摘 要

虚拟存储技术和Cache是计算机存储系统的重要组成部分，它对操作系统的实现提供了硬件上的支持，近年来随着云计算这一类技术的广泛应用，虚拟存储技术的研究日趋活跃。本文以虚拟存储系统的实现为课题，重点研究了虚拟存储器控制芯片以及Cache的Verilog实现，并实现了以组成完整计算机系统的MIPS架构的CPU，在FPGA芯片上运行。主要研究内容分为四个部分。

**关键词：**虚拟存储器，高速缓存，Cache，内存管理单元，Verilog，FPGA

ABSTRACT

Virtual memory and cache are two crucial components of computer storage system, it provides hardware support for the operating systems. With the increasing popularity of technologies such as cloud computing, researches on the virtual memory is on the rise. Topic of this dissertation is the implementation of virtual memory, it focus on implementing virtual memory and cache with Verilog, plus a MIPS CPU. The whole system is tested on an FPGA chip. The dissertation consists of four parts.

**Keywords: v**irtual memory, cache, memory management unit, Verilog, FPGA

目 录

[第一章 绪 论 1](#_Toc1034576787)

[1.1 研究工作的背景与意义 1](#_Toc2128488811)

[1.2 国内外研究历史与现状 1](#_Toc1559690402)

[1.3 本文的主要贡献与创新 2](#_Toc464895308)

[1.4 本论文的结构安排 2](#_Toc233756423)

[第二章 存储系统基本结构 3](#_Toc796458119)

[2.1 内存管理单元 3](#_Toc1088951966)

[2.1.1 地址翻译 3](#_Toc880683266)

[2.1.2 翻译缓存 4](#_Toc1233878998)

[2.1.3 进程间的内存隔离 5](#_Toc175255580)

[2.1.4 多级页表 6](#_Toc1278038612)

[2.1.5 MIPS的虚拟存储系统 6](#_Toc454171841)

[2.2 高速缓存 10](#_Toc940069706)

[2.2.1 映射方法 11](#_Toc444388896)

[2.2.2 替换算法和写策略 13](#_Toc1651225343)

[2.2.3 虚拟存储器中的高速缓存 14](#_Toc2097297419)

[2.3 本章小结 15](#_Toc716833810)

[第三章 系统的实现 16](#_Toc1591489736)

[3.1 系统整体结构及开发环境 16](#_Toc1980172434)

[3.1.1 Verilog HDL 16](#_Toc970657892)

[3.1.3 模块层次结构 17](#_Toc77967461)

[3.3 MMU模块的实现 18](#_Toc2111051605)

[3.3.1 TLB模块的设计 22](#_Toc1514112901)

[3.4 Cache模块的设计 25](#_Toc858734031)

[3.4.1 数据存储器与标签存储器 26](#_Toc223609842)

[3.4.2 功能的实现 27](#_Toc338317987)

[3.5 MIPS CPU的设计 30](#_Toc2116441638)

[3.5.1 子模块 31](#_Toc1901392019)

[3.5.2 异常处理 32](#_Toc1147624126)

[3.5.3 时序逻辑的实现和状态机 33](#_Toc871060652)

[3.6 模块的结合 34](#_Toc1029976144)

[3.6.1 CPU模块与MMU的结合 34](#_Toc34717265)

[3.6.2 顶层模块 35](#_Toc852065815)

[3.7 本章小结 35](#_Toc442182898)

[第四章 系统的测试 36](#_Toc499612573)

[4.1 内存接口模块 36](#_Toc1085822238)

[4.2 外设控制器 37](#_Toc1238641017)

[4.3 测试的顶层模块 38](#_Toc1588564540)

[4.4 测试程序的编写 38](#_Toc1966505504)

[4.5 测试的结果 39](#_Toc325036367)

[4.6 本章小结 39](#_Toc1763820120)

[第五章 全文总结与展望 40](#_Toc1097060469)

[5.1 全文总结 40](#_Toc779208208)

[5.2 后续工作展望 40](#_Toc556406178)

[致 谢 42](#_Toc1541449365)

[参考文献 43](#_Toc282949904)

[外文资料原文 44](#_Toc506219950)

[外文资料译文 45](#_Toc110799528)

第一章 绪 论

1.1 研究工作的背景与意义

虚拟存储器是计算机内存系统中的重要组成部分，通过对一系列存储设备虚拟化可以使得用户认为可用的内存空间是一段连续的存储空间，而实际的存储地址可能不是连续的，甚至处于不同存储设备上。这带来了许多优点，其中最显著的是使得操作系统中的各个应用程序不需要手动地管理共享的内存空间，并且内存的隔离增加了应用程序运行的安全性。使用内存的交换还可以使可用的内存大于实际内存设备的存储大小。目前主流操作系统都使用了虚拟内存技术来管理内存，虚拟存储器的实现技术也日趋成熟。

1.2 国内外研究历史与现状

在19世纪40至50年代，大型应用程序都必须自行实现一级存储器和二级存储器的管理。60至70年代，计算机的内存设备都较为昂贵。因此虚拟内存的提出不仅是为了扩展能够使用的内存大小，而且还使得应用程序的编写更加容易[[[1]](#endnote-0)]。最早的分页式虚拟存储是60年代曼彻斯特大学在ATLAS计算机上实现的，用于扩展它的主磁芯存储器[[[2]](#endnote-1)]，1961年，伯勒斯公司也独立地发布了第一个使用虚拟存储的商用计算机，但使用的方式是分段[[[3]](#endnote-2)]。

Intel 80286把虚拟内存引入了带保护模式的x86架构中，但其段交换技术在段较大时性能不佳。Intel80386在分段的层次下引入了分页的支持，使得页错误可以在不触发双异常的情况下与其他异常一同触发。但加载段描述符的操作对性能的消耗很大，这使得操作系统更加倾向纯粹的分页存储而不是段页式[[[4]](#endnote-3)]。

近年来对于虚拟内存的实现方法的研究侧重于对提高地址翻译的效率以及降低相应硬件电路所占用的芯片面积。特别实在内存密集型数据处理中，传统的地址翻译机制不能满足性能要求，此时一种新的地址翻译机制使得地址翻译过程和内存访问操作并行地完成[[[5]](#endnote-4)]，大大提高其效率。

对于近年来研究较为活跃的嵌入式设备，由于其对性能的要求很高，使用内存管理单元会出现一些不可预测的异常，而且内存管理单元需要占用相当的芯片面积，因此嵌入式系统的内存管理方式一般不会使用内存管理单元[[[6]](#endnote-5)]，比较常见的方式是使用实时或轻量级操作系统[[[7]](#endnote-6)]。

1.3 本文的主要贡献与创新

本课题是使用Verilog语言在FPGA上实现虚拟存储控制器以及Cache的，不同于传统的直接制造芯片的方式，采用FPGA设计整个方案使开发难度和成本大大降低，并且有利于系统的维护与升级。

1.4 本论文的结构安排

课题的工作任务是实现虚拟存储控制器和Cache，第二章重点讨论虚拟存储器的主要实现方式以及各种实现方法的优缺点，其中重点讨论了MIPS的虚拟内存管理和异常处理机制。第三章先简要地介绍了本课题使用的Verilog语言以及主要的开发方式，之后详细地讨论了本课题中所定义的各个模块及其实现方法。第四章主要讲整个系统的实际调试方法，包括额外模块的定义以及测试程序的编写。

第二章 存储系统基本结构

为了提高执行的效率以及满足操作系统和应用程序的需求，现代计算机采用了虚拟存储器技术。处理器通过向存储器发出访存信号以及地址来完成指令的读取和数据的读写。存储器发出的地址称为虚拟地址，它要先经过内存管理单元（MMU）翻译成物理（或实际）地址，然后再传递给各个层级的存储器完成访存的操作，最终得到访存的结果。这样不仅可以将各个进程的内存区域隔离开，还可以使得虚拟的内存空间大于实际的内存大小。

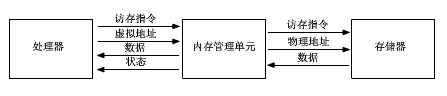


图2-1 虚拟存储器的系统结构

为了提高内存访问的效率，存储器一般由多级缓存构成，其中第一级一般为Cache，可保存最近使用频率较大的一部分内存数据。

2.1 内存管理单元

从虚拟地址到实际地址的转化操作往往是使用内存管理单元来实现的。这样的地址翻译过程使得用户可以把包含主存储器和大容量存储器的装置感觉成单个大容量的空间。当内存管理单元收到地址翻译请求后就会根据翻译的结果作出不同的响应：输出实际地址或触发异常，使得CPU完成不同的操作。

在操作系统中往往会对不同进程所能访问的内存单元做不同的限制，如有些内存区域是只读的，有些区域只允许内核进程访问。为了实现这些需求，MMU同样实现了内存访问的权限管理，对非法的内存访问触发异常。

硬件上，MMU可以是一个单独的芯片，如68851，也可以与处理器集成在一起，使得处理器可以直接输出物理地址。

2.1.1 地址翻译

为了实现虚拟地址到实际地址的翻译，MMU一般使用分页存储管理方式，即将虚拟地址空间和实际空间分成若干个大小相等的块，称为“页（page）”，并以页为单位来完成地址的映射。这样，虚拟地址和物理地址就都分别分成了页号和页内地址两部分，其中虚拟地址的页号被称为虚拟页（virtual page number, VPN），物理地址的页号被称为物理页（physical page number, PPN）。并使用一个页表来记录虚拟页号相对应的物理页号。

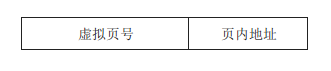


图2-2 虚拟地址与物理地址的组成

通常，一个页表的表项包含虚拟页号、物理页号以及一些用于标记该项状态的服务位（Service bit）：

（1）有效位，用于标示是否可以用该项进行地址转换。在机器初始化的时候由于还没有分配页，所以应将所有的项都标记为无效。

（2）修改位，或脏位，用于指示该项对应的页是否有被修改过。对于使用了大容量存储器来扩大内存的操作系统，当一个被修改过的页被替换时就需要将该页的数据先写回到大容量存储器上。

（3）引用位。用于实现页替换的LRU算法。当页被访问时会被写为1，并且会被定期地写为0。

实际中页表往往是保存在具有缓存系统的存储器中，每个表项相对于页表起始地址的位置就是虚拟页号，这样使得给出一个虚拟地址立即就能知道其表项的地址。此外，每个表项还包含用于标明此项是否有效和用于控制内存访问的状态位。对于一个一般的MMU其地址翻译过程大概如下：

（1）根据虚拟页号找出与所给虚拟地址的页号相符项，检查有效位是否有效，如无效，则说明此虚拟页还没有与之相对应的物理页，所以触发缺页异常；

（2）根据匹配到的项中的服务位来检查此次访问是否合法，若非法，则触发异常；

（3）取出物理页号，将其与虚拟地址中的页内偏移连接得到转化后的物理地址。

必须指出，地址翻译过程对于不同架构其具体的过程是不同的，有些与以上的描述有所差异。

2.1.2 翻译缓存

为了提高地址转化的效率，MMU中保存了页表的一小段缓存，称为翻译后备缓冲区（Translation Lookaside Buffer, TLB）。对于有TLB的MMU其地址翻译过程应为先在TLB中查找，若找到则TLB命中，直接从TLB中取出表项做后续的处理，否则先从内存中的页表里取出相应的项填入TLB。TLB通常采用全相联的缓存，实际中可使用相联存储器（CAM）来实现[[[8]](#endnote-7)]。

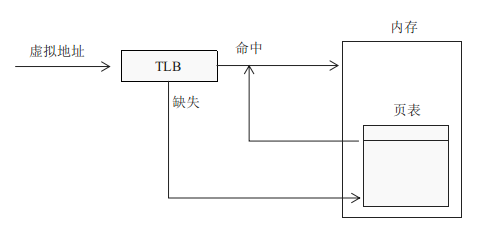


图2-3 翻译后备缓冲区

不同架构对于TLB的实现以及操作方法是有所差异的。在TLB中查找相匹配的项一般都是硬件完成，而TLB不命中时从内存中调出表项则不同：对于x86、PowerPC等架构这部分操作是由硬件来完成的，软件只需要维护位于内存中的页表以及处理页表项无效时的缺页中断，并且将页表的起始（物理）地址写入相应的寄存器即可。MIPS架构则是需要软件维护TLB，当TLB不命中时会触发TLB Miss中断，由软件完成页表项的加载。换句话说，硬件并不认为TLB是页表的缓存，相反，TLB被认为是整个“页表”。此时便不存在缺页中断了，当软件发现位于内存中的表项无效时直接调用缺页处理程序完成物理页框的分配以及将被替换的页写回大容量存储器。当软件在从内存中调入页表项的时候，如果页表的起始地址使用的也是虚地址此时就可能再次触发TLB Miss中断，而当此次中断处理完成后就会回到最初触发中断的地方，由于只更新了页表所在页的地址映射，这次访存还会触发TLB Miss异常。所以在最坏的情况下MIPS的一条访存指令会触发三次异常。但这样做的好处就是使得系统的编写更加灵活。由于加载表项到TLB中的过程是软件来完成的，所以操作系统实现的表项结构可以与TLB的表项有所不同。

