摘要

流水线的引入是处理器历史上的里程碑，他让指令的不同处理阶段在时间上重叠，从而达到了很高的处理速度。而程序中的分支指令发生跳转容易造成之前顺序取出的指令被废弃，从而令流水线停顿，极大影响流水线的性能。为了减少这种情况发生的概率，我们引入分支预测的机制，在取当前指令的同时预测下一条指令的地址，从而减少了流水线的停顿。目前嵌入式领域市场的增长促进了高性能嵌入式处理器的开发，高性能的嵌入式处理器也开始采用流水结构。由于流水线的特点，高性能嵌入式系统也开始应用分支预测技术来提高流水线的效能。同时，由于嵌入式处理器对功耗和面积的敏感特性，嵌入式处理器在应用分支预测技术时也必须考虑面积和功耗的问题。

本论文正是以嵌入式的 MiniSys 体系为基础，在此之上分析和实现 MiniSys 嵌入式流水处理器和分支预测部件。为了寻求适合于本系统的分支预测算法，在参照了 ARM11[15]的分支预测机制的基础上，选择并实现了 gshare bimode 和 bimodal 三种条件分支预测器。同时介绍了与分支预测以及性能测试密切相关的 mini-C 编译器的构造和特点。并通过实现动态和静态双预测器提高系统性能。最后，本文对三种预测器版本处理器进行了详细的性能测试和理论分析以选出适合本系统的分支预测器。

关键词:流水线处理器，SOC，分支预测器，编译器

删除的内容: [1]

删除的内容: [16]

删除的内容: [7]

删除的内容:

删除的内容: minisys

删除的内容: 流水线，mini-C

Abstract

Pipelines structure is a landmark of CPU developing history.It can overlap the instruction's different stages to achieve high speed.Sometimes,the pipeline may be stall for several clock cycles due to some branch instructions was token.It can cause Performance degradation.To reduce the chance of pipeline stall,we can implement branch prediction technique.This technique can predict the next instruction's address while feching current insreuction.If prediction is currect,the cost of branch is zero.The market of embedded system is growing so fast that it promotes the developing of high-permance embedded cpu.Now ,it also applies pipeline structure to high-performance embedded cpu.But chip area and power must be considered due to embedded system's power and chip area sensitivity.

The anasis and inplement of pipeline structure and branch predictor is based on MiniSys SOC architecture.To find a compatible and high accuracy branch prediction arithmetic,I refered to ARM11's branch predictior's structure.Finally,I choose and implement three branch prediction arithmetics,they are gshare,bimode and bimodal.The Closely related mini-C compiler is also introduced.The performance is also improved by implemented dual-precicor(static predictor and dynamic predictor).At the end of this paper,there are particular performance analysis and test to find the best predictor for MiniSys SOC.

Key Words: Pipeline, SOC, Branch predictor, Compiler

删除的内容: Minisys

删除的内容: Mini-C

目录

删除的内容: --------分页符--------

第 1 章 绪论

1.1 引言

在嵌入式处理器的发展史上，处理器经历了单周期处理器，多周期处理器，流水线处理器的发展。在单周期处理器时代，处理器所有指令均在一个周期内完成，处理器的速度取决于最复杂指令的执行时间，因此这个时期的嵌入式处理器主频都不高，速度很慢。进一步的分析发现，不同指令所需要的处理阶段的数量存在差异，于是发展出了多周期的处理器以取得更高的性能。嵌入式应用领域需求的增长促进了更高性能的嵌入式处理器的发展，于是在嵌入式处理器领域引进了流水线机制，进一步提高嵌入式处理器的性能。但流水线中存在的分支指令有时会造成取指方向背离顺序的方向而转向其他地址，就会造成之前取出的指令被废弃，流水线发生停顿，造成性能的损失。加上现在的高级程序中含有大量的分支指令，使得这种性能损失变得尤为明显。目前的解决方法是引入分支预测的机制，在取当前指令的同时预测下一条指令的入口地址，如果预测成功，那么流水线不会因为控制相关性而发生停顿，可以极大改善流水线的性能。

由上可知，分支预测的准确性对于流水线性能，因此伴随着流水线技术的发展，分支预测的算法也在不断地完善。分支预测起源于高性能计算架构（HPCA）领域，从最初的静态分支预测发展到基于分支指令本身的历史信息的单级动态分支预测。顺着研究的深入，发现了分支行为与之前执行的分支行为有关，于是又产生了 2 级自适应分支预测算法和关联分支预测算法。更进一步的研究指出，分支预测是本质是机器学习的过程，于是提出了目前最热门的人工神经元预测技术。由于神经网络预测器的复杂性，目前 HPCA 领域的预测器还是采用的结构相对简单的两级预测器或由简单预测器组成的混合预测器，比如 Alpha21264 采用的就是 gshare 与 bimodal 组成的混合预测器，而 AMD 的 K8 采用的是典型的两级预测器。所以，目前实际的处理器还是采用的结构较简单和成熟的预测器技术。

本课题面对的是一个嵌入式 RISC 型流水处理器的分支预测部件的设计，因此我们更关注嵌入式系统中分支预测部件设计应该注意的问题。

1.2 嵌入式处理器与分支预测

嵌入式处理器对于硬件成本和功耗都很敏感，这决定了 HPCA 领域的复杂预测器是不适合嵌入式处理器的。目前，就 HPCA 领域的商用处理器而言，都是采用的结构较简单的分支预测机制，至于像基于神经网络的预测器由于其巨大的延迟和复杂度，使得其目前只停留于学术研究领域。而像 ARM11 这款典型的嵌入式处理器采用的是 64 记录的 Bimodal 预测器，这是一种非常经典的单级预测器。除此以外，其他复杂度很低的预测器还有 gshare 和基本 bimode 关联预测器。本论文将详细阐述这三种预测器在 MiniSys 系统上的实现细节和性能测试。

1.3 系统概述和论文工作

本论文所使用的系统沿用的是我院“计算机系统综合课程设计”所采用的指令系统和存储结构以及 IO 结构。本系统的 IO 外设采用的是先前课程设计的部件。汇编器和编译器直接采用综合课程设计时的编译器和汇编器，其中编译器是由本人独立设计和完成的。由于目标系统是一个嵌入式的 SoC 系统，因此论文的重点是设计适合这种嵌入式领域使用的分支预测器。

论文首先介绍了 MiniSys SOC 架构，以及五段流水的基本结构和数据相关的解决方案。随后介绍了与分支预测算法中的返回地址预测以及性能测试密切相关的 mini-C 编译器的相关细节和实现方法。作为论文工作的重点，本文详细论述和分析了 MiniSys 中分支预测的可行性和几种常用的分支预测算法。对这几种分支预测算法的实现细节进行了详细的阐述，并对各种算法的性能进行了测试与分析。最后，论文通过实际的程序测试验证了分支预测算法的有效性。

1.4 论文内容的安排

本论文分共分 6 章，

- 第 1 章 分析了分支预测部件对于流水处理器的重要性，提出了嵌入式系统中分支预测部件应该注意的问题和目前的发展概况，最后给出了本论文的总体结构。
- 第 2 章从整个 MiniSys SoC 系统的角度，介绍了该系统的整体架构以及一些相关细节。
- 第 3 章介绍 MiniSys 的 mini-C 编译器的相关细节。
- 第 4 章介绍 MiniSys 分支预测器的设计和实现，性能测试，选型和进一步改进。
- 第 5 章介绍了最终版本的参数配置和下载测试。
- 第 6 章对本次毕业设计工作的总结。

第 2 章MiniSys SoC架构

2.1MiniSys整体框架

MiniSys[12]硬件部分主要分为两个部分一个 32 位的嵌入式 CPU 以及由 IO 总线连接的外设。软件部分由汇编器和编译器构成。以下是 Mini SOC 的结构配置[12]:

1.寄存器文件

寄存器文件由 32 个 32 位寄存器构成，其中 5 个定义为固定功能外，其他寄存器都可作通用寄存器使用。

2.指令系统

使用经过精简的 31 条 MIPS[3]指令，其中只有某些指令作了微小的调整，其他指令均符合 MIPS 体系。

3.存储系统

采用哈弗结构，其中 ROM(4KB),RAM(4KB),均使用 Cyclone II 的片上 M4K 存储资源。它们都采用字节编址，但以 32 位（4 字节）为一个存储单元，即他们和 CPU 之间的数据交换都以 32 位为单位进行。

4.存储系统与 IO 编址

MiniSys SoC 的 I/O 空间编址采用与存储器统一编址方式，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，因此，对 I/O 部件的访问采用与存储器访问相同的指令格式。

5.中断系统

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。

2.2MiniSysCPU结构

2.2.1MiniSys的流水线

图 2.1 是经过改进后的 MinSys 流水线结构简图。

删除的内容:

删除的内容: 改进版

带格式的: 正文

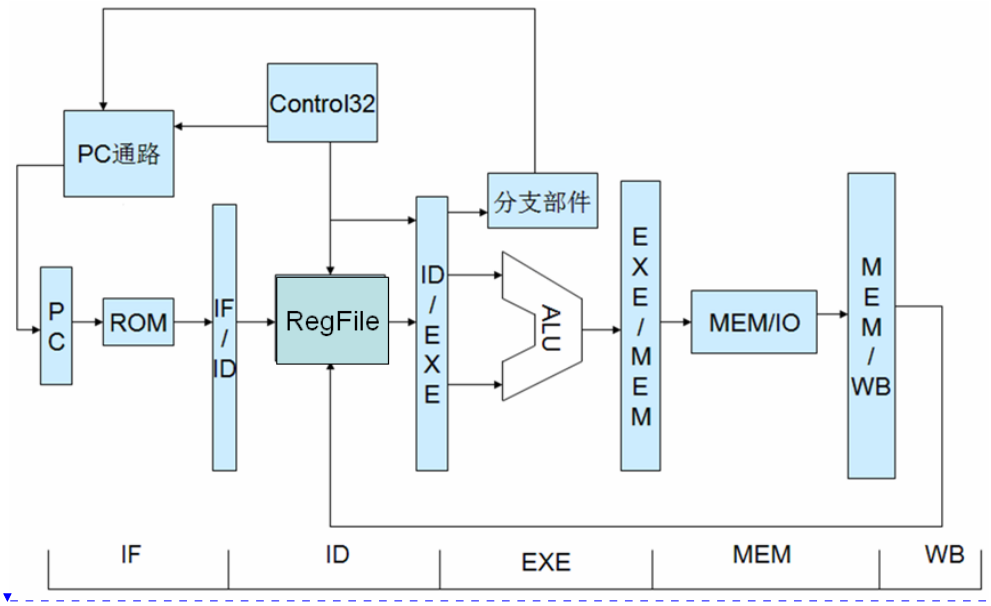


图 2.1 MinSys 流水线的结构

下面就 5 级流水段各个部分的功能作如下简单介绍:

- IF 阶段
  1) 根据 PC 值，从 ROM 中取出指令，送入 IF/ID 流水寄存器
  2) 将 PC 通路给出的下一地址写入 PC 寄存器
- ID 阶段
  1) 指令译码；
  2) 处理数据相关
  3) 把操作数从寄存器文件中读出
- EXE 阶段
  1) 使用 ALU 完成算术逻辑运算；
  2) 把运算结果保存到寄存器中，供 MEM 执行阶段使用；
  3) 给出各种控制信号
  4) 计算分支结果，给出分支地址，并进行分支验证，重定向 PC
- MEM 阶段
  1) 给出各种存取存储器的控制信号，并完成存储器的存取任务；
  2) 把从存储器中读入的数据保存到寄存器中，供 WB 执行阶段使用；
  3) 读写 IO
- WB 阶段
  1) 将指令执行结果写入到寄存器堆中；
  2) 数据过滤

2.2.2 相关性问题

流水型处理器设计的思想是，细分指令处理的各个阶段，让每个阶段都具有相似的低延迟，实现时间上的重叠处理以提高处理速度。然后，由于相关性的问题，导致流水处理器设计变得复杂。如何处理好相关性问题，是影响处理器性能的关键因素之一。相关性问题在处

删除的内容:

删除的内容: 改进后的

删除的内容: 图 2.1 是经过改进后的 MinSys 流水线结构简图，与原来的流水线相比，改进的流水线最大的不同就是将分支部件从 ID 段移到了 EXE 段。这样做的原因在于：对于 BEQ, BNE 指令，由于转移条件和两个寄存器内容相关，最坏的情况下，旧版系统需要从其他部分转发寄存器内容，加上译码的延迟，分支方向的产生过程包含了很大的延迟，而后移转移部件极大缓解了延迟问题，主频进一步提升。但这样也会带来一个问题，一旦分支发生转移，会造成 2 个时钟的 ... [1]

删除的内容: <#>根据下一指令的 PC 对分支地址进行分支预测，给出下一指令的分支地址。

带格式的: 项目符号和编号

删除的内容: <#>静态分支预测

删除的内容: 相关性的解决

理器设计中有三种:结构相关，控制相关和数据相关。

1. 结构相关

在本 MiniSys 处理器中，没有结构相关。

2.控制相关

在 MiniSys 中，引起控制相关的原因是分支转移导致顺序取指的破坏，需要重新定向取指入口。这样会导致流水线出现空泡，避免这种情况出现的一个方法是，在取下一指令前，提前预测下一指令的入口地址。本设计中就引入了分支预测机制来缓解控制相关导致的性能损失。

3.数据相关

在 MiniSys 中，导致数据相关的根本原因是，当前指令与他之前的 4 条指令中的某一条存在 RAW 相关性。为了最大限度防止流水线停顿，采用转发机制，其中只有一种可能性会导致一周期停顿，其他情况均不会带来流水线的停顿。下面就转发策略进行详细描述:

转发的原则是，当前操作所需要的寄存器内容肯定是最新的更新值。在本设计中，转发器开关位于 ID 段，因此最新的数据按时间顺序，可能是 EXE 的 ALU 结果，MEM 段的 ALU 数据或 MEM/IO 读出数据，WB 段还未回写的数据，最后是寄存器文件的数据。下面的转发策略就是按这个原理转发的，转发逻辑采用了优先级组合电路。

转发算法:

1) 当前指令需要地址为 X 的寄存器值
2) 查找 EXE 段是否有寄存器回写信号及回写地址是否为 X，若有，判断存储器寄存器传送信号时候有效，若无效，说明本段有需要转发的数据并进行转发。其他情况，转(3)。
3) 查找 MEM 段是否有寄存器回写信号及回写地址是否为 X，若有，判断存储器寄存器传送信号是否有效，若有效则转发存储器输出数据；若无则转发由 EXE 送来的 ALU 数据。其他情况转(4)
4) 检查 WB 段是否有寄存器回写信号及回写地址是否为 X，若有，转发 WB 段过滤后的数据，请他情况转(5)
5) 转发寄存器文件的内容

需要注意的是，当检测到 EXE 段的待写入地址是需要的目标地址，且 EXE 段是 LW 指令，则需要将流水线阻塞一个周期，让 LW 流入 MEM 段，EXE 段清零。

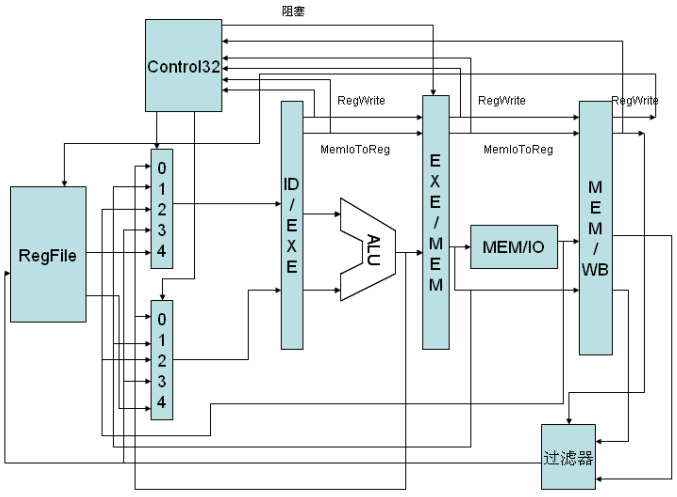


图 2.2 数据转发结构

图 2.2 就是数据转发的结构图，为了简洁起见，EXE,MEM,WB 三段保存的向控制器输出的寄存器写入地址没在图上表示出来。控制器中的转发部分就是根据上面的转发算法构造的。

2.2.3 中断控制器

本设计的中断控制器在下降沿检测中断。本系统提供两个中断 INT0 和 INT1,入口地址分别是 07F8H,07FCH。INT0 优先级大于 INT1,INT0 可以中断 INT1,但同级中断不能嵌套。为了更精确地进行中断，加入了长度为 4 的提交缓冲，用来记录最近在 EXE 段提交的指令信息，用于在 EXE 段空的时候，确定中断返回地址。同时，中断发生时，CPU 的 IF 寄存器可能会发生阻塞，所以，中断控制器会检测阻塞信号，并延迟 1 周期响应中断。

2.2.4 IO系统

IO 部分采用统一编址，使用存储指令 LW, SW 访问。其编址方式如图 2.3 所示:

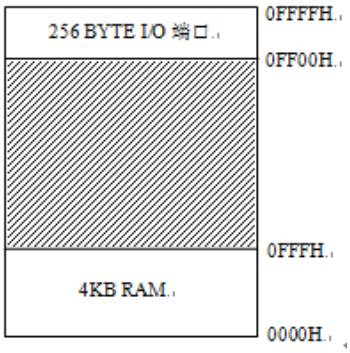


图 2.3 MiniSys 编址方式

IO 地址占 16 位，其中高 12 位用于设备选通，低 4 位用于端口寻址。这样一来，LW,SW 操作根据地址范围的不同，将产生内存操作或 IO 操作。

本 MiniSys 系统中，一共有四种 IO 部件，分别是 16 位计数器，扫描键盘，8 位 LED 显示和看门狗。其中，计数器和看门狗沿用自旧版系统。扫描键盘采用的是另一位同学设计。8 位 LED 是本人全新设计的。其结构如下:

