東南大學

毕业设计(论文)报告

题　　目　流水线处理器Cache及指令预取算法的研究实现

计算机科学与工程 院（系）计算机科学与技术 专业

学　　号　09005328

学生姓名　吴哲凯

指导教师　杨全胜

起讫日期　2009 年 1 月 10 日至 2009 年 6 月 2 日

设计地点　东南大学九龙湖校区

流水线处理器 Cache 及指令预取算法的研究实现

摘要

当前，SoC 设计领域的竞争日趋激烈。开发自主的处理器核、核心 IP 以及总线架构，将使我国的 SoC 发展更具竞争性。国内不少研究所和高校都在研制开发自主的 SoC，东南大学也不例外。经过多年的努力，东南大学计算机学院开发出了一种嵌入式 SoC 系统——MiniSys。它采用了 MIPS 32 位指令集中的 31 条指令，包括一个 32 位 RISC 型流水处理器、七种简单的接口以及一个集成开发环境——MiniSys IDE。

本文针对 MiniSys Soc 在 Cache 方面的缺失，设计并实现了适合 MiniSys 的 Cache 结构，并对 MiniSys 的存储结构做了进一步的改进。同时，为了降低 Cache 失效带来的损失，实现了一个指令预取算法——Wrong-Path 预取，并对其做了适当的改进。最后，在单核处理器的基础上，实现了一个简单的双核处理器。该处理器采用了对称式共享存储器的系统结构，利用监听协议保证 Cache 的一致性。同时，为了解决两个核的数据竞争，增加了一条原子互换指令，实现了硬件同步操作。

整个系统使用 Quartus7.2 开发，所有模块采用 RTL 级（寄存器传输级）描述，单核处理器通过了时序仿真和下载验证，双核处理器通过了时序仿真验证。

关键词： 流水线，Cache 结构，指令预取，双核处理器

Research and Implement of Pipeline Processor Cache, Instruction Prefetching Algorithm

Abstract

At present, there are fierce competitions in SoC design. It is very important for our country to develop independent processor core, IP core and bus architecture. In that case, the development of SoC design in our country will become more competitive. Many research institutes and colleges are developing their own SoC, including Southeast University. Computer Science and Engineering College of SEU have developed a SoC embedded systems called MiniSys until now. It uses 31 instructions of MIPS 32-bit Instruction Set, including a 32-bit RISC pipeline processor, seven simple interfaces and an integrated development environment called MiniSys IDE.

To make up for MiniSys Soc deficiencies in the Cache, a suitable Cache structure is designed and implemented for MiniSys in this paper. Further, some improvements are made to the storage structure of MiniSys. Besides, in order to reduce the losses caused by Cache Miss, an instruction prefetching algorithm is achieved and improved, which called Wrong-Path Prefetching. Finally, a dual-core processor is achieved based on the single-core processor. It adopts symmetric shared memory system structure. A monitor agreement is implemented to keep the consistency of Cache. At the same time, an atomic exchange instruction is described to solve the competition caused by two cores.

The entire system is developed by Quartus 7.2. All modules are described at RTL (Register Transfer Level). The single-core processor passes timing simulation and download authentication. The dual-core processor passes timing simulation.

Keywords: Pipeline, Cache Structure, Instruction Prefetching, Dual-core Processor

目录

第 1 章 绪论

1.1 引言

1946 年，第一台通用电子计算机诞生。经过 60 多年的时间，计算机技术取得了令人瞩目的发展。计算机的性能不断提高，性价比不断上升，应用的领域越来越广。当前的计算机既向巨型化发展也同时向微型化发展。其中嵌入式系统是计算机市场增长最快的领域。在日常生活中，这种智能设备随处可见，其范围涵盖从日常使用的电器到手持数据设备，以及视频游戏和数字机顶盒。

嵌入式系统是一个嵌入到对象体系中专用计算机系统，它以应用为中心，以计算机技术为基础进行设计[1]。大致分为两种类型，第一种类型以嵌入式处理器为核心，在系统板上配置满足系统要求的各种功能部件，比如 Power PC、MIPS、ARM 系列等；第二种类型是将整个专用计算机系统设计到一个芯片内，比如单片机系统和片上系统（SoC）。

SoC 是 20 世纪 90 年代发展起来的技术，它将微处理器、模拟 IP 核、数字 IP 核和存储器集成在单一芯片上。SoC 可以使应用产品小型化、多功能化，能够降低产品的功耗、提高运行速度和降低成本。SoC 广泛应用于计算机、通信、消费等多个领域。

SoC 设计的关键技术主要包括总线架构技术、IP 核可复用技术、软硬件协同设计技术、SoC 验证技术、可测性设计技术、低功耗设计技术、超深亚微米电路实现技术等。其设计观念与传统的设计观念完全不同。在 SoC 设计中，设计者面对的不再是电路芯片；而是能实现设计功能的 IP 模块库。设计者不必要在众多的模块电路中搜索所须要的电路芯片，只需要根据设计功能和固件特性，选择相应的 IP 模块。

相比于国外的研究水平，中国的 SoC 技术仍处于落后阶段。国内的大多数嵌入式系统都是基于国外的 SoC 芯片做应用开发，缺乏自主知识产权，而且很容易受制于人。因此开发自主的处理器核、核心 IP 以及总线架构，对于我国 SoC 技术的发展至关重要。目前，国内不少研究所和高校都在研制开发自主的 SoC。比如中科院计算所的中科 SoC、方舟 2 号、国芯 C3 Core 等。另外，培养学生的 SoC 设计能力同样也很重要，特别是计算机专业的学生。

在这些方面，东南大学计算机学院做了不少的努力。通过借鉴加州大学伯克力分校的经验，东大计算机学院为本科生开设了“计算机综合课程设计”实践课程。该课程要求学生设计一个 SoC 小系统 MiniSys，主要包括一个 32 位 RISC 型处理器、七种简单接口以及一个 MiniSys 汇编语言汇编器。它可以使学生在嵌入式专用芯片设计、计算机接口电路设计和系统软件设计等方面得到很好的训练，同时加深学生对计算机系统的理解。

1.2 MiniSys概述及存在问题

MiniSys 包含一个以 32 位 RISC 型处理器为核心，自带多个外围部件的 SoC 芯片和相关系统软件。系统软件包括一个 BIOS 和一个简单的 MiniSys 汇编语言汇编器。图 1.1 为 MiniSys 的系统结构。

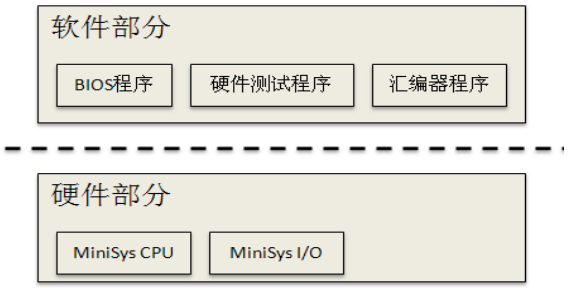


图 1.1　MiniSys 的系统结构

MiniSys 采用了 MIPS 32 位指令集中的 31 条常用指令，处理器的大致结构为 MIPS 经典的五级流水结构。处理器有 32 个 32 位寄存器，数据线和地址线分别为 32 位和 16 位。其中，32 个 32 位寄存器除了 5 个寄存器（$sp、$zero、$i0、$i1、$ra）被固定功能外，其余的都可以用做通用寄存器。外围部件包括一个 4 位 7 段 LED 控制器、一个 4×4 键盘控制器、两个 16 位定时/计数器、一个 16 位 PWM 控制器、一个看门口控制器、一个简易 UART 串行通信控制器和一个 32 位的并口。

MiniSys 的存储结构采用哈佛结构，在 CPU 中包含片内的 4KB ROM 和 4KB RAM，它们都采用字节编址，但以 32 位（4 字节）为一个存储单元。I/O 空间编址采用与存储器统一编址的方式，既将整个地址空间分为两个部分，一部分作为访问 RAM 的存储空间，另一部分作为访问 I/O 部件的 I/O 空间。系统内部提供两个中断源的控制电路，两个中断源为 INT0 和 INT1，其中 INT0 的优先级高于 INT1。

经过多年的努力，MiniSys 日趋成熟。硬件上主要实现了一个 32 位 RISC 型流水线处理器和七种简单接口。软件上实现了一个集成开发环境 MiniSys IDE。该集成开发环境包含一个汇编器和一个 Mini C 语言编译器，可以编译 Mini C 语言程序和汇编 MiniSys 汇编语言程序，最终生成机器代码；同时，它提供文本编辑功能，汇编器部分提供全局查错功能。

不过，MiniSys 也存在不足之处，特别是它的存储结构。MiniSys 的存储结构不含 Cache 结构，而是在 CPU 内部耦合了两个存储模块，一个用来存储指令，另一个用来存储数据。这样设计主要是因为系统选用的存储模块容量小、访问速度快。处理器和存储器之间的性能差距不是很大，所以没必要增加 Cache 结构。早期的计算机系统也是如此，1989 年之前的处理器是不带 Cache[2]。但是随着存储器容量的增加，处理器和存储器之间的性能差距日益显著。可以说，存储器层次结构的产生是计算机发展的必然结果。其目标就是提供一个存储系统，使之能够具有几乎相当于最便宜层次的存储器的价格，但是访问速度却与最快层次的存储器接近。图 1.2 给出了一个典型的多级存储器层次结构。

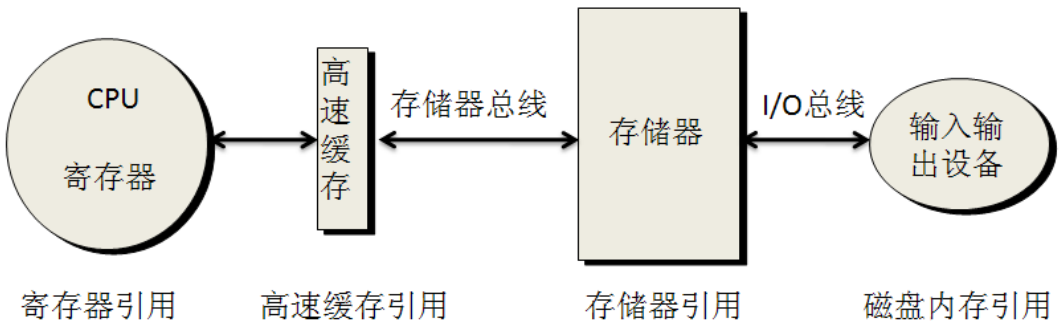


图 1.2　多级存储器层次结构

1.3 论文工作

为了完善 MiniSys 的系统结构，增强系统的可扩展性，需要对 MiniSys 进行必要的改进。本文的主要工作包括以下几点：

- 设计并实现一个 MiniSys 流水线处理器，包括四个基本接口以及中断控制器。
- 设计并实现符合 MiniSys 的 Cache 结构。
- 研究并实现一个适合 MiniSys 的预取算法，并做适当改进。
- 优化存储器结构，主要是增大存储器带宽。
- 设计并实现一个简单的双核处理器，解决 Cache 的一致性以及数据竞争。

整个系统使用 Quartus7.2 开发，所有模块采用 RTL 级（寄存器传输级）描述，单核处理器通过了时序仿真和下载验证，双核处理器通过了时序仿真验证。

1.4 论文结构

- 第 1 章介绍了课题的背景和论文的主要工作。
- 第 2 章介绍了系统的整体框架，主要是针对单核处理器。双核处理器将在第 7 章介绍。
- 第 3 章介绍了 MiniSys 流水线处理器的设计与实现，重点阐述了各种相关性的解决。同时简单介绍了中断控制器和接口单元的实现。
- 第 4 章介绍了适合 MiniSys 结构的 Cache 的设计与实现，同时提出了存储器的优化策略。
- 第 5 章研究了流水线处理器下的指令预取算法，并对多个算法进行比较。在此基础上，实现了一个适合 MiniSys 的预取算法——Wrong-Path 预取，并对该算法做了改进。
- 第 6 章介绍了所有的测试程序，并对测试结果进行了分析，以此验证系统的正确性。
- 第 7 章探讨了 MiniSys 双核处理器的设计与实现，同时重点提出了解决 Cache 一致性的策略以及硬件同步策略。
- 第 8 章总结了整个毕业设计的工作，并展望了未来。

第 2 章 系统整体框架概述

2.1 系统组成

整个系统包括一个 MiniSys CPU、四种 I0 接口、中断控制器、Cache 结构、指令预取控制器以及存储控制器。图 2.1 为系统的整体架构。

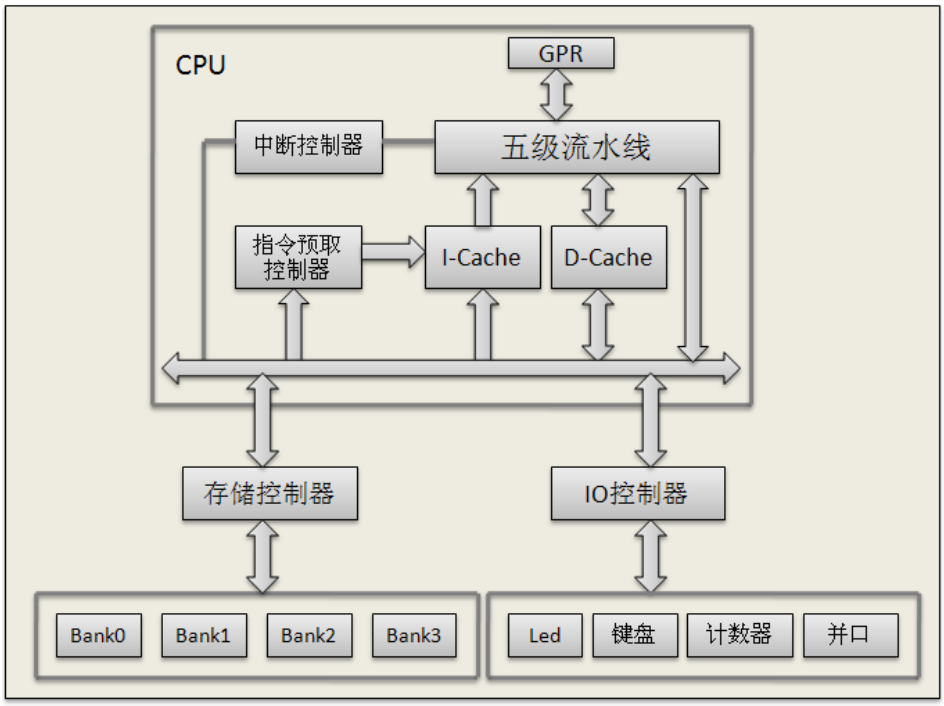


图 2.1 系统的整体架构

MiniSys CPU 采用 MIPS 经典的五级流水结构，即取指、译码、执行、存储、写回[3]。CPU 内部含有 32 个 32 位的通用寄存器、1 个程序计数器(PC)和若干流水线寄存器。MiniSys 设计有两个中断，INT0 和 INT1。INT0 用来响应时钟中断，INT1 用来响应键盘中断。另外，MiniSys 设计实现了七种接口，本设计实现了其中的 4 种。包括一个 4 位 7 段 LED 控制器、一个 4×4 键盘控制器、一个 32 位计数器和一个 32 位并口。

存储结构采用了哈佛结构，指令和数据有各自的高速缓存（分别称为 I-Cache 和 D-Cache）。这样，在 Cache 都命中的情况下，读取一条指令和存取一个数据的操作就能同时进行。指令 Cache 容量为 32 个字，数据 Cache 容量为 8 个字。Cache 采用 2 路组相联的映射方式，每一路由若干 Cache 行组成，每个 Cache 行为 128 位，可以放 4 条指令或数据。替换策略采用了 LRU 算法。另外，数据 Cache 采用了写直达＋写不分配的写策略。

指令预取采用了 Wrong-Path 算法，包括顺序预取和非顺序预取。顺序预取采用了 Next-Line 预取，当进入新的 Cache 行时，则将该行的下一行地址送往预取控制器。如果下一行不在 Cache 或预取缓冲器中，则发出预取请求；非顺序预取则是利用 ID 阶段的译码信息，当 ID 阶段判断出指令为 BEQ 或 BNE 时，则将目标行的地址送往预取控制器。如果目标行不在指令 Cache 或预取缓冲器中，则发出预取请求。另外，预取模块加入了一个 128 位的 Non-Referenced Cache。当一个未被引用的预取行被替换时，可以将它放在 Non-Referenced Cache，而不是简单的丢弃。

2.2 系统模拟运行

图 2.2 是一个汇编程序的代码和相应的机器代码。其中第一条指令从内存单元读取 1 个数，并把它存入 1 号寄存器中。第二条指令将 1 号寄存器的数值送 Led 灯显示。利用 MiniSys IDE 将汇编程序转换成机器代码。图 2.3-2.4 模拟了机器代码的执行过程。

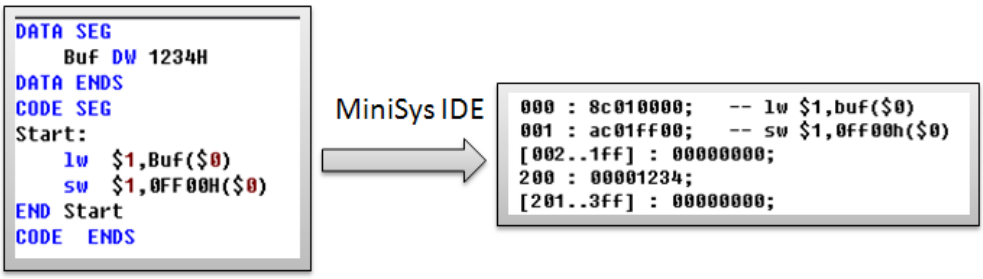


图 2.2 汇编代码到机器代码的转换

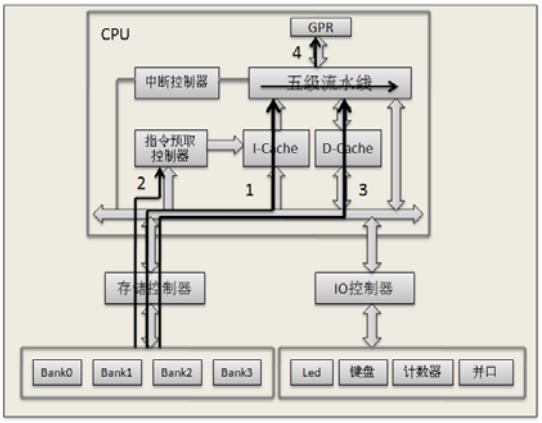


图 2.3 lw $1,Buf($0)的执行过程

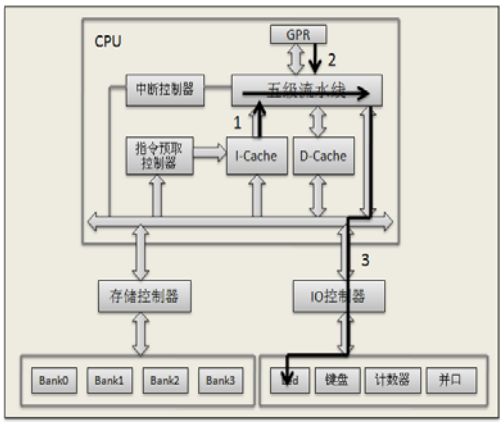


图 2.4 sw $1,0FF00H($0)的执行过程

图 2.3 显示了第一条指令的执行过程。系统启动后，利用 rst 信号进行系统的初始化工作。在第二个时钟的下降沿，CPU 的取指部件向指令 Cache 取指，结果造成 Cache 失效。第一次的 Cache 失效是无法避免的。Cache 控制器发出失效信号和读内存信号，获得总线控制权后就开始传输数据。完成后重新判断 Cache 是否失效，如果发现 Cache 命中，就将数据送往 CPU 的指令寄存器。即图 2.3 中的过程 1。

Cache 完成读操作后，预取控制器获得了总线的控制权，开始指令预取。即图 2.3 中的过程 2。同时 CPU 对第一条指令进行译码，经过 2 个周期进入存储阶段。由于这是一条读内存指令，所以 CPU 将访问数据 Cache，结果 Cache 失效。Cache 控制器发出失效信号和读内存信号，获得总线控制权后就开始传输数据。完成后重新判断 Cache 是否失效，如果发现 Cache 命中，就将数据送往 CPU 的一个流水线寄存器中。即图 2.3 中的过程 3。最后这条指令进入写回阶段，将 1234H 写入 1 号寄存器。即图 2.3 中的过程 4。