当TLB不命中而TLB已满时就需要将其中已有的一个表项替换掉。常用的替换方式有随机替换、最近最少使用、先进先出等。

2.1.3 进程间的内存隔离

在操作系统中不同进程间的内存需要隔离开，即同一虚拟地址在不同的进程中会被翻译成不同的物理地址，不同进程的虚拟地址空间是不同的，以此来达到内存隔离的目的。实现的方法一般有两种：

（1）每个进程有自己单独的页表（对于二级或多级页表，这里应该是页目录），在进程切换时将TLB的所有表项标记为无效，并修改保存页表起始地址的寄存器；

（2）在页表的每个表项中引入进程号用于标记该表项所属于的进程，并通过一个寄存器记录当前进程的进程号。在进行地址翻译时增加表项的进程号与当前进程匹配的要求。

x86使用的是第一种方式，它的页表项中没有用于区分进程的字段，在进程切换时处于TLB中的表项均属于之前的进程，必须先无效化，以便从内存中读取出新进程的页表。TLB中的所有表项总是属于同一个进程。

MIPS采用的是第二种方式，其表项中有一个8位的ASID字段用于标明进程，同时当前进程的进程号位于Context寄存器中。

2.1.4 多级页表

单级页表占用的空间可能会很大，而在一般的应用中很少会将整个页表填满。为了节约页表所占的空间，可以采用多级页表。对于二级页表而言，第一级页表也叫页目录，用于保存第二级页表的起始地址，第二级页表则为页表，保存了相应的物理地址。这样虚拟地址被分成三个部分：页目录号、虚拟页号、页内偏移。在翻译地址的时候先根据页目录号查出页表的起始地址，然后在页表中找到物理页号。类似地，n级页表的前n - 1级保存的是下一级页表的起始地址，最后一级则是物理地址。

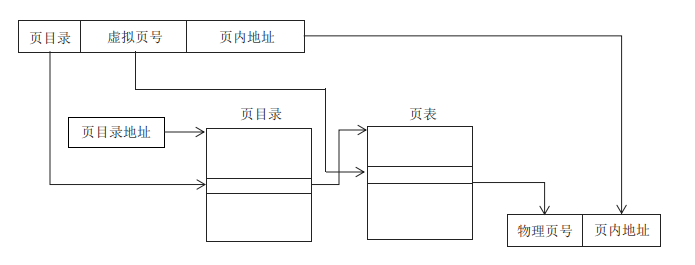


图2-4 二级分页的翻译过程

2.1.5 MIPS的虚拟存储系统

本课题是要实现MIPS32中的MMU，这里重点讨论一下MIPS的内存管理单元具体的工作原理。

2.1.5.1 虚拟地址空间与权限管理

MIPS32把4GB的虚拟内存空间划分为了几个段，如图2-5[[[9]](#endnote-8)]

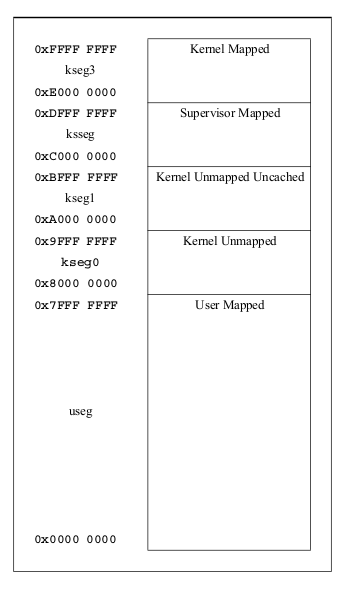


图2-5 虚拟地址空间的划分

对这些数据段[[10]](#footnote-0)的访问权限取决于处理器当前的运行模式。MIPS处理器存在用户模式（User Mode）、管理模式（Supervisor Mode）、内核模式（Kernel Mode）这三种不同的模式，其特权级别是递增的，高特权级可以访问相同或低特权级的段，非法的访问会触发地址错误异常（Address Error Exception）。

不同的数据段的内存访问方式也是不同的。useg、ksseg、kseg3的地址是经过虚拟地址映射的，对这三个段的访问的地址会经过MMU翻译。kseg0段是直接映射到物理地址0x000000-0x1FFFFFFF，在做地址翻译的时候只需将虚拟地址的最高位换成0即得到物理地址。这部分空间用于TLB初始化前的初始化操作，异常处理程序的入口地址也在这段中。

kseg1是用于IO操作的段，这部分地址所指向的通常不是存储器，而是外设，向这部分区域的读写操作就是在控制外设。由于在这段中向一个地址的读和写可能不是普通的内存读写，因此该段不会经过Cache缓存（Uncached）。

2.1.5.2 地址翻译过程

MIPS的地址翻译同样是基于TLB的，其TLB的条目包含EntryHi、EntryLo0、EntryLo1、PageMask四个部分，在MIPS32/64中，这四个部分都是32/64位的。其中虚拟页号位于EntryHi，一个TLB条目可同时记录两条地址映射信息于EntryLo0和EntryLo1中，分别对应于页内地址的最高位为0或1的两种情况。四个部分的具体字段及意义如下[[[11]](#endnote-9)]：

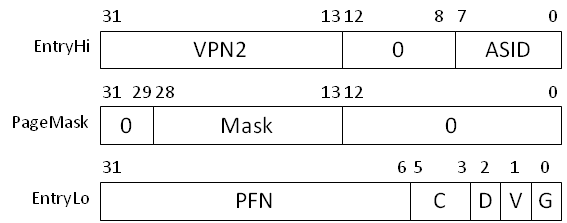


图2-6 TLB条目的组成

VPN2：虚拟页号。其中“2”表示一个条目中一个虚拟页号对应两个物理页号；

ASID：该条目属于的进程号。在地址翻译时必须和当前进程号相等才匹配成功；

Mask：虚拟页号掩码，用于实现可变大小的页；

FPN：物理页号；

C：用于控制Cache对该页的缓存方式，等于不同的值时的作用不一样。MIPS规定这个字段等于2和3时分别代表不允许和允许将该页放入Cache，而其他值不同的厂商可以定义不同的意义；

D：脏位。等于1时表示该页被修改过；

V：有效位；

G：全局位，为1表示该项是全局的，所有进程都可以用这项做地址翻译。如果一个表项的两个G 位都是1那么在查找TLB做地址翻译的时候就不需要匹配ASID。

一个表项总大小是16B，如果按所有页都是1KB来计算，页表总共包含0.5M条记录，总大小就是8MB，并且由于不同进程的表项可用ASID来区分，整个操作系统就只需要一张页表即可，因此页表大小在大多数情况下都是可接受的。

此处条目中的物理页号长度为26位，加上12位的页内地址，物理地址最长可以达到39位，也就是可支持0.5TB大小的内存。然而大多数情况下内存是小于这个值的，这时可直接将物理页号的高位置为零并在地址翻译的时候忽略。

此外，TLB表项中还包含可选字段RI和XI，当为1时就禁止对该页的读取和执行操作，否则会触发TLBRI和TLBXI异常。

TLB表项中的虚拟页号与实际的虚拟页号相比少了最后一位，这是因为TLB的一个表项有两个物理地址的记录，选择哪个记录由虚拟地址的最后一位决定，也就是说两个记录分别对应奇数页号与偶数页号。如对于1KB大小的页，其虚拟地址的12到31位是虚拟页号，0到11是块内地址，12位则是用于选择两个物理地址其中一个的。

页的大小的选择需要同时考虑多个相互冲突的条件[[[12]](#endnote-10)]。如果页太大，由于内存以页为单位进行分配，浪费的空间就会较大；如果页太小，总的页数就越多，页表也就越大。为了能在具体情况中综合分析得出最佳的页大小，MIPS中使用的是大小可变的页。页的大小可以通过Mask字段来控制，虚拟地址的虚拟页在与TLB的每个表项中的虚拟页做匹配前两者都要先与Mask的反码取与操作之后再比较，这就实现了可变长度的虚拟页号和页内地址。MIPS中，页的大小可以取4KB、16KB、64KB、256KB、1MB、4MB、16MB、64MB、256MB，相应的虚拟页号长度分别是20、18、16、14、12、10、8、6、4。

MIPS中与MMU有关的异常有：

（1）TLBS与TLBL：这两个异常都是当TLB中不存在与所给虚拟地址相匹配的项，或虽然存在匹配的项，但有效位为0时触发，具体触发哪个取决于当前的访存操作是读还是写。前一个条件触发的异常又称为TLB Miss（TLB Refill），后一个又称为TLB Invalid，具体触发的是哪个异常只能通过TLB查询指令来获取，一般情况下对这两个异常的处理方式都类似；

（2）TLB Modified：第一次修改页的时候触发。此时D位会被写为1,并且之后的写操作都不会再触发异常。这个异常用于向操作系统发出这个页已经被修改的警告；

（3）ADEL、ADES：这两个为地址错误，都在出现低特权级模式下访问高特权的内存段时触发，并根据访存操作是读取或写入分别选择触发前者还是后者；

（4）TLBRI和TLBXI：分别当RI或XI为1并且此次访存操作是读数据或读指令时触发。

2.1.5.3 MMU操作

MIPS的MMU是与CPU集成在一起的，和MMU相关的寄存器都在协处理器0（Coprocessor 0）中，可通过两个协处理器的寄存器读写指令来对寄存器进行操作。其格式如下：

|  |
| --- |
| MTC0 $reg, $cp0reg  MFC0 $reg, $cp0reg |

协处理器0是一个逻辑处理器，并不是一个实体，可以通过它来完成包含内存管理控制的一些操作。协处理器0中与MMU有关的寄存器有：

Context：上下文寄存器，用于记录TLB异常所涉及到的TLB条目在内存中的地址，其结构如图2-7所示

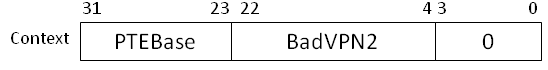


图2-7 Context寄存器

其中PTEBase是页表的起始地址，BadVPN2是触发异常的虚拟页号，当和TLB有关的异常被触发后这个字段就会被写为触发异常的虚拟页号。最低4位总是0,正好对应一个页表条目的大小16B。操作系统可通过修改PTEBase来将页表定位在其他以8MB为单位的地址上。

EntryHi：对应于TLB条目的EntryHi，当执行TLB写入指令时会将此寄存器的值写入相应条目中的EntryHi部分，执行TLB读取时相应条目中的EntryHi部分会被写到此寄存器中。此外，在其他情况下该寄存器的低8位用于记录当前进程的进程号。

EntryLo0、EntryLo1、PageMask：作用与EntryHi寄存器相同，只不过没有其他用处。

Index：TLB条目索引号，在执行TLB写入时用于指明写入的条目的索引号，在执行TLB查找的时候会将匹配的条目的索引写入该寄存器。

Random：随机数寄存器，是只读的。用于在执行随机写入时指明索引。

Wired：若大于0，则TLB条目索引为0到Wired-1之间的条目不能被随机写入。

MIPS提供了用于操作TLB的指令，可完成TLB条目的读取、写入以及查找操作：

TLBWI：按索引写入TLB条目，索引号由Index寄存器给出；

TLBWR：将数据写入一个随机的TLB条目，此时的索引由Random寄存器给出；

TLBR：读取索引为Index的TLB条目；

TLBP：找出与EntryHi寄存器中的虚拟地址和进程号相匹配的TLB条目，并将相匹配项的索引写入Index寄存器。若没有匹配的条目，写入的值就是最大的的TLB条目索引加一；

2.2高速缓存

高速缓存（Cache）是位于处理器和存储器之间的缓冲存储设备，其访存的速度是大于主存的。当处理器发出访存指令和地址时先会在Cache中查找该地址是否存在于缓存中，如果存在就立即访问并返回结果，否则再去访问主存，把主存中相应的数据块调入到Cache中。这个访问Cache的过程完全由硬件来完成，一般情况下对于用户（甚至处理器）来说是不可见的，当然也有例外，如MIPS就提供了操作Cache的指令。

Cache的结构一般是分为多个大小相等的数据块，内存中的块会通过一定的方式映射到Cache中的块，这个块称为行（line）。由于Cache不可能把整个内存都缓存下来，因此这个映射只能是内射（injection）。每个块都有相应的标记位，分别用于标明这个块是否有效、是否被修改过（脏位），以及用于匹配的标签位，因为一个Cache块可对应多个内存中的块。