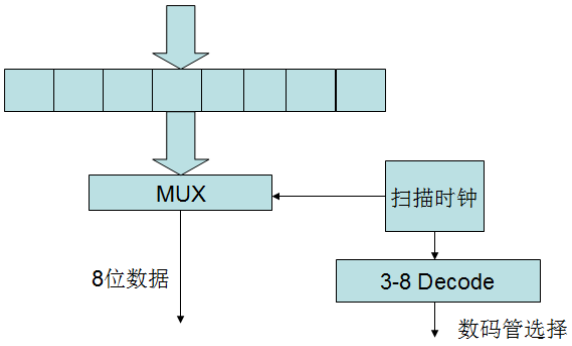


图 2.4 新的 LED 控制器结构

该 8 位 LED 采用硬件扫描，一次写入全部 32 位数据，十六进制输出。IO 系统通过了下载测试，性能和稳定性良好。

2.3 本章小结

本章节就分支预测所依托的处理器平台部分进行了详细介绍。本处理器是在综合课程设计处理器的基础上进行了相关的技术改造和技术升级，并重用了大部分器件，提高了开发效率。接下来的章节将介绍本毕业设计的非常关键的 mini-C 编译器部分，这个是分支预测部分实验的重要组成部分，测试程序全部用 mini-C 语言编写，并通过 mini-C 编译器编译成 MiniSys 能识别的机器代码。

第 3 章 mini-C编译器

3.1 编译器简介

为了降低 MiniSys 程序员编程的难度，提高 MiniSys 的易用性，我设计和实现了经过简化的 C 编译器，简称 mini-C 编译器。mini-C 并非利用其它编译器裁剪，而是利用编译原理课程设计的 SeuLex 和 SeuYacc 完成的分析器框架，其他部分完全是自主研发和设计的，具有完全自主知识产权。mini-C 编译器提供 while 循环，if-else 条件控制，一维数组，函数递归调用，端口操作，算术运算等，变量方面，只支持 32bit 有符号整数类型，变量声明只允许在全局区或函数内部开头部分定义。

由于 MiniSys 系统没有提供专用的函数返回指令，只能使用寄存器转移指令。而在 mini-C 编译器中，考虑到 MiniSys 代码空间大小，我们将寄存器转移指令作为专用的子程序返回指令，这样在分支预测器的设计中，特别设置了返回地址栈(Return Address Stack)针对编译器的函数调用过程优化，因此编译器和分支预测器是紧密联系的。同时，本设计的分支预测器的性能测试程序样本也是通过 mini-C 编译器进行编译的，因此本章将对 mini-C 编译器的结构、特点及有关技术给予较详细阐述，以便更好地说明性能测试中相关参数的特点。

3.2 编译器总体结构

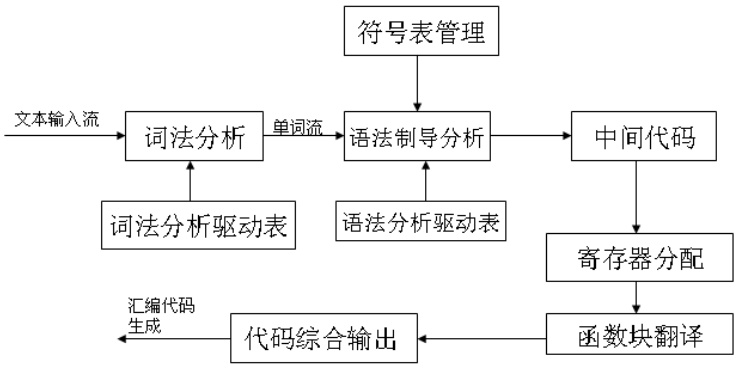


图 3.1 编译器结构

MiniC 编译器主要由词法分析器，语法制导分析筐架（含中间代码生成）,寄存器分配模块，函数块翻译模块，代码综合输出模块等几个部分组成。其结构示意如图 3.1 所示。

词法分析器在词法分析表的驱动下，从输入流中识别符号，为语法分析提供单词流。语法分析器采用了 LALR(1)分析法，并采用语法指导的方式进行中间代码的生成，符号表管理模块为语法分析过程提供单词登记，查询的功能，并参与变量的空间分配计算，为后续的代码生成,提供地址信息。寄存器分配部分负责对翻译过程中产生的临时变量进行寄存器关联，采用的策略是简单统计各个临时变量的使用频率，为使用最频繁的前 10 个临时变量分配寄

存器，对超出部分的寄存器，为其分配内存空间。函数块的翻译负责将中间代码生成部分产生的各个函数的中间代码翻译为汇编代码，并加上相应的现场保护，现场恢复的代码，完成子程序的翻译。最后的代码综合输出主要作用是，将各个子程序进行总装，加上中断入口，程序初始化代码，产生完整的一个汇编代码，完成翻译。产生的汇编代码只要通过 MiniSys 的汇编器翻译，就可以生成最终的机器代码，进入 CPU 运行。

3.3 词法和语法规则

在词法、语法规则中规定标识符是以字母开头的字母数字串。常数支持十进制和十六进制的常数。目前编译器只支持单文件且不允许包含任何其他库。鉴于此部分内容很多，而且也不是毕业设计的主要工作，因此在这里就不累赘，具体定义请参见附录 2

3.4 运行时的内存布局

Mini C 编译器产生的程序，运行时的活动记录基本类似于 ANSI C 的结构，考虑到 Mini C 对 C 语言进行了裁剪，对活动记录进行了适应性修改，活动记录采用了向上生长的方式，具体结构如图 3.2 所示。

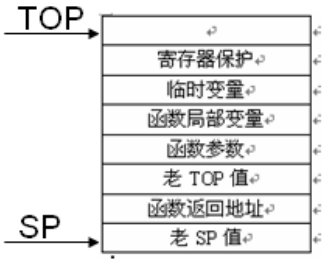


图 3.2 活动记录结构

SP 是指向当前活动记录底部的指针，TOP 是指向当前活动记录顶部的第一个空单元的指针。

当函数调用发生时，产生一个新的活动记录，当一个函数返回时，它的活动记录也被释放，从而实现了一个栈式动态分配。

- 活动记录的创建。当一个函数调用发生时，向 8(TOP),0(TOP)保存老 SP 和老 TOP,保护现场,初始化新的 SP 为 SP=TOP,新 TOP 初始化为老 TOP+新活动记录的大小。
- 活动记录的销毁和释放。当一个函数返回时,先完成现场的恢复，恢复 TOP=8(SP),恢复 SP=0(SP)。

图 3.3 是一般程序的内存布局示意图。

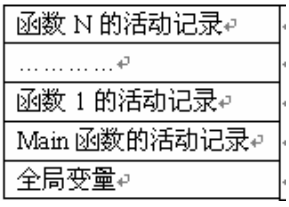


图 3.3 一般程序的内存布局

3.5 中间代码到汇编代码翻译的技巧

中间代码在完成临时变量的寄存器分配后，一共有三种变量形式，常数，t 类寄存器(10个),内存数，为了简化翻译过程,以四个 s 寄存器为基础，加入 MOV 中间指令，负责在三种变量类型之间进行转换，从而省略了其他中间指令翻译的代码量，实现了模块的最大化重用。MOV 中间指令的思想很简单，借鉴了 X86 指令集的 MOV,同时考虑到 MIPS 指令的规整性,以及 Mini C 的特点，将数据转换传送过程交给 MOV 处理，解决了代码重复问题。同时也要说明的是，本编译器采取的是简单的寄存器分配方法，分配效率较低，这样就会导致程序代码中访存操作会比较多，一定程度上影响系统的 CPI。

3.6 错误检测

提供一定的语法错误检测能力和丰富的语义错误检测功能。对于非法的符号，词法分析器有一定的能力跳过，从而使分析过程能够继续下去。语义检测功能中，提供了很好的类型匹配和参数匹配检测，像 ANSI C 一样，不提供对数组下标的检查。下面是一个简单的例子:

```
int fib(int n);
void main(void)
{ fib(1,2); }
```

编译器会检测出 fib 函数的参数数目不匹配，并给出提示。

3.7 本章小结

mini-C 编译器由于完成时间仓促，效率和性能还不甚理想，但其应付一般的应用足够有余。后面的分支预测器中的 RAS 结构是针对本编译器优化的，对于直接汇编的用户，需要将 JR,JAL 配合使用，而且 JR 不可用于函数返回以外的其他用途，否则 RAS 结构的优化特性将失效，但不影响程序正确性。下面将进入本论文最为核心的分支预测部分。

第 4 章 MiniSys分支预测部件的设计

带格式的：标题 2，不调整西文与中文之间的空格

带格式的：字体：四号

带格式的：字体：四号

4.1MiniSys五级流水线存在的问题

流水线只有在流水模式下才能获得最好的性能，所谓流水模式，即流水线中每个阶段都填充着指令，没有空流水段。而分支指令发生转移所导致的控制相关性会造成流水线出现空流水段而导致流水线效能下降。MiniSys 流水线就存在控制相关性导致的性能问题。下面，我们通过一个例子说明.首先，我们看一个简单 MIPS 汇编程序的片段，如图 4.1 所示

```
this:
ori $2,$0,10-----a
ori $3,$0,10-----b
beq $2,$3,this--c
slti $s4,$5,11---d
jal fun----------e
```

图 4.1 程序片段

为简单起见，我们用 a-e 标识每条指令，如图 4.1 所示。可以看到，本程序片段的分支指令为 c 和 e,而且由于 c 的转移条件成立，所以运行到 c 肯定发生转移。下面我我们来看无分支预测和有分支预测（且预测成功）的时序图对比:

无分支预测的时序图

周期	IF	ID	EXE	MEM	WB
1	c	b	a	X	X
2	d	c	b	a	X
3	e	d	c	b	a
4	a	NULL	NULL	c	b

有分支预测且预测成功的时序图

周期	IF	ID	EXE	MEM	WB
1	c	b	a	X	X
2	a	c	b	a	X
3	b	a	c	b	a
4	c	b	a	c	b

X:其他指令;NULL:本流水段空

图 4.2 有分支预测（且成功）与无分支预测的时序对比

从图 4.2 可以看到，在第一周期，C 指令已经处于 IF 取指段，而动态分支预测就发生在 IF 段。如果分支预测成功，下一周期将取得正确的下一条指令 a，而无分支预测的版本只能顺序取下一条指令 d。在第二周期，带分支预测的处理器取到了正确的转移目标地址的指令，而非预测版本顺序地取了错误的指令 d.在第三周期，C 进入了 EXE 段，将进行分支验证与指令重定向，由于分支预测版本成功预测了下条指令，因此不会发生转移重定向。而非预测版本取到了错误的指令，需要重定向指令流。在第四周期，预测的版本继续正常执行，而非预测的版本由于需要清除错误取到的指令而使得流水线出现了两段空泡（或流水线发生了两周期停顿），造成了流水线性能的下降。这个例子充分说明了分支预测的实际意义。

还有个与之相关的问题，就是分支部件的位置。这个问题会导致两个相互矛盾且与系统

性能紧密相关的参数的选择问题。由于 MiniSys 中有三种分支指令需要访问寄存器内容，如果将分支部件放在译码段(ID),势必会引起处理器中关键路径的延迟增加导致主频的下降，但在分支验证失败时只会损失一个流水段。如果将分支部件放在执行段(EXE)则会减少处理器关键路径延迟，带来主频的提升，但在分支验证失败时将损失两个流水段。因此，这两个参数的最佳搭配是一个很大的难题。经过测试我们发现，无分支预测的处理器其分支转移率为62%左右，这些分支转移都会引起流水线的停顿，由此我们得到无分支预测的系统每条分支空泡数期望为 1*0.62=0.62。而如果加上分支预测部件后，如果要使系统的每条分支空泡数的期望低于原系统，则分支预测失效率必须小于 0.62/2=0.31，即 31%。换句话说，从减少系统分支空泡率的角度说，要想让流水线系统性能提高，分支预测的准确率至少大于 69%。

同时我们也要注意到，我们的 MiniSys SOC 属于嵌入式系统范畴，其对功耗，资源消耗的需求也决定了我们设计分支预测器应该注意的问题。嵌入式处理器的主频一般不会很高，我们在设计 MiniSys SOC 分支预测器时应该尽量不影响主频，因此，我们将分支部件放在执行段。同时，我们设计预测准确率达到或超过 69%的分支预测器时，应当选择和设计硬件代价小且预测精度高的分支预测器。

因此，根据以上分析，我们设计 MiniSys 分支预测时的技术需求如下:

1.分支预测器预测精度要达到 69%以上

2.减少关键路径延迟，尽量减少对主频的影响

3.增加分支预测器后，对整体功耗影响应尽量小

4.分支预测器所用资源消耗不应太大

4.2 分支预测的方法

4.2.1 分支预测的本质工作

实验研究表明，分支指令的行为是可以预测的。减少分支开销，增加指令流吞吐率的一种主要方法就是对分支指令的目标地址和判定条件进行推测。一条静态分支指令在程序运行时经常会被重复地执行，他的动态行为是有章可循的，因此可以根据该指令过去的行为对它将来的行为进行有效的预测。分支预测(branch prediction)主要分为两部分，即分支目标地址的推测和分支条件的推测。任意一种推测（speculation）机制都必须能够验证分支预测结果，并在预测失败时能够恢复正确的执行方式。

4.2.2 分支目标地址的推测

分支目标地址的推测可以通过设置分支目标缓冲 BTB(Branch Target Buffer)来实现。这是一个本质上和 I-Cache 相似的小型 Cache，使用指令地址对 BTB 进行访问。图 4.3 就是 BTB 的一个原理图。

删除的内容: 4.1 分支预测的意义

流水型处理器只有在流水模式下才能达到最大的吞吐率。流水线处理器在取指时通常情况下都是按顺序取指的，而分支指令会造成控制流背离顺序的方向，使流水线停顿，只有等待新的地址产生后才能恢复正常的取指，这样一来就会造成流水线产生空的指令槽（空泡），严重影响流水线处理器的性能。分支预测技术就是在指令流发生变化前，预测下一条指令的入口，从而避免流水线中错误的指令流形成的空泡，提高流水线的效率。下面以一个简单例子说明分支预测的意义:

首先，我们看一个简单的程序片段

```
this:
ori $2,$0,10-----a
ori $3,$0,10-----b
beq $2,$3,this--c
slti $s4,$5,11---d
jal fun----------e
```

图 4.1 程序片段

为简单起见，我们用 a-e 标识每条指令，如图 4.1 所示。可以看到，本程序片段的分支指令为 c 和 e,而且由于 c 的转移条件成立，所以运行到 c 肯定发生转移。下面我们来看无分支预测和有分支预测（且预测成功）的时序图对比:

... [2]

带格式的: 字体: 四号

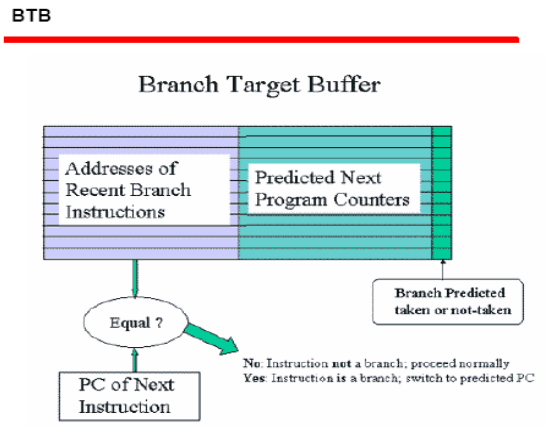


图 4.3 BTB 原理图

BTB 中一般包含两个域，分支指令地址 BIA 和分支目标地址 BTA。当访问的 PC 与其中一个 BIA 匹配时，即意味着 BTB 命中，当前指令是分支指令。而如果不命中，则认为当前指令不是分支指令，处理器继续按线性顺序取下一条指令。当分支指令最终执行后，根据最终结果更新 BTB 中的 BTA 内容。

对于间接转移分支地址的预测，目前还没有很好的办法。但其中有个特例就是对子程序返回地址的预测可以采取返回地址栈 RAS(Return Address Stack)的方法进行精确的预测。在函数调用指令执行后，将函数返回地址压入 RAS。当函数返回指令在 BTB 中命中时，根据指令的类型，选择从 RAS 中取出返回地址。在执行段对返回地址进行验证，同时弹出 RAS 栈顶。为了能识别指令类型，需要在 BTB 中加入指令类型的字段。

4.2.3 分支方向的推测

一般地，分支方向的预测分为两大类，静态预测和动态预测。

1.静态预测

静态预测技术中，最简单的方法就是将分支指令总是预测为不转移（转移）。这种方法实现简单，但效果不好。静态预测技术还有就是 M88110 所采用的软件静态预测技术，通过设置指令的预测位为转移或不转移来实现，但效果也是不理想。静态预测技术没能利用分支执行的历史信息，因此不能达到较高的精度。

2.动态预测

I.基于分支目标地址偏移的预测

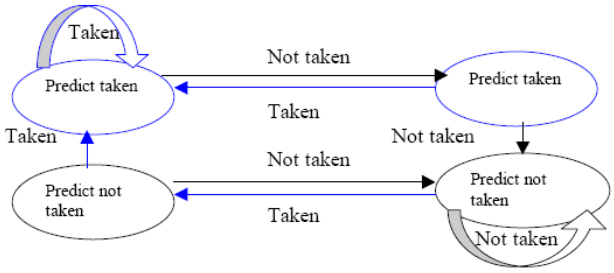
这种技术是根据当前分支指令地址和分支目标地址的相对偏移来确定转移方向。如果偏移为负，则很有可能是循环，则预测为转移。若相对偏移为正，则预测为不转移。这种技术还是没能充分利用动态的历史信息，故不能达到高精度。

II.基于分支执行历史的推测

a.单级预测技术

最经典也是在商业处理器中用得较多的是基于 2 位饱和计数器的 bimodal 预测算法，其状态转换图如图 4.4 所示。这种算法的历史信息采用一个 N 位饱和计数器器来实现。历史信息可以采用在 BTB 中附加历史位，或单独设置一个分支历史表 BHT 表实现。用程序指针的部分位访问 BTB, 查询 BTB 中是否有该项目，如果有则说明当前指令是转移指令，用历史信息

和 FSM 进行预测，当分支执行后，更新 BTB 和 BHT,若预测失败还要进行恢复。关于历史位的位数，Lee 和 Smith 进行了研究，当采用操作码类型的静态预测时，准确率为 55%-80%;采用一位历史位时，预测的准确率为 79.7%-96.5%;当采用两位历史位时，准确率为 83.4%-97.5%,再增加历史位提升的空间就非常小了。因此，两位历史位已经足够。可以看到单级的预测算法已经可以达到较高的预测成功率。