图 2.4 显示了第二指令的执行过程。这次 CPU 取指时 Cache 命中，所以不需要访存，直接将 Cache 中的指令送往 CPU 的指令寄存器。然后开始译码，读取 1 号寄存器的内容。由于存在数据相关性，流水线的 ID 阶段要停顿一个周期。停顿之后，将第一条指令的读内存数据直接引入，作为读 1 号寄存器的内容。经过 3 个周期进入存储阶段，由于这是一条写 IO 指令，所以直接将数据送往 LED 寄存器。最后 LED 显示数据 1234H。

上述 2 节简单介绍了系统的整体架构和模拟运行，具体的设计与实现将在后续章节依次介绍。

第 3 章 MiniSys CPU的设计与实现

3.1 流水线的概念

所谓流水线技术就是利用执行指令所需的操作之间的并行性，实现多条指令的重叠执行。它是高速 CPU 中采用的关键技术之一。流水线设计者的目标就是平衡各个流水线的长度。如果每一步都得到最佳的平衡，那么每条指令在流水线上的平均时间在理想情况下等于

$$\frac{\text{非流水线执行 1 条指令的时间}}{\text{流水线级数}}$$

MiniSys 采用了 MIPS 32 位指令集中的 31 条常用指令，所以 MiniSys 的流水线 CPU 选用 MIPS 经典的五级流水结构，同时做了稍许修改以适应 MiniSys CPU 的结构。理想情况下，每条指令的平均执行时间为单周期的 1/5。图 3.1 为流水线中指令的执行模式。其中，每一条指令在 5 个时钟周期内完成执行，当然这里不考虑因相关性而带来的流水线停顿。在每一个时钟周期内，硬件将启动一条新的指令，并执行 5 条不同指令的各自阶段。每一个阶段的任务如下：

- 取指（IF），根据 PC（程序计数器）指示的地址从存储器中读取一条指令，同时计算新的 PC。
- 译码（ID），对 IF 模块送入的指令进行译码，并且读取该指令源寄存器域指定的 CPU 寄存器内容。另外，数据的相关性也在这个阶段解决。
- 执行（EXE），完成各种算术逻辑运算。
- 存储（MEM），负责存储器和 IO 设备的读写操作。
- 写回（WB），将操作结果写入寄存器堆中。

时钟	1	2	3	4	5	6	7	8	9
指令i	IF	ID	EXE	MEM	WB				
指令i+1		IF	ID	EXE	MEM	WB			
指令i+2			IF	ID	EXE	MEM	WB		
指令i+3				IF	ID	EXE	MEM	WB	
指令i+4					IF	ID	EXE	MEM	WB

图 3.1　流水线中指令的执行模式

为了确保流水线不同段中的指令不会相互影响，我们需要在连续的流水段中引入流水线寄存器。在每个时钟周期结束之后，该段的所有执行结果都保存在流水线寄存器中，在下一个时钟周期作为下一个段的输入。这样在每个时刻，每条指令都只在一个流水段上是活动着，因此任何指令所作的动作都发生在一对流水线寄存器之间。图 3.2 为含流水线寄存器的流水线结构。

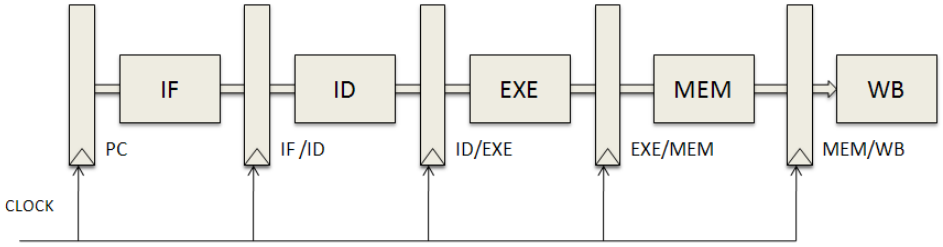


图 3.2　含流水线寄存器的流水线结构

3.2 流水线相关性的解决

CPU 采用流水线结构后，会产生三种相关。如果处理不好，会影响指令的正确执行同时会大大影响流水线的效率。

（1） 结构相关

所谓结构相关是指硬件不支持多条指令在同一个机器周期执行。当发生结构相关时，指令之间相互竞争硬件资源。在 MiniSys CPU 中，当取指和存储阶段都需要读写存储器；以及当译码和写回阶段都需要读写寄存器时，就会出现结构相关。

解决的方法很多，比如停顿流水线、资源重复。对于第一类的结构相关性，MiniSys CPU 采用了资源重复的方法，就是将指令和数据分别放在两个存储器体中，指令放在 ROM 模块，而数据放在 RAM 模块。这样读写数据和读指令就不会产生冲突。对于第二类的结构相关性，则采用了读取和写入不同步的方法，即读寄存器在时钟的上升沿完成，而写寄存器在时钟的下降沿完成。

因此 MiniSys CPU 的设计中避免了结构相关的问题。但是，当加入 Cache 结构后，结构相关的问题又会出现。此时的指令 Cache 和数据 Cache 相当于原来的 ROM 和 RAM，并且两个 Cache 共享一个大的存储器。当指令 Cache 和数据 Cache 同时失效时，就会产生存储器冲突。具体的解决方法将在 Cache 的设计和实现一章中介绍，这里就不在赘述。

（2） 数据相关

所谓数据相关是指一条指令的执行依赖于上一条或是上几条指令的执行结果。通常指令的运算结果先保存在流水寄存器中，直到 WB 阶段才会将运算结果写回寄存器堆中，如果后继指令在前面的指令执行写回操作前就要使用其运算结果，那么就会发生数据相关。如图 3.3 所示。

数据相关的解决策略有两种：转发和阻塞。

转发实质上就是先判断指令是否存在数据相关，若存在则将尚未写回寄存器堆的运算结果提前引入，作为后继指令的运算操作数。转发策略的关键是对数据相关性的判断，这种判断不但要得出是否发生数据相关，而且还能指出当前指令与流水线中哪个阶段的指令发生了数据相关以便转发。

阻塞其实就是当判断出相关后，暂停当前流水线直到之前引起数据相关的指令执行完。这种策略实现简单，但是会使执行效率大大降低。

MiniSys CPU 采用了转发策略，具体的判断放在 ID 阶段。为了完成相关性的判断，需要在 ID/EXE 寄存器中保存写回地址、相关的读写信号；在 EXE/MEM 寄存器中保存写回地址、ALU 运算结果、相关的读写信号。这些数据和信号都要输入到数据相关的判断模块中，另外还需要输入 EXE 阶段当前的运算结果、MEM 阶段的读 MEM/IO 数据。解决数据相关的核心代码如图 3.4 所示。送到 ALU 的两个操作数 SrcA 和 SrcB 有四种来源：读寄存器数

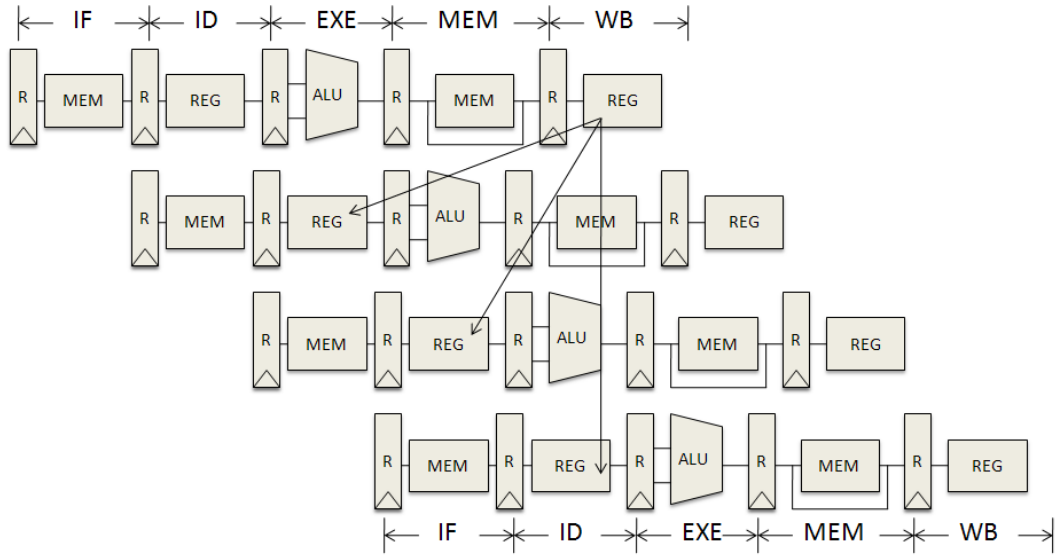


图 3.3　数据相关

据、ALU 阶段当前运算结果、EXE/MEM 寄存器保存的 ALU 运算结果、读存储器或 IO 接口数据。大多数的数据相关都可以通过转发解决，但是有一种特殊情况需要停顿流水线一个周期。当 EXE 阶段的指令是读存储器或 IO 指令，其写回地址正好等于 ID 阶段读寄存器的地址。因为读存储器或 IO 是在 MEM 阶段，所以流水线的 IF、ID 阶段需要停顿一个周期，以等待数据的到来。

```
//数据相关解决
wire exe_eq_rs,exe_eq_rt;
wire mem_eq_rs,mem_eq_rt;
wire [4:0] Rs,Rt;

assign Rs=INST[25:21];
assign Rt=INST[20:16];
assign exe_eq_rs=((EXE_WADDR==Rs)&&(EXE_WADDR!=5'b00000))?1'b1:1'b0;
assign exe_eq_rt=((EXE_WADDR==Rt)&&(EXE_WADDR!=5'b00000))?1'b1:1'b0;
assign mem_eq_rs=((MEM_WADDR==Rs)&&(MEM_WADDR!=5'b00000))?1'b1:1'b0;
assign mem_eq_rt=((MEM_WADDR==Rt)&&(MEM_WADDR!=5'b00000))?1'b1:1'b0;

wire RorI;
assign RorI=(Rtype||Itype)?1'b1:1'b0;
wire exe_rmemio,mem_rmemio;
assign exe_rmemio=(EXE_RMEM||EXE_RIO)?1'b1:1'b0;
assign mem_rmemio=(MEM_RMEM||MEM_RIO)?1'b1:1'b0;
wire [31:0] memio_data;
assign memio_data=MEM_RMEM?MEM_RDATA:IO_RDATA;

//SrcA
wire [31:0] SrcA1,SrcA2,SrcA;
assign SrcA1=(RorI&&mem_eq_rs&&MEM_WREG)?MEM_ALU_RESULT:RDATA1;
assign SrcA2=(mem_rmemio&&mem_eq_rs)?memio_data:SrcA1;
assign SrcA=(RorI&&exe_eq_rs&&EXE_WREG)?EXE_ALU_RESULT:SrcA2;

//SrcB
wire [31:0] SrcB1,SrcB2,SrcB;
assign SrcB1=(RorI&&mem_eq_rt&&MEM_WREG)?MEM_ALU_RESULT:RDATA2;
assign SrcB2=(mem_rmemio&&mem_eq_rt)?memio_data:SrcB1;
assign SrcB=(RorI&&exe_eq_rt&&EXE_WREG)?EXE_ALU_RESULT:SrcB2;
```

图 3.4　解决数据相关性的核心代码

（3） 控制相关

当指令执行到分支指令或是其他引起程序计数器 PC 值发生变化的指令时，这些跳转指令将和后面的若干条指令发生控制相关。解决控制相关问题需要做到以下两点：

- 尽早判断出分支条件是否满足。
- 尽早获得转移地址以便更新 PC。

MiniSys CPU 的指令集中含有两条条件转移指令（BEQ、BNE）和三条无条件转移指令（J、JAL、JR），这些指令会产生控制相关。解决控制相关的策略主要有三种：阻塞、预测、延迟。

本设计采用了如下策略：

- 对于无条件转移指令，在 ID 阶段就可以获得转移的目标地址，因此只需要清空 IF/ID 寄存器，并用目标地址更新程序计数器（PC）。
- 对于条件转移指令，当然也可以在 ID 阶段获得转移的目标地址，但是这会大大降低 CPU 的主频。这里采用了静态预测的策略——预测分支总是失败。真正的判断放在 EXE 阶段，如果发现预测失败，则需要清空 IF/ID 寄存器和 ID/EXE 寄存器，同时用目标地址更新程序计数器 PC。静态预测的准确率不是很高，因此会略微降低流水线的效率。当然可以采用更好的预测算法，提高预测准确率，比如 2 bit 分支预测[4]、GShare[5]等。

至此，MiniSys CPU 的相关性都得到了解决。流水线设计的主要难点就是各种相关性的解决，至于其他模块的设计和单周期类似，这里不做详细的介绍。

3.3 中断模块设计

在计算机系统中，中断是一个重要的部分。外设通过中断机构主动向 CPU 提供状态信息，或者在数据准备好后主动提请 CPU 去取数据。对于 CPU 而言，在中断允许的状态下，中断的到来会打断正在运行的程序，转到中断处理程序中。中断处理结束后，再回到被打断的程序处继续执行。

MiniSys 设计有两个中断，INT0 和 INT1。当中断到来时，CPU 响应中断的过程如下：

- 保存当前指令的下一条指令的 PC 值到$i0 或$i1 寄存器中。
- 设置相应的中断屏蔽位。
- PC 更新为 0FF8H 或 0FFCH，即中断程序的入口地址。

中断设计的难点在于第一条，即保存正确的返回地址。因为中断随时可能到来，所以当前指令的下一条指令的地址并一定是简单的 PC+4。比如被中断的是一条跳转指令，那么下一个指令的地址应该是跳转的目标地址。

（1） 中断返回地址保存

为了保存正确的返回地址，本设计采用了如下策略：当中断信号有效时，CPU 并不立即执行中断程序，而是停顿两个周期，即清空 IF/ID 寄存器两个周期。为此，中断信号需要保持三个周期。

图 3.5 为中断信号到来后，流水线的执行情况。中断信号在时钟的下降沿产生，并且保持三个周期。在时钟的第一个上升沿，当前指令 i 进入 ID 阶段，下一条指令进入 IF 阶段。其实，下一条指令并不会被真正执行，而是等到中断返回后才被执行。在时钟的第二个上升沿，中断程序的第一条指令 j 进入 IF 阶段，IF/ID 寄存器被清零。在时钟的第三个上升沿，IF 阶段的指令不变，IF/ID 寄存器仍然清零。在时钟的第四个上升沿，当前指令 j 进入 ID 阶段，下一条指令进入 IF 阶段。流水线恢复了执行，其中的 EXE 段和 MEM 段暂时为空。

上面提到 CPU 需要停顿两个周期，其实这是 CPU 至少需要停顿的周期数。因为中断到来时，当前的指令 i 可能是 J、JAL、BEQ 或 BNE，这样下一条指令的地址应该是跳转的目标地址。如果当前指令 i 是 J 或 JAL，那么在 ID 阶段就可以获得跳转的目标地址。也就是说，在时钟的第二个上升沿就能保存正确的返回地址。如果当前指令 i 是 BEQ 或 BNE，那么 EXE 阶段才可以确定跳转的目标地址（采用了静态预测的策略——预测分支总是失败）。也就是说，在时钟的第三个上升沿才能保存正确的返回地址。因此，为了保存正确的返回地址，CPU 至少需要停顿两个周期。

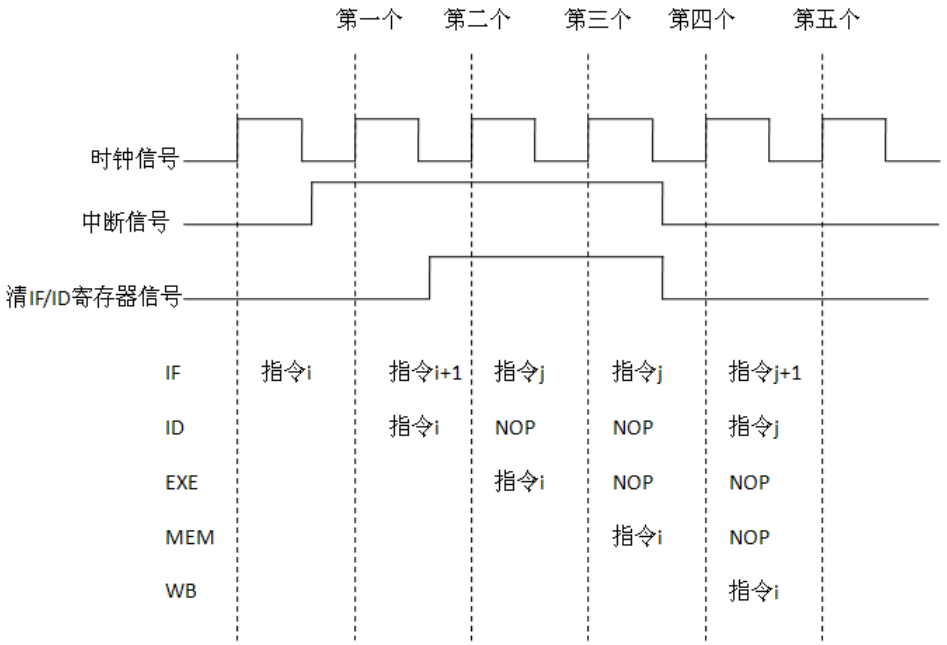


图 3.5　中断信号到来后，流水线的执行情况

停顿 CPU 的核心代码如图 3.6 所示。主要是利用一个状态机，该状态机含有四个状态：S0、S1、S2、S3。如果当前状态为 S0，并且有时钟或是键盘中断信号，那么中断信号（INT）发出，并且直到状态 S3 才撤销。INT_CLEAR 信号为 IF/ID 寄存器的清零信号。

```
//清空流水线2个周期
wire [1:0] NS,S;  //State Reg
wire S0,S3;
assign S0=!(S[0]||S[1]);
assign S3=S[0]&&S[1];

assign NS=(S0&&Int)?2'b01:((INT&&!S3)?(S+2'b01):2'b00);
dffe dffe_S0(.CLK(NCLK),.CLRN(CLRN),.PRN(1'b1),.ENA(ENA),.D(NS[0]),.Q(S[0]));
dffe dffe_S1(.CLK(NCLK),.CLRN(CLRN),.PRN(1'b1),.ENA(ENA),.D(NS[1]),.Q(S[1]));

//中断信号保持三个周期
wire INT,NINT0,NINT1;
assign NINT0=(S0&&Int0)?1'b1:(S3?1'b0:INT0);
assign NINT1=(S0&&Int1)?1'b1:(S3?1'b0:INT1);
assign INT=(INT0||INT1)?1'b1:1'b0;

dffe dffe_INT0(.CLK(NCLK),.CLRN(CLRN),.PRN(1'b1),.ENA(ENA),.D(NINT0),.Q(INT0));
dffe dffe_INT1(.CLK(NCLK),.CLRN(CLRN),.PRN(1'b1),.ENA(ENA),.D(NINT1),.Q(INT1));

//比中断信号慢一个周期，但同时撤销
wire NCLEAR;
assign NCLEAR=(INT&&!S3)?1'b1:1'b0;
dff dff_CLEAR(.CLK(NCLK),.CLRN(CLRN),.PRN(1'b1),.D(NCLEAR),.Q(INT_CLEAR));
```

图 3.6　停顿 CPU 的代码

（2）　中断屏蔽位设置

本设计暂时不支持中断嵌套，所以中断屏蔽的设置比较简单。核心代码如图 3.7 所示。INT0 被响应的条件是 Mask[0]、Mask [1]和 IntClose 都为 0，并且时钟中断信号为 1。INT1 被响应的条件是 Mask [0]、Mask [1]和 IntClose 都为 0，并且时钟中断信号为 0。这样，时钟中断的优先级比键盘中断高。另外，IntClose 为关中断信号，当指令 Cache 或数据 Cache 失效时，禁止中断。关中断的主要目的是为了简化中断返回地址的保存。

```
wire IRET0,IRET1;    //中断返回
assign IRET0=JR&&RS[4]&&RS[3]&&!RS[2]&&RS[1]&&!RS[0];
assign IRET1=JR&&RS[4]&&RS[3]&&!RS[2]&&RS[1]&&RS[0];

wire NMask0,NMask1; //中断屏蔽设置
assign NMask0=Int0?1'b1:(Mask0&&!IRET0);  //产生中断后，就设置中断屏蔽，直到中断返回再撤销屏蔽
assign NMask1=Int1?1'b1:(Mask1&&!IRET1);

wire Mask0,Mask1;
dffe dffe_Mask0(.CLK(NCLK),.CLRN(CLRN),.PRN(1'b1),.ENA(ENA),.D(NMask0),.Q(Mask0));
dffe dffe_Mask1(.CLK(NCLK),.CLRN(CLRN),.PRN(1'b1),.ENA(ENA),.D(NMask1),.Q(Mask1));

wire IntClose;  //IC_MISS DC_MISS Close Int
assign IntClose=(IC_MISS||DC_MISS)?1'b1:1'b0;

//Prior INT0>INT1 不支持中断嵌套
wire Int0,Int1,Int;
assign Int0=TIME_INT&&!Mask0&&!Mask1&&!IntClose;
assign Int1=!TIME_INT&&KEY_INT&&!Mask0&&!Mask1&&!IntClose;
assign Int=(Int0||Int1)?1'b1:1'b0;
```