Cache的依据是局部性原理，即相邻的几次访存操作的地址往往很接近。在这种情况下如果把内存中的一块数据加载到Cache中那么这几次的访存就都只需要访问缓存而不用访问主存，从而提高了程序执行的效率。

2.2.1 映射方法

常用的映射方法有全相联、直接映射和组相联。

2.2.1.1 全相联

全相联是指一个数据块可以放在Cache中的任何块上，并通过标签来区别不同的内存块。整个Cache是一个相联存储器（CAM）。对于全相联，内存地址被分成两部分：标签和块内偏移量，其中偏移量的长度由块的大小决定。在访存操作中，将所给地址的标签部分与Cache中所有的块的标签作比较，如果发现有匹配的块即说明Cache命中，此时只需要对该块做相应的操作。

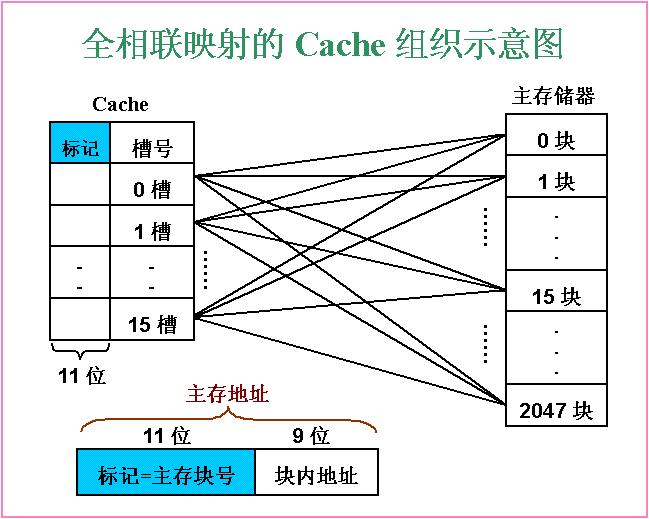


图2-8 全相联Cache

由于数据块可以被放在Cache中的任何地方，因此全相联是自由度最高的Cache，随之带来的问题就是使用的用于判断标签是否匹配的比较器较多，会占用很多电路空间。

2.2.1.2 直接映射

直接映射的Cache将地址分为三部分：标签、索引、块内地址，其对应成数据块直接就是索引所指的块，也就是说内存中的一个块对应一个固定的Cache块，Cache块的个数即为行数。

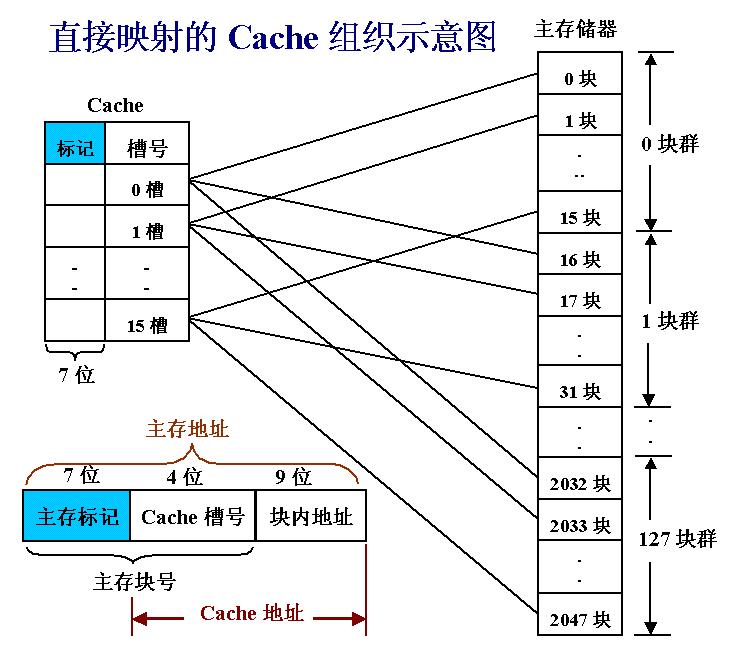


图2-9 直接映射

直接映射的Cache电路设计简单，资源少，但最大的问题就是一个内存行只能对应一个Cache行，如果出现程序交替访问两个标签不同，而索引相同的块的情况Cache将一直不命中，并且会一直从内存中调取块，造成极大的性能损耗。

2.2.1.3 组相联

组相联结合了全相联与直接映射，其地址结构与直接映射相同，但Cache中每个索引有多个块，这些具有相同索引的行组成的一个行组被称为一排（Row），每排具有的块数称为相联性（Associativity）。也就是说，在内存块到Cache中的排时采用直接映射，在一排内采用的是全相联映射。

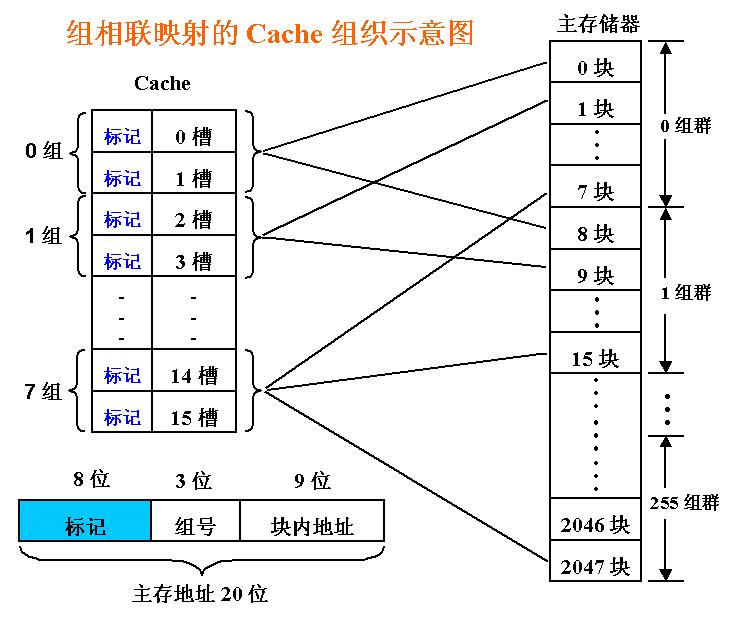


图2-10 组相联

2.2.2 替换算法和写策略

对Cache的读操作没有对内存产生影响，因此不用考虑，而写操作则可能有如下的情况：

（1）若Cache不命中，就先更新主存。此时可以将块加载到Cache中（写分配），也可以不加载。

（2）若命中，则可以采取以下的方式之一：

写直达（write through）：同时将数据写入缓存和主存中。

回写（write back）：只写缓存，并且当要替换该块时并且块被修改过才将该块写回到内存中。这种方法有效地克服了前一种方法的缺陷，如程序包含对同一个内存变量的多次修改时，第一种方法便体现不出Cache的优势。不过，这种方法存在内存和Cache数据一致性的问题，特别是在DMA中，外设需要访问内存来实现和软件的通信，而如果此内存区域还在Cache中内存中的数据就还是旧的值，外设就会读取到错误的数据。克服这种缺陷的有效方法是允许软件对Cache直接操作，当需要使用DMA等来通信时手动地让Cache将数据写回，或者软件规定一个内存块能否经过Cache缓存。

写后写（posted write）：将数据写入Cache，并在Cache能访问总线时将数据写回内存。这种方式是异步的读写，克服了回写可能造成的数据不一致的情况。

当需要将新的块加载进Cache，而所有的块都被占用时就需要将某个已有的块替换出去。一般有三种主要的策略，分别是：

（1）随机选择被替换的行（随机策略）；

（2）替换最先加载的行，这种情况下Cache相当于一个队列（先进先出策略，FIFO）；

（3）替换最近使用次数最少的块（LRU, Lease Recently Used）。这种策略最常用，而且需要在每行的记录中加入额外的字段。

2.2.3 虚拟存储器中的高速缓存

在使用虚拟存储器的系统中，由于存在虚拟地址和物理地址两种地址，Cache可以使用这两个地址中的一个来进行映射。根据映射使用的地址不同，Cache可以分成以下两种[[[13]](#endnote-11)]（图2-8）：

1. 物理地址Cache（physical address cache）：直接使用经MMU转化后的实际地址来映射。由于实际地址是唯一的，所对应的存储单元固定，所以标签也使用实际地址。在这种情况下虚拟存储机制对Cache来说是不可见的。如图2-11所示。

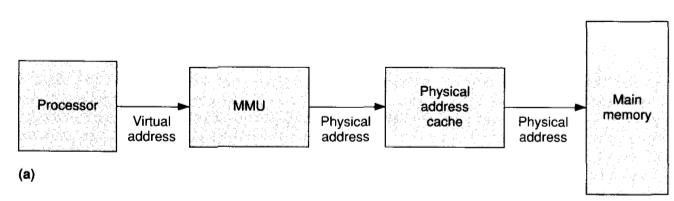


图2-11 物理地址Cache

1. 虚拟地址Cache（virtual address cache）：使用虚拟地址映射，实际地址作标签。由于同一个虚拟地址可以对应多个不同的物理地址，所以要使用物理地址作标签，在访问Cache时要将经MMU转化的物理地址与读出的标签相比较，匹配之后才能说明Cache命中。如图2-12所示。

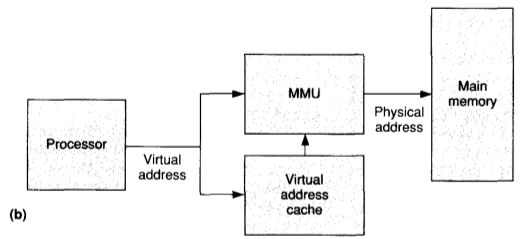


图2-12 虚拟地址Cache

虚拟地址Cache主要的优点就是检查是否命中和地址转换可以同时进行，缺点则是它必须处理虚拟地址重复的问题，因此在读取出块的标签后还需要和物理地址相比较。

2.3 本章小结

本章讨论了虚拟存储器系统和高速缓存的工作原理以及实现方法。对于虚拟存储器，列出了MMU中不同架构对于TLB实现方法的差异以及优缺点，TLB替换算法的基本原理。基本方法就是操作系统负责维护用于地址转换的页表，硬件读取页表并完成地址的转化或者根据情况触发不同的异常。操作系统需要在缺页异常发生时完成物理页的分配，并根据其修改位的值确定是否将该页写回到内存。

详细讨论了MIPS32中的虚拟地址空间划分和地址翻译过程。MIPS只允许访问与当前运行模式特权级相同或较低的特权级的段。地址的翻译过程是采用TLB，并使用ASID字段来区别不同进程的表项。通过页掩码来实现可变大小的页。当TLB不命中时会触发相应的异常，让操作系统从内存中取出页表项。

对于高速缓存，讨论了Cache常用的映射方法、替换算法和写策略以及其相应的优缺点。常用的Cache映射方法有直接映射、全相联和组相联，其中组相联可以看成是前面两者的结合，兼有两者的优点。写策略一般会使用回写或者写后写，因为写直达在程序可能有多次写操作时由于每次都要访存，会消耗大量的性能。

第三章 系统的实现

3.1 系统整体结构及开发环境

本课题需要使用Verilog语言实现一个内存管理单元（MMU）,高速缓存，以及MIPS架构的CPU，并在FPGA芯片上运行。系统的整体结构如图3-1所示。

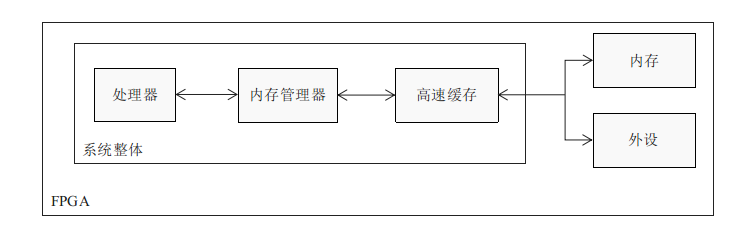


图3-1 整体结构

FPGA开发板采用ALINX系列开发板AX301，其使用的是ALTERA公司的Cyclone IV的FPGA芯片EP4CE6F17C8，其包含6272个逻辑单元、270KB内存，资源可以实现整个系统。开发软件是ALTERA公司配套的Quartus Prime 17.1.0精简版，此软件可以将Verilog代码以及引脚定义生成FPGA程序并下载到开发板上进行实际调试。

仿真测试使用的是Icarus Verilog，它可以直接运行输入的Verilog文件并生成波形图，可以使用gtkwave等软件查看生成的波形图。

3.1.1 Verilog HDL

整个系统采用Verilog语言实现。Verilog HDL，是一种硬件描述语言，用户可以使用类似于软件语言的方式定义各个信号间的逻辑和时序关系。一份Verilog设计包含一个模块的层次体系，一个模块在逻辑上相当于一个芯片，是一个封装好的具有一定功能的实体，并且包含若干个输入口、输出口以及双向口。在一个模块中可以实例化其他已经定义好的模块并使用导线，即wire类型的变量，将它们链接起来组成更大的模块。在设计中还应定义好并指明顶层模块。