The states in a two-bit prediction scheme

图 4.4 2 位计数器状态转换图

b.两级预测技术

前面提到的这些动态分支技术存在许多的局限性。对某条分支指令的预测仅仅是基于该静态分支有限的历史信息，实际的预测算法没有考虑执行的上下文信息。研究表明，某些分支的行为同在它之前的分支指令的行为有着很大关联，因此，更精确的分支预测算法应当考虑其他关联分支指令的历史行为，从而动态地调整分支策略。所以，前面提到的预测算法很难再有大的准确率的提升。

1992 年,Yeh 和 Patt 提出了两级自适应分支预测技术[4],预测的精度可以达到 95%以上。两级自适应技术的核心是一个高度灵活的预测算法，可以随着上下文的变化调整自身的内容。在前面的方案中，分支地址仅索引一个分支历史信息表，每个地址只有一条记录，而在两级自适应算法中使用一个分支历史信息表的集合，称为模式历史信息表（PHT）。每个分支地址将索引一些记录的集合，然后根据动态分支上下文选择其中的一条记录，上下文信息是由分支历史信息移位寄存器（BHSR）来保存的，称为模式(pattern)。可以使用 BHSR 检索 PHT 并选择一个相关记录，其内容作为预测算法 FSM 的状态信息并生成一个预测结果。分支执行完毕后，利用实际分支的结果更新 BHSR 和选中的 PHT 记录。这个技术只是一个框架，可以有很多实现方法。

之后,McFarling,Young,Gloy 等人，对两级自适应预测进行了深入研究，设计了一种关联分支预测器（correlated branch predictor）[1]。如图 4.5,它使用统一的 BHSR 和共享式 PHT,并使用两位饱和计数器作为预测 FSM。BHSR 记录最近的 k 条分支执行方向，并保存动态控制流的上下文信息，PHT 可以看作是一个二维的阵列表，其中包含了 2^j 列,2^k 行以及两位预测符。如果分支地址有 n 位,其中 j 位来索引 PHT，选择 2^j 列中的一列，由于 j<n,可能有两个不同的分支地址选中 PHT 的同一列，即别名。所以这种 PHT 称为共享 PHT。BHSR 的 K 位模式用来索引已选中的 PHT 列，并从 2^k 条记录中选取一项。选中的记录内容是两位历史信息，用来完成基于历史信息的分支预测。还有一种是采用分支地址的高位索引单独的 BHSR,低位索引 PHT 列,用来增加跟踪和利用不同指令间关联性的灵活程度。

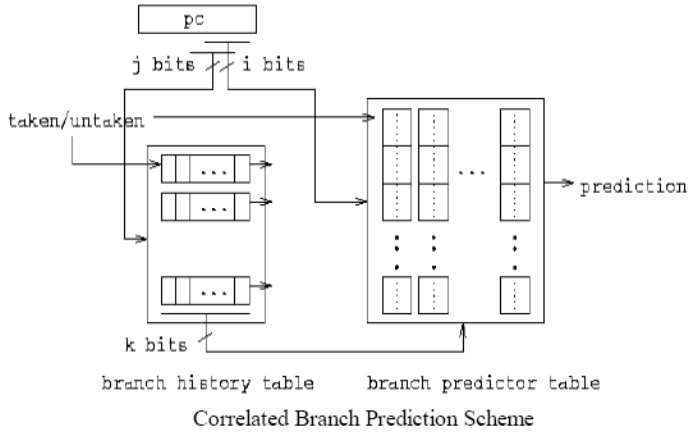


图 4.5 McFarling,Young,Gloy 关联预测器

前面的两级预测技术，特点是准确度很高，缺点是很高的硬件成本和逻辑复杂度，不太适合对性能和硬件成本敏感的嵌入式处理器。1993 年，Scott McFarling 提出一个非常高效的关联分支预测器,称为 gshare。如图 4.6,gshare 通过将分支地址的 j 位和统一的 BHSR 的 k 位进行 hash 计算（异或）,得出的结果为 max{k, j},索引 $2^{\max\{j,k\}}\times 2$ 位的 PHT 表，并选中一个两位分支预测符进行分支预测。gshare 只适用了一个统一的 k 位 BHSR 和一个很小的 PHT,就可以获得同其他关联分支预测器相当的精度。在 DEC Alpha 21264 超标量微处理器中就使用了该技术.

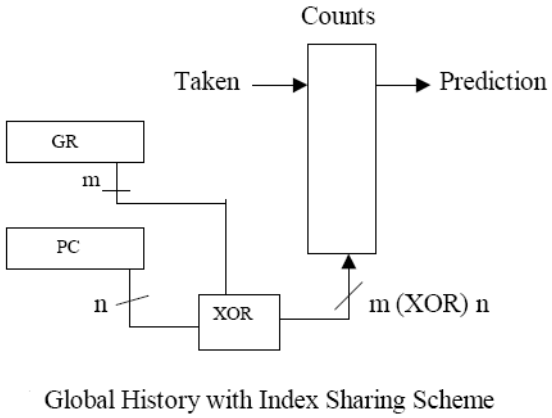


图 4.6 Gshare 预测器原理图

c.其他预测方法

近几年国际上学术界开始举办分支预测大赛,神经网络分支预测器成为当前的研究热点之一，比如将神经网络技术加入到分支预测中来，通过执行中对神经网络进行训练，来不断动态更新预测精度。

4.2.4 小结

本节就分支预测的本质，以及分支预测发展历史上最经典的预测器的原理和结构进行了阐述。可以看到，静态预测器由于其较低的精度，现在已经很少应用。而动态预测器方面，结构较简单的 Bimodal, gshare, bimode 预测器能以较低的硬件代价获得较高的准确性，这对于硬件待机敏感的嵌入式处理器来说具有十分重要的意义。而最新的神经网络预测算法则因为过高的硬件代价和延迟无法应用于商业处理器之中。本设计充分考虑了嵌入式处理器的特点，才选择了结构较简单的 Bimodal, gshare, bimode 进行分析并在 MiniSys 中实现。下面的章节将详细介绍 MiniSys 分支预测的实现过程。

4.3MiniSys 分支预测架构

4.3.1MiniSys中的分支指令

MiniSys 一共有 31 条指令，其中与分支有关的指令有 5 条，分为三种类型。

1.立即转移型 J

指令格式：J Address

J 指令是立即跳转型指令，其转移目标地址由该指令的立即数部分提供。

2.条件分支指令 BNE,BEQ

指令格式：BNE rt, rs, immediate

BEQ rt, rs, immediate

这两条指令转移条件为两个寄存器变量相等或不等时发生转移，其转移目标地址由该指令的立即数部分提供。

3.子程序调用和返回型 Jal, jr

指令格式：JAL target

子程序调用指令，转移发生时将它的下一条指令装入$ra 寄存器，转移目标由该指令的立即数部分提供。

指令格式：JR rs

子程序返回指令，转移目标地址由寄存器提供。(本预测器将 JR 针对 mini-C 优化)

4.3.2 MiniSys分支预测的策略

根据以上指令的分类和性质，将使用三种策略。

a. 立即型分支

这里将 J,Jal 归为立即型分支，对这类分支，通过设置分支目标缓冲(BTB)的方法可以进行非常高效的预测，只要 BTB 命中，且分支指令为立即型，则一定能成功预测。

b. 函数返回型分支

对 JR 的预测可以通过设置返回地址栈(RAS)的方法进行精确预测，当 Jal 执行后，将返

删除的内容: 4.3 在 MiniSys 5 级流水中分支预测时机的分析

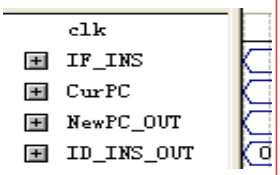


图 4.7 MiniSys 执行时序片段

图 4.7 是 MiniSys 执行时序的一部分。在图中：

IF_INS 是取指段的指令, ID_INS_OUT 是译码段的指令, CurPC 是程序存储器的输入地址, NewPC_OUT 是 PC 寄存器的输入值.

可以看到，在取指段时钟上升沿之前，为当前取指的地址，上升沿发生后 PC 装入了 PC+4。同时，当前指令也传入了译码段. 时钟下降沿来临时，程序存储器更新了指令。

因此，分支预测的时机在时钟上升沿来临之前。从时序上看，分支预测在 Minisys 是可以实现的。

删除的内容: 4

删除的内容: 4

删除的内容: 4

回地址压入 RAS,在 JR 于 BTB 命中时，从 RAS 栈顶取出目标地址，完成预测。

c. 条件分支预测

这个是本设计的一个重点。本设计中一共实现了 gshare,bimode,bimodal 三种预测器，通过实际性能的测试，对参数的讨论，选出最适合 MiniSys 的分支预测器，以及最合适的参数。

4.3.3 MiniSys 5 级流水中分支预测时机的分析

图 4.7 是 MiniSys 执行时序的一部分。

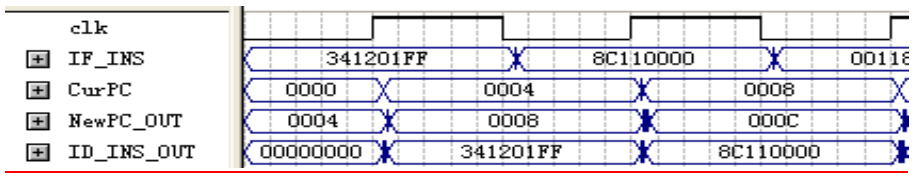


图 4.7 MiniSys 执行时序片段

在图 4.7 中：

IF_INS 是取指段的指令,ID_INS_OUT 是译码段的指令,CurPC 是程序存储器的输入地址,NewPC_OUT 是 PC 寄存器的输入值.

可以看到，在取指段时钟上升沿之前，为当前取指的地址，上升沿发生后 PC 装入了 PC+4 或转移目标地址。同时，当前指令也传入了译码段.时钟下降沿来临时，程序存储器装入了当前指令地址并更新了输出的指令。

因此，MiniSys 分支预测的时机在时钟上升沿来临之前。时序级的可行性分析为在 MiniSys 上实现分支预测提供了保证。

带格式的: 纯文本

删除的内容: 3

4.3.4 MiniSys分支预测器以及相关部件的关系

根据 4.4.3 的分支预测时机的分析，我们设计了与之相适应的分支预测结构，图 4.8 是带分支预测部件的流水线结构图。

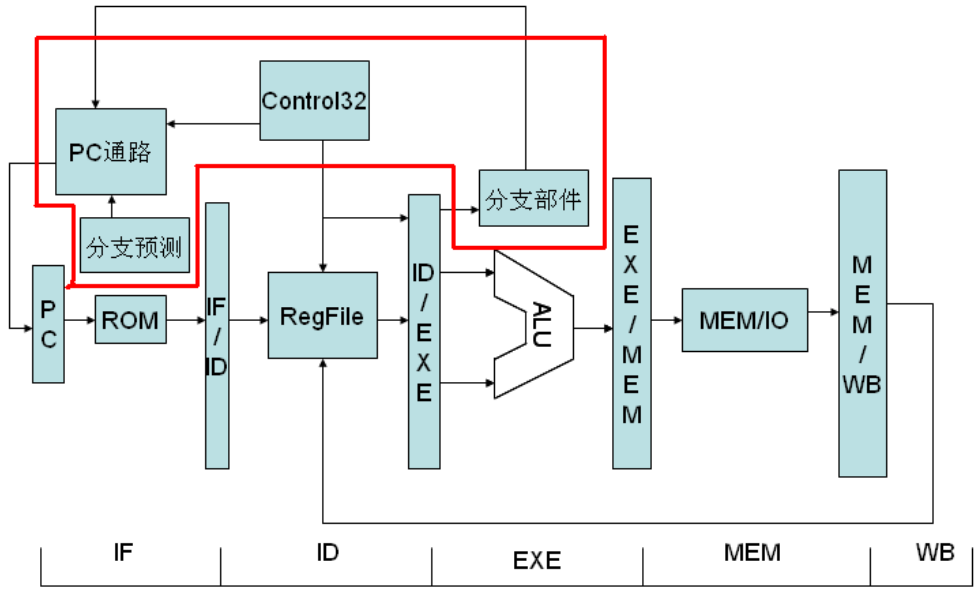


图 4.8 带分支预测部件的流水线结构

图 4.8 中有框的部分就是与分支预测相关的主要部件。PC 通路用于提供取值所需的目标地址，是一个 PC 地址的过滤器，其主要由多路开关组成，根据控制器(Control32)的信号选择正确的通路。分支预测器是本论文的核心，它向 PC 通路输出预测信息和预测的转移目标地址。同时，分支预测器也向 ID,EXE 段发送用于分支验证和预测器更新的信息。分支预测器的更新直接受控于 Control32 部件。分支部件连接着 PC 通路和 Control32 控制器，为 PC 通路提供最终转移目标地址，同时为 Control32 控制器提供分支验证和分支恢复的信息。在后文新加入的静态分支预测器位于 ID 段，它只连接 PC 通路，并合成来自 IF 段动态预测器的信息，送入 EXE 段进行分支验证。同时，静态预测器还直接影响 ID 段的清 0 操作。下面的小节，我们将对 PC 通路，分支预测器总体框架，更新控制逻辑和分支部件的结构进行详细的描述。

删除的内容: 4

4.3.5 MiniSys新的PC数据流

图 4.9 显示了为实现分支预测算法采取的新数据通路，下面分几种情况以及分支预测失败的恢复进行阐述。

1.中断发生

中断发生时，MUX1 选择数据通路 1，将中断入口地址引入 PC 寄存器，同时，清除 IF, ID 段未提交的指令。

2.分支预测与分支恢复

I. 取指阶段

若分支预测为不转移，则直接将 PC+4 装入 PC 寄存器。若分支预测有效且预测为转移，则将 BTB 的目标地址装入 PC 寄存器。

II.执行阶段

在这个阶段，分支的最终地址将被计算出来，分支方向最终确定。通过将实际的分支方向与分支地址同预测值进行比较，对分支预测进行验证。则有如下几种可能:

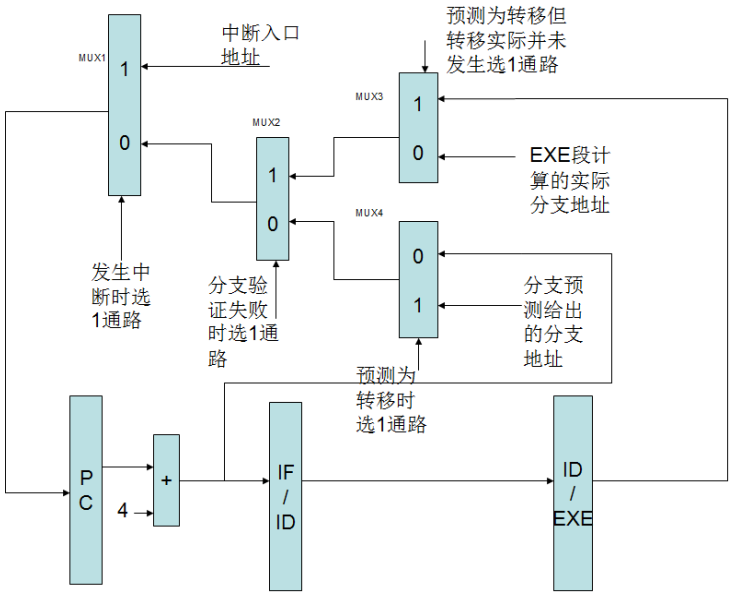


图 4.9 新的 PC 数据通路图

(1) 分支预测成功

这时，分支代价为 0，不需要重新定向指令流。

(2) 分支预测失败

若预测为转移但转移实际上并未发生，则需要将当前指令的下一条指令地址装入 PC 寄存器。同时清除 IF, ID 段指令。

若是预测为不转移但转移发生了，或转移地址错误，需要将 EXE 段计算的最终地址装入 PC 寄存器，同时清除 IF,ID 的指令。

删除的内容: 5

4.3.6 分支预测器的总体框架

图 4.10 显示了 MiniSys 中动态分支预测器部分的架构,下面分几个部分进行简单的说明.

1. BTB 分支目标缓冲

这个是一个类似于 I-Cache 的缓冲器，用来登记已经执行过的分支指令，分支地址，以及分支属性。BTB 采取 PC 访问的方式，如果 BTB 命中，说明当前是先前已经执行过的分支指令，输出的信息用于分支预测。更新控制由 EXE 段的信息控制.