图 3.7　中断屏蔽设置的代码

（3）　PC 值更新

PC 值的更新比较简单，本设计含有两个中断 INT0 和 INT1，其中 INT0 为时钟中断，INT1 为键盘中断。当中断信号有效时，PC 值更新为相应中断程序的入口地址，并且保持三个周期。

3.4 接口模块设计

MiniSys 含七种接口部件，包括一个 4 位 7 段 LED 控制器、一个 4×4 键盘控制器、两个 16 位定时/计数器、一个 16 位 PWM 控制器、一个看门口控制器、一个简易 UART 串行通信控制器和一个 32 位的并口。

由于 MiniSys 指令系统中没有专门的 I/O 指令，只能使用 LW 和 SW 两条指令进行存储器访问和 I/O 访问，所以 MiniSys 采用了 I/O 统一编址方式。MiniSys 的地址线为 16 位，可以寻址 64KB 的空间。在这 64KB 的地址空间中，将最后的 256 个字节分配给 I/O 端口，即 FF00H～FFFFH 的地址空间。整个地址空间如图 3.8 所示，其中中间部分暂时没有使用。

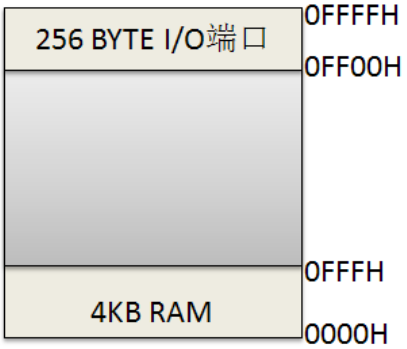


图 3.8　RAM 与 I/O 地址空间

```
IO io(
    .CLK(clk),
    .RST(rst),
    .IOR(exe_control[8]),          //读接口
    .IOW(exe_control[9]),          //写接口
    .COL(col),                     //列线
    .ADDR(exe_alu_result[15:0]),   //IO地址
    .WDATA(exe_wdata),             //写IO数据
    .RDATA(io_rdata),              //读IO数据
    .LINE(line),                   //行线
    .LED(led),                     //Led灯输出
    .P(parallel),                  //并口输出
    .KEY_INT(key_int),             //键盘中断
    .TIME_INT(time_int)            //时钟中断
);
```

图 3.9　接口顶层文件

本设计实现了四种接口，包括一个 4 位 7 段 LED 控制器、一个 4×4 键盘控制器、一个

32 位计数器和一个 32 位并口。图 3.9 为接口的顶层文件。在原有的 MiniSys CPU 中加入接口模块，只需修改 ID 模块和 WB 模块。接口的读写操作都是在时钟的下降沿。

接口设计的核心是总控制器模块的设计。该模块主要用来选择 4 个不同的接口。在设计时采用了集中译码的方式，它将 16 位地址线的高 12 位作为接口电路的基地址，低 4 位作为端口号。接口基地址加上端口号，最终形成 I/O 端口的地址。图 3.10 为总控制器的核心代码。

```
wire iorw,hn;
assign iorw=IOR||IOW;

assign hn=(ADDR[15]&&ADDR[14]&&ADDR[13]&&ADDR[12]&&ADDR[11]&&
           ADDR[10]&&ADDR[9]&&ADDR[8]&&!ADDR[7]);
wire ledcs,pcs,keycs,tcs;
assign ledcs=iorw&&hn&&!ADDR[6]&&!ADDR[5]&&!ADDR[4]; //FF00H
assign keycs=iorw&&hn&&!ADDR[6]&&!ADDR[5]&&ADDR[4];  //FF10H
assign tcs=iorw&&hn&&!ADDR[6]&&ADDR[5]&&!ADDR[4];    //FF20H
assign ppcs=iorw&&hn&&ADDR[6]&&ADDR[5]&&ADDR[4];     //FF70H

wire [31:0] DMux;
assign DMux=tcs?TimeRd:32'h00000000;
assign RDATA=keycs?{16'h0000,KeyRd}:DMux;
```

图 3.10　总控制器的核心代码

根据图 3.10 的译码信息可以推断出各个接口电路的基地址。表 3.1 列出了各个接口电路的基地址。

表 3.1　接口基地址分配表

部件名称	片选信号	接口基地址	部件名称	片选信号	接口基地址
4 位 7 段 LED 数码管	ledcs	0FF00H	32 位计数器	tcs	0FF20H
4×4 键盘控制器	keycs	0FF10H	32 位并口	ppcs	0FF70H

有了总控制器，添加接口就变得简单多了。各个接口电路可以单独调试，测试通过后再添加入到总控制器模块中，添加的关键在于连接片选信号和一些数据通路。本文不对各个接口电路的设计作详细地介绍。

以上四节论述了 MiniSys CPU、中断控制器、接口的设计与实现。所有模块整合在一起，就成为一个完整的设计。整个系统使用 Verilog HDL 语言[6]编写，故采用 Verilog HDL 语言的元件例化来描述各个单元的信号连接关系。需要注意的是顶层文件的文件名一定要和项目的名称一致。文件全部例化后，便可以编译、生成仿真波形进行检测，有关系统的测试将在第六章详细介绍。

第 4 章 MiniSys Cache的设计与实现

4.1 Cache简介

Cache 是一种特殊的存储器，也称高速缓冲存储器，它一般位于 CPU 和存储器之间。它由 Cache 存储部件和 Cache 控制部件组成。Cache 存储部件一般采用与 CPU 同类型的半导体存储器件，存取速度比存储器快几倍甚至十几倍。而 Cache 控制部件包括主存地址寄存器、Cache 地址寄存器，主存-Cache 地址变换部件及替换控制部件等。

Cache 的主要工作就是将存储器中最近读写过的数据在 Cache 存储部件中保留一个备份，使这些数据能够快速地返回给处理器。根据时间局部性原理，当前使用的数据或指令很可能在不久的将来再次被使用，所以将它们放在处理器能很快访问的 Cache 中是非常有意义的。另外，根据空间局部性原理，行内的其他数据也很可能在不久的将来被访问到。可以说，局部性原理是 Cache 性能的一个重要保障。本文中提到的 Cache 代表高速缓冲存储器，而存储器代表主存储器（内存）。

当处理器在 Cache 中找到要访问的数据时，称之为 Cache 命中；当处理器找不到所需要的数据时，称之为 Cache 失效。一个 Cache 包含若干行，这些行来源于存储器。发生 Cache 缺失时，数据的访问时间取决于存储器的时延和带宽。时延决定了读取第一个字的时间，而带宽决定了读取该行的其他部分所需要的时间。Cache 缺失通常由硬件处理，它会导致顺序执行处理器的停顿，直到所需数据可用为止。

Cache 设计主要包括以下几个方面：映射方式、行容量、Cache 容量、Cache 级数、写策略、一致性保持、替换算法。下面介绍关键的几个方面：

（1） 映射方式

根据存储器中的一个行（16 个字节）可以放在 Cache 中的位置，可以将 Cache 的组织形式划分成三种。如图 4.1 所示。其中，存储器的行容量和 Cache 的行容量一致。

- 如果一个行只能放在 Cache 中的唯一位置，那么这种映射方式称为直接映射。
- 如果一个行可以放在 Cache 中的任一位置，那么这种映射方式称为全相联映射。
- 如果一个行必须放在 Cache 中的某些位置，那么这种映射方式称为组相联映射。

组相联其实是直接映射和全相联映射的一种折中方案。组相联只有一路时，就是直接映射；组相联的每一路只有一行时，就是全相联映射。

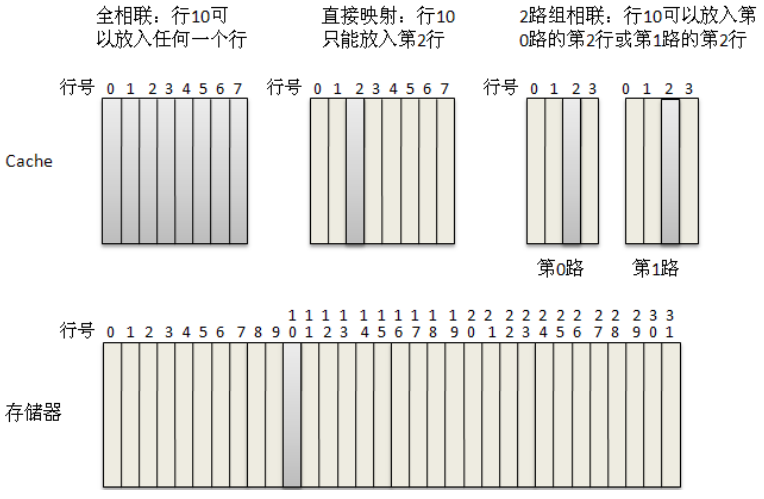


图 4.1 Cache 映射方式

（2） 写策略

通常有两种基本策略用来写 Cache：

- 写直达法（write through）：信息被同时写到 Cache 行中和存储器中。
- 写回法（write back）：信息只被写到 Cache 行中。只有当 Cache 中的脏行被替换出去时，信息才被写回到存储器中[7]。

两种方法各有自己的优点。在写回法中，写操作和 Cache 的速度是一致的，而且对同一行的多次写操作仅仅需要对下层存储器进行一次操作。因此，写回法需要的带宽较小。相对于写直达法，写回法对其他存储层次和存储总线的使用较少，节省了功耗，因而非常适合嵌入式应用程序。

相比于写回法，写直达法更容易实现。而且采用写直达法，下一级存储器总是有最新的当前数据副本，简化了 Cache 一致性的保持。

以上两种策略是在写命中的情况下采用的，在写缺失时则采用以下两种策略：

- 写分配：在发生写缺失时，存储器的行被读到 Cache 中，然后执行写命中时的操作。
- 写不分配：在发生写缺失时，只修改低层存储器的相应行，而不将该行读到 Cache 中，因而写缺失不影响 Cache 的内容。

一般情况下，写分配通常和写回法搭配，而写不分配通常和写直达法搭配。

（3） 替换算法

当缺失发生时，Cache 控制器必须选择 Cache 中的一行，并用欲获得的数据来替换它。如果使用直接映射方式，就不需要替换算法。如果使用全相联或组相联，就需要选用适当的替换策略。主要有三种替换策略：

- 随机替换策略：随机选择一行，将它替换出去
- 最近最少使用（LRU）替换策略：将最近最少用到的 Cache 行替换出去。
- 先进先出（FIFO）替换策略：将最早进入 Cache 的行替换出去。

4.2 MiniSys的存储结构

MiniSys 原先的存储结构很简单，并不含有 Cache 结构；而是在 CPU 内部耦合两个存储器，一个用来存储指令（ROM），另一个用来存储数据（RAM）。功能类似于指令 Cache 和数据 Cache。指令和数据都可以在一个周期访问到，相当于 Cache 总是命中。既然如此，为什么还要引入 Cache 结构呢？凭直觉，不加 Cache 比加要好。因为引入 Cache 后，势必会有 Cache 失效，这会造成流水线的停顿；而且会有新的结构相关产生，当指令 Cache 和数据 Cache 同时失效时，两者就会竞争存储器的访问权。

但是，随着计算机的发展，存储器的容量不断增加。这使得 CPU 与存储器之间的性能差距日益显著。在 Minisys 结构中，存储器采用的是 Altera 公司提供的宏模块，使用的容量很小（4KB），从该存储模块读写数据，延迟时间为 15ns；而 CPU 的时钟周期也就 20ns 左右。因此两者的速度差异并不是很明显。

如果 MiniSys 的存储器容量比 4KB 大得多，那么 CPU 和存储器之间性能差距将会变得很明显。如果 CPU 与存储器之间没有缓存，那么每次读写数据，流水线都会因访存延迟而停顿，CPU 的性能就会直线下降。

同时，如果流水线采取了乱序执行技术——现代处理器最常见的技术之一，那么 CPU 就需要维持一个指令窗。在每个时钟周期，CPU 可能需要获得多条指令，这就要求一个快速而且大带宽的存储体。在 CPU 和存储器之间加入 Cache 是一种很好的解决策略。

因此，为了增强 MiniSys 的可扩展性，同时平衡 CPU 与存储器之间的性能差异，应该

在 CPU 和存储器之间加入 Cache 结构。

4.3 Cache的设计与实现

4.3.1 基本结构设计

本设计采用哈佛结构，Cache 级数为一级，因此 Cache 的下一级存储器其实就是内存。指令和数据有各自的高速缓存，分别称为 I-Cache 和 D-Cache。如果 Cache 都命中的话，那么读取一条指令和读取一个数据的操作就能同时进行。指令 Cache 的容量为 32 个字，数据 Cache 的容量为 8 个字。由于受到 FPGA 逻辑单元数的限制，暂时只能这么大。当选用具有更多逻辑单元数的 FPGA 芯片时，可以扩展 Cache 的容量。

每一个 Cache 由若干 Cache 行组成。Cache 行是从存储器读入 Cache 的最小信息单元。每一个 Cache 行由标签域和数据域组成。指令 Cache 行的数据域为 4 个字（16 字节、128 位），标签域为 6 位（最高位为有效位）。数据 Cache 行的数据域也为 4 字（16 字节、128 位），但是标签域为 8 位（最高位为有效位）。主存储器容量为 4KB，其中低 2KB 放指令，高 2KB 放数据。程序中的指令地址和数据地址都为虚拟地址，实际上指令的虚拟地址和物理地址一样，而数据虚拟地址的最高位置 1 就是物理地址。图 4.2 为 MiniSys Cache 行的结构。

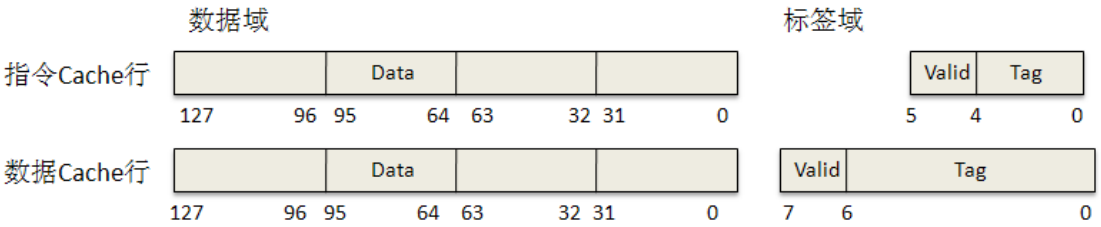


图 4.2　MiniSys Cache 行的结构

映射方式采用两路组相联。指令 Cache 每一路有 4 行，数据 Cache 每一路有 1 行。其实数据 Cache 相当于采用了全相联映射。两路的对应行（同一组中行）共享一位 LRU 位。Cache 的结构如图 4.3 所示。

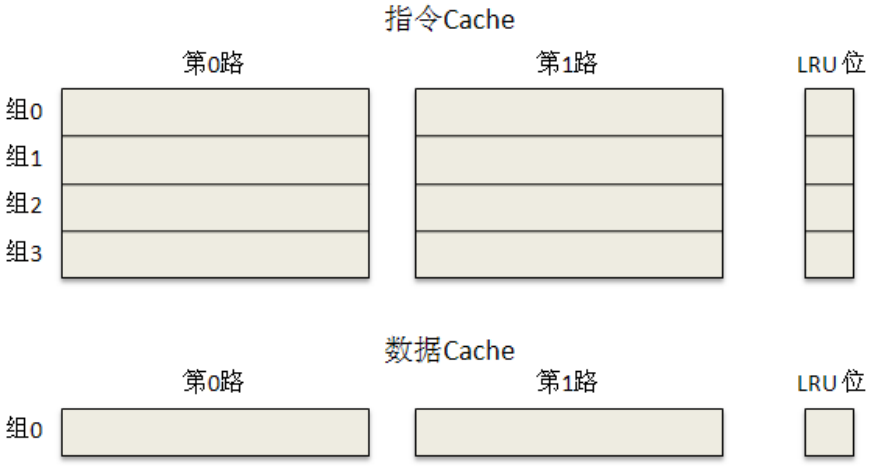


图 4.3　Cache 的结构

4.3.2 替换策略和写策略

一般来说，对于大容量的 Cache，LRU 和随机替换的性能差不多；对于小容量的 Cache，LRU 替换策略最好，先进先出算法比随机算法的性能要好一些[2]。本设计的 Cache 容量比较小，所以采用 LRU 替换算法。由于映射方式为 2 路组相联，所以只要为每一组设定一个 LRU 位。当第 0 路的 Cache 行命中时，就将该行对应的 LRU 位置为 0；当第 1 路的 Cache 行命中时，就将该行对应的 LRU 位置为 1。当 Cache 替换时，根据行索引检查相应的 LRU 位，当 LRU 位为 0 时，则替换第一路中的相应行；当 LRU 位为 1 时，则替换第 0 路的相应行。

在 Cache 的操作中，读操作占有的比例很大。比如，对于指令 Cache，基本上全是读操作，对于数据 Cache，平均只有一半的操作是写操作，但是由于写操作的特殊性，它会产生所谓的“脏”数据。如果处理不好，会影响到 Cache 的一致性保持以及系统整体的性能，因此采用合适的写策略是 Cache 设计的关键因素之一。

本设计采用了写直达+写不分配的策略。Cache 写命中时，信息被同时写到 Cache 行中和存储器的行中。Cache 写失效时，信息直接写到存储器的行中，而不用写到 Cache 行中。这种策略的硬件实现代价很小，而且简化了 Cache 一致性的保持。

4.3.3 Cache操作

不论是指令 Cache 还是数据 Cache，都是利用状态机实现数据的读写（指令 Cache 没有写操作）。指令 Cache 的控制器含有一个读状态机，数据 Cache 的控制器含有一个读状态机和一个写状态机，如图 4.4 所示。

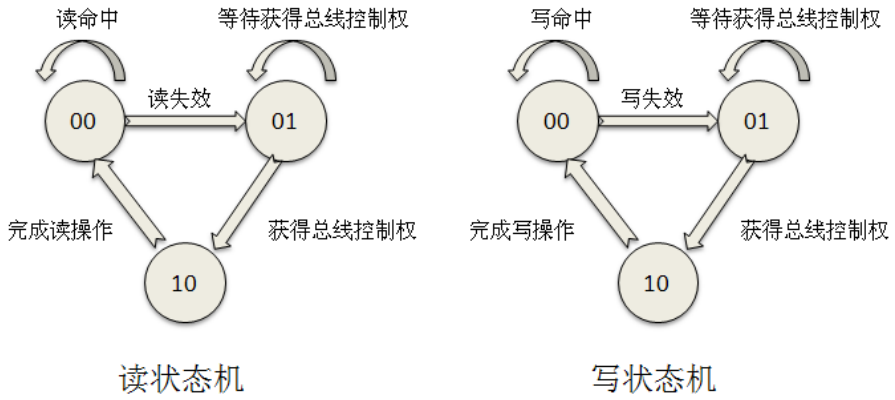


图 4.4　Cache 的读写状态机

CPU 通过如下步骤操作 Cache：

1) CPU 通过 Cache 控制器试图访问指令 Cache 中的指令或数据 Cache 中的数据。
2) Cache 控制器查看指令或数据是否在 Cache 中，图 4.5 为指令 Cache 的查询过程。如果指令/数据在 Cache 中，CPU 将得到该指令/数据，即 Cache 命中。如果指令/数据不在 Cache 中，则 Cache 控制器从存储器读取，即 Cache 失效。在读取数据之前，需要先获得总线的控制权。否则 Cache 控制器就一直等待，直到总线空闲。
3) CPU 从 Cache 中得到指令或数据后，恢复流水线的执行。

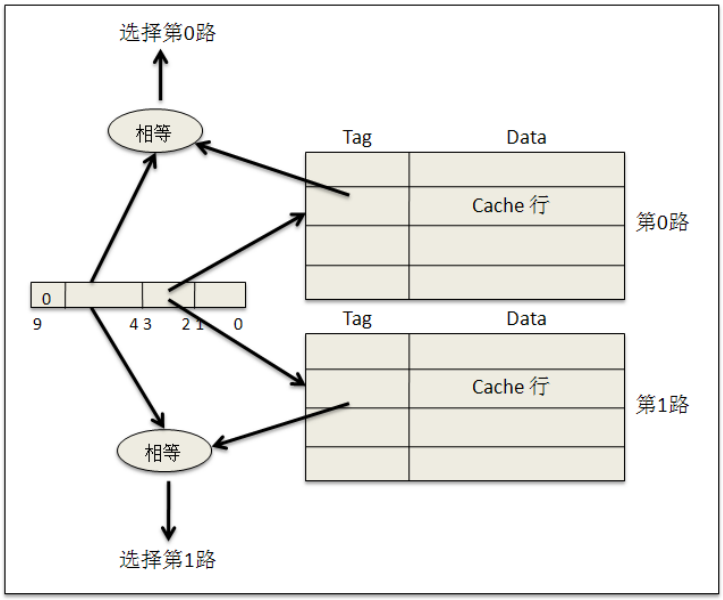


图 4.5　指令 Cache 控制器的查询过程

整个 Cache 采用模块化设计，基本的 Cache 行为一个 32 位触发器。图 4.6 为指令 Cache 的数据域描述。一共是四个 Cache 行，每一行有 4 个字（每个字为 32 位）。图 4.7 为指令 Cache 的标签域描述。