Verilog的所有语句中包含可综合和不可综合的语句，可综合代表可以通过该代码生成实际的电路下载到FPGA中。设计的主要部分应采用可综合语句。不可综合的语句通常用于仿真测试，包括延时、系统函数、wait语句等。Verilog提供了一些在仿真中有用的系统函数，经常用到的包括打印函数$display等。

根据描述的方式可分为开关级、门级、数据流级和行为级，本课题主要采用的是后两种描述方式。在实现时序逻辑时采用寄存器传输级描述（Register Transfer Level, RTL），这是硬件描述语言的一种编码风格，可以很方便地对同步逻辑电路建模。在实际电路设计中，很多设计模型，如本课题多次用到的状态机模型，都属于同步逻辑电路。这是一种包含组合逻辑和触发器的电路，典型结构如图3-2所示

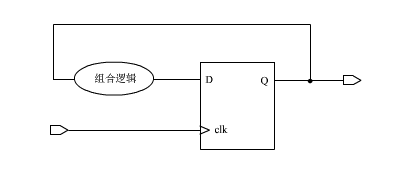


图3-2 同步逻辑电路

在RTL描述中使用数据流级来描述组合逻辑部分，使用行为级描述触发器。对于以上的电路，其RTL描述的Verilog代码大致如下

|  |
| --- |
| assign D = ......;  always @(posedge clk) begin  Q <= D;  end |

3.1.3 模块层次结构

图3-3给出了整个系统包含的主要模块以及层次结构。

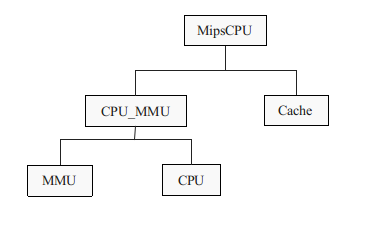


图3-3 模块层次结构

3.3 MMU模块的实现

MMU是本次课题的重点之一。MMU需要实现的功能包括将虚拟地址通过TLB翻译成物理地址或触发异常，以及响应处理器的TLB读写指令。根据这些功能可定义模块的各输入信号及意义如下：

clk：时钟信号输入；

res：复位信号；

addrValid：地址翻译指令，为高时表示应开始执行地址翻译；

vAddr：32位虚地址输入；

mmu\_reg：4位的输入，执行MMU中的寄存器的读写操作时用于指明被操作的寄存器。其编码与MIPS标准一致；

mmu\_accessType：2位的输入，表示本次访存操作的操作类型（读数据、写数据、取指令）；

mmu\_dataIn：32位值输入，写入寄存器的数据；

mmu\_cmd：4位输入，不为0时表示要对MMU执行的操作类型。具体的值与操作的对应关系见表3-1；

以及各个输出信号：

pAddr：32位翻译后的物理地址输出；

db\_io：所给虚拟地址是否位于IO段中；

mmu\_dataOut：读取到的寄存器数据；

mmu\_exception：所发生的MMU异常，其编码与MIPS标准中的异常码一致。

其中输入信号mmu\_cmd用于对MMU完成不同的操作，其操作的编码如表3-1所示。

表3-1 MMU指令编码

|  |  |
| --- | --- |
| mmu\_cmd的值 | 对应的操作 |
| 0 | 无操作 |
| 1 | 读MMU的寄存器 |
| 2 | 写MMU寄存器 |
| 3 | 写TLB，以Index为索引 |
| 4 | 随机写入TLB |
| 5 | 查找与EntryHi相匹配的TLB项 |
| 7 | 读索引为Index的TLB表项 |

该设计中MMU模块包含子模块TLB以及随机数生成模块，TLB模块用于保存所有的TLB条目以及按照所给的虚拟地址找出匹配的TLB条目并输出翻译后的物理地址。地址翻译和条目的读取逻辑部分采用的是组合逻辑，其中地址翻译只要输入一改变立即就能得到物理地址输出以及用于检测异常的信号，读取时只需要将读数据使能信号置高并输入要读取的索引，数据就能立即输出。对TLB条目的写入是时序逻辑，在上一个周期将需要写的条目的索引以及写入的数据准备好后要在下一个周期才能得到输出或完成操作。TLB模块的主要输入与输出口有：

（1）与TLB条目有关的四个寄存器的值的输入和输出，即EntryHi、EntryLo0、EntryLo1、PageMask。这四个寄存器位于MMU模块中，TLB在执行条目的读写以及查询指令时需要读取到或更新这四个寄存器的值；

（2）TLB索引输入，用于在TLB读写时指明被操作的条目的索引号；

（3）与地址翻译和TLB查找有关的状态位输出，包括匹配到的条目的索引matchedIndex、说明是否存在匹配的条目的信号found，匹配到的条目的状态位C、D、V，信号名分别是bitC、bitD、bitV；

（4）虚拟地址输入和物理地址输出。输入的虚拟地址将直接通过一系列组合逻辑最终。为了节省FPGA上的逻辑单元，用于地址翻译和TLB条目查找的虚拟地址用的是同一个逻辑电路，通过不同的操作来输入不同的虚地址。显然，当用于TLB条目查找时物理地址输出是没有意义的；

（5）TLB条目的读取和写入的使能信号re和we；

（6）时钟信号和复位信号。

随机数模块结构比较简单，其端口只包含时钟、复位以及输出。用于生成Random寄存器的值和随机写入TLB时被写入项的索引生成，对Random寄存器的读操作就是对随机数模块的读。

下面讨论MMU模块内部的设计方法。对MMU的寄存器和TLB条目的读写操作都只需要一个时钟周期，在TLB中找出与虚拟地址或EntryHi相匹配的项并取出该项的值这部分使用组合逻辑电路，也只需要一个周期，但如果发生异常则还需要写Context寄存器，并且在TLBP指令中还需要把查找的结果写到Index寄存器中，因此所有的MMU操作中最多需要两个周期，这样MMU应是一个状态机。虽然在实现的时候并没有按状态机的标准编码方法，即定义一个变量来保存当前状态，而是只使用了几个变量来标示当前是否在查找TLB，但严格的来讲，由于MMU在某个周期的输出可能依赖于上一个周期的数据，所以此时实现的MMU仍是一个状态机。

寄存器通过reg类型的变量实现，均为32位的变量，对这些变量的读写使用非阻塞的赋值，这样综合工具就会生成触发器。代码如下：

|  |
| --- |
| reg [31:0] reg\_index;  reg [31:0] reg\_entryHi;  ... |

对于寄存器的读写只需在每个时钟的上升沿判断指令输入mmu\_cmd是否为1或2，然后将信号mmu\_dataIn上的数据写入由mmu\_reg指明的寄存器，或将相应寄存器的值写到mmu\_dataOut上。为了便于仿真测试，在对每个寄存器的读写操作后都加上了$display系统函数来打印读写的信息。

而对于可能需要两个时钟周期才能完成的地址翻译和TLB查找指令，第一个周期先将输入的虚拟地址锁存到虚拟地址锁存器，在地址翻译的第二个周期时判断是否发生了异常，如果有就将虚地址的虚拟页号写入Context寄存器的4到22位；在TLB查询的第二个周期将所得的索引值写入Index寄存器。这里便需要两个变量来分别保存上一个周期是否在做地址翻译或查找TLB条目：

|  |
| --- |
| reg prob, convert; |

在初始化（输入信号res为1）的时候这两个变量都会被写为0，之后在每个时钟信号的上升沿时，当信号addrValid有效时convert会变为1，mmu\_cmd代表的指令是TLB查找时prob变为1，这两个信号为1的状态都只会保持一个周期，下一周期来临时都会再次变为0。在每个周期中如果convert为1并且mmu\_exception的值说明有异常发生，就会将输入的虚拟地址中的虚拟页号写入Context寄存器；如果prob为1，就检查是否有匹配的条目被找到，并将匹配的条目的索引写入Index寄存器。之前提到过，这部分逻辑实际上描述的是一个状态机的状态转移表。这两个变量可能取的值有四种组合情况，因此MMU是一个有四个状态的状态机，根据以上变量的转化条件可以画出其状态转移图如图3-4。

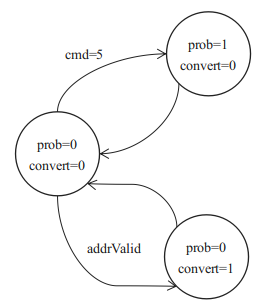


图3-4 MMU的状态图

在进行状态转移的时候由于后续的处理也会依赖之前的输入信号，因此对于地址翻译和TLB查询操作应在第一个周期结束，改变状态的同时将输入信号使用锁存器保存下来。在地址翻译时需要保存虚拟地址、CPU运行状态和内存访问类型，其中后面两者会在下一个时钟周期用于检查和触发异常，这三个寄存器分别为vAddrLatch、cpuModeLatch和accessTypeLatch；TLB查找只需要保存虚拟地址，但此时用于匹配的虚拟页号是由EntryHi寄存器给出而非来自处理器的虚拟地址。

除了以上提到的与时序有关的信号外，其余的都是组合逻辑。这些组合逻辑包含异常信号mmu\_exception的生成、控制是否经过Cache的cachable信号的生成以及TLB的index端口的输入。 异常信号等于0时说明没有异常，其他可能产生的异常有TLBS、TLBL、TLB Modified、ADES、ADEL。对于最后两者，本课题只对用户模式和内核模式进行检查。其具体判断过程为：

（1）首先判断当前CPU的模式，如果为用户模式并且虚拟地址的最高位为1，则异常信号为ADES或ADEL，由访存类型决定，否则继续第（2）步；

（2）判断虚拟地址是否处在映射的区域，如果不是，则异常信号等于0，否则继续第（3）步；

（3）如果TLB缺失或者匹配到的条目被标记为无效，则由访存类型决定异常信号等于TLBL或TLBS，否则继续第（4）步；

（4）如果匹配的TLB条目的D位为1并且访存操作是写，则异常为TLB Mod，否则没有异常。

cachable信号的逻辑则相对简单，如果虚拟地址对应的内存区域是非映射区，则cachable直接等于db\_io取反，后者是虚拟地址处于IO段的标志。如果是映射区，则取决于相应TLB条目的C位：MIPS标准中当C位等于2时表示不经过Cache，等于3则表示经过，其他的值可以根据实现自己赋予含义。本课题的实现中不对其他值定义含义。相应的代码如下：

|  |
| --- |
| if(mapped)  case(cacheC)  3'd3: cachable = 1'b1;  3'd2: cachable = 1'b0;  default: cachable = 1'b0;  endcase  else  cachable = !db\_io; |

TLB的index端口的输入口为信号为tlbWriteIndex，其取值为一个多路选择器：当mmu\_cmd为TLB写入或读取时等于Index寄存器的值，为随机写入时等于Random寄存器的值。

3.3.1 TLB模块的设计

TLB是MMU很重要的一个子模块，它完成了MMU最主要的功能。模块的端口已在前一节给出。TLB模块还包含一个参数ENTRY\_ADDR\_WIDTH（以下简称W），用于指明其TLB条目索引的位数，这样总的条目数N就是1<<W。受使用的FPGA逻辑单元数的限制，在最终实际调试的时候这里的W最大只能取4，即TLB有16个条目。

之前已给出一个条目的组成，一个条目包含四个32位的字段。为了设计的方便，这里用四个寄存器变量来保存，如下：

|  |
| --- |
| reg [31:0] tlb\_entryHi[ENTRY\_COUNT - 1:0];  reg [31:0] tlb\_entryLo0[ENTRY\_COUNT - 1:0];  reg [31:0] tlb\_entryLo1[ENTRY\_COUNT - 1:0];  reg [31:0] tlb\_pageMask[ENTRY\_COUNT - 1:0]; |

TLB条目的读写操作方式与MMU中的寄存器的读写类似，都是使用同步时序电路，写数据使用的索引是index输入。唯一的区别在于读，在读取TLB指令和地址翻译时都需要对TLB进行读取，并且这两个操作不会同时进行。因此为了简化生成的电路，节省逻辑单元，这两个读操作采用同一个地址译码电路，电路的地址输入端用一个多路选择器，信号名为readIndex，通过读使能信号是否有效来确定选择输入的index或匹配到的TLB条目的索引：

|  |
| --- |
| assign readIndex = re ? index : matchedIndex; |

下面讨论TLB条目匹配的实现，匹配逻辑最终需要输出匹配到的条目的索引。由于TLB条目的匹配和一般的相联存储器（CAM）有区别，不是将输入的虚拟页号简单得与每个条目的页号相比较，而是要先与该项的页掩码的反码取与操作再比较，所以不能使用FPGA芯片中提供的CAM而需要手动实现，这使得TLB会使用大量的逻辑单元。条目匹配的电路结构如图3-5所示。