2. Condition Branch Predictor 条件分支预测器

本设计一共实现了四种预测器，具体细节在后面的 4.5 节进行介绍。

3. RAS 返回地址栈

本部件是用于预测子程序返回的部件，内部由双向移位器组成。具体细节在后面章节介绍。

4. Fusion Logic&Select Logic

Fusion Logic 用于根据不同的分支属性选择不同的预测策略。

Select Logic 用于根据不同的分支属性选择不同的分支目标地址的来源。

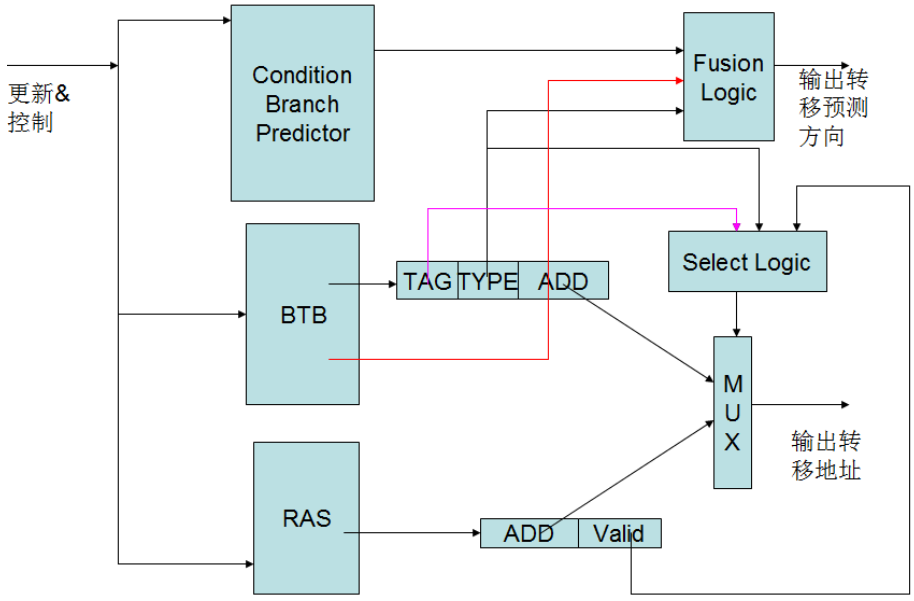


图 4.10 动态分支预测器的框架图

5. 预测的工作原理和策略

(1) 用 PC 访问 BTB,如果 BTB 命中，说明当前指令是分支指令(注:不命中也不说明当前不是分支指令，只是 BTB 中无此项),取出该分支的分支属性和分支目标地址，转(2)。不命中输出转移方向为不转移。

(2) 如果分支属性是立即型分支,则输出预测方向为转移，分支地址为 BTB 存储的分支目标地址。如果分支属性是条件分支，则分支方向由 Condiction Branch Predictor 提供，分支目标由 BTB 缓冲提供。如果分支属性为子程序返回型，则分支方向为转移，分支目标地址有两种情况，若 RAS 有效，则由 RAS 提供，否则由 BTB 提供。

4.4 通用部件部分设计细节

4.4.1 通用部件的含义

在 4.3.6 节中，我们展示了分支预测架构的示意图，在实际实现 gshare,bimode 和 bimodal 预测器时，除了条件分支预测器不同外，其他部件和信号连线时完全一致的，包括条件分支预测器的接口信号也是完全兼容的。我们将除条件分支预测器外的 BTB,RAS,Select Logic,Fusion Logic，流水段的改造和更新控制逻辑称之为通用部件。我们在下面将具体介绍这些通用部件的实现细节。

4.4.2BTB设计细节

1.BTB结构

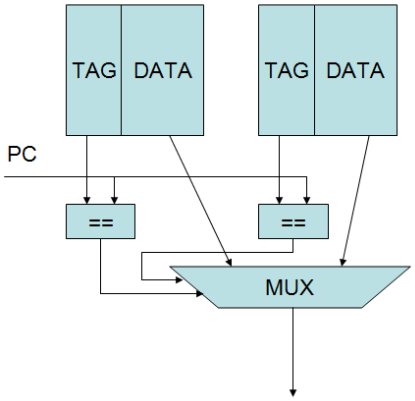


图 4.11 BTB 结构图

本设计的 BTB 如图 4.11 所示，该 BTB 采用的是 2 路组相联结构，每一路是 64 组。其中 TAG 字段是 15bit,DATA 字段是 16bit,BTB 总容量为 31*2*64=496Bytes.

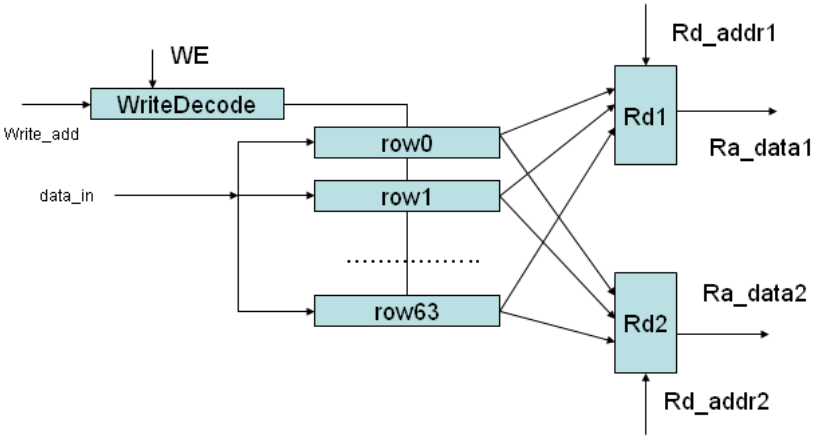
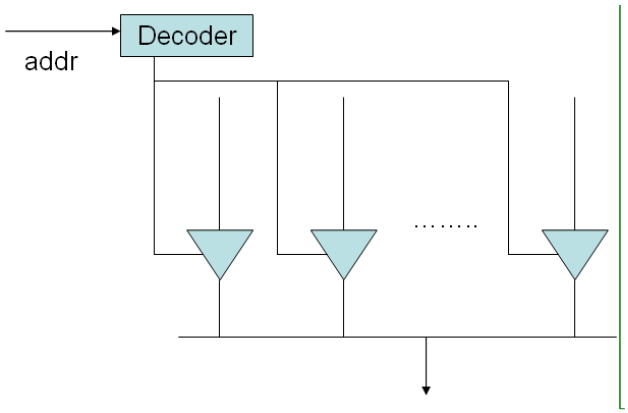


图 4.12 BTB 单组结构图

图 4.12 是 BTB 每一路内部采用双端口输出设计，其中一个输出端口用于正常输出，另一个端口用于置换算法中置换时候的输出。

图 4.13 是 BTB 双端口输出部分采取三态门总线的方式，这样延迟集中在译码器部分，可以减少数据输出时的延迟，节约器件的消耗。



批注 [y1]:

图 4.13 三态读出总线

2.BTB的表项结构及置换算法

I.TAG 区的表项

TAG 区表项如图 4.14。

删除的内容:（如图 4.14）

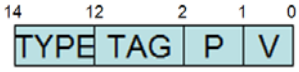


图 4.14 BTB 的 TAG 表项

a.TYPE 字段占 2bit,用于保存分支指令的属性，其编码规则为:

11: J/Jal (立即转移型)

01: bne/beq (条件转移型)

10: jr (子程序返回型)

00:-- (保留)

b.TAG 字段占 11bit，保存 PC 中地址的[15:5]位,用于命中对比。

c.P 字段占 1bit,并将其称之为置换位。具体的置换算法将在下文提到。

d.V 字段占 1bit,表示 BTB 中由 PC 索引的 TAG 和 DATA 项目有效，由硬件在第一次写入后置 1。

II.Data 表项

Data 表项如图 4.15 所示。

删除的内容: (如图 4.15)

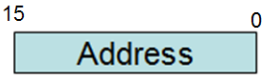


图 4.15 BTB 的数据区

该部分占 16bit,全部用于存储转移目标地址。

III.BTB 置换算法

(1)首先查找 PC[5:0]索引的 2 路 BTB 项目中是否存在 TAG=PC[15:5]的项，若存在，则将新的数据和分支属性写入该项。否则，转(2)

(2)查找 PC[5:0]索引的 2 路 BTB 项目中，是否存在 V(有效位)为 0（无效）的项，若存在，则写入该项，否则转(3)

(3)更新置换位(P)为 1 的项目

(4)若写入项的置换位为 1，则放弃写入，否则，采取随机置换(通过设置一个不断随时钟翻转的 1 位计数器实现)。

这个置换算法最大的创新之处在于，设置了一个置换位 P，实现了隐式删除。（所谓隐式删除就是将要删除的项目打上删除标记，而不是真正将其数据清零，这样，在新记录覆盖此记录之前，该记录是仍然有效的。）这个被隐式删除的项，在后续的执行中，可能没有被其他项替换，这样一来，当一段时间后，该项如被再次访问，就会被命中，这样就提高了 BTB 的命中率。

4.4.3RAS设计细节

1.RAS 的基本单元

删除的内容: I

带格式的: 字体: 加粗

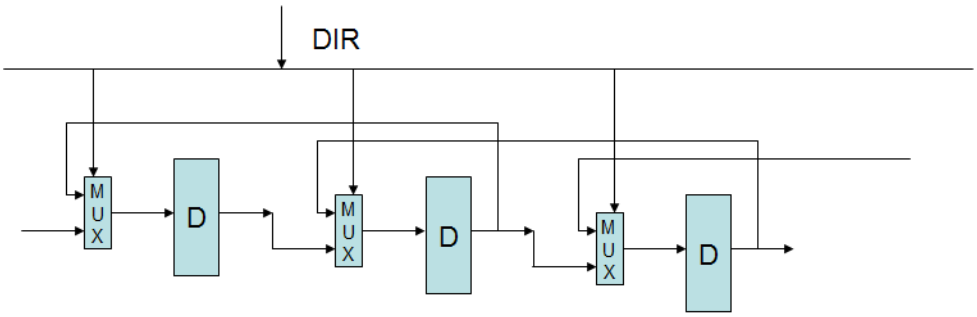


图 4.16 双向移位器结构

RAS 的基本单元如图 4.16 所示，RAS 的基本单元就是由 D 触发器和 2 路开关组成的双向移位结构，由 DIR 控制移位方向来实现一个硬件栈.

2.RAS 的表项

带格式的: 字体: 加粗

带格式的: 项目符号和编号

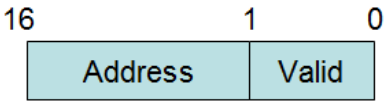


图 4.17 RAS 表项结构

RAS 表项由 16 位的转移地址 ADD 和一位有效位 Valid 组成，如图 4.17 所示。Valid 位在压栈时，由硬件将压入的一项置 1，在弹栈时，由硬件将由尾部移入的项目置 0.

3.RAS 的控制

带格式的: 字体: 加粗

带格式的: 项目符号和编号

为了保证 RAS 的弹栈和压栈操作是提交后指令所为，其控制逻辑位于 Control32 的总控制器中，通过监视 EXE 段提交的 Jal/Jr 指令完成入栈和弹栈操作。由于 IF 到 EXE 会有两周期的延迟，间隔在两条指令长度内的 jr 操作会造成一个 jr 的预测错误，所以程序员应当避免这样的情况发生。例如这样的情况，当前 JR 利用 RAS 预测的转移目标地址中刚好是另一个 JR 指令，这样，当第二个 JR 进入 IF 段时，前一个 JR 还在 ID 段，由于前一个 JR 未进入 EXE 段完成指令提交，RAS 不会弹出前一个 JR 的返回地址，从而使第二个 JR 拥有了前一个 JR 的地址，从而在分支验证时可能导致分支验证失败，流水线出现停顿。因此程序员编程的时候，在一个 JR 指令之后的第三条指令尽量不要也是 JR 指令。

4.RAS 设计参数

带格式的: 缩进: 首行缩进: 0 磅

本设计中 RAS 的深度为 10,可以精确预测深度小于 10 的函数嵌套调用过程。

删除的内容: IV.

带格式的: 字体: 加粗

4.4.4Fusion Logic设计细节

1.输出的信号

在实际 Verilog HDL 设计文件中，这部分主要输出两个信号,P_V 信号表明 BTB 是否命中，BTB 不命中则认为当前不是分支指令。还有个信号是 P_B 信号，主要表明分支的方向。

2.P_B 信号的合成

在 BTB 项目中，TYPE 字段给出的是指令的属性，如果 TYPE=11 或 TYPE=10 表示的是 J/JAL 或 JR 指令，这些指令肯定会发生转移，所以 P_B 输出 1.当 TYPE=10 时，分支方向由条件分支预测器给出。Fusion Logic 合成的结果就是根据这个原则产生的。

删除的内容: I

带格式的: 字体: 加粗

带格式的: 字体: 加粗

删除的内容: II

带格式的: 字体: 加粗

带格式的: 字体: 加粗

4.4.5Select Logic实现细节

这个部分的功能是要在 RAS 和 BTB 中选择一个分支目标地址的输出源。根据 BTB 表项的 TYPE 字段，当 TYPE=10（即 jr,子程序返回）时，且 RAS 栈顶有效，则选择 RAS 作为分支目标缓冲的输出，否则选择 BTB 作为输出。

4.4.6 更新控制逻辑

除 RAS 由 Control32 直接控制外，BTB,Condition Branch Predictor 的控制是由其自身完成的,Control32 只为 BTB, Condition Branch Predictor 提供分支最终结果，发出更新信号，EXE 的转移地址计算部件为 BTB 提供地址输入。

RAS，BTB 是整个框架的通用部件，在下面的章节会详细介绍条件分支预测器的原理和实现细节。

4.4.7 对MiniSys的分支预测适应性改造

为了兼容分支预测器，要对原有的 MiniSys 的 CPU 进行改造，下面就一些细节进行论述。

1. 流水寄存器的改造

为了进行分支验证和分支恢复，需要向 EXE 段的控制器传送相关的数据和信号。这些信号包括 P_V(预测有效)，P_B(分支方向)，预测的分支地址，被预测指令的下一条指令地址，2 位状态机状态信号。需要增加的信号如图 4.18 所示。

2. JR验证

增加一个比较器，对比最终地址和预测地址是否有偏差，如果有偏差，就说明分支预测失败。除 JR 指令以外的其他分支指令由于分支目标是固定的，因此只用检验分支方向。

3. 中心控制器的改造

需要增加的部件与其他部件基本独立，改造幅度不大。要增加的就是分支验证机制，判断分支是否成功，并输出相关的信号。如:PD_YN 表示分支预测为转移但实际未转移时输出 1。PD_FAIL 表示分支预测失败时输出 1。POP 在 EXE 段为 JR 且寄存器不是$i0 或$i1 时输出 1，

PUSH 在 EXE 段是 JAL 时输出 1。POP 和 PUSH 用于控制返回栈。UpdateGS 在 EXE 段是分支指令时置 1,用于发出更新信号。

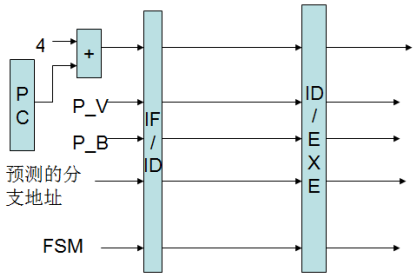


图 4.18 流水段的改造

4. 更新数据和信号

需要增加根据 EXE 段指令信息，产生分支转移地址的部件，用于更新 BTB 以及 RAS。更新信号由中央控制器产生，同时，EXE 段还应提供分支执行的最终分支方向用以更新分支预测器。中心控制器还要产生当前分支指令类型的编码，传给 BTB。BTB 置换位的信号需要由传来的 FSM 信号和分支结果产生，其规则是下次将被预测为转移时置 0,否则置 1。

5. PC数据通路的改造

这个改造的具体细节已在 4.3.5 节中给予阐述。

通过上述 5 个部分的改造，分支预测器将很容易与系统进行整合。这些改造措施在最大程度上实现了原有 CPU 的重用，而且在后面可以看到，分支预测部分的通用框架将很容易地整合多种条件分支预测器。这说明了整个处理器部分的结果的合理性和通用性，非常适合以后进行更进一步的加强改造。

6.静态分支预测器的加入

我们将在动态预测器实现细节,性能测试的讨论后进一步讨论模仿 ARM11 的机制加入一个新的静态分支预测部件，以及进行的相关技术改造和技术性能测试。

4.5 条件分支预测器

条件分支预测器是分支预测的核心部分，立即型分支和子程序返回型分支都能利用非常高效的 BTB 和 RAS 完成预测。条件分支的预测一直是分支预测研究和改进的热点。

在本毕业设计中，我们一共实现了三种非常有代表性的也适合在嵌入式 CPU 中采用的分支预测算法。分别是 gshare,bimode 和 bimodal。本节就三种方法的思想，原理以及实现细节进行阐述。

4.5.1bimodal预测器

1. bimodal的原理

bimodal 是一个相当经典的单级预测器，属于动态分支预测器的一种。他基于两位饱和计数器的方法来记录分支的静态历史信息，他通过 PC 值来直接索引一个 2 位饱和计数器，

通过饱和计数器的值来进行预测。

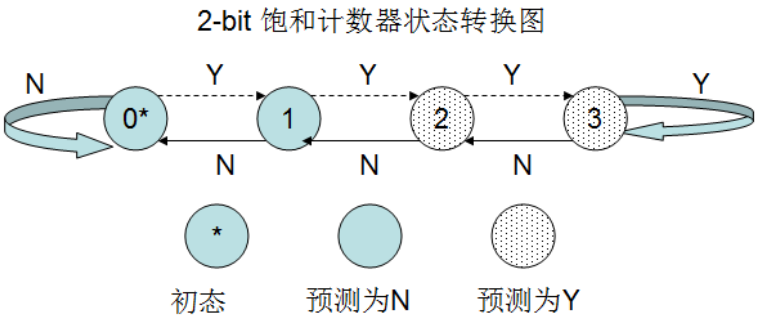


图 4.19 2 位饱和计数器状态转换图

图 4.19 所示就是一个 2-bit 饱和计数器的状态转化图，初始状态为 0。如果分支的最终结果是发生转移，则计数器加 1，否则计数器减 1，且在 0,3 饱和。如果计数器的值大于 1，则预测为转移，否则预测为不转移。

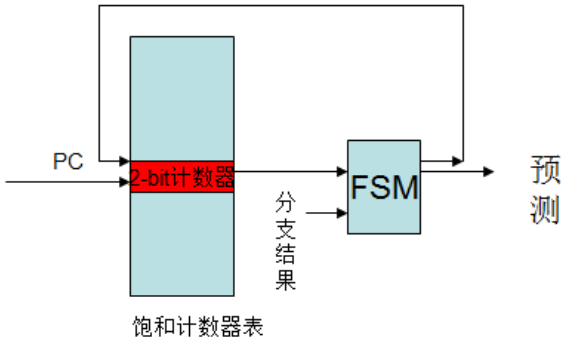


图 4.20 bimodal 原理

图 4.20 就是 Bimodal 预测器的原理图，通过 PC 索引一个 2 位饱和计数器，通过 FSM 生成预测位进行预测。当分支结果产生后，根据分支结果和当前的计数器的值更新饱和计数器表。

2. Bimodal实现细节

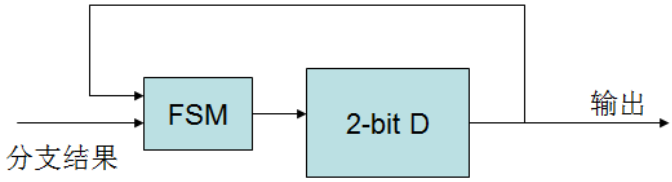


图 4.21 饱和计数器表的表项结构

图 4.21 展示了饱和计数器的一个表项，鉴于状态机很简单，每个单元采取独立状态机可以提高更新速度，不必采取代价更大，速度更慢的双端口输出来更新状态机。每个状态的存储采用 2 位 D 触发器实现。

