x00000000; // not touched
    SRIO_SP_IP_DISCOVERY_TIMER = 0x90000000; // 0, short cycles for sim
    SRIO_IP_PRESCAL = 0x00000018; // srv_clk prescalar=0x18 (250MHz)
    SRIO_SP0_SILENCE_TIMER = 0x20000000;
    SRIO_SP0_CTL = 0x00600001; // enable i/o;/////////////
    SRIO_SP_LT_CTL = 0xFFFFFF00; // long time
    SRIO_SP_RT_CTL = 0xFFFFFF00; // long time
    SRIO_SP_GEN_CTL = 0xC0000000; // host, master, undiscovered
    SRIO_ERR_DET = 0x00000000; // clear
    SRIO_ERR_EN = 0x00000000; // disable
    SRIO_H_ADDR_CAPT = 0x00000000; // clear
    SRIO_ADDR_CAPT = 0x00000000; // clear
    SRIO_ID_CAPT = 0x00000000; // clear
    SRIO_CTRL_CAPT = 0x00000000; // clear
    SRIO_SP_IP_PW_IN_CAPT0 = 0x00000000; // clear

    SRIO_PER_SET_CNTL = 0x0500026F; // bootcmpl=1
    //INIT_WAIT wait for lane initialization
    // polling SRIO_MAC's port_ok bit
    rdata = SRIO_P0_ERR_STAT;
    while ((rdata & 0x00000002) != 0x00000002) {
        rdata = SRIO_P0_ERR_STAT;
    }

    //Assert the PEREN bit to enable logical layer data flow
    SRIO_PCR = 0x00000004; // peren
}

void srio_read(unsigned int src, unsigned int dst, unsigned int len) {
    int rdata = 0;

    /* Create an LSU configuration */
    SRIO_LSU1_REG0 = 0;
    SRIO_LSU1_REG1 = src;
    /*This value is used in conjunction with BYTE_COUNT to create a 64-bit
       aligned
     RapidIO packet header address.*/

    SRIO_LSU1_REG2 = dst;/* 32-bit DSP byte address for the source of the LSU
       transaction */
    SRIO_LSU1_REG3 = len;/* Number of data bytes to read or write, up to 4K
       bytes */
    SRIO_LSU1_REG4 = 0x0000AB00; //outportID=0,priority=0,xambs=0,idsize=0 8
       bitid,DestID=0x00AB,Interrupt Req=0

```

```
#if NREAD_FUNC_OVERHEAD || NREAD_SPEED_TEST
    time_start = clock();
#endif
#if NREAD_FUNC_OVERHEAD && (!NREAD_SPEED_TEST)
    time_stop = clock();
    nread_overhead = time_stop - time_start;
    printf("NREAD with clock() overhead are %d cycles.\n", nread_overhead);
#endif
    SRIO_LSU1_REG5 = 0x00000024; //Drbll Info=0,Hop Count=0,Packet Type=0x24
        NREAD

    /* Wait for the completion of transfer */
    do {
        rdata = SRIO_LSU1_REG6;
    } while (rdata & 0x00000001);
#if NREAD_SPEED_TEST
    time_stop = clock();
    if(0 == rdata) {
        cycles = time_stop - time_start - NREAD_OVERHEAD;
        printf("\nCycles of reading %d bytes with NREAD are %d. ", len, cycles
            );
        printf("Transfer data rate with NREAD are %.2f MB/s.\n", (float)(len
            *953.67)/cycles);
    } else printf("\nTransfer error. I will give up this test.\n");
#endif
    printf("rlsu_reg6 = %d\n", rdata);
}

void srio_write(unsigned int src, unsigned int dst, unsigned int len) {
    int rdata = 0;

    /* Create an LSU configuration */
    SRIO_LSU1_REG0 = 0;
    SRIO_LSU1_REG1 = dst;
    SRIO_LSU1_REG2 = src;
    SRIO_LSU1_REG3 = len;
    SRIO_LSU1_REG4 = 0x0000AB00; //outportID=0,priority=1,xambs=0,idsize=0 8
        bitid,DestID=0x00AB,Interrupt Req=0

#if NWRITE_FUNC_OVERHEAD || NWRITE_SPEED_TEST
    time_start = clock();
#endif
#if NWRITE_FUNC_OVERHEAD && (!NWRITE_SPEED_TEST)
    time_stop = clock();
    nwrite_overhead = time_stop - time_start;
    printf("NWRITE with clock() overhead are %d cycles.\n", nwrite_overhead);
#endif
    SRIO_LSU1_REG5 = 0x00000054; //Drbll Info=0,Hop Count=0,Packet Type=0x54
        NWRITE