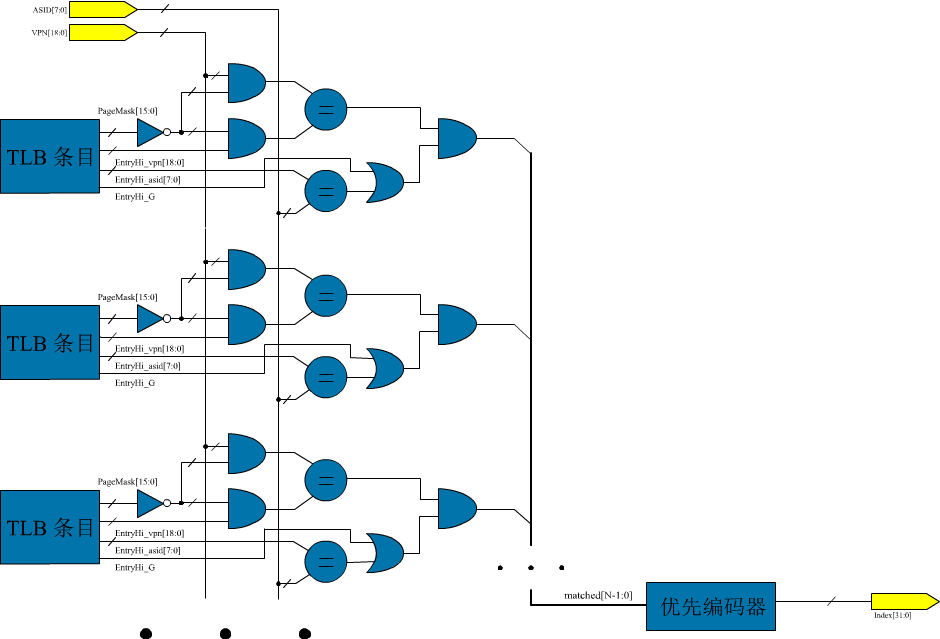


图3-5 匹配TLB条目的电路

所有的条目都需要与虚拟页号以及进程号作比较，每个条目都需要输出一个匹配是否成功的信号，然后将这些信号通过优先编码器转化成索引。在每个条目中，匹配成功需要满足一下两个条件之一：

（1）虚拟地址的页号和表项的页号都与该项的页掩码的反码取与操作后相等；

（2）该项的进程号与EntryHi中的进程号相等或两个G标志位都为1。

实现以上用于匹配TLB的组合逻辑电路使用的是generate语句，先定义一个保存每个条目是否匹配成功的宽度与TLB条目数相同的信号，然后使用该语句为这个信号的各个分量生成一个组合逻辑。这部分代码如下：

|  |
| --- |
| wire [ENTRY\_COUNT - 1:0] matched;  genvar i;  generate  for(i = 0; i < ENTRY\_COUNT; i = i + 1) begin  wire [31:0] entryHi = tlb\_entryHi[i];  wire [31:0] entryLo0 = tlb\_entryLo0[i];  wire [31:0] entryLo1 = tlb\_entryLo1[i];  wire [15:0] mask = tlb\_pageMask[i][28:13];  wire [18:0] tlb\_vpn2 = entryHi[31:13];  wire [18:0] vpn2 = vAddr[31:13];  wire g = entryLo0[0] & entryLo1[0];  assign matched[i] = ((tlb\_vpn2 & ~mask) == (vpn2 & ~mask)) && (g || entryHiIn[7:0] == entryHi[7:0]);  end  endgenerate |

之后，信号matched会被输入优先编码器，编码成匹配成功的条目的索引matchedIndex。另外，由于没有条目匹配和第1个条目匹配时编码器的输出都是0，所以还应定义一个额外的用于指示是否有条目匹配的信号：

|  |
| --- |
| assign found = |matched; |

找出匹配成功的条目的索引后该索引又会被通过readIndex读入从而得到匹配的条目数据，根据这个数据生成翻译出来的物理地址。这部分的逻辑并没有考虑匹配不成功的情况，也就是说当TLB缺失的时候得到的物理地址是没有意义的。为了组合出物理地址先要根据虚拟页号的奇偶性选择出两个物理页号记录中的一个，这依赖于页掩码的取值，因为不同的页大小其虚拟页号的长度不一样，奇偶位的位置也就不一样。同样，在进行地址拼接的时候也需要页掩码，因此先对页掩码进行分析。由于在不止一个地方需要依赖由页掩码确定的页大小，这里先将页的大小进行编码，根据掩码得到编码的值，即把1KB至256MB的页大小分别编码为0至8，编码过程本质上是一个优先编码器，通过一系列if语句实现，大致如下：

|  |
| --- |
| if(~|mask)  matchedPageMaskKind = 4'd0;  else if(~|mask[15:2] && &mask[1:0])  matchedPageMaskKind = 4'd1;  ...  else  matchedPageMaskKind = 4'dx; |

其中最后一项代表页掩码不合法。

将页大小进行编码之后就可以获取到虚拟地址中确定奇偶页号的位，即虚拟页号的最后一位。对于1KB的页，奇偶位位于12位，4KB位于14，以此类推。这里使用case语句实现一个多路复用器来选择出奇偶位。通过该奇偶位的值又可以从两个物理页号记录中选择出相应的那一个，得到selectedEntryLo。输出信号bitV、bitD、bitC直接分别取自该信号的1、2、5到3位。最后物理页号和块内地址根据页的大小拼接成物理地址，同样使用多路复用器来实现。由于这里采用的物理地址是32位的，所以需要忽略物理页号的高6位。1KB的页为将selectedEntryLo的6到25位与虚拟地址的0到11位连接；4KB为将8到25位与0到13位连接，以此类推。至此，所有信号都已生成完毕。

3.4 Cache模块的设计

本课题中的Cache采用虚拟地址Cache，即映射使用虚拟地址，物理地址作标签。由于在之后的测试中会向可执行的区域写数据并执行，所以这里没有将指令Cache和数据Cache分开设计，而采用一个统一的Cache。

具体的映射方法使用相联度为2的组相联，写策略使用回写，写操作缺失时采用写分配。选择组相联的原因是为了克服直接映射中出现需要不断交替访问两个标签不同但Cache索引相同的内存块时出现的连续缺失的情况。如果将指令和数据Cache分开，这种情况出现的可能性就很小，但在本课题中由于是统一的一个Cache，程序在读指令和读写数据时两者的地址所映射的索引就有可能会相同，从而使Cache不断出现缺失的情况。曾使用过的设计就是直接映射Cache，而测试的结果就是程序运行所需要的时间大大增加。而对于替换算法，采用如下的方式：如果一排中的两个块有无效的，则替换无效的，如果都有效，则替换脏位为1，即被修改过的，如果都被修改过，则随机替换。

Cache的主要端口及作用包含：

（1）时钟及复位信号，这是每个模块都有的；

（2）就续指示输出ready。在复位的时候Cache需要使用一个时序逻辑来将所有的块的标签以及服务位初始化，在初始化之前处理器应该等待，当该信号为1时才能开始运行；

（3）物理地址和虚拟地址的输入pAddr与vAddr；

（4）处理器访存接口，用于Cache与处理器之间的连接，接收来自处理器的访存请求并响应。这些信号包括来自处理器的数据db\_dataOut、返回给处理器的数据db\_dataIn、访存操作是否完成的指示信号db\_ready以及访存类型输入db\_accessType；

（5）Cache与内存（或下一级缓存）间的接口。这部分接口和处理器访存接口相似，区别在于信号名，都为dbOut\_\*而不是db\_\*，并且没有db\_accessType信号，还有额外的信号dbOut\_re和dbOut\_we，分别是读取和写入的使能，输出。

Cache模块还包含两个参数：块索引的位数和块内地址的位数。在最终的设计中这两个参数的和不能大于32，而且由于标签使用的是物理地址，所以一个块只能在一个页中，不能出现块的大小大于页的情况。

3.4.1 数据存储器与标签存储器

Cache中包含两个存储器，数据存储器和标签存储器。为了设计的方便，在设计中采用两组以上的存储器，分别对应于一排中的两个行，这样，同一索引对应的两个数据块以及标签和服务位分别来自这两个存储器。数据存储器中每个数据都是32位的类型。每个标签是由物理地址标签和服务位拼接而成。其中的物理地址标签由将物理地址去掉块内地址得到，这样才能保证同一个块中对应的物理地址标签都相同。服务位包含有效位和脏位，当一个内存块被加载到Cache中时相应的有效位就会被写为1。脏位的含义与TLB中的D位相同，即该块是否有被修改过，用于在要将该块替换掉时判断是否需要将该块写回到内存中。这样，标签存储器的地址宽度等于索引的宽度，地址的值等于索引；数据存储器的地址宽度为块索引宽度加上块内地址的宽度减2，其中减2是因为数据的地址是字节编址的，而这里的数据寄存器的数据宽度是32，包含4个字节，对应地址中的两位。数据存储器的输入地址应为块索引和块内地址拼接而成。

存储器使用的是同步读写的RAM，即在一个周期内读取或写入，上第一周期将数据准备好后下一个周期才能得到数据或完成写入。RAM包含数据的输入输出口、地址输入以及读取和写入的使能信号。RAM的实现大致如下：

|  |
| --- |
| reg [WIDTH - 1:0] data[SIZE - 1:0];  reg [ADDR\_WIDTH - 1:0] addrLatch;  assign dataOut = data[addrLatch];  always @(posedge clk) begin  if(we)  data[writeAddr] <= dataIn;  else if(re) begin  addrLatch <= readAddr;  end  end |

以上实现的RAM称为同步RAM（synchronous RAM），因为其读或写都是在一个周期内进行的，两者不会同时进行。之所以采用同步RAM来保存，是因为本课题使用的FPGA中有内建的同步RAM，Quartus软件在综合的时候会根据设计的逻辑来将其用内部资源实现。对于以上的存储器，该软件会生成使用内部RAM的电路（inferred RAM）而不会直接用逻辑单元来实现，后者的地址译码逻辑会占用大量的逻辑单元。

如果将上面代码中的

|  |
| --- |
| assign dataOut = data[addrLatch]; |

改为

|  |
| --- |
| assign dataOut = data[addr]; |

此时读取数据的输出与地址之间就是一个纯组合逻辑，读操作和写操作相互独立并可同时进行。此时这个存储器也叫异步RAM（asynchronous RAM）。由于选择的FPGA芯片中没有异步RAM资源，因此Quartus软件就会直接用逻辑单元来实现。

3.4.2 功能的实现

下面讨论Cache的实现方法。Cache需要完成的功能包括检查是否命中、对存在于缓存中的数据进行读写、通过访存将位于内存中的数据加载到Cache中以及将缓存中的数据块写回到内存中。这些功能是通过一套时序逻辑电路和组合逻辑实现的。

时序电路用于实现Cache的缓存读写以及缺失时候的访存操作。整个Cache的时序电路部分实现用的是有限状态机模型，状态的编码及意义如下：

（1）S\_RES：初始化状态，代表Cache正在完成初始化，即把所有的有效位都写为0。初始化需要一个时序操作，因为RAM是同步的，一次只能写入一个条目而不能一次将所有块都写入。当res信号为1时当前状态就会变为该状态；

（2）S\_IDLE：空闲状态，当初始化完成后就会进入该状态，并且会一直保持在该状态直到有访存请求；

（3）S\_CHK\_HIT：检查是否命中。这是访存请求到来之后到达的第一个状态。当转化到这个状态时标签以及数据应已经从存储器中读取出来了，并且这些信号已经通过一个组合逻辑产生了是否命中等信号。此时就会根据是否命中以及服务位来确定下一个状态；

（4）S\_LOAD\_BLOCK、S\_LOAD\_BLOCK\_WAIT：均表示从内存中加载块。加载时是以字为单位加载的，在加载一个字时需要等待dbOut\_ready为1时才能读取下一个。使用两个状态的原因是为了产生合适的访存信号dbOut\_re，因为在读取一个字时的第一个周期内读使能信号才为1，在等待访存的过程中读使能信号应为0。因此在第一个状态中若就续信号为0则应转入第二个状态；

（5）S\_WRITE\_BACK、S\_WRITE\_BACK\_WAIT：与上一个类似，表示将块内的数据写回到内存中。

（6）S\_READ\_FIRST\_W：从缓存中读取块内的第一个字。在写回中写第一个字时需要先将第一个字从数据存储器中读取出来，而之后写回的数据可以在上一次写的时候读；

（7）S\_WRITE\_LAST\_W、S\_WRITE\_LAST\_W\_WAIT：写回最后一个字。在写回的时候当地址为最后一个地址时还没有完成整个写回操作，还需要将最后一个字写回；