饱和计数器表的各个单元采用与 BTB 相同的方法，使用三态输出总线方式挂载，保证了较低的输出延迟并降低输出电路的复杂性。

对分支表的索引采取直接索引法，其索引值就是 PC[9:0]的内容。

3. Bimodal预测器的参数设置

Bimodal 预测器共设置了 1024 个表项，总共 1024*2=2048bits 的容量，而且就目前处理器的寻址空间来看，几乎完全覆盖每条指令，所以后面 Bimodal 的测试数据可以认为是无限入口的 Bimodal 预测器。

4. Bimodal预测器的优劣

Bimodal 最大的优势在于其实现非常简单，能够利用指令自身的强烈偏向特性，缺点在于没有考虑分支执行的上下文信息，不能达到很高的精度。

4.5.2 gshare预测器

1. gshare的原理和思想

进一步的研究表明，分支转移行为与之前其他分支执行的上下文信息有关，也就是说当前分支的行为可能与之前执行的若干分支存在关联性。特别在高级程序中大量存在的 if-else 结构等都存在着分支关联特性。在以前针对单一静态分支行为的预测基础上，学术界提出了两级关联分支预测器。Yeh 和 Patt 提出的两级自适应预测框架的硬件代价过于庞大，只有 Pentium Pro 等少数处理器实现过。Scott McFarling 经过深入研究，提出了非常高效的 gshare 预测器，在拥有和其他关联预测器类似精度的情况下，具有非常小的硬件代价。DEC Alpha 21264 的混合预测器中，其中一个预测器就是用的 gshare 预测器。研究表明，在 SMT 中，gshare 获得了出人意料的高准确度。本文中，gshare 是在设计之初就是优先考虑的，后面的测试数据也表明了 gshare 在低硬件代价处理器上的优势。

2. gshare框架

图 4.22 是 gshare 预测器的原理图。gshare 预测器通过 BHSR 的 J 位移位寄存器来跟踪最近 J 条分支指令的历史，然后和 K 位 PC 进行异或计算，来离散 PHT 表的索引，减少由不同分支访问同一 PHT 记录带来的分支别名干扰。PC 与 BHSR 异或后，产生的 Max{J,K} 位索引来索引 PHT。PHT 被称作为模式历史表(Pattern History Table),PHT 的表项存储的是 2 位饱和计数器，也就是说预测器的第二级采用和 Bimodal 相同的预测方法。

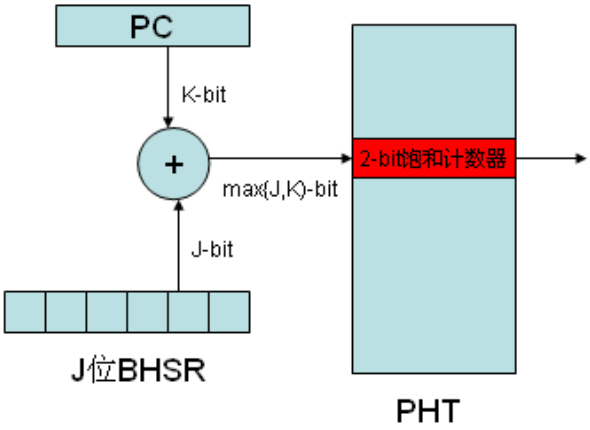


图 4.22 gshare 原理图

3. gshare实现细节

I. PHT 表

采用 Bimodal 预测器设计中的饱和计数器表，从结构和功能上讲，两者完全相同。Bimodal 实际上可以看作是 BHSR 退化为 0 的 gshare 预测器的一个特例。

II. BHSR 历史移位寄存器

采用 K 位 D 触发器搭建的移位寄存器。

4. gshare参数配置

PHT 采用 1024 表项，一共 1024*2=2048bits 预测位，BHSR 采用 10 位移位寄存器。对于目前的 MiniSys 结构，1024 入口已经足够。具体的性能测试在后面章节中详细论述。

4.5.3Bimode预测器

1. Bimode预测器的原理和思想

在所有的分支指令中，有相当多数的分支指令有强烈的偏向特性。由于 PHT 表的容量限制，可能发生两种完全相反偏向特性的分支指令访问同一 PHT 表项，从而导致精度的损失。为了减少这种情况的发生，把容易发生跳转和不跳转的模式历史信息分布放入不同的 PHT。它由 3 部分组成，一部分用来选择 PHT，另外两部分表示 PHT 的方向，分别为跳转和不跳转，PHT 的方向被全局历史索引。

2. Bimode预测器结构

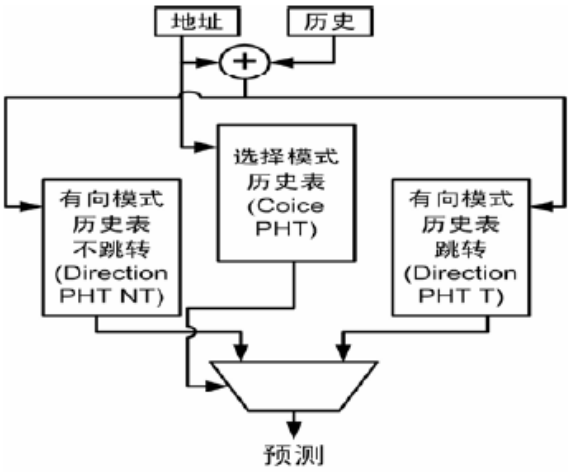


图 4.23 bimode 结构图

如图 4.23 所示，Bimode 的结构和 gshare 类似，可以认为是 gshare 的改进版本。Choice PHT 用于根据当前选中的 FSM 选择 Token PHT 和 UnToken PHT 中的一个作为预测输出。

3. Bimode实现细节

I. 三张 PHT 表

三张 PHT 表除了容量不同外，其他参数和功能完全相同。可以采用 gshare 使用的 PHT 表。

II. 地址索引

这部分采用 gshare 的地址索引器。

III. 更新策略

BHSR,Choice PHT 是随分支指令的执行不断更新的，Token PHT 和 Untoken PHT 只有在预测过程中被选中的表才进行更新。

4. Bimode参数选择

I. PHT 表参数选择

Choice PHT,Token PHT,Untoken PHT 的容量比为 2:1:1,大小分别采用 512,256,256 入口大小，总容量和 gshare 相同。

II. BHSR 容量选择

采用 9bits 大小的移位寄存器，PC 选择 PC[8:0]进行 XOR 操作，产生的 9 位索引区索引三张 PHT。

5. Bimode初步评价

Bimode 预测器是对 gshare 的改进，而且硬件复杂度几乎没有增加，从理论上讲，Bimode 比 gshare 更优化。但考虑到 MiniSys 结构较简单，gshare 更大的 PHT 可能会更有优势。后面章节的实验数据将验证这一说法。

4.5.4BTB置换位策略

根据前面提到的三种条件分支预测器的特征可以看到，他们的模式历史信息都采用的是 2 位饱和计数器，这样一来，在分支结果产生后，我们可以以很小的代价知道下次该分支是否转移，从而可以优化置换位的策略，提高 BTB 的效率。

最佳的 BTB 置换位策略是，取出当前分支的旧历史记录，在当前分支执行的最终结果产生后，通过有限自动机获得下一状态，若下一状态预测为转移，则设置置换位为 0,否则设置为 1。

这样的好处是，如果下次预测为不转移的项目被隐式删除，那么在 BTB 中可能被替换掉，再次访问时就会发生不命中，从而预测为不转移。这样可以使 BTB 得到充分利用。

4.6 性能测量与分析

我们实现了基于 gshare,bimode,bimodal 的三种条件分支预测器，根据这三种条件分支预测器，我们实现了三种分支预测器，本节我们将对这三种预测器进行性能测量和分析。

带格式的：正文

带格式的：字体：五号

4.6.1 测试方法和测试程序集

在本设计中，将使用第三章中介绍的为 MiniSys 设计的 Mini-C 编译器来编译测试程序集。Mini-C 编译器的特点是存在大量的访存操作，CPI 可能会偏高。

为充分考察预测器对三种类型分支预测的性能，特别选择了有代表性的 QuickSort 作为

综合性能测试的程序。同时,还包含了循环为主的 KMP 算法，以及与输入无关的 8 皇后问题，用来测试系统的分支预测准确性。

这些测试程序都采取一遍运行的方式，考察 CPU 首次执行时的分支预测准确性。

本测试采用的是硬件计数器法，分别统计分支执行数量，分支预测成功的数量，提交指令数量和时钟周期数，并分别计算预测准确率和 CPI，其中：

CPI=程序执行完成所需的时钟周期数/提交指令数量

预测准确率=分支预测成功的数量/分支执行数量

下面分别介绍这三个测试程序。

1.quick sort（快速排序）

I.mini-C 源代码

```
int array[200];
void quicksort(int array[20],int left,int right);
void InterChange(int array[20],int left,int j);
void main(void)
{
    int i;
    i=0;
    quicksort(array,0,29);
}
void quicksort(int array[20],int left,int right)
{
    int i;int j;int pivot;
    if(left<right)
    {
        i=left;j=right+1;pivot=array[left];
        while(1)
        {
            while(1)
            {
                i=i+1;
                if(array[i]>=pivot) break;
            }
            while(1)
            {
                j=j-1;
                if(array[j]<=pivot) break;
            }
            if(i<j) InterChange(array,i,j);
            if(i>=j) break;
        }
        InterChange(array,left,j);
        quicksort(array,left,j-1);
        quicksort(array,j+1,right);
```

```
        }
    }

    void InterChange(int array[20],int left,int j)
    {
        int temp;
        temp=array[left];array[left]=array[j];array[j]=temp;
    }
```

II. 程序特点及考察方向

这是一个基本快速排序的源代码，从程序代码可以看到，程序中有大量的循环，递归调用，条件分支。涵盖了 MiniSys 所含有的全部分支类型。对综合分支预测能力是一个非常好的考验。表 4.1 是编译过后的快速排序程序汇编代码中分支语句统计。

表 4.1 快速排序程序中分支语句统计表

分支指令	j	jal	jr	bne	beq
数量	9	6	5	1	11

总指令数为 329,分支指令占 9.7%,访存指令占 56.8%,其他指令占 33.5%,条件分支占分支数的 37.5%。QuickSort 还有一个最大特点，输入集合对分支预测准确性有较大的影响，主要是 pivot 基准是比当前值大还是小是一个比较随机的。

2.KMP

KMP 是非常有名的模式匹配算法，线性时间复杂度。

I.KMP 的 mini-C 源代码。

```
int pat[10];
int f[10];
int res;
int string[300];
void fail(int pat[10]);
int FastFind(int f[30],int string[300]);
void main(void)
{
    fail(pat);
    res=FastFind(f,string);
}
void fail(int pat[10])
{
    int j;int LengthP;int i;
    j=1;
    LengthP=10;
    f[0]=-1;
    while(j<LengthP)
    {
        i=f[j-1];
```

```
                while((pat[j]!=pat[i+1])&&(i>=0)){ i=f[i];}
                if(pat[j]==pat[i+1]) f[j]=i+1;
                else f[j]=-1;
                j=j+1;
            }

        }

        int FastFind(int f[30],int string[300])
        {
            int j;int i;int LengthP;int LengthS;
            LengthP=10;LengthS=300;
            j=0;i=0;
            while((j<LengthP)&&(i<LengthS))
            {
                if(pat[j]==string[i])
                {
                    j=j+1;i=i+1;
                }
                else
                {
                    if(j==0) i=i+1;
                    else j=f[j-1]+1;
                }
            }
            if((j<LengthP)||(LengthP==0)) return -1;
            else
            {
                return i-LengthP;
            }
}
```

II.KMP 的特点和考察方向

从源代码可以看到,KMP 算法中主要特点就是循环和 if-else 结构，这对于关联预测器是非常有利的。表 4.2 是编译过后的 KMP 程序汇编代码中分支语句统计。

表 4.2 KMP 程序分支语句统计表

分支指令	j	jal	jr	bne	beq
数量	16	3	5	4	9

总指令数为 403,分支指令占 9.2%,访存指令占 63.5%,其他指令占 27.3%,条件分支占分支数的 35.1%。此程序立即分支较多，对 BTB 预测有利。

3. 8 皇后问题（不考虑棋盘对称性）

本程序只是遍历整个解空间，求出所有的可能性，并给出解的数量。

I.mini-C 源代码

```
int c;
    int abs(int n)
    {
        if(n<0) return 0-n;
        else return n;
    }
    int CheckTheChessBoard(int queens[20], int i)
    {
        int j;int k;
        j=0;
        while(j <= i )
        {
            k=0;
            while (k <= i)
            {
                if ((j != k)&& ((queens[j] == queens[k]) || ( abs(queens[j] - queens[k]) ==
    abs(j - k))))
                {
                    return 0;
                }
                k=k+1;
            }
            j=j+1;
        }
        return 1;
    }
    void Trial(int queens[20], int i, int n)
    {
        int j;
        if (i >= n)
        {
            c=c+1;
        }
        else
        {
            j=0;
            while(j < n)
            {
                queens[i] = j;
                if(CheckTheChessBoard(queens,i)==1)
                {
                    Trial(queens, i + 1, n);
                }
```

```
                    j=j+1;
                }
            }
        }
        void main(void)
        {
            int i;int queens[8];
            c=0;
            i=0;
            while(i<8)
            {
                queens[i]=-1;
                i=i+1;
            }
            Trial(queens,0,8);
            $0xff00=c;
}
```

II.8 皇后问题特点及考察方向

表 4.3 是编译过后的 8 皇后问题程序汇编代码中分支语句统计。

表 4.3 8 皇后程序中分支语句统计表

分支指令	j	jal	jr	bne	beq
数量	16	6	6	5	12

总指令数为 420,分支指令占 10.7%,访存指令占 54.5%,其他指令占 34.8%,条件分支占分支数的 37.8%。总体看指令分布除访存外，都比较平均。本程序与输入无关，测试只需一次即可。

4.6.2 对比测试

1.QuickSort随机序列长度为 30

随机序列由 PC 的一个 C 程序产生，用于 Ram.mif 文件初始化。图 4.24 是来自于一次测试的 qsort(30)测试结果。

带格式的：居中

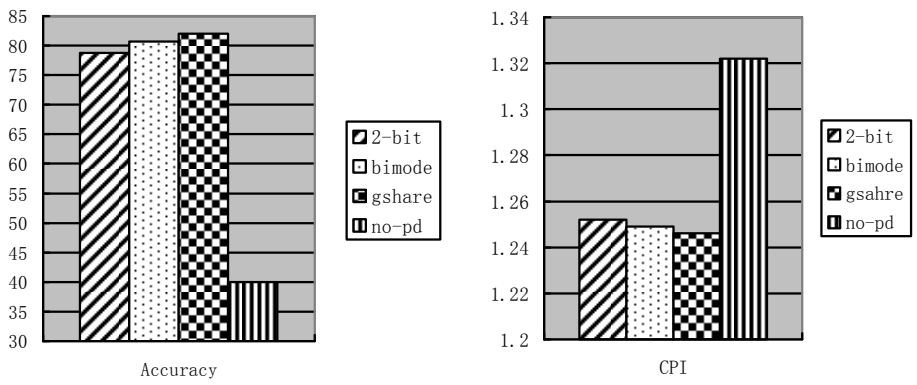


图 4.24 qsort(30)测试结果

根据测试结果，从准确率的观点，显然 gshare 最好，Bimode 稍差，Bimodal 最差。前面我们已经论述过，1024 表项对 Bimodal 来说已经无穷大，也就是说理想 Bimodal 预测器精度在本次测试中已经低于了关联分支预测器。单级预测器的局限性体现出来。从 CPI 上看，性能与分支预测准确率是呈现正比关系。装入分支预测后，性能确实有提高。no-pd 代表的无预测器的情况，可以看到，没有预测器时，CPI 很高，性能较差。

2. 当QuickSort随机序列长度为 60

随机序列由 PC 的一个 C 程序产生，用于 Ram.mif 文件初始化。图 4.25 是来自于一次测试的 qsort(60)测试结果。

带格式的：居中

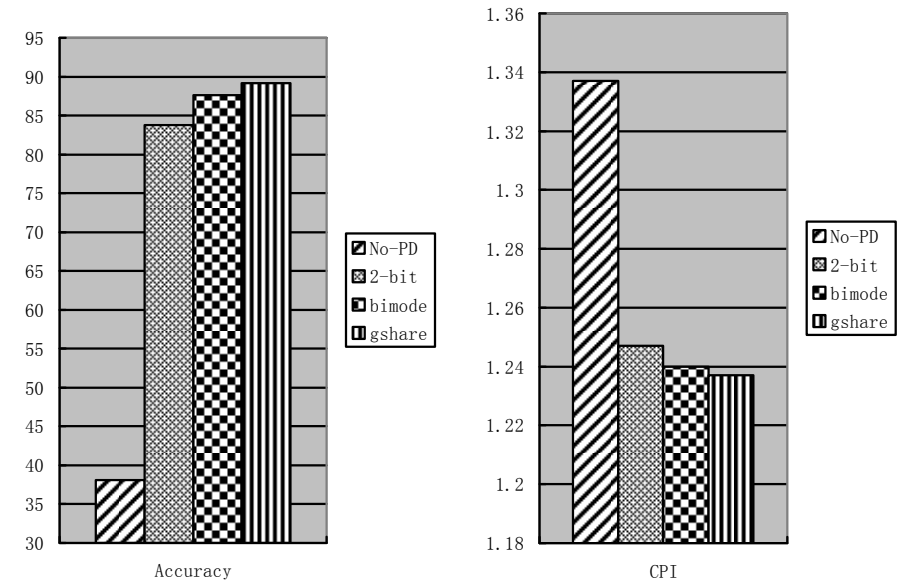


图 4.25 qsort(60)测试结果

从测试结果看到，更长的输入序列使得预测器有了更充足的学习时间，预测准确度有很大的提升。就三种预测器对比来看，gshare 还是获得了最好的准确率，Bimode 次之，Bimodal 最差。三者与不预测版本相比，性能提升十分明显。