由于所有的模块采用 RTL 级描述，因此大大增加了实现的难度。在具体实现中，首先采用行为级描述，主要是利用 case 语句构建自动机。测试通过后，再将行为级描述逐渐改成 RTL 级描述。采用 RTL 级描述的好处在于更容易找出系统的瓶颈所在，但是难点在于必须搞清所有数据流的运动路径、运动方向和运动结果。

```
//DATA
Dffe32 Dffe_L0_Q0(.CLK(NCLK),.CLRN(CLRN),.ENA(ENA[0]),.D(D0),.Q(L0[31:0]));
Dffe32 Dffe_L0_Q1(.CLK(NCLK),.CLRN(CLRN),.ENA(ENA[0]),.D(D1),.Q(L0[63:32]));
Dffe32 Dffe_L0_Q2(.CLK(NCLK),.CLRN(CLRN),.ENA(ENA[0]),.D(D2),.Q(L0[95:64]));
Dffe32 Dffe_L0_Q3(.CLK(NCLK),.CLRN(CLRN),.ENA(ENA[0]),.D(D3),.Q(L0[127:96]));

Dffe32 Dffe_L1_Q0(.CLK(NCLK),.CLRN(CLRN),.ENA(ENA[1]),.D(D0),.Q(L1[31:0]));
Dffe32 Dffe_L1_Q1(.CLK(NCLK),.CLRN(CLRN),.ENA(ENA[1]),.D(D1),.Q(L1[63:32]));
Dffe32 Dffe_L1_Q2(.CLK(NCLK),.CLRN(CLRN),.ENA(ENA[1]),.D(D2),.Q(L1[95:64]));
Dffe32 Dffe_L1_Q3(.CLK(NCLK),.CLRN(CLRN),.ENA(ENA[1]),.D(D3),.Q(L1[127:96]));

Dffe32 Dffe_L2_Q0(.CLK(NCLK),.CLRN(CLRN),.ENA(ENA[2]),.D(D0),.Q(L2[31:0]));
Dffe32 Dffe_L2_Q1(.CLK(NCLK),.CLRN(CLRN),.ENA(ENA[2]),.D(D1),.Q(L2[63:32]));
Dffe32 Dffe_L2_Q2(.CLK(NCLK),.CLRN(CLRN),.ENA(ENA[2]),.D(D2),.Q(L2[95:64]));
Dffe32 Dffe_L2_Q3(.CLK(NCLK),.CLRN(CLRN),.ENA(ENA[2]),.D(D3),.Q(L2[127:96]));

Dffe32 Dffe_L3_Q0(.CLK(NCLK),.CLRN(CLRN),.ENA(ENA[3]),.D(D0),.Q(L3[31:0]));
Dffe32 Dffe_L3_Q1(.CLK(NCLK),.CLRN(CLRN),.ENA(ENA[3]),.D(D1),.Q(L3[63:32]));
Dffe32 Dffe_L3_Q2(.CLK(NCLK),.CLRN(CLRN),.ENA(ENA[3]),.D(D2),.Q(L3[95:64]));
Dffe32 Dffe_L3_Q3(.CLK(NCLK),.CLRN(CLRN),.ENA(ENA[3]),.D(D3),.Q(L3[127:96]));
```

图 4.6 指令 Cache 的数据域

```
//TAG
Dffe6  Dffe_L0_T0(.CLK(NCLK),.CLRN(CLRN),.ENA(ENA[0]),.D(D),.Q(T0));
Dffe6  Dffe_L1_T1(.CLK(NCLK),.CLRN(CLRN),.ENA(ENA[1]),.D(D),.Q(T1));
Dffe6  Dffe_L2_T2(.CLK(NCLK),.CLRN(CLRN),.ENA(ENA[2]),.D(D),.Q(T2));
Dffe6  Dffe_L3_T3(.CLK(NCLK),.CLRN(CLRN),.ENA(ENA[3]),.D(D),.Q(T3));
```

图 4.7　指令 Cache 的标签域

4.3.4 结构相关解决

原先的 MiniSys 的结构冲突已经全部解决，但是引入 Cache 结构后，又会产生新的结构冲突。当指令 Cache 和数据 Cache 同时失效时，两者都要访问存储器，这就出现了资源竞争。

为了解决上述问题，我们规定数据 Cache 的访存优先级比指令 Cache 高。因为数据 Cache 失效时，需要停顿四个流水段（IF、ID、EXE 和 MEM）；而指令 Cache 失效时，只需要停顿一个流水段（IF）。也就是说，数据 Cache 失效对流水线造成的影响比指令 Cache 失效大，尽快地处理数据 Cache 失效有助于提升 CPU 的性能。

这样，当两者的访存请求同时到达时，总线仲裁机构会将总线控制权交给数据 Cache，而让指令 Cache 等待。当数据 Cache 完成数据读写后，释放总线控制权，此时指令 Cache 就可以获得总线的控制权。

4.4 存储器结构改进

本设计中，存储器使用 Altera 公司提供的宏模块来实现。最初的设计如下：总容量为 4KB，其中低 2KB 放指令，高 2KB 放数据，带宽为 1word/cycle。具体定义如图 4.8 所示。

```
lpm_ram_dq ram(.data(wdata),.address(addr),.we(we),.inclock(clk),.q(data));
defparam ram.lpm_width=32,
         ram.lpm_widthad=10,
         ram.lpm_outdata="UNREGISTERED",
         ram.lpm_indata="REGISTERED",
         ram.lpm_address_control="REGISTERED",
         ram.lpm_file="DRam.mif";
```

图 4.8　原先存储器的定义

这种设计严重影响了系统的性能。因为每次 Cache 失效时，流水线至少需要停顿 6 个周期，其中的 2 个周期用来通信、4 个周期用来传输一个 Cache 行。加入预取模块后，存储器的低带宽成为了系统主要瓶颈。很多时候，尽管预取命中（预取的 Cache 行中正好含有 CPU 要访问的指令），流水线还是得停顿下来，因为预取还没完成。

因此，为了提高系统的性能，有必要对原先的存储器结构进行改进。还是使用 Altera 公司的提供的宏模块实现，但是采用交叉存储的结构。如图 4.9 所示。在该结构中，新的存储器被设计成由 4 个 Bank 构成，每个 Bank 容量为 1KB。数据按地址横跨在 4 个 Bank 中（既按地址交替存放）。

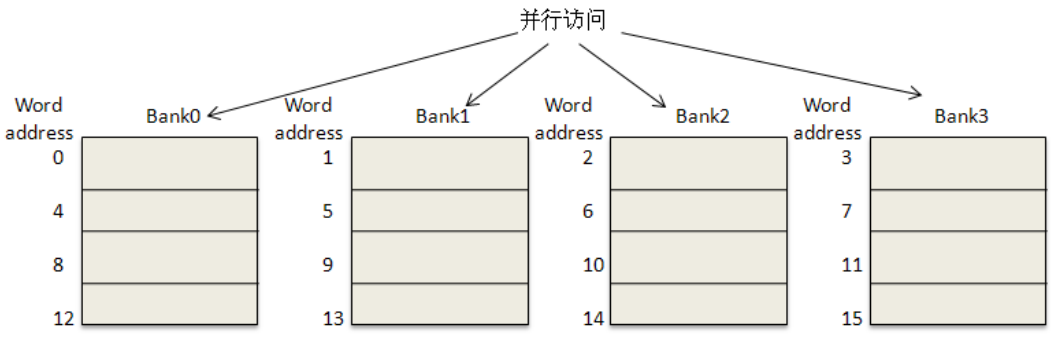


图 4.9 交叉存储结构

采用上述结构后，由于 4 个 Bank 可以并行操作，因此在一个时钟周期内就可以同时取到 4 个字，存储器的容量没有变化，但是带宽扩大了 4 倍。这样一次 Cache 失效只需要停顿流水线 3 个周期，其中两个周期用来通信、一个周期用来读写数据。新存储器的定义如图 4.10 所示。

```
lpm_ram_dq bank0(.data(D),.address(ADDR[9:2]),.we(WCS[0]),.inclock(CLK),.q(Q[31:0]));
defparam bank0.lpm_width=32,
         bank0.lpm_widthad=8,
         bank0.lpm_outdata="UNREGISTERED",
         bank0.lpm_indata="REGISTERED",
         bank0.lpm_address_control="REGISTERED",
         bank0.lpm_file="Bank0.mif";

lpm_ram_dq bank1(.data(D),.address(ADDR[9:2]),.we(WCS[1]),.inclock(CLK),.q(Q[63:32]));
defparam bank1.lpm_width=32,
         bank1.lpm_widthad=8,
         bank1.lpm_outdata="UNREGISTERED",
         bank1.lpm_indata="REGISTERED",
         bank1.lpm_address_control="REGISTERED",
         bank1.lpm_file="Bank1.mif";

lpm_ram_dq bank2(.data(D),.address(ADDR[9:2]),.we(WCS[2]),.inclock(CLK),.q(Q[95:64]));
defparam bank2.lpm_width=32,
         bank2.lpm_widthad=8,
         bank2.lpm_outdata="UNREGISTERED",
         bank2.lpm_indata="REGISTERED",
         bank2.lpm_address_control="REGISTERED",
         bank2.lpm_file="Bank2.mif";

lpm_ram_dq bank3(.data(D),.address(ADDR[9:2]),.we(WCS[3]),.inclock(CLK),.q(Q[127:96]));
defparam bank3.lpm_width=32,
         bank3.lpm_widthad=8,
         bank3.lpm_outdata="UNREGISTERED",
         bank3.lpm_indata="REGISTERED",
         bank3.lpm_address_control="REGISTERED",
         bank3.lpm_file="Bank3.mif";
```

图 4.10 新存储器的定义

为了将程序按照新的存储结构存储，需要修改汇编器的输出文件。具体实现如下：在汇编器输出时，将指令或是数据地址做模 4 运算。运算结果为 0 输出到 Bank0.mif、为 1 输出到 Bank1.mif、为 2 输出到 Bank2.mif、为 3 输出到 Bank3.mif。

经过测试，系统的性能得到大幅提升。具体的测试将在系统测试一章中详细介绍。

第 5 章 指令预取算法的研究与实现

5.1 指令预取算法的研究

指令预取是预测未来的指令，提前将它们从存储器中取到指令预取缓冲中。实质上就是利用指令读取和指令执行之间的并行性。当 CPU 执行指令时，除了 Cache 失效，总线一般是空闲的。因此，可以利用这段空闲时间，读取可能的后继指令到指令预取缓冲中。 指令预取通常是由 Cache 之外的硬件完成。一般情况下，Cache 失效时，Cache 控制器和预取控制器都会发出读存储器的信号。完成读操作后，造成 Cache 失效的行被装入指令 Cache 中，预取的行被装入预取缓冲中。当 CPU 请求的指令不在 Cache 中时，先查询预取缓冲。如果预取缓冲中的行包含这条指令，则将该行放入相应的 Cache 中，然后发出下一个预取请求。UltraSparc III[8]采用了这种预取技术。

近几十年来，人们对指令预取技术进行了大量的研究。预取策略必须完成两项工作以保证预取的有效性。第一，预取策略必须尽可能准确地预测将被引用的行；第二，预取发起时间应尽可能早，以保证在 CPU 使用预取指令前完成预取操作，这样就可以大大减少甚至完全消除失效造成的延迟。从理论上讲，最优的预取算法可以将所有指令预取到 Cache 中，以此消除所有的 Cache 失效[9]。不幸的是，程序的非顺序执行会使预取部件预测错误。人们做了大量的工作，设计出了不少算法。这些算法预测程序流的方向并预取这个方向的指令。

5.1.1 各种预取算法比较

指令预取的实现一般有两种方式：第一，修改 Cache 的组织结构，比如增大行容量；第二，增加额外的硬件实现一个预取算法。下面研究了多个算法，并对算法进行了比较。

（1） 长 Cache 行

使用长的 Cache 行是最简单预取方式，属于预取实现的第一种方式。该方式的优点是减少了 Cache 标签域的存储空间。缺点在于重填时间长，而且替换的粒度大，容易造成 Cache 污染。下面的预取算法都是采用预取实现的第二种方式。

（2） Next-Line 预取

这种方法总是尝试预取当前行的后继行，尽量保证在 CPU 使用后继行的指令前完成预取。其中，当前行定义为 CPU 当前所取指令所在的 Cache 行，而后继行定义为当前行的顺序下一行。如果下一行不在 Cache 中，那么当 CPU 访问的指令位于当前行的某个特定位置时，预取请求发出。这个特定的位置称为 Fetchahead Distance。

其优点在于硬件实现代价小，因为下一行的地址很容易计算，即简单的 PC+16（因为一行是 128 位）。缺点在于该方式不适合非顺序执行的程序。因为它总是预取当前行的下一行，如果程序执行的路径是非顺序的，比如条件分支、跳转、子程序调用等等，那么顺序预取几乎不可能减少 Cache 失效。 另外，这种算法的性能还受到 Fetchahead Distance 的制约。如果该距离足够长，即可以保证预取完成，但是预取仍然可能无效。因为如果 CPU 执行了当前行所含的一条分支指令，而且分支成功，那么预取的行可能就用不到；如果该距离很短，那么预取可能无法及时完成。

（3） Target-Line 预取

Target-Line 预取弥补了 Next-Line 预取的不足，它可以正确预取非顺序的 Cache 行。Target-Line 预取需要一张硬件维持的预取目标表，表中含有当前行和相应后继行的地址对。程序执行中进入了一个新 Cache 行，硬件就会根据当前行的地址查询预测表。如果表中含有后继行的地址，并且该后继行不在 Cache 中，那么预取请求发出；反之，预取请求不发出。

如果程序执行路径和先前的一致，那么这种策略是比较有效的。但是它的缺点也很明显，首先，它严重依赖于预测表，还要有执行相应预测算法的硬件部件，因此硬件开销比较大。其次，预测表需要有一个建立的过程，也就是说在开始阶段，预测表根本起不到作用。另外，既使预测表已经建立，并且预测也是正确的，但是如果目标行已经在Cache中，那么预取请求还是不会发出。

因此，Target-Line预取适合Cache行需要频繁替换的Cache，即比较适合于相联度不高并且容量较小的Cache。试想如果Cache行的替换不频繁，那么已经建立了的预测表就发挥不出应有的作用。

（4） Hybrid 策略[9]

Next-Line预取和Target-Line预取各有优势，将两种策略结合起来可能是一种很好的策略。因为这样不论程序是顺序执行还是非顺序执行，预取都能发挥作用。

Hsu和Smith提出了将Next-Line预取和Target-Line预取结合的混合策略[10]。目标行和下一行都可能被预取，这样就提供了双层保护来避免Cache失效。Next-Line预取和先前描述的一样。而Target-Line预取和先前的类似，但是如果后继行是下一个顺序行，则不需要将它的地址加入到预测表中。这样就节省了表的空间，可以让表含有更多的非顺序后继行地址。

这种策略的优点在于不受程序执行路径的限制，不管是顺序执行的还是非顺序执行的，它都能发挥作用。但是，它的缺点在于系统必须要有较大的存储器带宽。因为这种策略会频繁的进行预取，会占用总线更多的时间。如果预取是无效的，那么就浪费了带宽，甚至影响了正常的操作，比如Cache失效引起的正常访存。

因此，Hybrid策略适合于存储器带宽大的系统。从理论上讲，在足够带宽的保证下，混合策略的性能应该是Next-Line预取和Target-Line预取的性能总和。

（5） Wrong-Path 预取[9]

这种算法其实也是一种混合策略，将 Next-Line 和 Target-Line 方法结合起来。其思想主要如下：顺序预取采用的是 Next-Line 算法，实现也类似；非顺序部分和 Target-Line 不一样，Wrong-Path 的非顺序预取部分不需要建立预测表。因为它总是预取分支的目标指令所在行，该地址可以在 CPU 的译码阶段直接获得。如果分支不会在近期成功，而是要经过一段时间，那么当分支成功时，目标行已经被预取。所以，这种算法希望程序在遇到分支指令时，先沿分支的失败路径执行，等过了一段时间再沿正确路径执行。如果第一次遇到分支时，分支就成立，那么这种算法的非顺序部分就会失效。这也是这种算法名字的由来。

Wrong-Path 的硬件实现代价和 Next-Line 预取差不多，但是性能优于 Next-Line。因为它既能预取顺序行，又能预取非顺序行。另外，Wrong-Path 非顺序预取的行并不是立即被执行的行，这就意味着有更多的预取时间来保证预取的完成。

这种策略的优点在于硬件实现代价小，同时又具有了混合策略的性能。缺点在于受到程序执行路径的制约。如果分支总是成功，那么预取的路径可能永远不被访问。这不仅浪费了带宽，而且造成了 Cache 污染[9]。

（6） 以基本块为单位的非顺序指令预取[11]

以基本块为单位的非顺序指令预取。其思想为将程序划分成若干个基本块，在每个基本块的第一条语句后面插入预取指令。这样执行当前基本块的同时就可以预取下一个基本块。这种算法需要编译器的支持，要将程序划分成若干基本块，同时要在程序中插入预取指令。

5.1.2 衡量预取算法性能的标准

大部分的 Cache 研究使用 Cache 失效率或者 CPI 作为性能衡量的尺度。但是对于带预取算法的 Cache，失效率可能就不再是一个合理的衡量尺度。比如，对于一个处理器来说，完全的失效延迟假设为 5 个 CPU 时钟周期。如果一个算法提前三个周期发起预取，而另一个提前一个周期发起预取。Cache 的失效率可能是一样的，但是最终的性能却是有差异的。比起第二个算法，第一个可以减少两个周期的延迟。第一个算法的性能应该是优于第二个。

用总 CPU 周期数来衡量算法的性能应该会更合理一点。总 CPU 周期数等于指令正常执行的周期数加上 Cache 失效周期数。有了总 CPU 周期数，算 CPI 或 IPC 也很简单，只要知道指令的执行条数即可。

另外，预取算法的性能还和Cache的重填率有关。如果Cache的重填率很低，比如一个周期只能填一个字。这样一次预取会占用总线很长时间。如果预取刚开始就出现Cache失效，则Cache控制器必须等待预取完成才能获得总线的控制权。如果预取的行是一个无用行，那么就会影响Cache的正常操作。

5.2 MiniSys 指令预取算法的实现

5.2.1 算法的选用

MiniSys 的指令 Cache 采用了 2 路组相联的映射方式。总容量为 32 个字，其中每个 Cache 行含有四个字。为了不修改现有的 Cache 结构，故不采用长 Cache 行的方式。另外，一些需要编译器支持的算法也不适合 MiniSys，比如以基本块为单位的非顺序指令预取[11]、基于控制流的混合指令预取[12]等等。

经过改进，MiniSys 的存储器带宽得到了扩大，Cache 的重填率也因此而上升。有了这些保证，Next-Line 预取、Target-Line 预取以及 Hybrid 策略都适合于 MiniSys。但是考虑到 Target-Line 预取实现的硬件代价大，而且对于 Cache 行替换不频繁的系统（MiniSys 的汇编程序通常很小，Cache 行替换不频繁），其性能很低，因此本设计不采用 Target-Line 算法。本设计实现了三个算法：OnMiss 预取、改进的 OnMiss 预取、WrongPath 预取。

（1） OnMiss 预取

OnMiss 预取是 Next-Line 预取的一种类型，只有在 Cache 失效时，预取控制器才发出预取请求。这样，每次 Cache 失效，处理器会从存储器取两个行。其中，造成 Cache 失效的行被放入 Cache 中，而预取的行被放入预取缓冲中。

（2） 改进的 OnMiss 预取

OnMiss 预取的实现很简单，硬件代价也小。但是这种策略很被动，只有当 Cache 失效时才发出预取请求，效率很低。因此，可以稍微做一些改进——当预取命中时也发出一个预取请求。其实，UltraSparc III 的预取算法类似于改进的 OnMiss 算法。如果程序一直是顺序执行的，改进的 OnMiss 预取算法效果很好。因为 CPU 执行当前行的指令时，预取控制器就会预取当前行的下一行。但是，如果程序是非顺序执行的，预取可能就会失效。