（8）S\_LOAD\_LAST\_W：加载最后一个字。在加载数据块时从内存读取到最后一个字后还需要一个周期将读取到的字写到数据存储器中；

（9）S\_WRITE、S\_WRITE\_WAIT：写数据回内存。当执行写操作时发生缺失，应先访问内存将数据写到内存中，然后再到内存中读取出整个块。

当前状态保存在state变量中，一个组合逻辑根据当前状态以及其他输入信号来得到下一个状态nextState，其他的一些信号也依赖于下一个状态的值，因为所有的信号需要在时钟信号上升沿前准备好。

Cache模块的组合逻辑电路包括判断是否命中并从索引对应的两个行中选择出匹配的行和数据，以及输出的访问内存成地址的生成和Cache内部存储器的地址及数据的生成。其中后面两者依赖于下一个状态的值。其中较为重要的信号包括块选择信号hitEntry、victim和whichEntry。hitEntry信号用于指示一排中命中的块，victim信号用于指示将要被替换掉的块，而whichEntry则用于在排中选择出需要用到的块，其取值在Cache命中时等于hitEntry，在缺失的时候等于victim。对两对数据存储器和标签存储器的写操作是针对选中的那对存储器，因此在存储器的写使能信号前需要加入是否选中该存储器的条件（与门），对于第一个存储器，条件为~whichEntry，第二个为whichEntry。其余信号的具体依赖关系将在下面的时序逻辑中一并讨论。

输入信号cachable用于指明该地址是否能够被放入缓存，如果不能，那与内存的接口信号将与输入的与处理器的接口信号直接相连。这通过一个多路复用器实现。

下面讨论Cache模块在实现不同操作时的状态转移条件和路径。首先是初始化操作，与初始化有关的状态只有S\_RES，相关的变量是resetIndex，用于标签存储器的地址输入。有关的信号还包括两个标签存储器的写使能，都为1，以及标签存储器的输入数据，也为0，这里只要保证其有效位为0即可。状态转移逻辑较为简单：在每个时钟上升沿，如果resetIndex等于最后一个索引即块数减一，则转移到S\_IDLE状态，否则resetIndex加一。

处于空闲状态时如果有访存请求，则下一个状态将会是S\_CHK\_HIT，并且在状态改变时会将虚拟地址索引对应的标签读取出来以检查是否命中。下一个状态依赖于是否命中、访存的类型以及是否需要改变标签存储器中的服务位。对于命中的情况，如果命中的块的脏位不为0，并且访存的操作为写入则需要将脏位改变为1，所以下一个状态是S\_IDLE，需要一个周期来对标签存储器进行写入；如果不需要写脏位，那么此时虽然状态不是空闲状态，但已经完全可以响应下一个访存请求，因此在这种情况下处理器接口的db\_ready信号应为1，并且下一个状态根据是否有访存请求而变为S\_CHK\_HIT或空闲态。对于缺失的情况，如果访存操作是写入，那么下一个状态就是S\_WRITE，因为当缺失时需要先将数据写入内存再处理后续的；否则，根据选中的（此时排内选择信号whichEntry等于victim）的块的脏位是否为1来确定下一个状态为读第一个字（S\_READ\_FIRST\_W）或加载块（S\_LOAD\_BLOCK），另外S\_WRITE完成后也会进行同样的判断和状态转移。读取第一个字完成后就是写回S\_WRITE\_BACK，之后就是加载块。当不命中时离开状态S\_CHK\_HIT的时候需要将新的块的标签以及有效位写入标签存储器，这样当加载块完成后又会回到状态S\_CHK\_HIT，此时标签和数据存储器就都已更新，因此一定会命中。

在访存缺失时会执行块的加载和写回，这里先讨论这两种操作的时序。与块的加载有关的状态包括S\_LOAD\_BLOCK、S\_LOAD\_BLOCK\_WAIT以及S\_LOAD\_LAST\_W，相关的变量是inBlockAddr和addedInBlockAddr，前者是一个与resetIndex类似的变量，是一个寄存器，用于记录当前正在加载的字的地址，后者是下一个地址的值，由于访存的单位是字，因此该变量每次应该增加4，所以这两个变量的关系为

|  |
| --- |
| assign {memEnd, addedInBlockAddr} = inBlockAddr + 4; |

这里的memEnd变量用于指示是否是最后一个地址了，当表达式右侧的值溢出时memEnd就会等于1。相关的信号包括与内存的接口中的地址、读使能和Cache中数据存储器的地址、写使能。其中数据存储器的地址是由索引和块内地址拼接成的，而内存接口的地址则是由标签与块内地址拼接。整个加载的操作逻辑上是一个循环，每次循环就加载一个字到缓存，因此一次循环时需要等待内存就续才能进行下一次循环，退出循环的条件就是inBlockAddr加载到了最后一个字。在执行加载时索引和标签分别由虚拟地址和物理地址给出。数据存储器的块内地址部分是inBlockAddr，而内存接口的地址的块内地址则应该是下一个块内地址，即addedInBlockAddr，这是因为在一个循环中，一次循环结束的时候会将这次加载完成的字写到数据存储器，而将下一次循环所需要读取的字的地址准备好。另外，在整个循环开始的时候此时inBlockAddr变量还没有被写为0，仍是上次操作时的最终值，因此此时块内地址应单独考虑，并且为0。在每个循环中，当存储器就续，即dbOut\_ready为1时就会将该数据写入数据存储器，即数据存储器的写使能信号置为1，并根据memEnd信号判断是否为块内的最后一个地址，如果是那么下一个状态就会是S\_LOAD\_LAST\_W，因为还需要一个周期才能将数据写入存储器，否则下一个状态仍是S\_LOAD\_BLOCK，并且需要将inBlockAddr加1。

写回操作的流程与加载类似，也是以字为单位来写，逻辑上也是一个循环结构，使用的块内地址变量也是inBlockAddr。只不过在整个写操作开始之前需要一个额外的周期来将选中的块中的第一个字读出来，并且输出的地址的标签部分来自被写回的块的标签而不是物理地址。在每个循环中，因为在每个循环结束的时候需要从存储器读取出下一个字，内存接口成块内地址应为inBlockAddr，而数据存储器的应为addedInBlockAddr，与加载时正好相反。在循环退出后由于此时的输出数据线上还存在没有被写到内存的最后一个字，所以还需要用一个额外的状态S\_WRITE\_LAST\_W来将这最后一个数据写回内存。整个写回操作完成后，将进入S\_LOAD\_BLOCK状态来加载新的数据块。

3.5 MIPS CPU的设计

本课题同样需要实现一个MIPS架构的CPU来对之前实现的各种内存管理模块发出访存操作，并通过实际地运行程序的方式来检验其正确性。标准的MIPS处理器应为5级流水线，无硬件互锁机制的处理器，对于所有的跳转指令需要在其后插入延时槽，但受所使用FPGA内部资源的限制，以及本课题的重点并不是处理器，这里的设计采用无流水线的多周期CPU。为了能使之后的测试程序顺利运行，本课题实现了如下的必要指令：

（1）访存指令lw、lb、lbu、lh、lhu、sw、sb、sh；

（2）所有的运算指令，包括add、addi、addu、addiu、sub、subu、mult、multu、or、ori、xor、xori、nor、sll、sllv、srl、srlv、sra、srav、slt、sltu；

（3）分支跳转指令，包括beq、bne；

（4）跳转指令，包括j、jr、jal、jalr、eret；

（5）数据传送指令，包括li、mfc0、mtc0；

（6）TLB操作指令，包括tlbp、tlbwi、tlbwr、tlbr；

其他功能方面，CPU模块还实现了部分的异常处理机制，特别是与TLB有关的异常。对于数据的存储，本课题的CPU采用的是大端存储方式。CPU模块对于TLB操作，CPU会直接和MMU进行通信，访问内存则需要外部的一个逻辑电路，将CPU输出的虚拟地址传递给MMU翻译成物理地址后再传递给存储器完成访存操作。因此，CPU模块的主要端口包括：

（1）时钟与复位信号输入，这两个信号是每个模块都有的；

（2）与MMU通信的接口，包括访问的MMU寄存器号输出mmu\_reg、输出给MMU的数据mmu\_dataIn、来自MMU的数据mmu\_dataOut、发送给MMU的命令mmu\_cmd以及MMU产生的异常mmu\_exception；

（3）访问内存的信号，包括输出的虚拟地址db\_addr、数据输出db\_dataOut、数据输入db\_dataIn、访问内存的操作类型db\_accessType、内存设备的就续状态输入db\_ready以及访问的内存的长度db\_memLen。最后一个信号用于实现对8位的数据以及16位数据的读写。这是因为整个系统是32位的，内存的地址虽然是按字节编址，但访问也是以32位的字为单位，如果需要完成其他长度的读写，就需要有一套额外的逻辑电路来实现。具体的实现方式将在3.6节中讨论；

（4）其他信号包括外部设备是否就续的输入ready、用于MMU检查内存访问权限的CPU运行状态的输出cpuMode以及外部请求输入irq。ready信号是用来等待Cache初始化的，对于irq输入本课题暂时没有在外部使用到。

3.5.1 子模块

整个CPU模块是本课题中较为复杂的主要模块。其大致的结构是由若干个子模块以及用于连接各个模块的逻辑构成的。这些子模块包括：

（1）控制信号产生器（Controller）：用于根据当前的指令来产生各种控制信号，如传递给ALU的操作码、寄存器的写使能信号、是否需要访问内存、访问的内存的长度（db\_memLen）以及各种用于CPU中选择数据的选通信号等。该模块是纯组合逻辑；

（2）下地址逻辑（InstructionFetcher）：用于根据控制信号、指令中的数据以及当前程序计数器输出下一条指令的地址。这个模块同样也要处理跳转指令和分支跳转指令，因此输入信号中包含算数逻辑单元输出的指明结果是否为0的信号zero。这个模块是纯组合逻辑；

（3）算数逻辑单元（ALU）：将两个输入的数据做给定的算数运算之后输出，并产生零和溢出标志位。做的运算的类型由控制信号产生器产生。该模块也是纯组合逻辑；

（4）寄存器堆（RegFile）：MIPS处理器中的32个寄存器位于这个模块中。该模块包含两组读取的寄存器号输入和读取到的寄存器的值的输出，以及一组写寄存器的信号端口。与Cache中的存储器类似，该模块使用的是同步读写逻辑，使得综合软件能够生成使用内部RAM的电路而不是用逻辑单元来实现。该模块是时序逻辑模块；

（5）协处理器0寄存器（CP0Regs）：用于保存需要用到的逻辑上属于协处理器0的寄存器，这些寄存器包括EPC、Status、Cause、BadVAddr。该模块用于响应有关协处理器0的寄存器访问指令MTC0、MFC0的读写请求，如果指令作用到的寄存器属于上面四个中的就会直接对相应的寄存器进行读写，如果是属于MMU的（之前提到过），就会将请求重定向到MMU中。同时，该模块对寄存器的读写是异步的，四个寄存器的值直接被作为了输出。该模块是时序逻辑模块；

（6）异常控制器（ExceptionController）：用于根据各种控制和状态信号产生和异常处理有关的信号和数据，如是否有异常产生（exception）、是否需要将PC的值写入EPC、在写入EPC时是否需要将PC的值加4以及Status和Cause寄存器的新值。

3.5.2 异常处理

CPU模块中的异常处理机制大致与MIPS标准中的处理方法一致。MIPS中，与异常有关的寄存器都位于协处理器0中，之前提到过，这是一个逻辑处理器。有关的寄存器包括：

（1）EPC：用于保存当异常处理程序完成后需要返回的指令的地址，其取值为出异常的指令的PC或该PC加4，这是因为不同异常处理方式不同，有些异常在处理完成后需要重新执行出错的指令，比如TLBL和TLBS异常，需要重新完成内存的访问。而有些异常，如外部中断、系统调用等，不需要重新执行发生异常的指令；

（2）Status：用于保存当前的状态，包括是否处于异常中以及当前CPU的基本运行状态。MIPS标准中Status寄存器中的字段组成如图3-6所示。

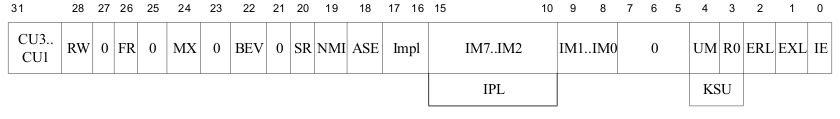


图3-6 Status寄存器

在这里仅使用0到4位和10到15位。其中EXL和ERL分别表示目前是否处于异常和错误的处理程序中，KSU字段表示CPU的基本运行状态。Status寄存器的值直接决定CPU的运行模式。在本课题中只实现了用户模式和内核模式。如果KSU为0或者目前处于异常处理程序中（EXL和ERL为1），则当前处于内核模式；若KSU为2并且不处于异常处理中，则为用户模式。Status寄存器的读写都由异常控制模块在控制；