3. KMP测试

模式串长度 10，待匹配的主序列长度 300,由 PC 通过 C++程序随机生成 Ram.nif 初始化文件。图 4.26 是 KMP 测试结果。

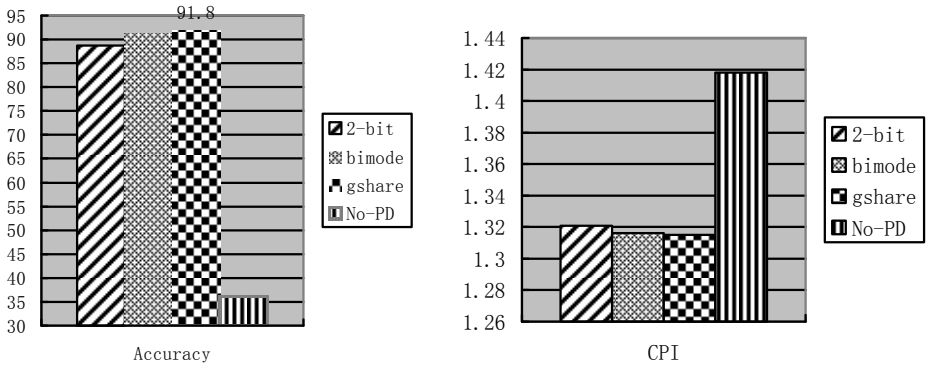


图 4.26 KMP 测试结果

测试结果与前面类似，gshare 依然取得了最好的预测准确度，bimode 次之，bimodal 最差。我们看到，关联预测器确实比单级预测器有更好的性能，单级预测器的局限也得到体现。通过这三个例子的测试，我们已经看到了 ghare 在三种预测器中取得了最好的预测准确度和最好的 CPI 值，考虑到处理器的实际性能和功耗面积的需求，我们将在下面的章节进行更详细的分析和论证，以选出最适合 MiniSys 的分支预测器。

4.7 使用资源与功耗上的分析

嵌入式应用对功耗和芯片面积是很敏感的。芯片的功耗直接决定了便携式应用的电池续航时间。而芯片的面积则直接决定了设备的成本。应此，我们需要对三种预测器的功耗和资源使用进行分析。前面 4.5 节已经提到，除条件预测器本身的区别外，包括条件预测器接口在内的其他所有部件都是完全一样的，因此，我们通过考察整合以后系统的资源使用和功耗来更全面和客观地评估三种预测器在功耗和面积上的优劣。

如下表 4.8.1 所示，就是三种预测器的功耗与资源消耗情况:

表 4.8.1 功耗与资源消耗

预测器	gshare	bimode	bimodal	无预测器
器件消耗	13142(40%)	12776(38%)	13035(39%)	3454(11%)
功耗	114.94mw	115.05mw	115.07mw	114.99mw

从表格 4.8.1 看到，gshare 虽然器件使用最多，却取得了和无预测器版本极为相近的功耗。从这点说，gshare 功耗是这张表中最好的。再来看器件消耗，最少的是 bimode,最多的是 gshare,估计的原因是 bimode 的三张 PHT 虽然总容量和 gshare，bimodal 一样，但 bimode 每张 PHT 大小却只有 Bimodal 和 Gshare 的 1/2 或 1/4,而且是并行读取，因此，其表的互联结构要比 bimodal 和 gshare 简单，考虑到 FPGA 的分布互联结构，因此其取得了器件消耗代价的最低。总的看来，三种预测器在功耗和资源使用上各有优劣。

带格式的: 居中

删除的内容: 是

删除的内容: 对

4.8 实际性能测试与分析

CPI 作为处理器的一个重要指标，是衡量 CPU 性能的总要参数，一般我们希望 CPI 越低越好，但可能有一个误区，不是 CPI 越低，处理器性能就越好。CPI 低的处理器其结构相对高 CPI 的处理器复杂，这样会导致时钟频率的下降，如果 CPI 改善的比率小于时钟频率减小的比率就会导致实际性能的下降，我们引入 MIPS(每秒百万指令数)来衡量处理器的实际性能。下面的工作就是对比加装分支预测前后，处理器实际性能的对比。

采用的数据是前面性能测试章节的数据。由于 FPGA 的分布式互联结构，其综合布局布线后的延迟无法预期，所以统一采用关闭优化选项，使用 EP2C25 的芯片综合，其结果如表 4.6 至表 4.9。

表 4.6 使用不同预测器的处理器主频

	最高主频（MHz）
无预测	33.46
gshare	32.37
bimode	32.73
bimodal	34.04

表 4.7 执行快速排序程序的实际性能

样本 1	最高主频（MHz）	CPI	MIPS
无预测	33.46	1.322	25.31
gshare	32.37	1.246	25.98
bimode	32.73	1.249	26.20
bimodal	34.04	1.252	27.19

表 4.8 执行 KMP 程序的实际性能

样本 2	最高主频（MHz）	CPI	MIPS
无预测	33.46	1.337	25.03
gshare	32.37	1.237	26.17
bimode	32.73	1.240	26.40
bimodal	34.04	1.247	27.30

表 4.9 执行 8 皇后程序的实际性能

样本 3	最高主频（MHz）	CPI	MIPS
无预测	33.46	1.418	23.60
gshare	32.37	1.315	24.62
bimode	32.73	1.316	24.87
bimodal	34.04	1.321	25.77

从三张表格看来，所有加装预测器的处理器实际效能均高于不加装的版本。而且我们还应注意到，gshare 虽然预测精度最高，但其最高主频较低使得他的实际性能位于三种预测器中最低的位置。所以，CPI 不完全决定处理器的实际性能，鉴于 FPGA 互联的复杂性，以上的测试数据只作为参考，只能在一定程度上说明问题。

4.9 分支预测选型与进一步优化

4.9.1 分支预测选型

前面的章节，我们已经对三种预测器的性能进行了充分的测量以及对他们在 FPGA 上的器件消耗和功耗进行了深入的分析。同时还对他们在 FPGA 上的实际性能进行了分析。从分支预测准确性和功耗上讲，对于 MiniSys 系统 gshare 无疑是最优的。但实际性能的分析和资源的使用却表明 gshare 一定程度上不如 bimode 和 bimodal。然而，我们要充分考虑 FPGA 的互联的特点。FPGA 由于高度的灵活性，其采用的分布式互联结构，延迟和布线存在不确定性。特别是编译器优化时往往不能确定出准确的关键路径，都会导致最终主频最高值的不确定性。最直接的证明就是无预测的流水其实际主频与带预测器的版本相差无几，甚至低于 bimodal 预测器版本。这个是实际流片中不可能发生的。因此，处理器最高实际性能的分析只能作为一个参考。同时，我们注意到，不同版本的处理器其时钟频率的差异都在 5%以内，这说明了各个预测器版本在主频方面并无本质的差异，如果实际流片中优化关键路径，各个版本都能获得很高的主频。同时当三种处理器工作在相同的时钟时，gshare 无疑最有优势。所以，综合全局信息，我们选择 gshare 作为 MiniSys 的最佳预测器。下面我们就来更深入地分析 gshare 的性能。

4.9.2 深入讨论gshare性能

1. QuickSort中不同序列对预测精度的影响

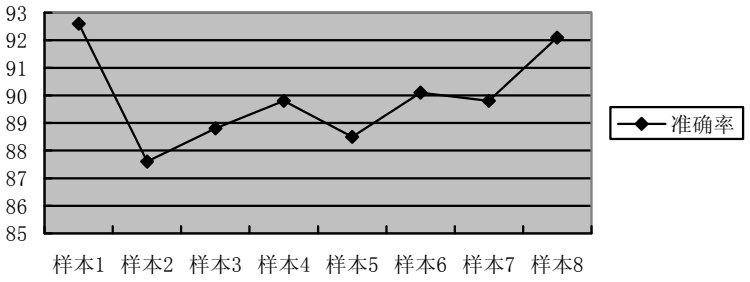


图 4.27 qsort(60)多样本测试曲线

图 4.27 是 qsort(60)多样本测试的曲线。从测试数据看到，随机生成的 8 个长度 60 的序列对精度影响是比较大的，颠簸幅度达到了 5 个百分点。这个也印证了前面介绍 QuickSort 算法时所说的序列对精度的影响。关联预测器需要利用先前的分支信息与当前分支行为之间的关联特性，有些情况下，这种关联是存在的，而且关联度很大，但有些情况下，比如快速排序中，由于要排序的序列是随机的，这样的关联并不一定确定，因而表现出快速排序中预测精度对输入序列比较敏感，这就是为什么图 4.27 的预测准确率出现差异的原因。但从 8 组随机序列的结果看，gshare 平均性能是非常优秀的。

2. BTB大小对预测精度的影响

为了获得最小的预测延迟，IF 段是不对指令译码的，要确定一条指令是不是分支指令，

只能通过 PC 在 BTB 中查找该指令是否已经执行过且在 BTB 中登记过，然后对其进行预测。因此 BTB 大小理论上越大越好，而且其最佳大小与程序中分支数量和程序代码大小成正比。下面我们看一个测试数据，如图 4.28 所示，其中横轴是 BTB 的大小，纵轴是预测准确率。测试程序选择的是 QuickSort,序列长度 60 中的一组。

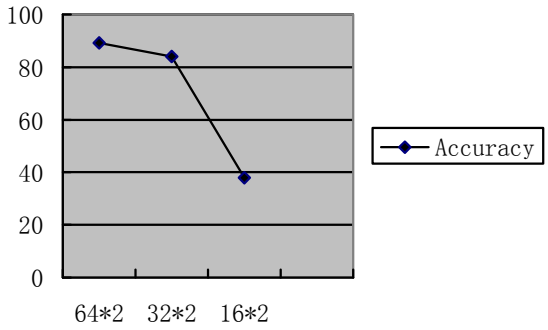


图 4.28 BTB 大小与预测精度的关系

从图 4.28 可以看出，BTB 减小时，预测精度会减小，并且减小到一定值后急剧下降。因此 BTB 应该足够大。根据 MiniSys 的代码空间大小和硬件资源的限制，BTB 设为 64 组 2 路组相联是比较合适的。

3. RAS对精度的影响

RAS 返回地址栈的用途是对函数的返回过程进行精确的预测，在 QuickSort 中有大量的函数递归调用，有 RAS 对预测精度有较大的帮助，我们就 QuickSort,序列长度 60 中的一组的实验数据来说明，如图 4.29 所示。

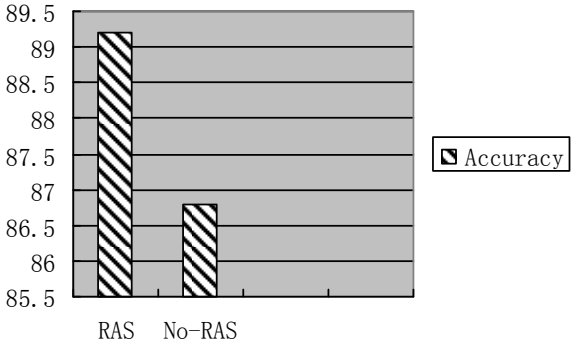


图 4.29 RAS 大小与预测精度的关系

可以看到，RAS 确实可以提高有较多函数调用过程的程序的分支预测准确性。

4. Gshare PHT大小对预测精度的影响.

本参数测试中，使用的是 8 皇后问题和 QuickSort(60)测试程序进行测试。测试输入如图 4.30 所示。

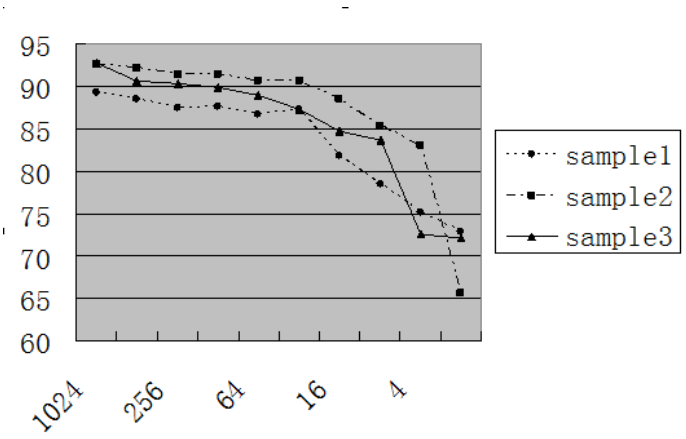


图 4.30 PHT 大小与预测精度的关系

图 4.30 中，横轴是 PHT 的入口数，纵轴是预测准确率（百分比）。从图中可以看到，QuickSort 的两条曲线除了预测精度有所区别外，其精度与 PHT 大小几乎呈现相同的趋势。而 8 皇后问题的曲线则和 QuickSort 的趋势不同。这说明了不同程序其预测精度与 PHT 大小的关系也是不同的，相同程序不同输入的程序其曲线特性是相似的。从测试中我们还发现，程序分支预测精度与 PHT 大小的总体关系是成正关系的，随 PHT 大小的增加，预测精度也增加，但增加到一定程度后，预测精度趋于平缓，再增加 PHT 大小对预测精度的增加没有意义。对 QuickSort(60)而言，其最佳 PHT 大小位于 16-32。8 皇后问题的最佳 PHT 大小在 128-256。对于不同的程序，其最佳 PHT 大小也是不同的。在 MiniSys 应用中，其程序大多是比较小巧的，1024 的 PHT 大小是足够的。曲线中，出现了随 PHT 增大预测准确率出现下降又上升的现象，原因可能是由于分支指令的关联性是非常复杂的，PHT 的增大也导致了 BHSR 长度的增加，当增大到特定长度时，可能导致在这个特定长度的历史信息跟踪上发生相关性下降的情况，一般认为跟踪的历史信息越长，精度越高。正是因为分支行为的复杂性，所以不是越大的 PHT 越好，而是对不同的程序都有其自身的最佳大小。

5. 性能测试总结

通过 3 种类型程序的大量测试，证明了三种分支预测器的有效性。通过大量测试，所有的测试结果得到的预测准确率均高于 2.2.1 节中得出的 69%的要求。gshare 和 bimode 预测器更是达到了 80%以上的预测准确率。RAS 在提高精度方面确实起到了很大的作用。MiniSys 分支预测器结构的正确性与有效性得到了验证。

表 4.4 是 gshare 预测器的最终参数配置。

表 4.4 gshare 预测器的参数配置表

	BTB	PHT/饱和计数器	RAS
gshare	64 组*2 路	1024	10

测试过程中，为了减少仿真时间，加入了 21 级流水除法器和 LED 灯连接，通过下载执行测试数据，效率大幅提高。这也体现出了仿真与硬执行的区别。测试过程非常顺利，测试结果达到了预期。

4.9.3 对预测架构的进一步改进

前面章节我们讨论的分支预测架构属于动态分支预测结构。我们知道，BTB 中存储的

分支是已经被执行过的分支，当这条分支再次执行时，就可能在 BTB 中命中，从而不必到 EXE 段再判断是否分支以及分支方向，而在 IF 阶段就可以进行动态分支预测。前面的架构在整体预测准确性上已经达到了很高的性能。我们来考虑分支第一次执行或分支在 BTB 不命中的情况，这时，分支最终决断要到 2 周期后，指令进入 EXE 阶段后作出，如果分支结果为转移，则会空出 2 段空泡。参考 ARM11 的设计，我们在 ID 段就能够知道本指令的类型，对于 J,Jal 立即型转移，其转移目标为指令的立即数，如果 BTB 不命中，在此我们改变指令流只会损失 1 个时钟周期，代价比 EXE 段决断小 1 个周期。像 JR 指令，由于其转移目标来自寄存器，ID 段转发延迟较大，加上要经过 3 个多路开关，延迟会更大，影响主频。于是，对 JR 指令，我们不在 ID 段进行分支操作,仅当 JR 流入 EXE 才进行分支操作。对于 Bne,Beq 指令，我们来看之前测试的数据中，条件分支的转移情况统计如表 4.5 所示。

表 4.5 条件分支转移情况统计表

	QuickSort1	QuickSort2	8-Queen
条件分支执行数	1832	2681	4245
条件分支转移数	745	1330	2238
转移发生概率	40.7%	49.6%	52.7%

从统计数据来看，条件分支发生转移的可能是在 50%左右，分支方向不确定性很大。对于 bne、beq 在 ID 段对条件分支的预测我们采取这样的措施，看 bne、beq 是向前跳还是向后跳,向后跳很有可能是循环，预测为转移，否则预测为不转移。

于是，我们在 ID 段加入一个新部件，称之为静态分支预测器(static branch predictor)。应当注意到，静态预测器仅在 BTB 命中失败时才进行预测。

我们要对 PC 通路进行适当的改造，其改造如图 4.31 所示:

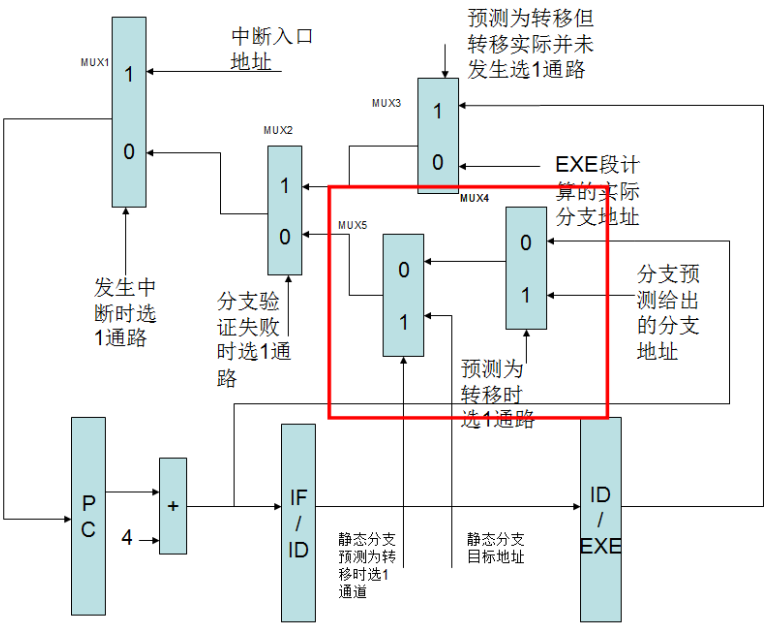


图 4.31 加装静态分支预测器的 PC 通路

图 4.31 中，框中所示的部分就是 PC 通路上的修改，加入了一级多路开关，用途是,一旦 BTB 命中失败，指令进入 ID 段后，静态预测器若预测为转移，则需要直接从 ID 重新获得新的取值 PC。若 BTB 命中成功，则静态预测器不工作。