（3） Wrong-Path 预取

为了进一步提升系统的性能，本设计还实现了一种 Hybrid 算法——Wrong-Path 预取算法。顺序行的预取采用了 Next-Line 预取；非顺序行的预取则是利用 ID 阶段的译码信息，

当 ID 阶段判断出指令为 BEQ 或 BNE，则将目标行的地址送往预取控制器。如果目标行不在指令 Cache 和预取缓冲中，则发出预取请求，预取的行放在预取缓冲中。

由于 Wrong-Path 预取既可以预取顺序行，又可以预取非顺序行，同时硬件实现代价和 OnMiss 预取差不多。因此，系统最终选用了 Wrong-Path 预取。

5.2.2 算法的硬件实现

考虑到 OnMiss 预取和改进的 OnMiss 预取的实现比较简单，并且没有被系统选用，所以本文不做介绍。这里主要介绍一下 Wrong-Path 预取的实现。

（1） Wrong-Path 预取的实现

图 5.1 是 Wrong-Path 预取算法的顶层文件。由于 Wrong-Path 是一种混合预取算法，所以会有两种预取地址。一种为当前的 PC 值加 16，另一种为译码阶段保存的分支目标地址。当 PC 值的最低四位为 0000，或者正在被译码的指令为分支指令时，预取控制器检查需要预取的行是否在 Cache 中，其实就是查询 Cache 的标签域。这就要求 Cache 的标签域能够同时被取指和预取单元访问。解决的方法主要有两种：标签域多端口设计和标签域可复制。本设计采用了后者，将 Cache 中的标签域复制了一份，然后输入到预取控制器中。

```
WrongPath wrongpath(
    .CLK(clk),
    .RST(rst),
    .ADDR0(pc),     //预取地址1
    .ADDR1(bpc),    //预取地址2
    .ID_PREQ(branch),       //预取请求1
    .IC_PHIT(ic_phit),      //预取行命中
    .IC_TAG0(ic_tag0_all), //Cache的第1路标签域
    .IC_TAG1(ic_tag1_all), //Cache的第2路标签域
    .NRC_TAG(nrc_tag),      //NRC的标签域
    .PERMIT(m_ppermit),     //总线获得信号
    .D(m_data),             //读内存数据
    .MREAD(pf_read),        //读内存信号
    .NWRITE(pf_write),      //写NRC信号
    .FETCH(pf_fetch),       //预取信号
    .TAG(pf_tag),           //预取行的标签域
    .Q(pf_rdata)            //预取数据
);
```

图 5.1　指令预取算法的顶层文件

预取控制器的内部主要包含标签域的比较逻辑、预取行的数据寄存器、预取行的标签域寄存器以及一个状态机。其中，状态机是预取控制器的核心，如图 5.2 所示。标签域的比较逻辑用来判断预取行是否在 Cache 中，如果已经在 Cache 中，那么就不需要发出预取请求。预取行的数据寄存器由四个 32 位触发器构成，标签域寄存器则由一个 10 位触发器构成。

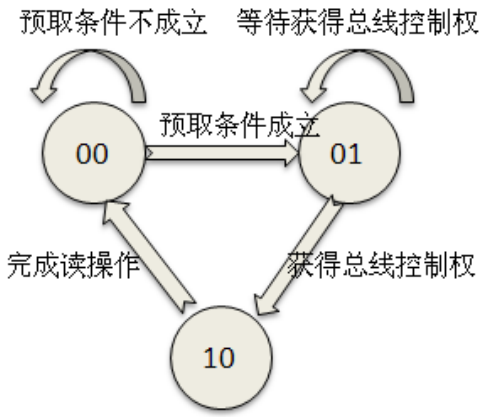


图 5.2　预取控制器的状态机

当两个预取条件同时满足时，顺序预取的请求被发出，而非顺序预取的请求被简单丢弃。采取这种策略主要是考虑到了 Wrong-Path 的特性。它希望程序一开始沿分支的失败路径执行，经过一段时间后再沿分支的目标路径执行。如果程序在实际执行中确实是这样的，那么首先满足顺序的预取请求是合理的，因为非顺序的行并不会马上被引用，而且非顺序的预取请求很有可能会再次出现。

（2） 预取算法的加入

要想在原先的系统中加入预取算法，不管是 OnMiss 算法还是 Wrong-Path 算法，都需要对指令 Cache 和存储器做一些修改。对于指令 Cache，主要修改的是读状态机。图 5.3 为加入预取算法后，指令 Cache 读状态机的变化。

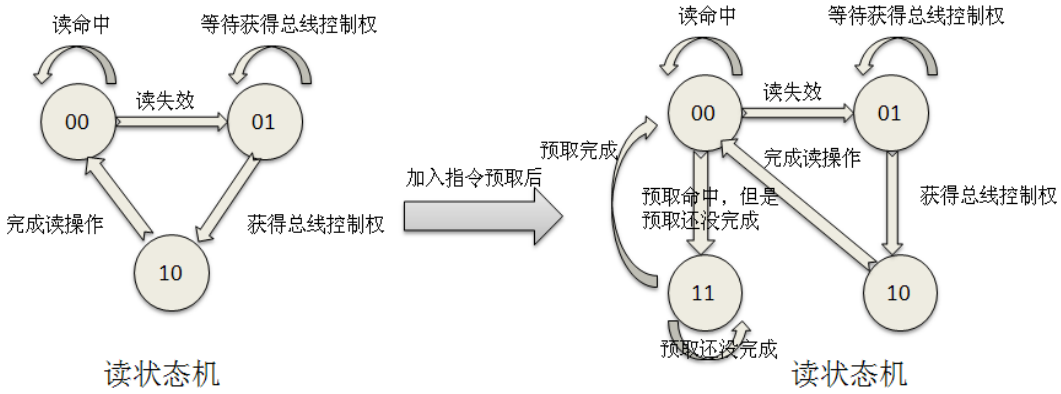


图 5.3 加入预取算法后，指令 Cache 读状态机的变化

对于存储器，主要是修改仲裁机构。加入预取算法后，存储器的访问请求由原来的 3 个变成了 4 个，分别是指令 Cache 的读操作、数据 Cache 的读操作和写操作、指令预取的读操作。当多个访问请求同时到来时，就需要有一个仲裁机构。该机构通过某种仲裁机制，让其中的一个操作获得总线的控制权，而让其他的操作暂时进入等待状态。

为了解决上述问题，规定访存的优先级如下：DCache>ICache>Prefetch。首先满足数据 Cache 的请求，然后是指令 Cache，最后才是指令预取。这样规定是基于如下考虑：第一，数据 Cache 失效对流水线造成的影响比指令 Cache 失效大，尽快地处理数据 Cache 失效有助于提升 CPU 的性能。所以规定数据 Cache 的优先级高于指令 Cache。第二，预取请求并不是必须的，当预取操作影响到正常的 Cache 访存操作时，预取操作应该让位于正常的 Cache 访存操作。这里，正常的 Cache 访存操作是指数据 Cache 的读写操作，指令 Cache 的读操作。所以规定预取的优先级是最低的。

5.2.3 Wrong-Path预取算法的改进

在本设计中，预取缓冲的容量为 16Bytes，相当于一个 Cache 行。我们知道预取的行先是放在预取缓冲中，当 CPU 要求的指令在预取缓冲时，才将预取缓冲中的行放入 Cache 中。这样做主要是为了避免 Cache 污染。

由于预取缓冲的容量很小，所以每次预取都会覆盖缓冲区中原先的预取行，而不管该行有没有被 CPU 使用。如果该行没有被 CPU 使用，但是可能等会就会被使用（按程序局部性原理，这是很容易出现的情况），那么简单地把它丢弃显然不是一个很好的选择。因此有必要对算法做一些改进——增加一个 Non-Referenced Cache[13]。在新的预取行进入预取缓冲前，预取控制器检查预取缓冲中的行有没有被 CPU 使用过，如果使用过了，那么简单将它丢弃即可；如果没有被使用，则把它放入 Non-Referenced Cache，而 Non-Referenced Cache

原来的数据则简单的丢弃。

Non-Referenced Cache(NRC)的实现相对来说比较简单。该模块主要由一个数据寄存器、一个标签域寄存器以及简单的控制逻辑组成。当未被引用的预取行被替换时，NRC 模块执行写操作，将该预取行的数据和地址保存下来。

每次 CPU 取指时，Cache 控制器先将 PC 值和 Cache 的标签域比较。如果命中，就把指令送往 CPU 的指令寄存器，如果失效，则将 PC 值与预取行标签域比较；如果命中，就把预取行送往 CPU 的指令寄存器，同时更新 Cache 的数据域和标签域。如果失效，则将 PC 值与 NRC 的标签域比较；如果命中，则将 NRC 的数据送往 CPU 的指令寄存器，同时更新 Cache 的数据域和标签域。如果失效，则发出 Cache 失效信号和读内存信号。

由于系统选用了 Wrong-Path 预取算法，因此这里的改进是指在 Wrong-Path 预取的基础上添加 Non-Referenced Cache。下面以图 5.4 所示的程序（计算菲波那契数列第 10 项）为例，具体说明 Non-Referenced Cache 的作用。

```
CODE SEG
Start:
    ori  $1,$0,0  ;f(0)=0
    ori  $2,$0,1  ;f(1)=1
    ori  $3,$0,0  ;f(n)
    ori  $4,$0,10 ;n=10
    ori  $5,$0,1
Loop:
    beq  $4,$5,Finish
    add  $3,$2,$1   ;f(n)=f(n-1)+f(n-2)
    add  $1,$2,$0   ;update f(n-2)=f(n-1)
    add  $2,$3,$0   ;update f(n-1)=f(n)
    addi $5,$5,1
    j Loop
Next:
    addi $1,$0,10
    addi $2,$0,10
    addi $3,$0,0
    addi $4,$0,16
    addi $5,$0,1
    ...
Finish:
    sw  $3,0($0)
    j Next
END Start
CODE  ENDS
```

图 5.4　计算菲波那契数列第 10 项的程序

系统启动后，CPU 取第 1 条指令，结果造成 Cache 失效。Cache 控制器发出访存请求，由于当前 PC 值的低 4 位为 0000，预取控制器也会发出访存请求。根据规定的优先级，Cache 控制器先获得总线的控制权，然后读取要求的行。Cache 控制器访存结束后，预取控制器获得总线的控制权，然后读取预取行。这样，当预取结束后，程序的前 4 条指令（ori　$1,$0,0、ori　$2,$0,1、ori　$3,$0,0、ori　$4,$0,10）位于 Cache 中，第 5 条至第 8 条指令（ori　$5,$0,1、beq　$4,$5,Finish、add　$3,$2,$1、add　$1,$2,$0）位于预取缓冲中。

接下来程序顺序执行，当执行到第 5 条指令 ori $5,$0,1 时，由于当前 PC 值的低 4 为 0000，因此预取控制器发出访存请求。此时总线处于空闲状态，预取控制器获得总线的控制权，然后读取预取行。预取结束后，第 9 条至第 12 条指令（add　$2,$3,$0、addi $5,$5,1、j Loop、addi $1,$0,10）位于预取缓冲中。当第 6 条指令 beq　$4,$5, Finish 进入译码阶段时，由于预取控制器正在预取顺序行，因此非顺序的请求被简单丢弃。

当程序执行到第 9 条指令 add　$2,$3,$0 时，与先前类似，预取控制器预取顺序行。预取结束后，第 13 条至第 16 条指令（addi $2,$0,10、addi $3,$0,0、addi $4,$0,16、addi $5,$0,1）位于预取缓冲中。

第一遍循环结束后，程序再次执行第 6 条指令 beq　$4,$5, Finish。当这条指令进入译码阶段后，预取控制器发出非顺序预取请求。此时总线处于空闲状态，预取控制器获得总线的

控制权。由于预取缓冲中的指令没有被 CPU 使用，所以在读取新的预取行之前，先将预取缓冲中的数据放入 Non-Referenced Cache，然后预取控制器读取新的预取行。预取结束后，程序的最后两条指令（sw　$3,0($0)、 j Next）位于预取缓冲中。

循环结束后，程序执行第 12 条指令。然后是第 13 条指令。由于第 13 条指令位于 Non-Referenced Cache 中，所以 Cache 命中，然后把 Non-Referenced Cache 中的数据放入 Cache 中。如果没有 Non-Referenced Cache，那么 Cache 会失效。这就是 Non-Referenced Cache 的作用所在。

加入 Non-Referenced Cache 后，本设计所采用算法的最终思想如下：

1） 顺序预取，当 PC 的低四位为 0000 时，生成预取地址 PC+16。然后，将生成的预取地址和指令 Cache 的标签域复本进行比较。如果要预取的行不在 Cache 中，就发出预取请求。否则，就不用发出预取请求。

2） 非顺序预取，当流水线的 ID 阶段判断出当前指令为 BEQ 或 BNE 时，生成预取地址。接下来的操作与顺序预取一样。

3） 当顺序预取请求和非顺序预取请求同时成立时，首先满足顺序预取请求，而将非顺序的请求简单丢弃。

4） 当预取请求成立时，首先检查预取缓冲中的数据是否被 CPU 访问过。如果没有，则把该数据放入 Non-Referenced Cache 中，Non-Referenced Cache 原来的数据则简单丢弃。否则，说明该数据已经在指令 Cache 中，简单丢弃即可。

5.2.4 算法适用的程序结构

OnMiss 预取和改进的 OnMiss 预取适合于顺序执行的程序，而 Wrong-Path 预取适合的程序结构有些复杂，这里做一个简单的总结。其中，程序结构主要是针对 MiniSys 的汇编程序结构。

1) 程序中含有少量的子程序调用指令和无条件跳转指令。遇到这些指令，Wrong-Path 基本失效，除非跳转的距离很短，就在下一个 Cache 行。

2) 循环的出口条件放在循环的开始而不是末尾。图 5.5 展示了两种循环结构，Wrong-Path 算法更适用于左边的结构。该结构符合了算法的特性，即算法希望程序一开始沿分支的失败路径执行，经过一段时间后再沿分支的目标路径执行。

3) 循环体较小，最好少于 32 条指令。因为指令 Cache 的容量就 32 words，如果循环很大，则很有可能出现 Cache 颠簸。

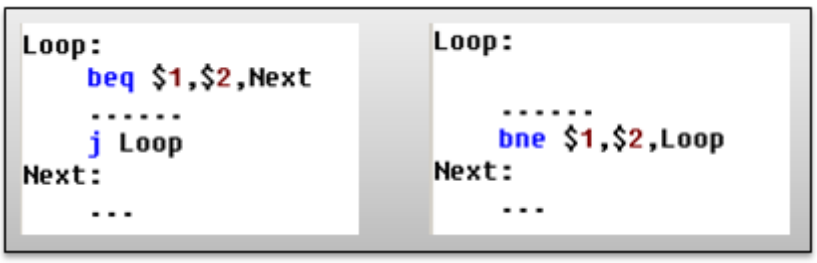


图 5.5 两种循环结构

对于第 1 条，当遇到子程序调用或无条件转移指令时，很多预取算法也是无能为力，除了 Target-Line 预取。但是这种算法的硬件实现代价高，本设计不采用；对于第 2 条，所有的循环都可以转为图 5.5 中左边的结构。相比于右边的结构，左边的结构需要增加一条语句；对于第 3 条，一般的循环基本上可以在 32 条指令内完成。

因此，Wrong-Path 所适用的程序结构可以满足编程的需求。这里的编程主要是指 MiniSys 的汇编语言编程。

5.3 预取算法的性能比较

本设计实现了三个预取算法，OnMiss 预取、改进的 OnMiss 预取、Wrong-Path 预取。其中，Wrong-Path 预取已经加入了 Non-Referenced Cache 结构。这里选用了三个测试程序，Test1、Test2、Test3。其中 Test1 实现了一个冒泡排序算法；Test2 完成三个任务，首先将一批成绩数据由百分制转换为等级制，然后计算第 N 项菲波那契数列，最后计算两个 8 位整数的原码乘法；Test3 是一个重点测试 CPU 的程序，测试了大部分的 MiniSys 指令。

使用总 CPU 周期数和 Cache 失效次数来衡量算法的性能。其中总 CPU 周期数等于指令执行的周期数加上 Cache 失效的周期数。图 5.6 为三种算法运行测试程序的总 CPU 周期数对比，图 5.7 为 Cache 失效次数对比。存储器的带宽为 1word/cycle，即 Cache 的重填率为 1word/cycle。

图 5.6 表明 Cache 中引入预取算法，确实有助于提升 CPU 的性能，这里的性能提升主要是考虑总 CPU 周期数。三种算法中，OnMiss 改进与 Wrong-Path 的性能差不多，都优于 OnMiss。图 5.7 表明用 Cache 的失效次数衡量预取算法不是很合理。Wrong-Path 的失效次数最少，性能应该是最优的。但是根据图 5.6 的周期数统计，它的性能和 OnMiss 改进差不多。

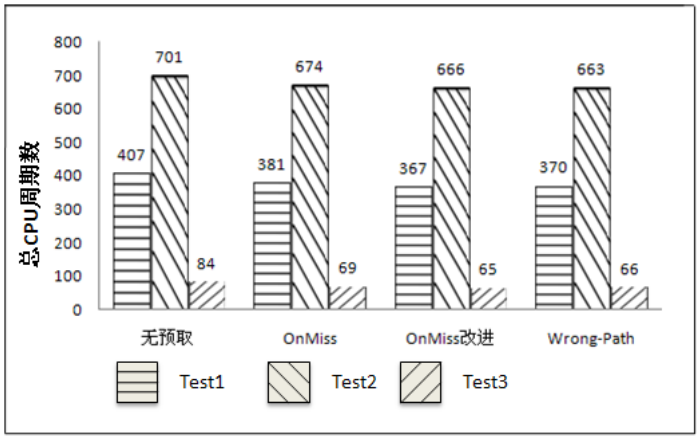


图 5.6　总 CPU 周期数对比，存储器带宽为 1word/cycle

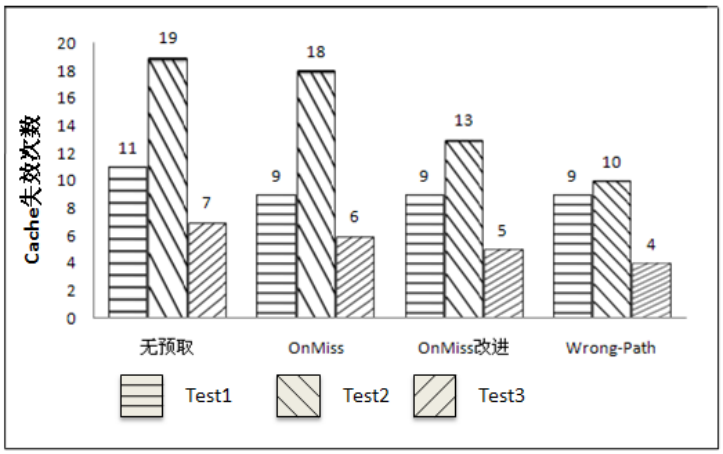


图 5.7　Cache 失效次数对比，存储器带宽为 1word/cycle

图 5.8、图 5.9 的统计数据分别与图 5.6、图 5.7 一样，区别在于图 5.8、图 5.9 的存储器带宽为 4words/cycle，即 Cache 的重填率为 4words/cycle。

当存储器的带宽扩大后，CPU 的性能大大提升。图 5.9 的 Cache 失效次数能很好的反映预取算法的性能，这与先前的结论不一致。原因在于当存储器的带宽较小时，经常会出现这种情况：预取虽然命中，但是预取还没有完成，Cache 仍然失效。当存储器的带宽扩大后，Cache 的重填率增加了，出现这种情况的概率就大大降低。因此，这个时候 Cache 的失效次数能很好的反映预取算法的性能。