（3）Cause：保存触发异常的原因。该寄存器的2到6位用于保存异常码（excCode）。

当出现异常时需要将程序计数器（PC）值写入异常程序计数器（EPC）寄存器，并将PC跳转到异常处理程序的入口地址处。在MIPS中，获取异常处理的入口地址有两种方式：查询中断（Polled interrupt）和向量中断（Vectored interrupt）。本课题采用较为简单的查询中断，即当中断或异常发生时跳转到一个固定的地址，然后中断处理程序中通过读取Cause寄存器的值来获取到触发中断的原因，并调用相应的异常处理程序。这个固定的地址为0x80000080，该地址处于kseg0段中，为内核非映射段，这样就不会出现进入异常处理程序时取指令发生TLB缺失异常，从而无限循环的情况。

在每条指令执行完成时检查是否有外部中断，取指令和执行访存指令的时候会检查是否有与MMU有关的异常，即检查mmu\_exception是否不等于0。

3.5.3 时序逻辑的实现和状态机

CPU模块的实现同样使用的是状态机模型。各个状态的编码以及描述如下：

（1）S\_INITIAL：复位后进入的状态。CPU会保持在该状态直到ready输入信号变为1，即Cache初始化完成；

（2）S\_FETCH\_INSTRUCTION、S\_FETCH\_INSTRUCTION\_WAIT：取指令状态。由于取指令实质上是读内存，所以需要等待内存就续。与之前类似，当下一个状态是前者的时候发出读数据的指令，之后前者会转移到后者，并一直保持直到内存就续，之后会转移到S\_INS\_DECODE状态上；

（3）S\_INS\_DECODE：指令译码，转移到该状态之后会将来自内存的指令写入指令寄存器，从而产生一系列的控制信号。该状态的下一个状态为S\_EXEC；

（4）S\_EXEC：执行状态；

（5）S\_READ\_MEM、S\_READ\_MEM\_WAIT：读内存状态；

（6）S\_WRITE\_MEM、S\_WRITE\_MEM\_WAIT：写内存状态；

（7）S\_EXCEPTION：异常处理状态。

一条指令的执行过程中状态的转移路径可分为四种。由于其出发的状态始终是S\_FETCH\_INSTRUCTION，在下面的路径中将略去该状态。第一种是S\_INS\_DECODE、S\_EXEC，然后回到取指令状态，路径为这种的指令包括不需要访存但需要访问寄存器的指令，在离开状态S\_INS\_DECODE时从寄存器堆或协处理器0中读取出值，离开S\_EXEC状态时将数据写回寄存器。第二种是S\_INS\_DECODE、S\_EXEC、S\_READ\_MEM，对应于读内存指令lw、lh、lhu、lb、lbu，在离开最后那个状态时将读取到的数据写到寄存器中。第三种与第二种类似，但最后一个状态为S\_WRITE\_MEM，对应于写内存的指令sw、sh、sb。第四种的路径只包含状态S\_INS\_DECODE，对应的指令为既不需要访问寄存器也不需要访存，这类指令包括tlbp、tlbwr、tlbwi、tlbr、nop（实际为sll）、eret。

3.6 模块的结合

主要的模块实现完成后需要将这些模块相互结合起来成为一个模块。

3.6.1 CPU模块与MMU的结合

首先将CPU与MMU相结合，成为模块CPU\_MMU。该模块用于将来自CPU的虚拟地址传递给MMU进行地址翻译，并实现宽度小于32的数据的读写。其端口包括：

（1）访存接口，包括数据输出db\_dataOut、数据输入db\_dataIn、物理地址输出db\_addr、访存类型db\_accessType以及内存是否就续的输入db\_ready；

（2）输出的地址是否处于IO段的信号db\_io以及标明是否经过Cache的信号cachable；

模块响应CPU的访存请求的流程为先将虚拟地址传递给MMU，一个时钟周期后会输出物理地址，或产生异常，这里的异常信号mmu\_exception是在CPU和MMU之间直接相连的。当得到物理地址后就可以访问内存了，此时需要实现非32位数据的读写，对于读操作，只需先从内存读取出数据所在的字，即把地址的0、1位改为零作为访存地址，然后再根据访存的数据的长度以及地址的0到1位来从这个字中进一步地选择出数据。这里需要考虑到CPU使用的是大端的存储方式。如对于字节的访问，地址的0到1位为0、1、2、3分别应该选择字中的31到24、23到16、15到8、7到0位作为返回给CPU的数据值。对于写操作，如果数据为32位，则直接将数据写到内存，否则，需要先读取出数据所在的字，再根据数据长度和地址的0到1位将读取出的字的不同位置改为需要写的数据，再将修改后的字写回到内存。

该模块同样采用的是状态机模型。其状态的定义与作用如下：

（1）S\_IDLE：空闲状态，模块复位时会进入该状态。模块会一直保持在该状态直到有访存请求；

（2）S\_SAVE\_ADDR：保存虚拟地址的状态。当有访存请求到来时会首先进入该状态，将来自CPU的虚拟地址写入寄存器，同时写入MMU用于地址转换；

（3）S\_ACCESS\_MEM、S\_ACCESS\_MEM\_WAIT：访存状态。对于32位数据的访问以及非32位数据的读取，该状态完成的是数据的读或写，对于非32位数据的写操作，该状态为读取出数据所在的字；

（4）S\_WRITE\_BACK、S\_WRITE\_BACK\_WAIT：将数据写回到内存，仅用于非32位数据的写操作，将数据读取出后的写回。

对于状态S\_ACCESS\_MEM，如果发生了异常那么状态直接转换为S\_IDLE，CPU此时会处理相应的异常。

3.6.2 顶层模块

将CPU\_MMU模块和Cache模块结合就得到了整个系统的顶层模块MipsCPU。这两个模块的结合不需要额外的逻辑控制，直接将对应的端口相连即可。得到的MipsCPU模块的端口主要包含与内存访问有关的端口。具体包括：

（1）时钟信号与复位信号；

（2）内存访问接口，包括数据输入db\_dataIn、数据输出db\_dataOut、（物理）地址输出db\_addr、内存就续标志db\_ready、地址是否处于IO段的标志db\_io、内存的读使能信号db\_re、内存写使能信号db\_we。

至此，整个系统都已实现完毕。

3.7 本章小结

本章详细地讨论了系统中各个模块的实现方法，整个系统的主要模块包括内存控制单元MMU、高速缓存Cache以及MIPS处理器。对于时序逻辑采用的主要方法是状态机模型，模块中的各个信号大多都依赖于模块当前的状态。

MMU模块采用的是MIPS架构中内存管理的方式，即基于TLB的地址翻译机制。TLB条目的匹配采用的是组合逻辑的方式，一个周期内就可得到物理地址或产生异常，其代价就是占用了大量的逻辑单元。在MMU模块中实例化了TLB模块，并使用一些状态变量来实现了状态机模型，完成地址的翻译。

Cache模块采用的是相联度为2的组相联映射方式，写策略采取的是写回，替换算法是随机替换。之所以要采取组相联是因为在本设计中Cache不分为指令Cache和数据Cache而是统一的一个，因此较容易出现需要交替访问两个索引相同标签不同的内存块，而组相联则可以克服这种情况下不断出现的Cache缺失。

CPU模块实现了部分对于测试程序必要的指令。并实现了部分异常处理机制，用于检测并处理TLB异常。

第四章 系统的测试

下面需要将整个系统下载到FPGA开发板上进行实际调试，即让CPU运行一段程序。为了使系统能够在芯片中顺利运行，还需要一个内存模块。另外为了使测试程序运行的效果能够被展示出来，这里加入了外设控制模块，使得程序可以控制开发板上的外设。

4.1 内存接口模块

内存的实现方案有使用开发板上的同步动态随机存储器（SDRAM）、使用开发板上的SD卡以及直接与计算机通信。前两个外设的操作方式都较为复杂，而且其控制模块会使整个系统所使用的逻辑单元数超过FPGA的资源。因此这里采用的是直接与计算机进行串口通信，并在计算机上编写一个模拟内存并与之交互的上位机程序。

这里采用的串口为异步串口通信（UART），数据是以字节为单位进行传输的。其一个字节的传输时序如图4-1所示。



图4-1 串口数据帧

完整的串口协议还包括握手等操作，较为复杂，而且接口数也较多，这里采用简化的串口通信协议，其端口只包含发射（TX）和接收（RX）两个口，这两个口的收发可以互不相关，因此称为异步通信。串口通信通过两个模块UART\_tx和UART\_rx来实现，其中前者为发射，后者为接收。串口的发射和接收口通过FPGA的引脚引出后到达开发板上的USB转串口模块，与计算机建立连接。

内存模块通过传输命令的方式来完成内存的访问操作。传输的第一个字节为命令代码，之后为数据，不同命令的数据也不同。这些命令包括：

（1）读数据（代码：0x01）：读取一个32位的字。第一个命令代码字节后应加上4个字节长度的地址。上位机在接收到该命令后应回复4个字节的读取结果；

（2）写数据（代码：0x02）：写一个32位的字，命令的数据包括4字节的地址和4字节的待写数据；

（3）打印字符（代码：0x03）：在上位机的终端上打印一个字符，该命令主要用于调试。命令的数据为1个字节的字符。

以上命令中所有的数据都是以小端的方式传输的。

该模块已对内存是否为IO操作作出了区分，并对IO段的部分地址做了映射。将IO段的地址0x00000001映射为打印字符，该地址是一个只写地址，对该地址的写操作就会发送打印字符的命令。

上位机程序采用Python编写，其模块Pyserial可以很方便地实现对串口的读写操作，并且文件的读写也很方便。

4.2 外设控制器

外设控制器实现了对开发板上的7段数码管的控制，对IO段中部分地址的写操作将被重定向到对数码管的操作。数码管共有6个，每个数码管有8个LED，正好可以使用一个字节来保存一个数码管成状态。数码管在开发板上的电路如图4-2所示。

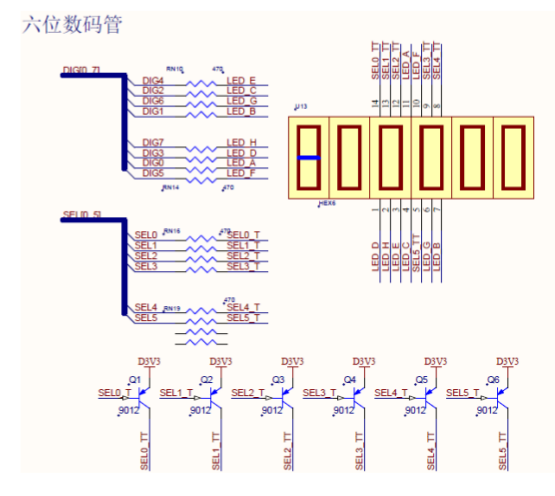


图4-2 数码管电路

6个数码管共用同一组LED控制信号，并另有6个选通信号用于控制数码管的亮灭，一般情况下各个数码管显示成数字应是不同的，这就需要对数码管进行扫描，即快速逐个地用选通信号选中各个数码管并输出控制信号，使得这些数码管看起来是同时显示了不同的数字。

模块中对6个数码管使用了6个8位的寄存器来保存其LED的状态，并通过一个时序逻辑对选通信号进行扫描。这6个寄存器被分配了IO段上的地址，使得可以从程序中实现对数码管的控制。6个寄存器的地址分别为0x00000010到0x00000024，地址间隔为4。

4.3 测试的顶层模块

将系统顶层模块MipsCPU与内存接口模块以及外设模块结合构成了最终用于测试的顶层模块。该模块的所有端口应在FPGA上分配引脚。顶层模块的端口及引脚分配情况为：

（1）时钟信号clk、复位信号res\_n。其中时钟分配到了开发板上的50MHz时钟源，复位信号为按钮RST。该复位信号是低电平有效，所以在连接到各个子模块时应先取反；

（2）串口收发端口rx与tx，其分配的的引脚为与USB转串口模块相连的引脚。此处与串口模块的连接是交叉连接，即rx连接到模块的tx口，tx口连接到rx口；

（3）数码管LED控制信号。应直接分配到与数码管的LED相连；

（4）数码管选通信号。

4.4 测试程序的编写

本课题采用MIPS交叉编译器mips-linux-gnu-gcc编译测试程序，采用汇编语言与C语言混合编程。在程序链接的时候使用自定义的链接脚本来对程序中不同的段定义虚拟地址和加载地址。程序编译的结果是一个ELF文件，这里使用objcopy命令将其转化为可供上位机程序读取的二进制内存镜像文件。

程序的启动代码以及异常处理的入口处使用的是汇编语言编写。其中启动代码初始化了栈，异常处理程序负责保存上下文，即将所有的寄存器保存到内存，以及根据Cause寄存器中excCode的值来查异常处理程序表从而调用不同的处理程序。