静态预测器控制器主要根据要进行静态预测的指令类型负责确定在预测为转移时的目

标 PC 的来源，并计算出目标 PC。其输出 PC 通路中静态分支预测的多路开关控制信号 ST_PD 有效的前提是 BTB 不命中，否则 ST_PD 信号不能为有效。若 ST_PD 有效，则说明指令流要进行重定向，需要输出清除 IF/ID 寄存器的清零信号 ST_CLFAR。

还有一个需要修改的地方在于，向 EXE 分支部件传递的预测有效，预测方向和预测地址要进行过滤，因为在 ID 段有两种预测器的结果，预测有效和预测方向信号只需要和 ST_PD 信号进行“或”操作，而预测地址则根据 ST_PD 信号进行选择。

可以看到，静态预测器的加入对原系统的改动非常小，但却可以改善系统的 CPI，使系统的执行效能更高。图 4.32 就是加入静态预测器后的分支预测模块图。

批注 [y2]:

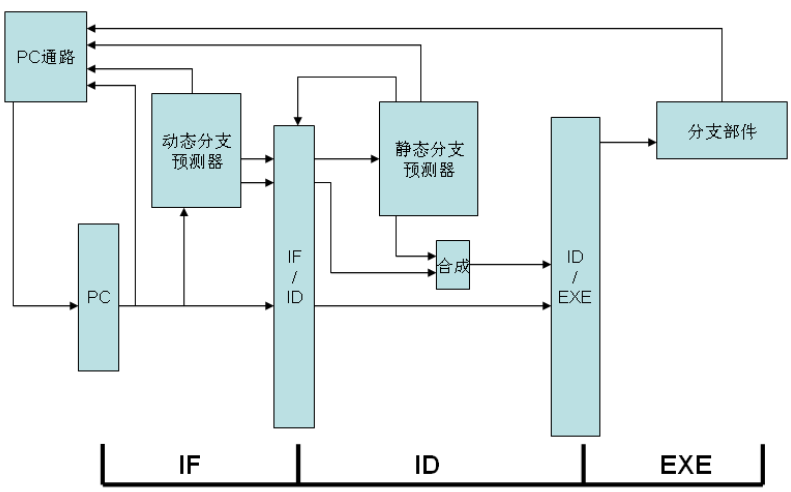


图 4.32 双预测器与系统的结构关系

图 4.32 中我们可以看到 3 大分支部件和 2 个分支预测器的关系，下面我们通过 QuickSort(60)的程序对比测试加入静态分支预测前后的效果。

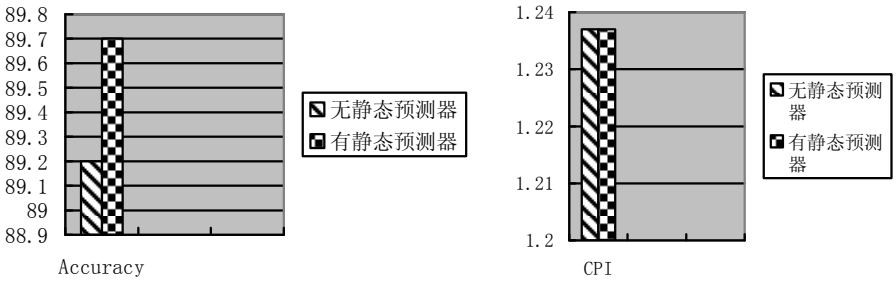


图 4.33 加装静态预测器后的对比测试结果

从图 4.33 看到，在预测准确性上，得益于静态预测器的帮助，准确性提高了 0.5%,而在 CPI 上，小数点后三位以内，已经看不出差异了，原因很简单，由于本程序存在大量的重复执行的分支指令，加上 BTB 大小充足，第一次不命中的分支，在执行后基本都能在以后的执行中命中，加上执行的指令数在 3 万条以上，所以在千分位精度上，看不出 CPI 的提升。下面我们来看更精确的 CPI 的区别。

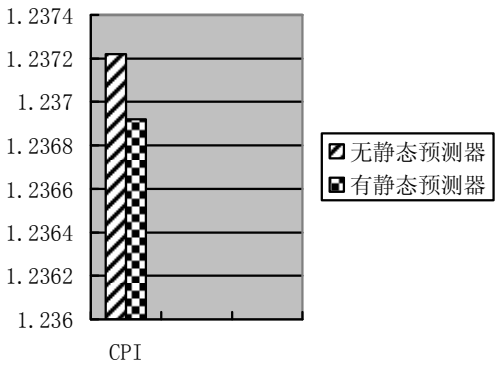


图 4.34 加装静态预测器后对 CPI 的影响

从图 4.34 看到，更高精度的数据表明，静态分支预测确实改善了系统的 CPI 特性，提高了流水线效率。

静态分支预测器在很低的硬件代价上带来了系统性能的提升，改善了流水线的效率，使得系统在相同主频下，有更高的指令吞吐率。

4.10 本章小结

分支预测器是本毕设的重点，是整个毕设的核心。本章通过详细的原理，设计细节和性能测试的描述，证明了所实现的三种预测器的正确性和有效性。而且均超越了开题报告所规定的性能指标。本章通过详细的参数讨论分析了各种因素对预测器的影响，取得了非常好的效果。设计中借鉴了 ARM11 的结构，对预测器结构进行了进一步的优化。本文还通过实际性能的计算，充分证明了加装分支预测器对性能的提升，说明分支预测器是切实有效的。

第 5 章 带分支预测的CPU的下载验证

5.1 最终版本参数配置

用于下载验证的 CPU 共三个版本，分别是不带分支预测，带动态分支预测和带动态和静态预测的版本。其中，动态预测器共有 bimodal,bimode 和 gshare 版本。经过上述参数的讨论，决定动态预测器采用 gshare 作为下载测试版本。其中，BTB 大小配置为 64 组 2 路，动态预测位为 1024*2bits,RAS 返回栈深度 10。

5.2 目标试验台简介

采用博创科技的 FPGA 综合试验台。其中核心板的 FPGA 采用的 EP2C35F672C8,属于 Cyclone II 系列。板上默认时钟 50MHz,在使用时经 FPGA 内分频模块分为 25MHz 使用。采用的板上其他资源为 8 位扫描式数码管和 5×4 扫描键盘，定时计数器由 miniSOC 模块提供。图 5.1, 5.2 即试验台及核心板:

图 5.1 实验台

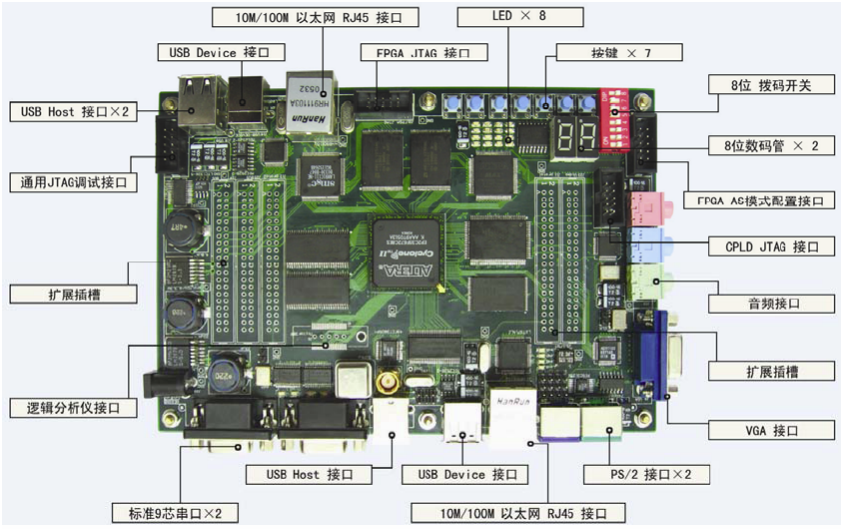


图 5.2 实验台核心板

管脚配置见附录。

5.3 验证程序

5.3.1 时钟键盘中断测试程序

1.测试目的

用于验证系统的下载正确性和中断机制的可靠和稳定性。

2. 程序源代码及说明

```
int a;
void init(void)
{
    a=0;
}
void delay(void)
{
    int cnt;
    cnt=50000;
    while(cnt>0)
    {
        cnt=cnt-1;
    }
}
void main(void)
```

```
{
    init();
    $0xff22=25000000; //向定时器写入计数值
    $0xff20=2; //设定定时器为自动装填模式
    while(1) $0xff00=a; //向数码管写显示值
}
void interuptServer0(void) //中断 0 服务程序
{
    int temp;
    temp=a&0xf0000;
    a=a+1;
    a=a&0xffff;
    a=a|temp;
}
void interuptServer1(void) //中断 1 服务程序
{
    int key;
    key=$0xff10;
    delay();
    key=key<<16;
    a=a&0xffff;
    a=a|key;
}
```

本程序在数码管的低 3 位显示秒表计数值，由秒中断驱动。第 4 位显示键盘按键值，由键盘中断驱动。同时，可以测试中断优先级的单级嵌套，测试方法是按住按键不放，秒表正常计数。

3.下载测试的截图

图 5.3 就是定时键盘下载测试的照片。

图 5.3 定时键盘测试

从测试结果看，系统工作完全正常。键盘和时钟中断工作正常。

4.试验结果

经过下载测试表明，此 SOC 系统在目标板上工作正常，中断响应和中断嵌套正常。IO 系统及 CPU 工作正常。系统正确性得到验证。

5.3.2 九皇后问题

1. 测试目的

这个程序的特点是遍历的 9 皇后问题的解空间很大，执行时间较长，能充分考察分支预测器的性能。通过对比加装分支预测器和未加装分支预测器的 MiniSys 系统执行时间来证明，加装分支预测器后，对于 CPI 有非常直观的改善，性能得到提高。

删除的内容:

2. 九皇后问题的源代码及测试方法

```
int c;
int abs(int n)
{
    if(n<0) return 0-n;
    else return n;
}
int CheckTheChessBoard(int queens[20], int i)
{
    int j;int k;
    j=0;
    while(j <= i )
    {
        k=0;
        while (k <= i)
        {
            if ((j != k)&& ((queens[j] == queens[k]) || ( abs(queens[j] - queens[k])
== abs(j - k))))
            {
                return 0;
            }
            k=k+1;
        }
        j=j+1;
    }

    return 1;
}
void Trial(int queens[20], int i, int n)
{
    int j;
    if (i >= n)
```

```
    {
        c=c+1;
    }
    else
    {
        j=0;
        while(j < n)
        {
            queens[i] = j;
            if(CheckTheChessBoard(queens,i)==1)
            {
                Trial(queens, i + 1, n);
            }
            j=j+1;
        }
    }
}
void main(void)
{
    int i;int queens[9];
    c=0;
    i=0;
    while(i<9)
    {
        queens[i]=-1;
        i=i+1;
    }
    Trial(queens,0,9);
    $0xff00=c;
}
```

这个程序通过遍历解空间找出九皇后问题的全部解的数目，并在最后输出到 LED 显示装置上。当程序下载进入后，LED 等亮起，按下秒表进行计数。因为九皇后问题的解个数为 352，换算成 16 进制为 160H，当运算结束后，LED 立即显示 160，此时按下秒表停止键停止计数，得到运算时间的粗略值。考虑到人的反应到动作的时间一般认为是小于 0.2s，因此，只要对比样本的时间差在 0.5s 以上认为是对比样本是有明显差别的。

3.测试结果及讨论

对比的处理器分别是不带分支预测的版本和带动态加静态预测的加强版本。均采用 50MHz 进行 2 分频作为处理器系统时钟。

测试数据如表 5.1 所示。

表 5.1 9 皇后问题运行时间对照表

九皇后问题测试运行时间(s)	
不带分支预测	18.51

带动态+静态	16.57

从测试时间上看，加装分支预测的版本比不带分支预测器的版本在本程序测试中提高了10.5%的性能。下面我们进行更深入的讨论。通过加装专用计数器装置，下载运行得到测试结果，如表 5.2 所示。

表 5.2 9 皇后问题 CPI 与预测准确率对照表

执行开始的 8191 个周期的 CPI 和预测准确率		
	CPI	分支准确率
带动态+静态	1.251	81.4%
不带分支预测	1.354	28.0%

还有个重要的测试数据，执行开始后当执行到 8191 个分支时的预测准确率为 92.5%。下面我们来估计参数，以不带分支预测的参数为基准，取 CPI=1.354,运行时间=18.51,带分支预测的 CPI=1.251,则估算带分支预测的运行时间为 18.51/1.354*1.251=17.1s,与实际测试值 16.57 偏差 3.2%,其原因在于，随着运行时间的增加，动态预测器能够进行更充分的训练，从而使预测精度进一步提高，CPI 数值进一步下降，因此流水线效率进一步提升，从而执行时间低于估计时间。

接下来我们估算一下带分支预测的处理器的 CPI,我们以不带分支预测的数据和分支预测版本的执行时间为基准，估计带分支预测版本的最终 CPI=1.354*16.57/18.51=1.212。比周期 8191 时的 1.251 提高不少。这个还说明了，关联预测器适合于体积较大，执行时间较长的程序，以使预测器有充分的学习时间。整个测试过程非常顺利，进一步证明了系统的有效性和正确性。

5.3.3 简单计算器程序

1. 测试目的

进一步证明系统的正确性以及系统的实用性。考察编译器对于较大的程序编译的正确性。

带格式的: 字体: 五号

删除的内容:

2. 程序代码及其说明

```
int a;
int b;
int op1;
int op2;
int op;
int disp;
int stat;
int cnt;

void init(void)
{
    a=0;b=0;op=0;
    disp=0;stat=0;
```

```
    op1=0;op2=0;
    cnt=0;
}
int cal(int a,int b,int op)
{
    if(op==10) return a+b;
    if(op==11) return a-b;
    if(op==12) return a*b;
    if((op==13)&&(b!=0)) return a/b;
    else return 0;
}

int div(int a,int b)
{
    return a/b;
}

int mult(int a,int b)
{
    return a*b;
}

void delay(void)
{
    int a;
    a=300000;
    while(a>0)
    {
        a=a-1;
    }
}

int HexToDec(int n)
{
    int a0;int a1;int a2;int a3;
    int tmp1;
    a3=div(n,1000);
    tmp1=n-mult(a3,1000);
    a2=div(tmp1,100);
    tmp1=tmp1-mult(a2,100);
    a1=div(tmp1,10);
    a0=tmp1-mult(a1,10);
    tmp1=0;tmp1=(a3<<12)|(a2<<8)|(a1<<4)|a0;
    return tmp1;
```

```
}

void interuptServer1(void)
{
    int key;
    key=$0xff10;
    delay();
    if(key==15)
    {
        init();
    }
    if(stat==0)
    {
        if(key<10)
        {
            if(cnt<4)
            {
                cnt=cnt+1;
                key=key&0x0f;
                a=(a<<4)|key;
                disp=a;
                op1=key+mult(op1,10);
            }
        }
        if((key>9)&&(key<14))
        {
            if(cnt>0)
            {
                op=key;
                stat=1;
                cnt=0;
            }
        }
    }
    else
    {
        if(stat==1)
        {
            if(key<10)
            {
                if(cnt<4)
                {
                    cnt=cnt+1;
                    key=key&0x0f;
```

```
                    b=(b<<4)|key;
                    disp=b;
                    op2=key+mult(op2,10);
                }
            }
            if(key==0x10)
            {
                disp=HexToDec(cal(op1,op2,op));
                stat=0;op1=0;op2=0;a=0;b=0;
                cnt=0;
            }
        }
    }
}

void main(void)
{
    init();
    while(1)
    {
        $0xff00=disp;
    }
}
```

这是一个对于 MiniSys 系统来说较庞大的程序，使用前面提及的键盘和 LED 两大 IO 外设的一个综合例子。本实例可以计算两个数的加减乘除，鉴于 LED 灯对负数还没支持，目前只演示正数的情况。ENT 键输出结果，NUM 键清 0。程序核心是键盘中断，内部有一个状态机。

3.测试结果及截图

如图 5.4 是一个简单计算器程序的下载测试照片。

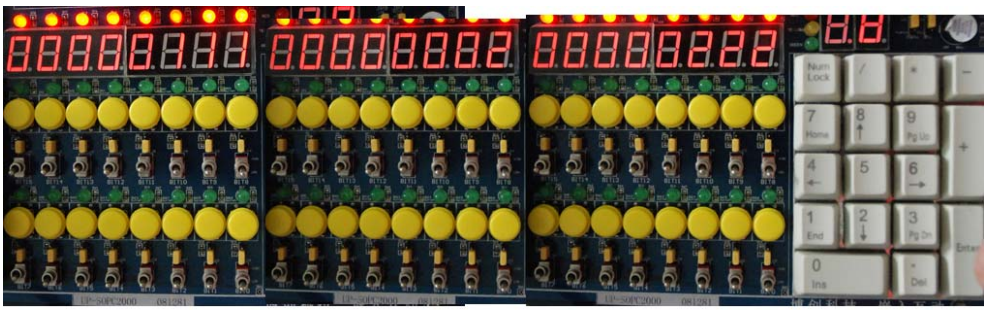


图 5.4 简单计算器 111×2=222

通过键盘输入 111×2 得到结果 222，结果正确。

5.4 本章小结

通过本章和前面分支预测章节的测试，证明了本系统的分支预测确实提高了系统的性能，同时也证明了系统的正确性和稳定性，而且证明了 MiniSys 具有一定的实用性。本系统在设计上是正确的和高效的。

第 6 章 总结

为了提高 MiniSys 流水线的性能，改善系统的 CPI 特性，在本次毕业设计中特别设计和实现了适合于 MiniSys 的分支预测部件。在设计过程中，通过查阅相关的资料，参考 ARM11 的分支预测机制，设计了几种适合于嵌入式系统的分支预测框架和预测器，并通过在 MiniSys 的流水处理器上进行加装测试，证明了加装分支预测器对于提高系统的性能确实有非常大的帮助。此外，通过扩大存储器容量和设计相关使用程序，进一步证明了 MiniSys 的实用特性。下面就是本设计的特色:

- 将分支部件后移至执行段，提高主频。同时，改进了 PC 通道，可以方便加装各种预测器。
- 动态分支预测器部分使用通用框架，并给出条件预测器的通用接口，可以最大程度上实现重用。本设计的三种条件预测器的框架部分完全相同。
- 使用 2 路组相联 BTB,且在 BTB 中引入置换位，实现了隐式删除，以及优化了置换策略，提高了 BTB 的效率。同时增加了 TYPE 字段标识分支类型，实现了不同分支采取不同预测方式，提高了预测率。
- 加装了 RAS 返回地址栈，针对 mini-C 编译器的子程序调用过程的优化。
- 拥有 mini-C 编译器的支持，且函数调用过程与预测器紧密结合。(对汇编程序员来说，鉴于 MiniSys 无专用函数返回指令，需要将 JR 只用于函数返回才能发挥 RAS 功效)
- 寄存器组和相关高速存储部件使用了三态总线挂载方式，减少了数据输出的多重门延迟。
- 与 ARM11 相似的预测机制，且采用了比 ARM11 更先进的条件分支预测器和更大的 RAS。

本课程设计从全局上讲是非常成功和有意义的。

删除的内容: 上将

致谢

感谢我的指导老师杨全胜。本论文的选题、工作、撰写都是在他悉心的指导下完成的。他丰富的学术知识、严谨的治学态度是我学习的榜样。也要感谢他为我们的毕业设计提供的良好的硬件条件是我顺利完成的前题。

参考文献

[1]现代处理器设计-超标量处理器基础/(美)沈(Shen,J.P)著;张承义等译 2004.05

[2] John L.Hennessy , David A.Patterson,《Computer Architecture¬——A Quantitative Approach》, China Machine Press,2007

[3] Dominic Sweetman ,《See MIPS Run Linux》 China Machine Press , 2007

[4]冯子军 肖俊华 章隆兵. 处理器分支预测研究的历史和现状. ftp://202.120.1.155/deng-qn/Advanced Computer Architecture/处理器分支预测研究的历史和现状.pdf

[5]黄 伟，王玉艳，章建雄,嵌入式处理器动态分支预测机制研究与设计 ,2008.11

[6]Chitaka Iwama, Niko Demus Barli, Shuichi Sakai, Hidehiko Tanaka ,Improving Conditional Branch Prediction on Speculative Multithreading Architectures , Joho Shori Gakkai Shinpojiumu Ronbunshu, VOL.2001;NO.6;PAGE.165-172(2001)

[7] Iain Bate and Ralf Reutemann ,Efficient Integration of Bimodal Branch Prediction and Pipeline Analysis , Embedded and Real-Time Computing Systems and Applications, 2005. Proceedings. 11th IEEE International Conference on,2005.08

[8] Chih-Cheng Cheng ,The Schemes and Performances of Dynamic Branch predictors, http://bwrc.eecs.berkeley.edu/Classes/CS252/Projects/Reports/terry_chen.pdf

[9]武萌 沈海斌. 一种 gshare 分支预测器的低功耗设计方法 , <<微电子学与计算机>>2007 年 第 24 卷 第 03 期

[10]王晶 樊晓桠 叶曾. 一种基于综合历史信息的 SMT 结构分支预测算法. 计算机科学. Vol52, No.2,2008

[11]陈跃跃, 周兴铭. 一种精确的分支预测微处理器模型. 计算机研究与发展. Vol40,No.5,2003

[12]杨全胜 ,计算机系统综合课程设计 ,清华大学出版社 ,2008.10

[13]Daniel A. Jiménez, Piecewise Linear Branch Prediction, Proceedings of the 32nd International Symposium on Computer Architecture (ISCA-32), June 2005

[14]Daniel A. Jiménez and Calvin Lin, Neural Methods for Dynamic Branch Prediction, ACM Transactions on Computer Systems, Vol. 20, No. 4, November 2002

[15]ARM11 MPCore Processor Technical Reference Manual ARM Limited http://infocenter.arm.com/help/topic/com.arm.doc.ddi0360e/DDI0360E_arm11_mpcore_r1p0_trm.pdf

[16]Kenji Kise , Takahiro Katagiri , Hiroki Honda , Toshitsugu Yuba The Bimode++ Branch Predictor , IEEE ,2005

删除的内容:

... [3]

删除的内容:【1】

带格式的 ... [4]

删除的内容:【2】

删除的内容:【3】

删除的内容:【4

删除的内容:】

带格式的 ... [5]

删除的内容:【5】

带格式的 ... [6]

删除的内容:【6】

删除的内容:【7】

删除的内容:【8】

删除的内容:【9】

删除的内容:【10】

删除的内容:【11】

带格式的 ... [7]

删除的内容:【12】

带格式的 ... [8]

删除的内容:【13】

删除的内容:【14】

删除的内容:【15】

带格式的 ... [9]

删除的内容:【16】

带格式的 ... [10]

附录一

实验板管脚分配

```
################################7 段数码管#####################################
set_location_assignment PIN_J24 -to seg_D[0]
set_location_assignment PIN_W17 -to seg_D[1]
set_location_assignment PIN_J8 -to seg_D[2]
set_location_assignment PIN_J9 -to seg_D[3]
set_location_assignment PIN_H8 -to seg_D[4]
set_location_assignment PIN_G6 -to seg_D[5]
set_location_assignment PIN_F7 -to seg_D[6]
set_location_assignment PIN_L4 -to seg_D[7]

set_location_assignment PIN_V18 -to DIG[0]
set_location_assignment PIN_G12 -to DIG[1]
set_location_assignment PIN_J2 -to DIG[2]
set_location_assignment PIN_H1 -to DIG[3]
set_location_assignment PIN_K2 -to DIG[4]
set_location_assignment PIN_J1 -to DIG[5]
set_location_assignment PIN_F6 -to DIG[6]
set_location_assignment PIN_K3 -to DIG[7]
################################KeyBoard#####################################

set_location_assignment PIN_J7 -to KEYI[0]
set_location_assignment PIN_K8 -to KEYI[1]
set_location_assignment PIN_J5 -to KEYI[2]
set_location_assignment PIN_J10 -to KEYI[3]
set_location_assignment PIN_G9 -to KEYI[4]

set_location_assignment PIN_K6 -to KEYO[0]
set_location_assignment PIN_K9 -to KEYO[1]
set_location_assignment PIN_H12 -to KEYO[2]
set_location_assignment PIN_G5 -to KEYO[3]
```

附录二

Mini-C 编译器的词法和语法规则(MyLex&MyYacc 描述)
词法规则(Lex 描述)

```
digit ([0-9])|([1-9][0-9]*)
letter [a-zA-Z]
id {letter}({letter}|{digit})*
num {digit}[1-9]*
hex 0(x|X)({letter}|[0-9])*
%%
"void"                {return VOID;}
"continue"            {return CONTINUE;}
"if"          {return IF; }
"while"           {return WHILE;}
"else"            {return ELSE;}
"break"               {return BREAK;}
"int"         {return INT;}
"return"              {return RETURN;}
"\|\|"                {return OR;}
"&&"                      {return AND;}
{id}              {yyval.IDENT_v.ID_NAME=yytext;return IDENT;}
{hex}             {yyval.int_literal_v.int_val=ConvertHexToint(yytext);return HEXNUM;}
{num}             {yyval.int_literal_v.int_val=atoi(yytext);return DECNUM;}
"\<="                     {return LE;}
"\>="                     {return GE;}
"\=\="                    {return EQ;}
"\!\="                    {return NE;}
"\>"                              {return '>';}
"\<"                              {return '<';}
"\,"                  {return ','; }
"\;"              {return ';'; }
"\{"              {return '{'; }
"\}"              {return '}'; }
"\%"              {return '%'; }
"\*"              {return    '*'; }
"\+"              {return '+'; }
"\-"              {return '-'; }
"/"               {return '/'; }
"\="                {return '='; }
"\("              {return '('; }
"\)"              {return ')'; }
```

```
"~"                     {return '~';}
"&"                     {return '&';}
"^"                     {return '^';}
"\["                    {return '[';}
"\]"                    {return ']';}
"\<\<"                  {return LSHIFT;}
"\>\>"                  {return RSHIFT;}
"\|"                    {return '|';}
\t|\    { }
\n|\r\n       {Lineno++;yylineno++;}
"$"                     {return '$';}
%%
语法规则(Yacc 描述)
%token IDENT VOID INT WHILE IF ELSE RETURN EQ NE LE GE AND OR DECNUM
CONTINUE BREAK HEXNUM LSHIFT RSHIFT
%left OR
%left AND
%left EQ NE LE GE '<' '>'    /*关系运算*/
%left '+' '-'
%left '|'
%left '&' '^'
%left '*' '/' '%'    /*算术运算*/
%right LSHIFT RSHIFT
%right '!'
%right '~'
%nonassoc UMINUS
%nonassoc MPR
%start program
%%
program    : decl_list    ; /*程序由变量描述或函数描述组成（decl）*/
decl_list    : decl_list decl | decl ;
decl    : var_decl | fun_decl ;
var_decl    : type_spec IDENT ';' | type_spec IDENT '[' int_literal ']' ';' ; /*变量包括简单变量和一
维数组变量*/
type_spec    : VOID | INT ; /*函数返回值类型或变量类型包括整型或 VOID*/
fun_decl      : type_spec FUNCTION_IDENT '(' params ')' compound_stmt
          | type_spec FUNCTION_IDENT '(' params ')' ';' ; //要考虑设置全局函数信息为假，和函
数表为申明
FUNCTION_IDENT : IDENT {cout<<"function ident"<<endl;} ; /*建立全局函数名变量*/
params      :    param_list | VOID ; /*函数参数个数可为 0 或多个*/
param_list    : param_list ',' param | param ;
param       :    type_spec IDENT | type_spec IDENT '[' int_literal ']' ;
stmt_list      : stmt_list stmt | ;
stmt       :     expr_stmt  |  block_stmt  |  if_stmt  |  while_stmt  |return_stmt  |  continue_stmt  |
```

删除的内容:
%}

```
break_stmt   ;
expr_stmt   : IDENT '=' expr ';' | IDENT '[' expr ']' '=' expr ';' | '$' expr '=' expr ';' | IDENT '(' args ')'
';' ; /*赋值语句*/
while_stmt : WHILE_IDENT '(' expr ')' stmt ; /*WHILE 语句*/
WHILE_IDENT : WHILE {cout<<"while ident"<<endl;} ;/*建立入口出口信息全局变量*/
block_stmt : '{' stmt_list '}'   ; /*语句块*/
compound_stmt : '{' local_decls stmt_list '}'   ; /*函数内部描述，包括局部变量和语句描述*/
local_decls   : local_decls local_decl | ; /*函数内部变量描述*/
local_decl    : type_spec IDENT ';' | type_spec IDENT '[' int_literal ']' ';' ;
if_stmt   :   IF '(' expr ')' stmt %prec UMINUS | IF '(' expr ')'   stmt ELSE stmt %prec MPR ;
return_stmt   : RETURN ';' | RETURN expr ';' ;
expr   :   expr OR expr   /*逻辑或表达式，运算符为'||'*/
  | expr EQ expr | expr NE expr /*关系表达式*/
  | expr LE expr | expr '<' expr | expr GE expr | expr '>' expr /*关系表达式*/
  | expr AND expr /*逻辑与表达式，运算符为’ &&’ **/
  | expr '+' expr | expr '-' expr /*算术表达式*/
  | expr '*' expr | expr '/' expr | expr '%' expr   /*算术表达式*/
  | '!' expr %prec UMINUS | '-' expr %prec UMINUS | '+' expr %prec UMINUS | '$' expr %prec
UMINUS   /*$ expr 为取端口地址为 expr 值的端口值*/
  | '(' expr ')'
  | IDENT | IDENT '[' expr ']' | IDENT '(' args ')'   /* IDENT ( args )为函数调用*/
  | int_literal   /*数值常量*/
  | expr '&' expr /*按位与*/
  | expr '^' expr /*按位异或*/
  | '~' expr   /*按位取反*/
  | expr LSHIFT expr /*逻辑左移*/
  | expr RSHIFT expr /*逻辑右移*/
  | expr '|' expr /*按位或*/
    ;
int_literal   :   DECNUM
                |   HEXNUM ; /*数值常量是十进制整数*/
arg_list    :   arg_list ',' expr | expr ;
args    :    arg_list | ;
continue_stmt : CONTINUE ';' ;
break_stmt : BREAK ';' ;
%%
```

注:本程序的词法和语法分析器由我们自己设计的 SeuLex+SeuYacc 工具生成，详细细节请参考 SeuLex 和 SeuYacc 的设计文档。

页 4: [1] 删除的内容 Justfly 2009-6-8 16:05:00

图 2.1 是经过改进后的 MinSys 流水线结构简图，与原来的流水线相比，改进的流水线最大的不同就是将分支部件从 ID 段移到了 EXE 段。这样做的原因在于：对于 BEQ, BNE 指令，由于转移条件和两个寄存器内容相关，最坏的情况下，旧版系统需要从其他部分转发寄存器内容，加上译码的延迟，分支方向的产生过程包含了很大的延迟，而后移转移部件极大缓解了延迟问题，主频进一步提升。但这样也会带来一个问题，一旦分支发生转移，会造成 2 个时钟的流水停顿（2 段空泡），原版系统只会有 1 个时钟（1 段空泡）。这样会造成性能的损失。但另一方面，引入分支预测机制后，不仅主频提高，而且也减少了分支转移带来的流水停顿的机会。比如经过测试，无分支预测的分支转移率为 62%左右，这些分之转移都会引起流水线的停顿，因此无分支预测的系统每分支空泡数为 1*0.62=0.62。而加上分支预测部件后，只有在分支失效的时候才会引起流水线停顿，如果要使系统的每分支空泡数低于原系统，则分支预测失效率必须小于 0.62/2=0.31，即 31%。换句话说，从减少系统分支空泡率的角度说，要想让流水线系统性能提高，分支预测的准确率至少大于 69%，考虑到其他因素，这个比率可能会稍高一点。

页 12: [2] 删除的内容 User 2009-6-8 18:20:00

4.1 分支预测的意义

流水型处理器只有在流水模式下才能达到最大的吞吐率。流水线处理器在取指时通常情况下都是按顺序取指的，而分支指令会造成控制流背离顺序的方向，使流水线停顿，只有等待新的地址产生后才能恢复正常的取指，这样一来就会造成流水线产生空的指令槽（空泡），严重影响流水线处理器的性能。分支预测技术就是在指令流发生变化前，预测下一条指令的入口，从而避免流水线中错误的指令流形成的空泡，提高流水线的效率。下面以一个简单例子说明分支预测的意义:

首先，我们看一个简单的程序片段

```
this:
ori $2,$0,10-----a
ori $3,$0,10-----b
beq $2,$3,this--c
slti $s4,$5,11---d
jal fun----------e
```

图 4.1 程序片段

为简单起见，我们用 a-e 标识每条指令，如图 4.1 所示。可以看到，本程序片段的分支指令为 c 和 e,而且由于 c 的转移条件成立，所以运行到 c 肯定发生转移。下面我我们来看无分支预测和有分支预测（且预测成功）的时序图对比:

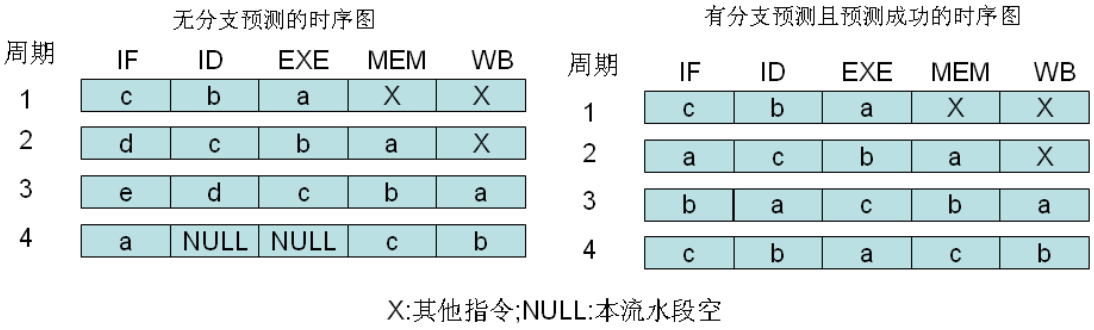


图 4.2 有分支预测（且成功）与无分支预测的时序对比

可以看到，在第一周期，C 指令已经处于 IF 取指段，而动态分支预测就发生在 IF 段。如果分支预测成功，下一周期将取得正确的下一条指令 a，而无分支预测的版本只能顺序取下一条指令 d。在第二周期，带分支预测的处理器取到了正确的转移目标地址的指令，而非预测版本取了错误的指令 d.在第三周期，C 进入了 EXE 段，将进行分支验证与指令重定向，由于分支预测版本成功预测了下条指令，因此不会发生转移重定向。而非预测版本取到了错误的指令，需要重定向指令流。在第四周期，预测的版本继续正常执行，而非预测的版本由于需要清除错误取到的指令而使得流水线出现了两段空泡（或流水线发生了两周期停顿），造成了流水线性能的下降。这个例子充分说明了分支预测的实际意义。

页 56: [3] 删除的内容	Justfly	2009-6-8 16:45:00

页 56: [4] 带格式的	Justfly	2009-6-8 16:45:00

缩进: 左侧: 0 磅, 悬挂缩进: 0.85 字符, 首行缩进: -0.85 字符

页 56: [5] 带格式的	Justfly	2009-6-8 16:45:00

缩进: 左侧: 0 磅, 悬挂缩进: 0.85 字符, 首行缩进: -0.85 字符, 不调整西文与中文之间的空格

页 56: [6] 带格式的	Justfly	2009-6-8 16:45:00

缩进: 左侧: 0 磅, 悬挂缩进: 0.85 字符, 首行缩进: -0.85 字符

页 56: [7] 带格式的	Justfly	2009-6-8 16:45:00

缩进: 左侧: 0 磅, 悬挂缩进: 0.85 字符, 首行缩进: -0.85 字符, 不调整西文与中文之间的空格

页 56: [8] 带格式的	Justfly	2009-6-8 16:45:00

缩进: 左侧: 0 磅, 悬挂缩进: 0.85 字符, 首行缩进: -0.85 字符

页 56: [9] 带格式的	Justfly	2009-6-8 16:45:00

缩进: 左侧: 0 磅, 悬挂缩进: 0.85 字符, 首行缩进: -0.85 字符, 不调整西文与中文之间的空格

页 56: [10] 带格式的	Justfly	2009-6-8 16:45:00

缩进: 左侧: 0 磅, 悬挂缩进: 0.85 字符, 首行缩进: -0.85 字符