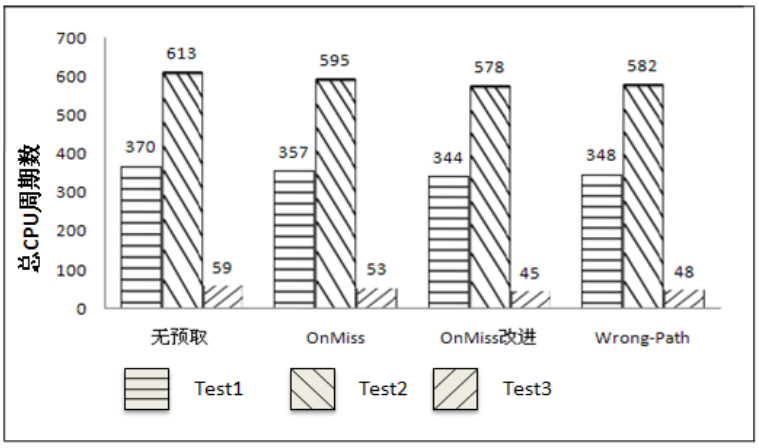


图 5.8　总 CPU 周期数对比，存储器带宽为 4words/cycle

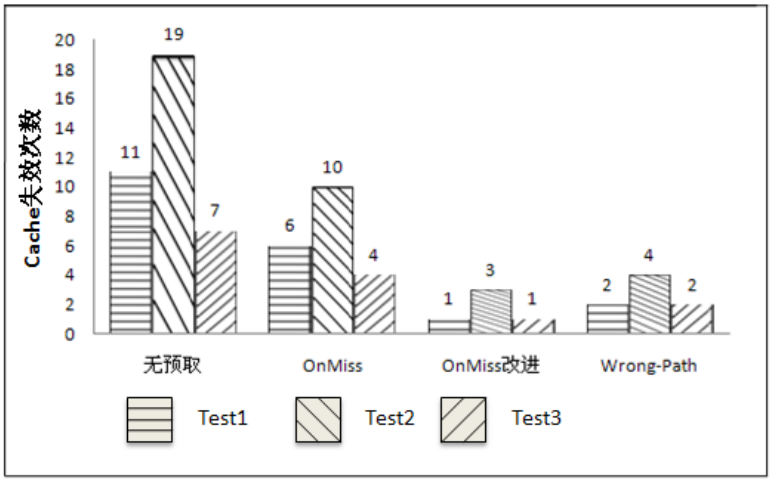


图 5.9　Cache 失效次数对比，存储器带宽为 4words/cycle

图 5.9 的统计数据表明，OnMiss、OnMiss 改进、Wrong-Path 分别可以减少 45%、75%、80%左右的 Cache 失效次数。OnMiss 改进的性能比 Wrong-Path 稍微好一点。按照设想，Wrong-Path 的性能应该优于 OnMiss 改进算法。出现反常情况的原因主要有两个：

- 选取的测试程序中所含的分支指令少，Wrong-Path 的非顺序预取发挥的作用不明显。
- 测试程序中，循环出口就在循环结束的下一个 Cache 行。OnMiss 改进算法也可以预取这些 Cache 行，Wrong-Path 的优势不明显。如图 5.10 所示。

尽管如此，Wrong-Path 预取的性能已经和改进的 OnMiss 预取差不多。这里，Wrong-Path 预取的性能主要是依靠顺序预取获得，而非顺序预取发挥的作用不明显。如果 Wrong-Path 的非顺序预取发挥的作用更大，那么它的性能应该会比改进的 OnMiss 预取好。下面以一个程序来说明这一点。

```
CODE SEG
        Start:
            ……
        Loop:
            beq $1,$2,Next  ;判断条件
            ……
            J Loop
        Next :                  ;循环出口
            ……
        end Start
CODE ENDS
```

图 5.10　测试程序中常见的情况

图 5.11 的左边为一个统计考试不及格人数的汇编程序，右边为等价的 C 程序。依次采用改进的 OnMiss 预取、Wrong-Path 预取算法进行测试，并统计了两种算法的总 CPU 周期数和 Cache 失效次数。其中，图 5.12 为两种算法的总 CPU 周期数对比，图 5.13 为两种算法的 Cache 失效次数对比。

```
DATA SEG
ORG_DATA 0000h
    SCORE DW 90,81,70,98,61
          DW 87,63,64,100,89
ORG_DATA 0040h
    RESULT DW 0
    NOPASS DW 0
DATA ENDS

CODE SEG
    Start:
        ori  $1,$0,1
        ori  $2,$0,10
        ori  $3,$0,0
        ori  $5,$0,60
Loop:
        beq  $1,$2,Next
        lw   $3,SCORE($4)
        addi $4,$4,4
        addi $1,$1,1
        slt  $6,$3,$5
        beq  $6,$0,Loop
        addi $7,$7,1
        sw   $3,NOPASS($8)
        nop
        nop
        addi $8,$8,4
        j Loop
Next:
        sw   $7,RESULT($0)
    END Start
CODE ENDS
```

等价的C程序

```
#include <stdio.h>
#define SIZE   10
#define SIXTY  60
void main()
{
    int A[SIZE]={90,81,70,98,61,87,61,64,100,89};
    int B[SIZE]={0};
    int i=0,j=0;
    int num=0;
    while(i>=SIZE)
    {
        if(A[i]>=SIXTY) continue;
        else
        {
            B[j++]=A[i]; //低于60的分数存入数组B中
            num++;       //不及格人数统计
        }
    }
    return;
}
```

图 5.11　统计考试不及格人数的程序

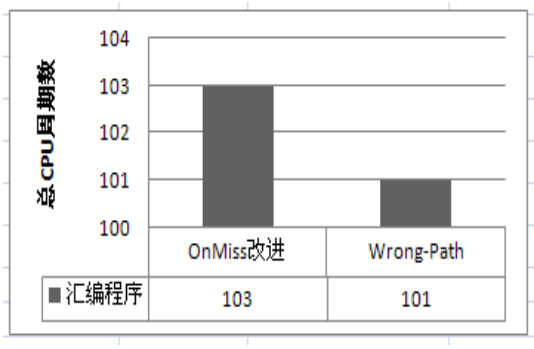


图 5.12　总 CPU 周期数对比

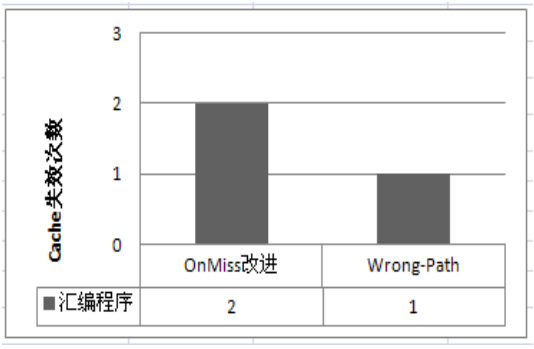


图 5.13　Cache 失效次数对比

测试结果表明 Wrong-Path 的性能要比 OnMiss 改进好一点。这是因为所有分数都大于 60 分，对于汇编程序，beq $6,$0,Loop 与 sw $7,RESULT($0)之间的指令不会被执行；对于 C 程序，if 语句的 else 部分不会被执行。循环结束后直接执行 sw $7,RESULT($0)（C 程序中为 return 语句），Wrong-Path 的非顺序部分预取了该条指令所在的行，结果 Cache 命中；而 OnMiss 改进无法预取到这个行，结果 Cache 失效。因此，Wrong-Path 比 OnMiss 改进少了一次失效。如果有不及格的情况，Wrong-Path 和 OnMiss 改进的性能一样，总的 CPU 周期数都为 107，失效次数为 1。这种情况下，Wrong-Path 的性能全部来自顺序预取。

综合考虑，本设计最终采用了 Wrong-Path 预取算法。相比于 OnMiss 预取以及改进的 OnMiss 预取，它适用的程序范围更广。因为它不受程序执行路径的限制，不管是顺序的还是非顺序的，都能发挥作用，而且实现的硬件代价与改进的 OnMiss 预取差不多。

5.4 结论

通过对多种算法的研究讨论以及对实验结果的分析，得出以下几点结论：

- 各种预取算法都可以有效的提升系统的性能。
- 如果存储器的带宽有限，用总 CPU 周期数来衡量算法性能是一个不错的选择，而不是失效率或是失效次数。
- 预取算法的性能在一定程度上受到存储器带宽的制约。一般情况下，存储器带宽越大越利于预取。
- 各种预取算法都有一个适用程序集。比如 Next-Line 适用于顺序执行的程序，Target-Line 预取适用于 Cache 频繁替换的程序，Wrong-Path 适用于顺序执行和非顺序执行的程序。

第 6 章 系统综合测试

经过前面 3 章的设计，基本完成了整个系统的设计。如果经过测试和验证，并得到正确的结果，那么设计就算真正完成。本章介绍系统的测试方法和验证程序设计。

6.1 测试方法简介

整个系统使用 Quartus7.2 开发，所以主要是利用 Quartus 软件进行仿真测试。所有的测试程序都是用 MiniSys 的汇编语言编写，然后利用 MiniSys 的集成开发环境将汇编语言转换成机器代码，并以一定的格式输出。这里需要对汇编器的输出做一些修改，最后的输出文件为 4 个 mif 文件。如图 6.1 所示。

图 6.1　汇编器输出文件

时序仿真结束后，分析波形以及存储器的数据变化，以此判断运行结果的正确性。时序仿真测试通过后，进行下载验证。下载验证使用了 MiniSys 实验板和博创的 UP-SOPC 实验板。图 6.2 为 MiniSys 实验板，图 6.3 为博创 UP-SOPC 实验板。

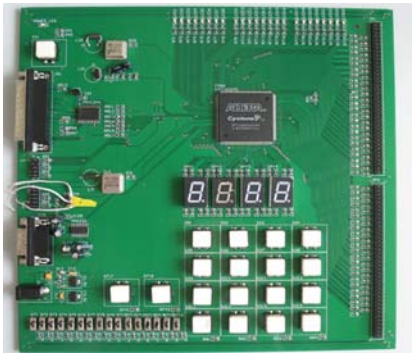


图 6.2　MiniSys 实验板

图 6.3　博创 UP-SOPC 实验板

MiniSys 实验板以一片大容量 FPGA 芯片（Cyclone EP1C6Q240C8）为核心，配合 JTAG、AS 以及并口三种下载方式，给用户提供了充分的设计平台，可支持外围接口到 CPU 的多层次、大规模的设计。同时，提供了标准串口以便于实验板与计算机之间的通信。另外，系统提供了最基本输入输出，方便实验调试和演示。

博创的 UP-SOPC 实验板包括两部分：主板和核心板。其中主板资源相当丰富，包括 8 寸 640*480 真彩 LCD、8 位扫描 LED 数码管、16 位按键输入模块等等。核心板以一片 FPGA 芯片（Cyclone2 EP2C35F672C8）为核心，配以多种接口设备，给用户提供了充分的设计平台。本设计主要使用了 8 位扫描 LED 数码管和 16 位按键输入模块。

6.2 测试程序说明

测试程序一共有 7 个，分别命名为 Test1、Test2、Test3、Test4、Test5、Test6 和 Test7。本章使用了 Test1、Test2、Test3、Test4 和 Test7，而 Test5 和 Test6 主要用来测试双核处理器。有关双核处理器的测试将在双核处理器的设计与实现一章中介绍。

Test1 实现了一个冒泡排序算法；Test2 完成三个任务，首先将一批成绩数据由百分制转换为等级制，然后计算第 N 项菲波那契数列，最后计算两个整数的原码乘法；Test3 是一个重点测试 CPU 的程序，测试了大部分的 MiniSys 指令。该程序主要是用来波形分析；Test4 是一个跑马灯程序，通过 24 个彩灯显示四种效果。同时利用时钟中断和键盘中断，进行计数和按键响应，并通过数码管显示。Test4 主要是用来下载验证（Cyclone EP1C6Q240C8）；Test5 是一个生产者和消费者程序，用来验证双核硬件同步的正确性；Test6 用来验证双核中 Cache 一致性；Test7 是一个数码钟程序，可以用来下载验证（Cyclone2 EP2C35F672C8）。

6.3 测试结果分析

（1） Test 1 测试程序

数组 A 含有 6 个数 3、2、1、4、5、6，程序对数组 A 进行冒泡排序，排序的结果输出到数组 B 中。程序的数据段定义如图 6.4 所示。仿真结束后，查看主存中的数据段，结果如图 6.5 所示，数据采用 16 进制表示。数组 B 的内容用红框标注，从每一个 Bank 的 89 位置开始，依次为 1、2、3、4、5、6，证明数组 A 已经排好序了。

```
;BubbleSort Small to Big
DATA SEG
ORG_DATA 0000h
    SIZE DW 6
    A DW 3,2,1,4,5,6
ORG_DATA 0090h
    B DW 0
DATA ENDS
```

图 6.4　Test1 的数据段定义

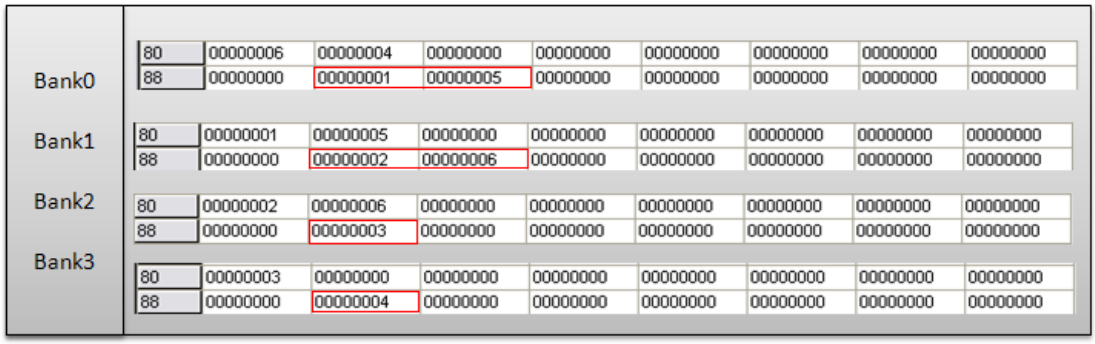


图 6.5　Test1 的仿真结果

（2） Test2 测试程序

程序先将 20 个分数由百分制转化为等级制，然后计算菲波那契数列的第 10 项，最后计算 10×11。数据段定义如图 6.6 所示。仿真结束后，查看主存中的数据段，结果如图 6.7 所

示。所有的运算结果用红框标注。20 个分数转化为等级制的结果为 5、3、3、3、6。菲波那契数列计算结果位于 Bank3 的 90，为 59H（89）。两个数相乘的结果位于 Bank2 的 92，为 6EH（110）。

```
DATA SEG
ORG_DATA 0000h
    SIZE  DW 20
    SCORE DW 90,81,70,98,60,59,87,67,98,100
          DW 33,76,88,60,59,93,44,76,30,49
ORG_DATA 0090h
    GRADE DW 0

ORG_DATA 00A4h
    Fibonacci DW 10
ORG_DATA 00ACh
    Fibonacci_Result DW 0

ORG_DATA 00B0h
    Multi_A DW 10
    Multi_B DW 11
    Multi_Result DW 0
DATA ENDS
```

图 6.6　Test2 的数据段定义

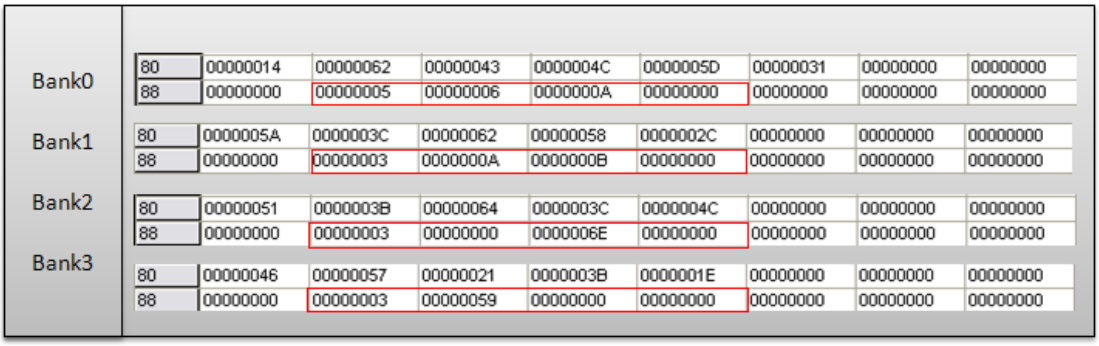


图 6.7　Test2 的仿真结果

（3）　Test3 测试程序

该程序测试了 MiniSys 指令集大部分指令，基本上验证了 CPU 的正确性。图 6.8-6.9 为测试程序的汇编代码。图 6.10-6.16 为时序仿真的波形图。下面对波形图进行具体的分析。叙述中的其中第 n 条指令的 n 表示指令进入流水线的顺序。

```
DATA SEG
ORG_DATA 0000H
        Dat dw 00001234H,98760000H
DATA ENDS

CODE SEG
ORG_CODE 0000H
    Start:
        addi $1,$0,4          ; $1 = 4
        lw $2,0000H($0)       ; $2 = [0] = 00001234H
        lw $3,0000H($1)       ; $3 = [1] = 98760000H
        add $3,$3,$2          ; $3 = $3 + $2 = 98761234H
        lui $2,09876H         ; $2 = 98760000H
        sub $3,$3,$2          ; $3 = $3 - $2 = 00001234H
        ori $2,$0,01231H      ; $2 = 00001231H
        xor $4,$2,$3          ; $4 = $2 ^ $3 = 00000005H
        sub $3,$3,$2          ; $3 = $3 - $2 = 00000003H
        slt $2,$1,$3          ; $2 = ($1<$3) = 00000000H
        bne $2,$0,Error       ; if $2!=$0, goto Error

        jal SubProc           ; goto SubProc
        beq $5,$4,Next        ; if $5==$4, goto Next
        j start
```

图 6.8　Test3 测试程序的上半部分

```
SubProc:
        and $5,$3,$4            ; $5 = $3 & $4 = 00000001H
        slti $4,$4,6            ; $4 = ($4 <6) = 00000001H
        jr $31                  ; SubProc Return

Error:
        j Start                 ; Reset

Next:
        sll $5,$5,2             ; $5 = $5<<2   = 00000004H
        srlv $5,$5,$4           ; $5 = $5>>$4 = 00000002H
        addi $1,$1,4            ; $1 = $1 +4   = 8
        sw $5,0000H($1)         ; [2] = $5 = 00000002H
        lui $4,08000H           ; $4 = 80000000H
        lw $2,0000H($1)         ; $2 = [2] = 00000002H
        srav $4,$4,$2           ; $4 = $4>>2   = E0000000H
        srl  $4,$4,20           ; $4 = $4>>20 = 00000E00H
        sw $4,0ff00h($0)
        j Start

    END Start
CODE ENDS
```

图 6.9　Test3 测试程序的下半部分

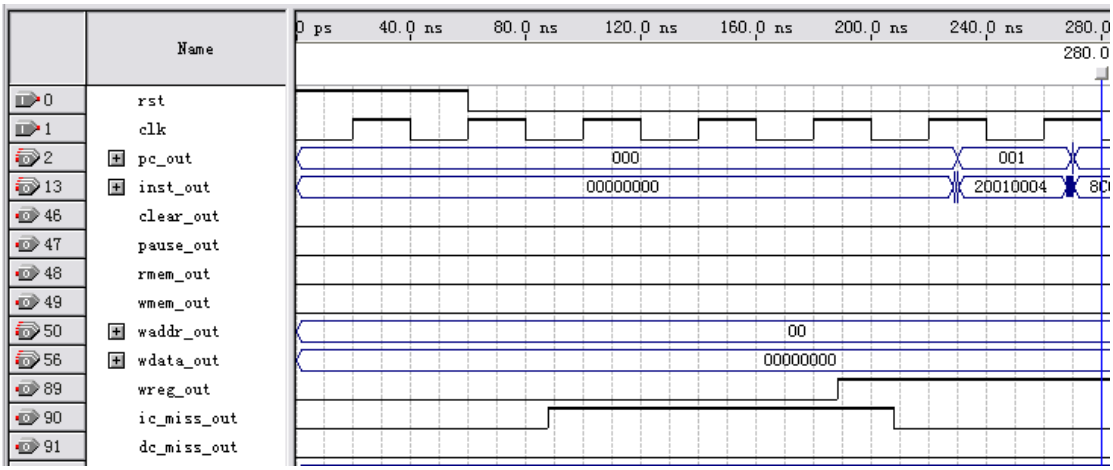


图 6.10　0～280ns 的执行情况

图 6.10 表示从 0 到 280ns 的执行情况。系统启动后，利用 rst 信号进行系统的初始化工作。第 2 个时钟的下降沿，CPU 取第一条指令，结果指令 Cache 失效。图 6.9 中的 ic_miss_out 变为 1，并且维持了三个周期，即指令 Cache 失效需要停顿流水线 3 个周期。Cache 命中后，开始执行程序的第一条指令 20010004（addi $1,$0,4）。一般的指令需要经过五个时钟周期才能执行完成。