    /* Wait for the completion of transfer */
    do {
```

```
270         rdata = SRIO_LSU1_REG6;
    } while (rdata & 0x00000001);
272 #if NWRITE_SPEED_TEST
    time_stop = clock();
274     if(0 == rdata) {
        cycles = time_stop - time_start - NWRITE_OVERHEAD;
276         printf("\nCycles of writing %d bytes with NWRITE are %d. ", len,
            cycles);
        printf("Transfer data rate with NWRITE are %.2f MB/s.\n", (float)(len
            *953.67)/cycles);
278     } else printf("\nTransfer error. I will give up this test.\n");
#endif
280     printf("wlsu_reg6 = %d  ", rdata);
}
```

致　谢

毕业论文暂告收尾，这也意味着我在西安电子科技大学的四年的学习生活既将结束。回首既往，自己一生最宝贵的时光能于这样的校园之中，能在众多学富五车、才华横溢的老师们的熏陶下度过，实是荣幸之极。在这四年的时间里，我在学习上和思想上都受益匪浅。这除了自身努力外，与各位老师、同学和朋友的关心、支持和鼓励是分不开的。

论文的写作是艰辛而又富有挑战的，一方面需要了解 RapidIO 协议，另一方面还需要熟悉 DSP 及 FPGA 两大硬件平台并能对其进行编程调试。短短的几个月里，在杜老师和张师兄的帮助下，我顺利地完成了 RapidIO 系统的测试。强大计算能力与高速数据传输的结合在一定程度上释放了网络通信能力的巨大潜能。而 RapidIO 协议的出现则解决了嵌入式领域的互连问题，使得 DSP 的计算能力和 RapidIO 高速通信结合在了一起。

在撰写论文的这几个月里，我首先要感谢的是我的毕设指导老师杜栓义教授。杜老师在毕设初期给我制定了详细的毕业设计任务书，给我指明了方向，让我明白了该怎样将论文逐步分解为一系列的小课题。任务书的末尾提供的参考资料和文献更是给了我无穷的力量，让我漫步在 RapidIO 协议的殿堂。

感谢实验室的张新硕师兄，在我遇到问题时他总是耐心地教导我，这使我受益颇多。同时还要感谢的是实验室的其他老师和师兄，是你们给了我一个非常好的学习和调试环境。最后要感谢我的家人以及我的朋友们对我的理解、支持、鼓励和帮助，正是因为有了他们，我所做的一切才更有意义；也正是因为有了他们，我才有了追求进步的勇气和信心。

时间的仓促及自身专业水平的不足，整篇论文肯定存在尚未发现的缺点和错误。恳请阅读此篇论文的老师、同学，多予指正，不胜感激！

谨以本文献给我最敬爱的父母亲以及所有关心我的人！

参考文献

[1] Fuller S．RapidIO 嵌入式系统互连 [M]．王勇，林粤伟，吴冰冰，译．北京：电子工业出版社，2006．

[2] 杨卿．RapidIO 高速互联接口的设计研究与应用 [D]．成都：电子科技大学，2009．

[3] Association R T. RapidIO, PCI Express and Gigabit Ethernet Comparison. 2005. http://www.rapidio.org/education/documents/InterconnectComparison_v02.pdf.

[4] 邓豹，赵小冬．基于串行 RapidIO 的嵌入式互连研究 [J]．航空计算技术，2008，3： 34．

[5] 李琼，郭御风，刘光明，等．I/O 互联技术及体系结构的研究与发展 [J]．计算机工程，2006，32 (12)：93–95．

[6] 梁小虎，王乐，张亚棣．高速串行总线 RapidIO 与 PCI Express 协议分析比较 [J]．航空计算技术，2010，3：035．

[7] Bouvier D. RapidIO: The Interconnect Architecture for High Performance Embedded Systems. 2003. http://www.rapidio.org/zdata/techwhitepaper_rev3.pdf.

[8] Association R T. RapidIO Specifications. 2013. http://www.rapidio.org/specs/current.

[9] Association R T. RapidIO Specifications 2.2. 2013. http://www.rapidio.org/specs/current/RapidIO_Rev_2.2_Specification.zip.

[10] Association R T. RapidIO Interconnection Specifications: Common Transport Specification. 2013. http://www.rapidio.org/specs/current/RapidIO_Rev_2.2_Specification.zip.

[11] Association R T. RapidIO Interconnection Specifications: LP-Serial Physical Layer Specification. 2013. http://www.rapidio.org/specs/current/RapidIO_Rev_2.2_Specification.zip.

[12] 张娟，苏海冰，吴钦章．基于多处理器的高速 RapidIO [J]．计算机工程，2010，36 (18)：2–5．

[13] 黄克武，吴海洲．基于 TMS320C6455 的高速 SRIO 接口设计 [J]．电子测量技术，2008，31：143–146．

[14] Instruments T. TMS320C645x DSP Serial RapidIO User's Guide (Rev. E). 2009. http://www.ti.com/lit/ug/spru976e/spru976e.pdf.

[15] 冯华亮. 串行 RapidIO: 高性能嵌入式互连技术 [J]. 今日电子, 2007 (9): 78–80.

[16] Xilinx．Virtex-5 系列概述．2009．http://www.seeddsp.com/download.php?id=439.

[17] 吴海燕．基于 RapidIO 总线的信号处理平台设计 [D]．成都：电子科技大学，2009．

[18] Xilinx. LogiCORE IP Serial RapidIO v5.6. 2011. http://www.xilinx.com/support/documentation/ip_documentation/srio_ug503.pdf.

[19] 史卫民，施春辉，柴小丽，等．基于 FPGA 的 RapidIO-FC 转接桥设计 [J]．计算机工程，2010，36 (19)．

[20] Instruments T. TMS320C6000 Optimizing C Compiler Tutorial. 2009. http://www.ti.com/lit/ug/spru425a/spru425a.pdf.

[21] 邓豹，赵小冬．基于 TMS320C6455 DSP 的 Serial RapidIO 传输性能测试研究 [M]．2009．

[22] Instruments T. TMS320C6672 Multicore Fixed and Floating-Point Digital Signal Processor. 2009. http://www.ti.com/lit/er/sprz335f/sprz335f.pdf.