程序入口处的代码和数据均分配在kseg0中，即非映射段，这是因为在启动的时候TLB还没有初始化。程序中所用到的页表和一段测试用的内存空间分别位于两个不同的物理页中。

测试程序完成的功能是模拟操作系统的启动过程，具体为先在页表中分配一个虚拟页并将一段实现数码管计时器的程序复制到该虚拟页中，切换到用户模式并运行该程序。其具体的流程为先初始化TLB，再向TLB中增加页表处在的页并初始化页表，将虚拟页号0x00000映射到页表所在的页，0x00004映射到一段用于保存程序的内存空间。之后调用内存复制函数来将一个定义好的控制数码管的程序复制到该内存空间，然后跳转到该区域。程序的运行过程中会在关键的地方通过串口向上位机的终端打印信息以跟踪程序的运行。

4.5 测试的结果

将整个系统下载到开发板，并将串口与计算机连接，然后启动上位机程序。可以看到，终端上依次打印的信息如图4-3所示。

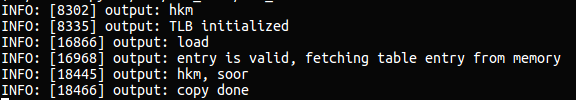


图4-3 终端输出

在最后一条信息打印出来的同时开发板上的数码管开始计数，即被复制到虚拟内存空间中的程序成功运行，并符合预期。

在数码管开始计数前可以看到开发板上的串口通信指示灯不断闪烁，说明不断地有Cache缺失情况出现，需要访存，这是由于此时程序的指令以及数据在内存中较分散，局部性较差，当数码管开始计数时程序和数据的读写都仅限于特定的区域，而且均Cache命中，此时串口指示灯不再闪烁。

4.6 本章小结

本章中讨论了整个系统的实际调试方法以及调试成结果。内存访问操作采取的是与计算机串口通信，并使用上位机程序模拟内存的方式。另外还编写了用于控制数码管的外设控制器使得可以从程序中控制数码管以检查程序的运行是否符合预期。

测试程序实现的功能是先完成页表与TLB的初始化，然后分配一个虚拟页，将一段用于实现数码管计时器的程序复制到该虚拟内存中并运行。

最终程序的运行结果符合预期，初步说明该系统能够正确地实现虚拟内存的管理以及使用Cache对内存进行缓存。

第五章 全文总结与展望

5.1 全文总结

本文以虚拟存储器和高速缓存的实现方法为研究背景，使用Verilog语言实现了一套运用在FPGA芯片上的内存管理单元和Cache控制器。

对于内存管理单元MMU，其主要成功能是根据TLB将来自处理器的虚拟地址转换为物理地址，这其中关键的一步就是找出与虚拟地址项匹配的TLB条目，该功能的实现使用的是组合逻辑，把每个条目是否匹配的信号输入一个优先编码器从而得到匹配的条目的索引。

Cache的映射方式采用的是相联度为2的组相联，写策略是回写，替换算法是随机替换，写操作发生缺失时采用的策略是写分配。模块会处理来自处理器的访存请求，若命中则不需要访存，否则将会把需要替换的块写回内存，然后将需要访问的块加载到缓存。

MIPS处理器实现了对测试程序必要的指令，以及异常处理机制，主要是为处理TLB异常。为了完成访存操作，CPU采用的是多周期，访存操作（包括取指令和读写数据）需要等待内存的就续。

为了使整个系统能在FPGA上运行，采用了和计算机中的上位机程序串口通信成方式模拟访存操作，并且实现了控制数码管的外设控制模块，在内存空间中的IO段中分配了用于访问数码管成地址，使得测试程序可以在数码管上显示数据从而确定程序的运行是否符合预期效果。

整个系统使用的是Verilog语言编写的，采用的设计方法是寄存器传输级描述（RTL）。对于时序逻辑的实现，多次采用了状态机模型。

5.2 后续工作展望

近年来虚拟存储器的研究发展日益活跃，本文只是实现了其基本的功能，还有很多可扩充和改进的地方：

（1）目前本系统中的Cache对用户是不可见的，为了能更好地处理例如DMA等操作，可以实现MIPS中的CACHE命令，允许应用程序直接对Cache操作。

（2）可以实现不同的Cache的替换算法和写策略并做成可配置的，可以通过LB条目中的C字段控制；

（3）MIPS会在TLB缺失时触发TLB缺失异常，让软件完成TLB重装。但在大多数情况下重装的过程都较为简单，而且触发异常同样有开销，因此可以实现页表的硬件读取电路，并使用一个特殊寄存器，通过寄存器的值来确定使用软件还是硬件读取页表项。

致 谢

本论文的工作是在我的导师邓建老师悉心指导下完成的，感谢邓建老师对我课题的指导和在工作之余抽出时间答疑，以及在论文撰写上给我提供的建议。

参考文献

1. Robert Glass, Ed. In The Beginning: Personal Recollections of Software Pioneers[M]. UK:IEEE Press, 1997, 1-2
2. Ferranti Ltd, [Ferranti Atlas 1 & 2 – Systems Architecture](http://www.ourcomputerheritage.org/ccs-f5x2.pdf)[EB/OL]. http://www.ourcomputerheritage.org/ccs-f5x2.pdf, November 12, 2009
3. Cragon, Harvey G. [Memory Systems and Pipelined Processors](https://books.google.com/?id=q2w3JSFD7l4C). Jones and Bartlett Publishers[M]. Texas:Jones & Bartlett Learning, 1996, 1-2
4. Virtual memory[EB/OL]. https://en.wikipedia.org/wiki/Virtual\_memory
5. Javier Picorel, Djordje Jevdjic, Babak Falsafi. Near-Memory Address Translation[C]. 2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT), Oregon, 2017, 303-304
6. Ioannis Deligiannis, George Kornaros. Adaptive memory management scheme for MMU-less embedded systems[C]. 2016 11th IEEE Symposium on Industrial Embedded Systems (SIES), Krakow, 2016, 1-1
7. Arm limited. IOT SOLUTIONS MBED IOT DEVICE PLATFORM[EB/OL]. https://www.arm.com/products/iot-solutions/mbed-iot-device-platform, 2017
8. 李亚民.计算机原理与设计——Verilog HDL版[M]. 北京:清华大学出版社, 2011, 368-369
9. MIPS. The MIPS32 and microMIPS32 Privileged Resource Architecture v6.02[EB/OL]. https://www.mips.com/?do-download=the-mips32-and-micromips32-privileged-resource-architecture-v6-02, 2015, 29-30
10. MIPS. The MIPS32 and microMIPS32 Privileged Resource Architecture v6.02[EB/OL]. https://www.mips.com/?do-download=the-mips32-and-micromips32-privileged-resource-architecture-v6-02, 2015, 31-32
11. Gerard Blanchet, Bertrand Dupouy. Computer Architechture[M]. (陈宗斌). 北京:北京大学出版社, 2017, 126-127
12. M. Milenkovic. Microprocessor memory management units[J]. IEEE Micro, 1990, 10:74-75

外文资料原文



外文资料译文

基于MMU的动态内存隔离的严格证明

摘要——出于安全性的原因，保证进程间的内存隔离是很有必要的。内存管理器因而是操作系统内核中重要的组成部分。内核一般通过一种叫做内存管理单元（MMU）的硬件设备来保证内存隔离。然而MMU本身并不提供内存隔离。它只是可供内核使用来保证这一属性的工具。

绪论

现代操作系统允许不被信任的进程间共享计算机资源，并且快速的地处理来自外部的事件，比如网络数据包，如果处理不够迅速包就会丢失。在这种情况下，处于安全和安保原因，分别防止一个进程对属于自己内存空间以外的内存的意外或恶意的访问是很重要的。现代计算机中，内核通过一种称为内存管理单元（MMU）的硬件设备来保证内存隔离。MMU是一个所有进程的内存访问都需要经过的硬件模块。它会将虚拟地址翻译为虚力地址，如果目前的设置存在这样的一个。同时也检查在目前的设置下这样的访问是否合法。总是将内核空间映射来提高效率但在用户模式下无法访问也是一个常用的设计。为了使该功能正常工作，内核需要维护一个以每个进程编码的将虚拟地址翻译成物理地址，以及检查访问合法性的页表。在此需要强调的是MMU本身并不会保证内存隔离，它只是供内核使用的来实现内存隔离的工具。在内核中用于管理内存（也就是MMU）的代码出现的漏洞将会造成重大的安全性问题。

由于内核是运行在所谓的内核模式下的（也就是硬件的特权模式），从安全性的角度来看权限越小越好。这种观点是基于可信计算需要保持最小这样的一条一般原理。这就是本文着重于一个支持优先调度和保存内存隔离的微内核的原因。

贡献：我们主要的贡献是在Coq工具中给出了基于MMu的动态内存隔离的严格证明。更准确地，它包括：

（1）一个将硬件架构看作单体的严格模型：对于内存隔离很重要的部分（也就是MMU）被按照所有有用的细节建模，而其他无关的部分被忽略。

（2）一个支持优先调度和内存隔离的微内核的抽象模型，其选择了合适的抽象级别使得它保持为一个实际模型而不会与具体实现相关。

（3）对一致性属性必须被操作MMU的微内核保持的具体描述，为了使内存隔离被保持。

相关成果：已经有许多内核安全的严格证明。我们下面对比与我们工作较为相关的其他工作。

其中最显著的是对微内核seL4的功能正确性和安全属性的使用Isabelle/HOL工具的严格证明。用于云计算的管理程序CertiKOS也是被严格地验证了的。特别地，它的内存管理器BabyVMM是按层构建的以允许使用一系列改进来严格验证，这一系列是在Coq工具中成为定形的。相反，我们的工作不是给出对一个具体微内核的严格证明，而是表明什么是微内核假设的硬件架构以及微内核为了随时保证内存隔离而必须服从的约束条件。

1. 1. []Robert Glass, Ed. In The Beginning: Personal Recollections of Software Pioneers[M]. UK:IEEE Press, 1997, 1-2

   [↑](#endnote-ref-0)
2. 1. [] Ferranti Ltd, [Ferranti Atlas 1 & 2 – Systems Architecture](http://www.ourcomputerheritage.org/ccs-f5x2.pdf)[EB/OL]. <http://www.ourcomputerheritage.org/ccs-f5x2.pdf,> November 12, 2009

   [↑](#endnote-ref-1)
3. 1. [] Cragon, Harvey G. [Memory Systems and Pipelined Processors](https://books.google.com/?id=q2w3JSFD7l4C). Jones and Bartlett Publishers[M]. Texas:Jones & Bartlett Learning, 1996, 1-2

   [↑](#endnote-ref-2)
4. 1. [] Virtual memory[EB/OL]. <https://en.wikipedia.org/wiki/Virtual_memory,>

   [↑](#endnote-ref-3)
5. 1. [] Javier Picorel, Djordje Jevdjic, Babak Falsafi. Near-Memory Address Translation[C]. 2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT), Oregon, 2017, 303-304

   [↑](#endnote-ref-4)
6. 1. []Ioannis Deligiannis, George Kornaros. Adaptive memory management scheme for MMU-less embedded systems[C]. 2016 11th IEEE Symposium on Industrial Embedded Systems (SIES), Krakow, 2016, 1-1

   [↑](#endnote-ref-5)
7. 1. [] Arm limited. IOT SOLUTIONS MBED IOT DEVICE PLATFORM[EB/OL]. https://www.arm.com/products/iot-solutions/mbed-iot-device-platform, 2017

   [↑](#endnote-ref-6)
8. 1. [] 李亚民.计算机原理与设计——Verilog HDL版[M]. 北京:清华大学出版社, 2011, 368-369

   [↑](#endnote-ref-7)
9. 1. [] MIPS. The MIPS32 and microMIPS32 Privileged Resource Architecture v6.02[EB/OL]. [https://www.mips.com/?do-download=the-mips32-and-micromips32-privileged-resource-architecture-v6-02,](https://www.mips.com/?do-download=the-mips64-and-micromips64-privileged-resource-architecture-v6-03,) 2015, 29-30

   [↑](#endnote-ref-8)
10. 这里的段不同于段式内存管理中的段。 [↑](#footnote-ref-0)
11. 1. [] MIPS. The MIPS32 and microMIPS32 Privileged Resource Architecture v6.02[EB/OL]. [https://www.mips.com/?do-download=the-mips32-and-micromips32-privileged-resource-architecture-v6-02,](https://www.mips.com/?do-download=the-mips64-and-micromips64-privileged-resource-architecture-v6-03,) 2015, 31-32

    [↑](#endnote-ref-9)
12. 1. [] Gerard Blanchet, Bertrand Dupouy. Computer Architechture[M]. (陈宗斌). 北京:北京大学出版社, 2017, 126-127

    [↑](#endnote-ref-10)
13. 1. [] M. Milenkovic. Microprocessor memory management units[J]. IEEE Micro, 1990, 10:74-75

    [↑](#endnote-ref-11)