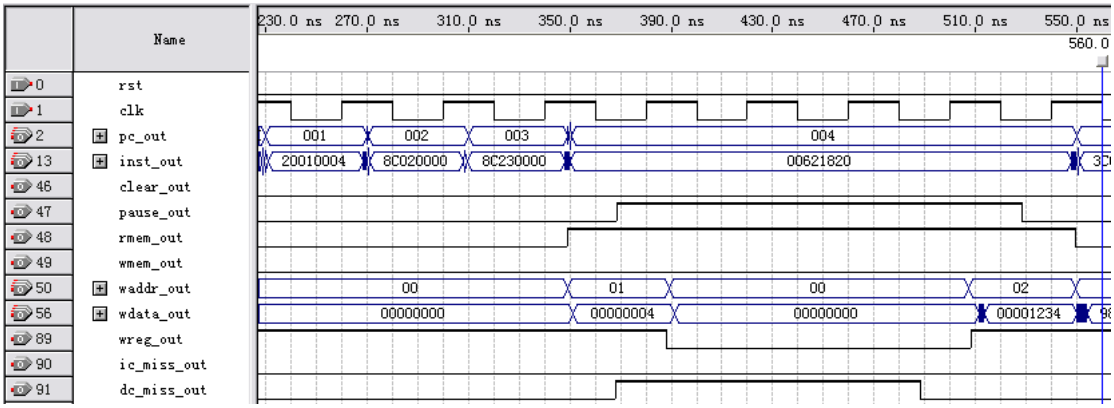


图 6.11　260～560ns 的执行情况

图 6.11 表示从 260 到 560ns 的执行情况。一个 Cache 行可以放 4 条指令，所以如果程序是顺序执行的，CPU 至少在近 4 个周期内不会遇到 Cache 失效的情况。从 260ns 开始，第 2、3、4 条指令依次进入流水线。340ns 的时候，第一条指令进入最后的 WB 阶段。写 1 号寄存器，写入数据为 4。此时，第 2 条指令 lw $2,0000H($0)正位于 MEM 阶段，这是一条读内存数据的指令，造成数据 Cache 失效，流水线停顿三个周期。读内存结束后，由于 ID 阶段指令与之前的存在数据相关，而且是需要阻塞的情况，pause 信号一直维持，直到相关性消除。因此，从 340ns 到 540ns，PC 的值始终没有变化。500ns 时，第 2 条指令进入 WB 阶段，写 2 号寄存器，写入数据为 00001234H。

图 6.12 表示从 540ns 到 840ns 的执行情况。Cache 总是命中，主要得益于指令的预取。540ns 时，第 3 条指令进入 WB 阶段，写 3 号寄存器，写入数据为 98760000H。第 4、5、6、7、8 指令依次进入 WB 阶段，写入数据、写入位置与程序一致。

图 6.13 表示从 820ns 到 1.12us 的执行情况。820ns，第 9 条指令 sub $3,$3,$2 进入写回阶段，写 3 号寄存器，写入数据为 3。然后第 10 条指令 slt $2,$1,$3 进入写回阶段，写 2 号寄存器，写入数据为 0。860ns 时，第 12 条指令 jal SubProc 是一条子程序调用指令，第二次出现 Cache 失效，流水线停顿三个周期，并在 1.02us 恢复执行。

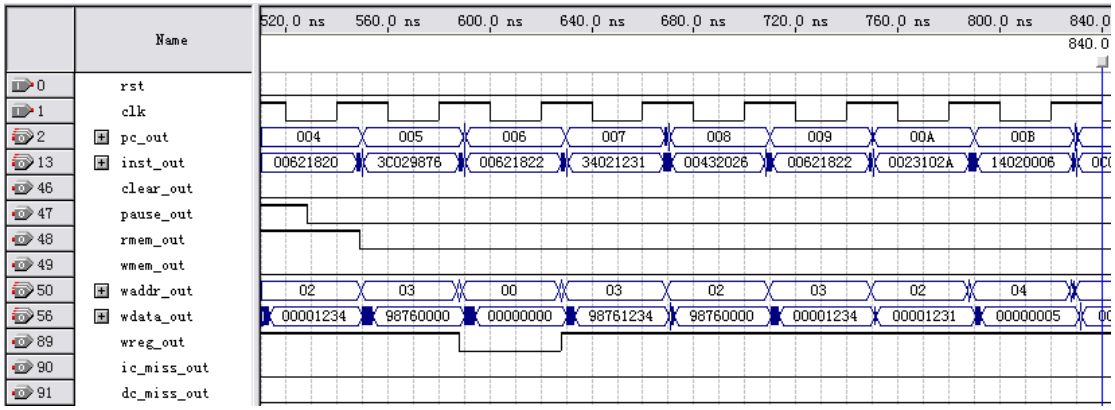


图 6.12　540～840ns 的执行情况

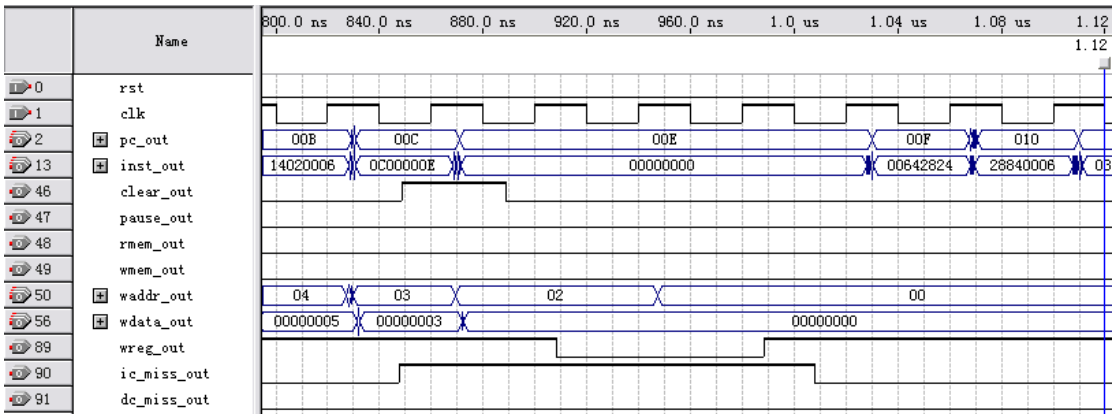


图 6.13　820ns～1.12us 的执行情况

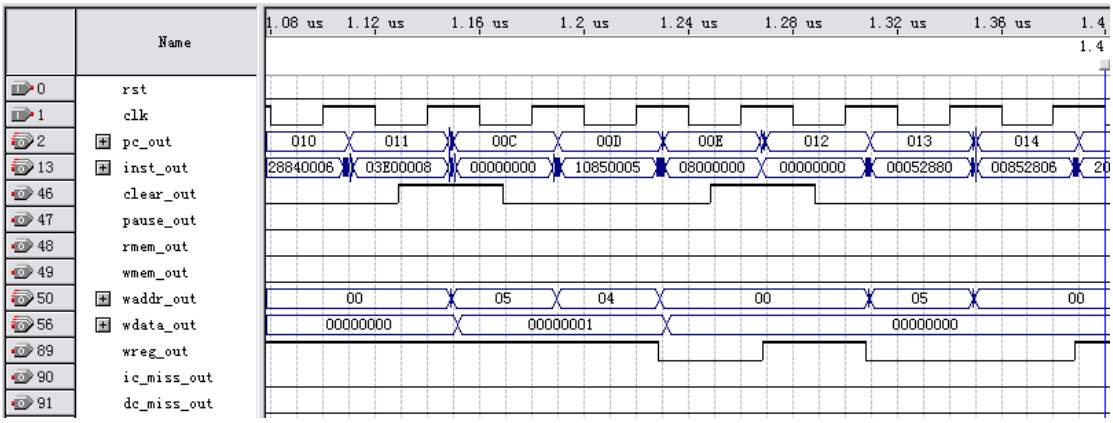


图 6.14　1.10us～1.40us 的执行情况

图 6.14 表示从 1.10us 到 1.40us 的执行情况。从 1.14us 开始，第 13 条指令 and $5,$3,$4、第 14 条指令 slti $4,$4,6 进入 WB 阶段，分别写入 5 号寄存器和 4 号寄存器，写入数据都为 1。第 15 条指令 jr $31 是子程序返回指令。然后执行第 16 条指令 beq $5,$4,Next，分支成功进入 Next 阶段执行。

图 6.15 表示从 1.38us 到 1.68us 的执行情况。从 1.42us 开始，第 17、18、19 条指令依次进入写回阶段。写入数据、位置与程序一致。第 20 条指令为写内存指令，1.50us 进入 MEM 阶段，数据 Cache 写失效，停顿流水线 3 个周期。

图 6.16 表示从 1.66us 到 1.96us 的执行情况。从 1.42us 开始，第 21、22、23、24 条指令进入 WB 阶段。写入数据、位置和程序一致。第 25 条指令 sw $4,0ff00h($0)是一条写接口指令，图 6.16 中没有显示接口的数据，因此看不到。最后执行第 26 条指令，即 j start，又跳回到程序的起始位置，重新开始执行。

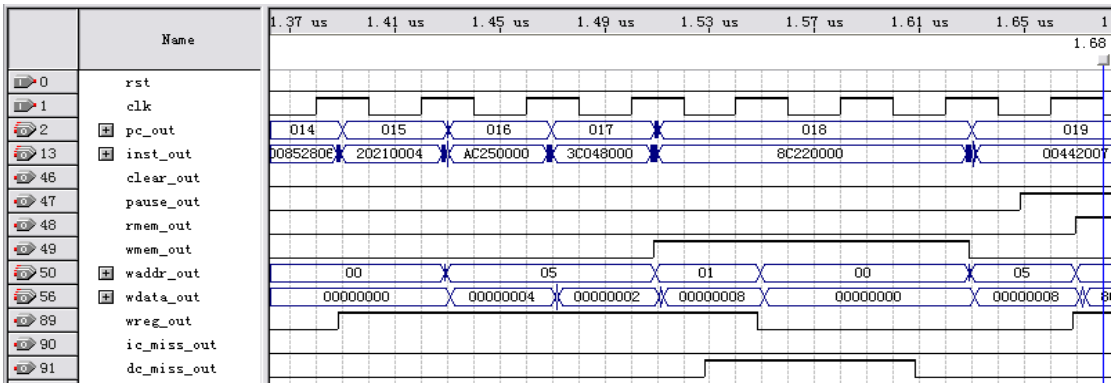


图 6.15　1.38us～1.68us 的执行情况

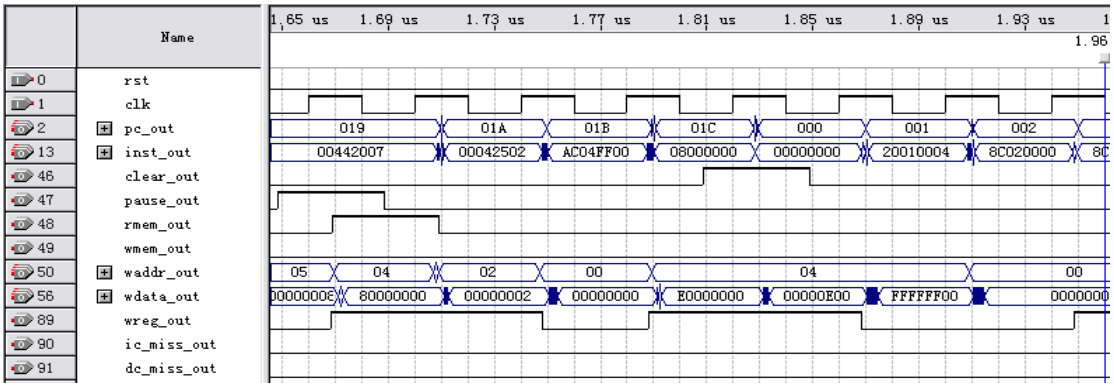


图 6.16　1.66us～1.96us 的执行情况

（4） Test4 测试程序

测试程序 Test4 通过了 MiniSys 实验板的下载验证。图 6.17 为拍摄的照片。24 个彩灯依次显示四种彩灯效果，数码管的低三位用来计数，第四位用来显示键盘的按键值。左起第三幅是按了键盘 8 后的情况，可以观察到数码管的第四位显示为 8。左起第四幅是按了键盘 1 的情况，可以观察到数码管的第四位显示为 1。表明测试程序在系统中运行正常。

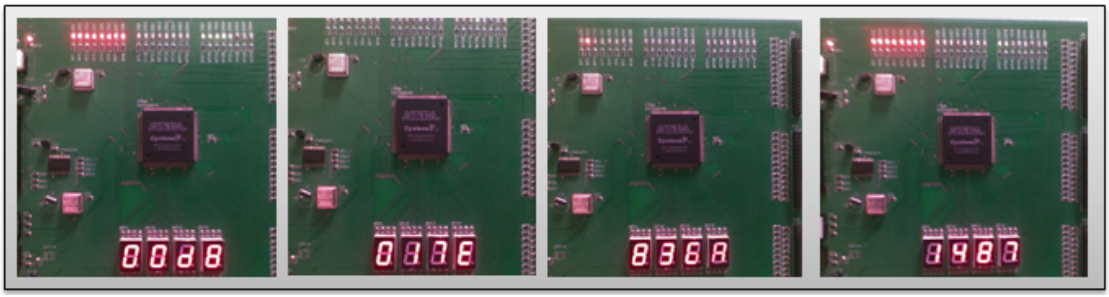


图 6.17　Test4 的下载结果

（5） Test7 测试程序

测试程序 Test7 通过了博创 UP-SOPC 实验板的下载验证。图 6.18 为拍摄的照片。数码管的低六位作为数字钟，第七位用来显示键盘的按键值。

观察数码管低六位可知，三幅图片拍摄的时间依次为 00:06:21、00:06:42、00:07:01。拍摄的顺序为从左往右，即左起第一幅最早拍摄。

图 6.18　Test7 的下载结果

左起第一幅是按了键盘 5 后的情况，可以观察到数码管的第七位显示为 5。左起第二幅是按了键盘 9 的情况，可以观察到数码管的第七位显示为 9。左起第三幅是按了键盘 2 的情况，可以观察到数码管的第七位显示为 2。

6.4 测试总结

经过上述的测试，整个系统运行正常，而且程序的执行结果都是正确的。同时，实现的指令预取算法具有一定的效率，可以有效减少 Cache 的失效次数。其中，Wrong-Path 预取和改进的 OnMiss 预取可以减少 70%以上的失效次数。至此，可以认为整个系统的设计已经完成。

第 7 章 简单双核处理器的设计与实现

1965 年，戈登. 摩尔发现了这样一个规律：半导体厂商能够集成在芯片中的晶体管数量大约每 18～24 个月翻一番。这就是总所周知的摩尔定律。在过去 40 多年中，摩尔定律一直引导计算机设计人员的思维和计算机产业的发展。工艺发展至今，半导体厂商已经可以在单个物理处理器内集成多个核，即多核处理器。多核处理器支持真正意义上的并行执行，因为多个线程或者进程能够在多个核上同时执行[14]。

本章主要是针对双核处理器，首先介绍了双核处理器的几种架构，然后在 MiniSys CPU 的基础上，设计并实现了一个简单的 MiniSys 双核处理器。

7.1 双核处理器的架构

双核处理器是指在一块CPU基板上集成两个处理器核心，并通过一定的方法将各处理器核心连接起来。“双核”的概念最早是由IBM、HP、Sun等支持RISC架构的高端服务器厂商提出的。早在 2001 年，IBM就推出了基于双核的POWER4 处理器；随后Sun和HP都先后推出了基于双核架构的Ultra SPARC以及PA-RISC 芯片。不过由于RISC架构的服务器价格高、应用面窄，没有引起广泛的注意。目前我们熟知的双核处理器主要是Intel和AMD的产品，下面简单介绍一下Intel和AMD的双核架构。

7.1.1 Intel的双核架构

Intel 早期的双核处理器采用了 NetBurst 架构，比如 Pentium D。图 7.1 是 Pentium D 的架构图。它沿用了 Prescott 架构（增强的 NetBurst 架构）及 90nm 生产技术生产。Pentium D 内核实际上由两个独立的 Prescott 核心组成，每个核心拥有独立的 1MB L2 缓存及执行单元，两个核心加起来一共拥有 2MB[15]。两个核心之间的协调工作需要通过前端总线 FSB 由外部的 MCH（北桥）芯片处理，前端总线带宽有限，因此，这里成为了影响双核发挥作用的性能瓶颈。

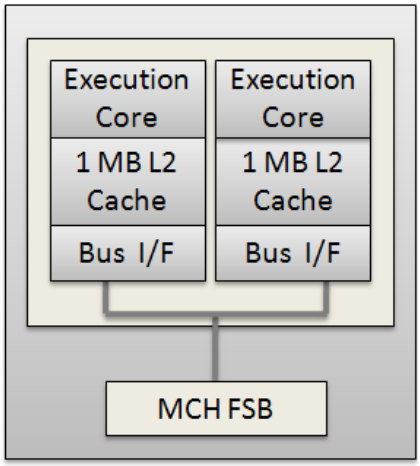


图 7.1 Intel Pentium D 的架构

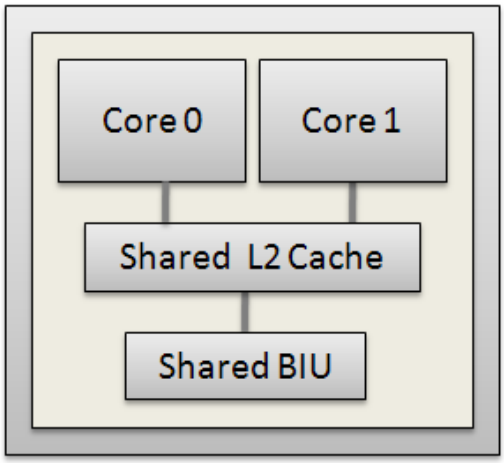


图 7.2 Intel Core 架构

Pentium D 的架构更像是一个双 CPU 平台， 其中两个核心的协调工作影响了处理器整体性能的发挥。不过，Intel 很快就摒弃了 NetBurst 架构，采用了全新的 Core 架构（Yonah 的改进）。图 7.2 是 Core 的架构图。Core 架构拥有双核心、64bit 指令集、4 发射的超标量体系结构和指令融合等技术，使用 65nm 和 45nm 制造工艺生产。每个核心拥有 32KB 的一级指令缓存、32KB 的双端口一级数据缓存，两个核心共同拥有 4MB 或 2MB 的共享式二级缓存。目前 Intel 主要的双核处理器都是基于 Core 架构。

相比于 NetBurst 架构的双核处理器，Core 架构的双核处理器性能大幅提升。其中的一个主要因素就是在 Core 架构中，两个核心采用共享二级缓存，因此两个核心的协调工作及数据交换降低了对前端总线 FSB 的依赖。

Intel 最新的 Core i7 四核处理器在 Core 双核处理器基础上，利用一个多达 12MB 的 L3 Cache 将两个双核结合起来，并将存储器控制器（Memory Controller）集成到 CPU 中，从而进一步加大了内核到存储器之间的带宽。另外，Core i7 在各个核心之间以及 CPU 与北桥之间采用带宽更宽的，点对点通信为基础的 QuickPath 总线（废除了 FSB），提高了核与核之间以及 CPU 与北桥之间的数据交换速度。

7.1.2 AMD的双核架构

AMD 早期的双核处理器主要是基于 K8 架构，比如 Athlon64 X2。图 7.3 是 Athlon64 X2 架构图。它的内部提供了一个称为 System Request Queue(系统请求队列)的技术，在工作的时候每个核心都将其请求放在 SRQ 中。当获得资源之后请求将会被送往相应的执行核心，也就是说所有的处理过程都在 CPU 范围之内完成，并不需要借助外部设备。K8 架构同样将存储器控制器（Memory Controller）集成到 CPU 中，同时，K8 架构采用 CrossBar 开关进行核间通信，速度明显快于 Intel 的 FSB。

之后，AMD 又推出了基于 K10 架构的双核处理器。K10 架构引入了共享三级缓存，同时每个核心拥有自己的一级缓存和二级缓存。如果处理器请求的数据存在于一级缓存中，则直接载入；如果数据在任何一个二级缓存中，则直接或者通过交叉开关载入一级缓存，并将二级缓存中的原数据标记为无效，这也是 AMD 的独特设计；如果数据在三级缓存中，则数据载入后仍然存在，其他核心还能继续访问，从而实现共享。

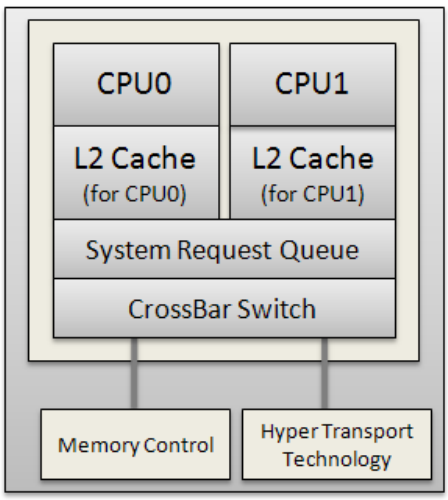


图 7.3　AMD Athlon64 X2 的架构

7.1.3 小结

从上述 Intel 与 AMD 的双核结构中可以看到，提高双核的性能，互连结构非常重要，通常采用的双核间的互连结构主要有：

- 较大的共享 Cache
- 带宽较宽的互连总线或者 CrossBar 开关

除此以外，将存储器控制器集成到 CPU 内部，以期扩大 CPU 到内存的带宽也是各 CPU 厂家目前常用的方法。

7.2 MiniSys双核处理器的设计

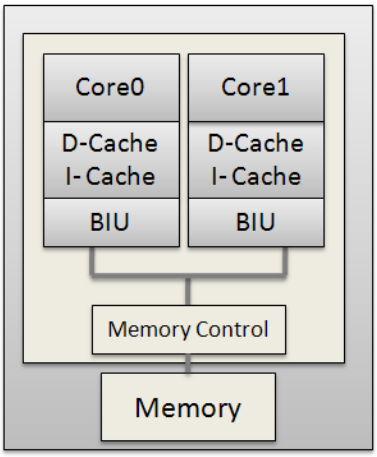
7.2.1 整体架构

上节总结了提高多核性能常用的一些互连结构和方法，但我们在设计 MiniSys 的时候还要综合考虑各种因素。比如，由于芯片资源有限，我们不可能做很大的一个共享 Cache 或者足够大的系统请求队列，并且由于处理器实际主频有限，不便执行较为复杂的队列管理算法，因此不宜采用 SRQ 和 CrossBar 开关的技术。

因此，MiniSys 双核处理器采用了类似于 Pentium D 的架构，其内核实际上由两个独立的 MiniSys CPU 组成，每个核心拥有 32 字的指令 Cache 和 8 字的数据 Cache。两个核心通过各自的总线接口连接到存储器控制器上。存储器容量为 4KB，其中低 2KB 放指令，高 2KB 放数据。与 Pentium D 不同的是，存储器控制器被集成到了 CPU 芯片中。这样，CPU 和存储器间的带宽得到了进一步扩展。图 7.4 为 MiniSys 双核处理器的整体架构。

采用双核结构后，总线变得异常忙碌，因此有必要简化预取算法。原先的单核处理器中采用了 Wrong-Path 算法，可以预取顺序行和非顺序行。如果双核处理器中也采用 Wrong-Path 算法，那么总线可能会过于繁忙，从而影响系统的整体性能。因此，双核处理器采用了简单的 OnMiss 预取算法，当 Cache 失效时才发出预取请求。虽然这种预取算法很简单，但是基本上可以满足双核处理器的性能要求。

由于没有操作系统的支持，所以只能指定线程运行的核心。这里规定线程 0 从地址 0 开始，运行在 Core 0 上；线程 1 从 0400H 开始，运行在 Core 1 上。



7.4 MiniSys 双核处理器架构

7.2.2 BIU的设计

本设计中的 BIU 主要就是一个仲裁机构。每个核心都有四种访存请求，指令 Cache 读操作、数据 Cache 的读和写操作、指令预取的读操作。当多种请求同时出现时，BIU 通过某种机制选择其中的一个请求，并将选中的请求发送到存储器控制器，未被选中的请求暂时处于等待状态。当前的操作结束后，如果有新的操作，BIU 会发送这个请求。BIU 选用的是基于优先级的仲裁机制，优先级规定如下：DCache>ICache>Prefetch。即首先满足数据 Cache 的请求，然后是指令 Cache，最后才是指令预取。理论依据与单核处理器一样。

7.2.3 Cache一致性的保持

本设计中两个核心通过总线共享一个物理存储器。这就带来了一个问题，两个不同的核心所保存的存储器视图是通过各自的 Cache 得到的，如果没有其他的防范措施，则会导致两个核心得到不同的值。图 7.5 说明了这个问题。

时间	事件	Core 0的Cache内容	Core 1的Cache内容	存储器位置X的内容
0				1
1	Core0读X	1		1
2	Core1读X	1	1	1
3	Core0向X写入0	0	1	0

图 7.5 双核处理器的 Cache 一致性问题

为了维护 Cache 的一致性，小规模多处理器系统并不是通过软件而是通过在硬件上引入一个协议来解决这个问题的。这个用于维护多个处理器一致性的协议称为 Cache 一致性协议。目前广泛采用的有两种协议：

- 目录式：把物理存储器的共享状态存放在一个地点，称之为目录。它比监听式的实现开销略微高一点，但是可以用来扩展更多的处理器。Sun 公司的 T1 设计就是采用目录式。
- 监听式：Cache 通过广播媒介（总线或是交换机）访问，所有的 Cache 控制器对总线进行监听或是监视，来确定它们是否含有总线或交换机上请求的数据块的副本。

监听协议的实现很简单，但是不利于扩展。目录式比监听式的实现要复杂一点，但是扩展性也更好。

本设计采用了基于监听式的写无效协议。两个核心都要不断地监听总线来检测地址，然后检查各自的 Cache 中是否有总线上广播的地址。如果有，数据 Cache 中相应行的标签域置为无效。

当向一个共享变量执行写操作时，核心必须获得对总线的访问权才能广播无效性操作。如果两个核心同时对一个共享块进行写操作，其广播无效操作的请求会通过总线的仲裁实现串行化。

由于采用的是写直达 Cache，所以要找出数据最新的值就变得很简单。写入的数据总要送到存储器中，因此可以在存储器中找到某个数据项的最新值。

7.2.4 原子互换指令的实现

写无效协议解决了 Cache 的一致性问题，但是还存在一个问题：当两个核心同时访问共享变量时，就会产生数据竞争。对于单核环境，可以简单的加以解决：访问共享变量时禁止中断。这样就能确保当前的指令顺序的执行而不被中断；对于多核环境，这种解决方案是不可行的。这里主要是依靠特殊的硬件指令来解决同步问题，比如提供一条 TestAndSet 指令[16]。图 7.6 为 TestAndSet 指令的定义。图 7.7 为利用 TestAndSet 指令实现的同步操作。

```
bool TestAndSet(bool &target)
{
    bool temp=target;
    target=true;
    return temp;
}
```

图 7.6　TestAndSet 指令的定义

```
bool lock=false;
......
while(TestAndSet(lock) {}
......        //操作共享变量
lock=false;  //释放锁
......        //不涉及共享变量的访问
```

图 7.7　利用 TestAndSet 指令实现的同步操作

当一个核心要对共享变量进行访问时，它首先执行 TestAndSet 指令。如果没有其他核心正在访问共享变量（lock 为 false），那么它获得对共享变量的访问权(lock 置为 true)，访问结束后就释放锁（lock 置为 false）。

本设计实现了一条类似 TestAndSet 的硬件指令——原子互换指令，以此来解决同步问题。这条指令将一个寄存器的值和存储器的值进行互换。特别要注意的是存储器的值并不放入数据 Cache 中，即不缓存。图 7.8 为该指令的定义和格式。

```
int AtomicExchange(int &reg,int &mem)
{
    int temp=mem;
    mem=reg;
    reg=temp;
    return temp;
}
```

op	rs	rt	immediate
010000	rs	rt	offset

For Example : ae $1,0($0) , ($1)=Memory[0] ,Memory[0]=($1)

图 7.8　原子互换指令的定义和格式

```
int mem=0;
......
int reg=1;
while(AtomicExchange(reg,mem) {}
......        //操作共享变量
reg=0;
AtomicExchange(reg,mem)    //释放锁
......        //不涉及共享变量的访问
```

图 7.9　利用原子互换指令实现的同步操作

有了这条指令就可以构造软件的同步机制，图 7.9 为利用原子互换指令实现的同步操作。基本思想如下：

1） 构造一个简单的锁，初始化为 0，表示锁可以被占用。

2） 如果某个核心要想访问一个共享变量，将一个寄存器的值设为 1。然后执行原子互换指令。

3） 执行完原子互换指令，检查寄存器的值。值为 0 表示获得了锁，为 1 表示未获得锁，需要继续等待。

4） 获得锁的核心结束对共享变量的访问后，将一个寄存器的值设为 0，然后执行原子互换指令，以此释放锁。

当两个核心试图同时进行原子互换操作时，则采用写串行机制解决竞争。通过仲裁机构让其中的一个获得锁，而让另一个进入等待状态。这里规定 Core 0 的优先级大于 Core 1（也可以是 Core 1 的优先级大于 Core 0），当两个核心试图同时进行原子互换操作时，仲裁结构让 Core 0 获得锁，而让 Core 1 进入等待状态。

7.3 验证测试

由于 MiniSys 双核处理器不含有接口和中断，所以只能通过时序仿真来验证系统的正确性。这里选用了三个测试程序：Test2、Test5、Test6。其中 Test2 完成三个任务，首先将一批成绩数据由百分制转换为等级制，然后计算第 N 项菲波那契数列，最后计算两个整数的原码乘法；Test5 是一个生产者和消费者程序，用来验证硬件同步和 Cache 一致性；Test6 用来验证 Cache 一致性。

（1） Test2 测试程序

首先将原程序进行并行化处理，主要是利用任务划分的思想。将成绩转换任务交给 Core 0 处理，将计算第 N 项菲波那契数列和两个整数的原码乘法合并成一个任务，交给 Core 1 处理。最终的运行结果与单核处理器的执行结果一致。但是执行时间大大减少，图 7.10 为单核和双核的总 CPU 周期数对比，双核的执行时间近似于单核的一半。图 7.11 为单核和双核的 Cache 失效次数对比，双核的失效次数为单核的两倍。这是因为双核采用了简单的 OnMiss 预取，只能预取顺序行；而单核采用 Wrong-Path 预取，可以预取顺序行和非顺序行。

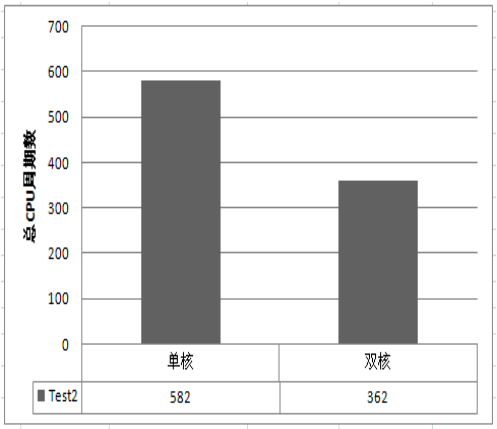


图 7.10 总 CPU 周期数对比

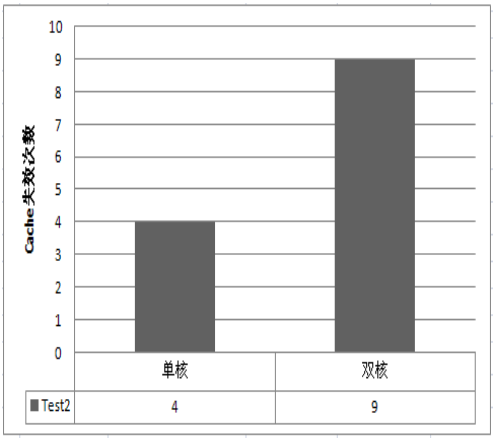


图 7.11 Cache 失效次数对比

（2） Test5 测试程序

Test5 模拟了一个典型的同步程序：生产者和消费者模型。它们共享一个变量，生产者进行增 1 操作，消费者进行减 1 操作。在对共享变量进行操作前，必须先获得锁。当缓冲区

满了时，生产者停止生产，等待消费者消费。当缓冲区空了时，消费者停止消费，等待生产者生产。这里 Core 0 运行生产者线程，而 Core 1 运行消费者线程。具体的程序中，缓冲区的大小为 10，生产者将生产 20 个产品，而消费者将消费 13 个产品。最终的执行结果如图 7.12 所示。Bank0 的 80 位置为 0，表示锁未被占用，这是正确的。因为程序已经执行完成，锁被释放了。Bank1 的 80 位置为 10，表示缓冲区的容量为 10。Bank2 的 80 位置为 7，表示当前剩余的产品，这个结果是正确的。因为生产者生产了 20 个商品，而消费者消费了其中的 13 个，所以还剩下 7 个。

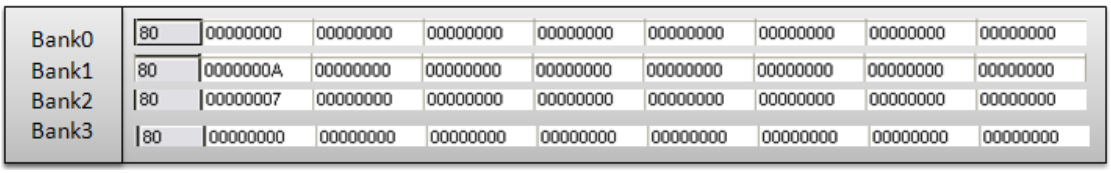


图 7.12　Test5 的运行结果

（3）　Test6 测试程序

程序代码如图 7.13 所示。Core 0 的线程不断的对共享变量进行读写操作，Core1 的线程不断的读共享变量。图 7.14-7.15 为仿真波形图，从 740ns 到 1.24us。820ns 开始，Core 0 和 Core 1 将读取的 Buf 值写入各自的 1 号寄存器，数值都为 2。经过两个周期，Core 0 更新 1 号寄存器的值，数值为 3。然后更新内存中 Buf 值。

```
DATA SEG
    Buf DW 1
DATA ENDS

CODE SEG
    Start:
        lw $1,Buf($0)
        addi $1,$1,1
        sw $1,Buf($0)
        j Start
ORG_CODE 0400H
        lw $1,Buf($0)
        j 0400H
    END Start
CODE ENDS
```

图 7.13　Test6 的汇编代码

当 Core 0 和 Core 1 再次读取 Buf 值时，Core 0 的数据 Cache 命中，Core 1 的数据 Cache 失效。因为之前 Core 0 执行写操作时，会在总线上广播无效地址和信号，Core 1 收到后会将相应 Cache 行的标签域置为无效。最后，两个核心读入的都是最新的数据，数值为 3。证明 Cache 保持了一致性。

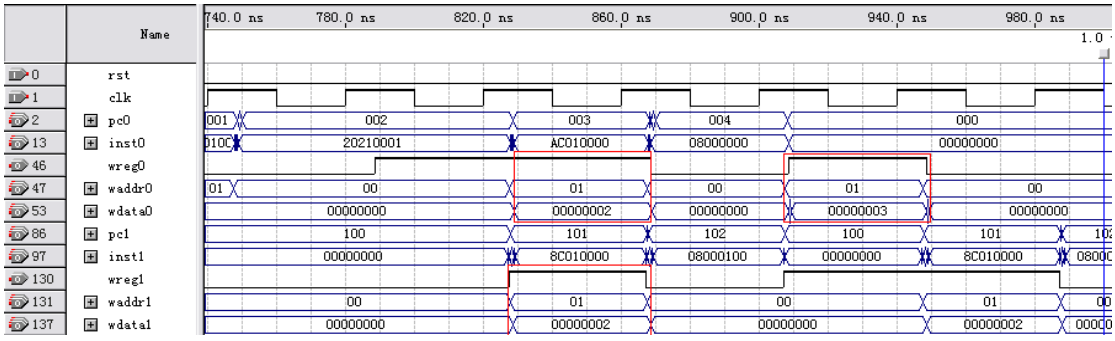


图 7.14　740ns～1.0us 的执行情况

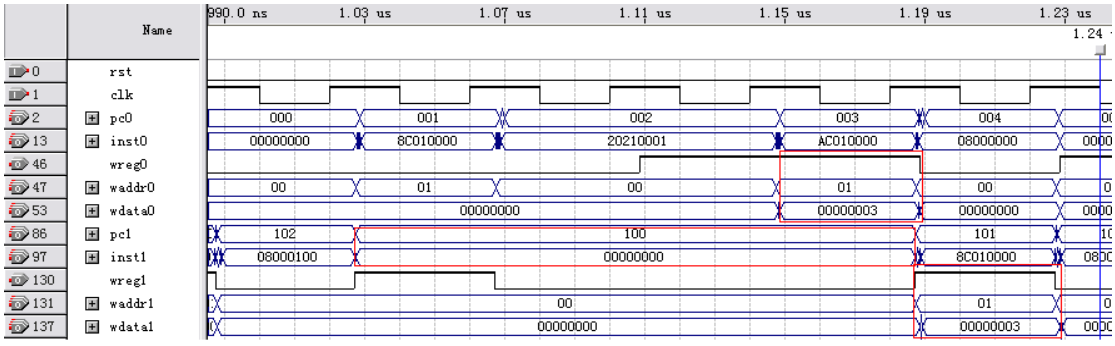


图 7.15　1.0us～1.24us 的执行情况

第 8 章 总结及展望

8.1 设计总结

整个设计历时三个月，不仅完成了任务书上的所有要求，还成功的设计和实现了一个简单的双核处理器。因此，整个毕业设计包括单核版本上的基本流水线的设计、Cache 结构的设计实现、预取算法的研究实现和双核版本上的双核结构设计、一致性的解决以及硬件同步指令的实现。所有设计通过了时序仿真测试，单核版本还完成了下载验证。

本设计的难点之处：

- 所有模块采用 RTL 实现。
- 流水线相关性的解决，包括数据相关、结构相关和控制相关。
- 适合 MiniSys 预取算法的设计与实现。
- 双核处理器的设计与实现。
- 双核处理器中 Cache 一致性的保持。

本设计的创新之处：

- 适合 MiniSys 的交叉存储结构的设计与实现。
- 带 Non-Referenced Cache 的 Wrong-Path 预取算法实现。
- MiniSys 双核处理器的设计与实现。

8.2 展望未来

经过本设计的改进与完善，MiniSys 的结构变的更加合理，同时为今后的进一步扩展打下了良好的基础。接下来，可以增加分支预测或实现乱序执行技术。

当然，本设计还存在一些不足之处：

- Cache 的容量偏小，当初主要是考虑到 Cyclone 的逻辑单元数有限，后来选用 Cyclone 2 后就不存在这个问题。
- 双核处理器暂时不含接口和中断，给测试造成了一些麻烦。
- 整个系统的测试还不够完备，肯定还有隐藏的 Bug。需要进一步的测试，以使系统更加健壮。

致谢

首先，感谢我的指导老师——杨全胜老师。整个设计期间，他给予了我细致地指导和帮助，同时还提供了良好的设计环境和硬件实验平台。他严谨的治学态度和丰富的知识更是给我留下了深刻的印象，是我学习的榜样。

其次，感谢我周围的朋友。每当我遇到问题或困惑时，他们总能给予我关怀和帮助，使我能克服困难，奋勇向前。

最后，特别感谢我的家人。他们是我坚强的后盾，不仅在物质上给予了我最大的帮助，而且在精神上支持着我。

参考文献(References)

[1] 杨全胜.计算机系统综合课程设计[M].北京:清华大学出版社,2008.8

[2] John L.Hennessy,David A.Patterson.计算机系统结构——量化研究方法[M].北京:电子工业出版社,2007.386-412

[3] Dominic Sweetman.See MIPS Run Linux[M].第 2 版.李鹏,鲍峥,石洋译.北京:机械工业出版社,2008.1-5

[4] Milena Petrovic,Igor Tartalja.Evaluating two-bit branch predictors for user/kernel code.[J] IEEE,1997:827-830

[5] David Tarjan, Kevin Skadron.Merging path and gshare indexing in perceptron branch prediction[J]. ACM,2005,2(3):280-300

[6] 夏宇闻.Verilog HDL 数字系统设计教程[M].北京:北京人民邮电出版社,2004

[7] 朱子玉,李亚民.CPU 芯片逻辑设计技术[M].北京:清华大学出版社,2005:52-56

[8] Horel Tim,Lauterbach Gray.UltraSparcIII:Designingthird-generation 64-bit performance[J]. IEEE,1999,19(3):73-85

[9] Jim Pierce, Trevor Mudge. Wrong-Path Instruction Prefetching[J].IEEE,1996.165-175

[10] W.-C. Hsu , J. Smith. Prefetching in supercomputer instruction caches[J].Supercomputing, 1992.588-597

[11] 沈立,戴葵,王志英.以基本块为单位的非顺序指令预取[J].电子学报,2003,25(4):94-98

[12] 沈立,戴葵,王志英,鲁建壮.基于控制流的混合指令预取[J].电子学报,2003,31(8): 1141-1144

[13] G.-H. Park, T.-D. Han, S.-D. Kim, O.-Y. Kwon Non-referenced prefetch (NRP) cache for instruction prefetching[J] .Computers and Digital Techniques,IEE Proceedings,1996,143(1): 37-43

[14] Shameem Akhter,Jason Roberts.多核程序设计技术——通过软件多线程提升性能[M].北京:电子工业出版社,2007.1-19

[15] 刘磊.对片上多核系统的系统结构的研究[J]. Computer Knowledge and Technology,2008, 4(2):485-487

[16] Abraham Silberschatz.Peter Baer Galvin.Greg Gagne.Operating System Concepts[M].第 6 版.郑扣根译.北京:高等教育出版社,2005.148-150