**深 圳 大 学 实 验 报 告**

**课 程 名 称： 计算机系统(3)**

**实验项目名称： 处理器结构实验二**

**学 院： 计算机与软件学院**

**专 业： 计算机科学与技术/软件工程**

**指 导 教 师： 王 毅**

**报告人： 郑雨婷 学号： 2021150122 班级： 高性能**

**实 验 时 间： 2023年11月16日**

**实验报告提交时间： 2023年11月30日**

**教务处制**

## 一、实验目的

了解控制冒险分支预测的概念

了解多种分支预测的方法，动态分支预测更要深入了解

理解什么是BTB（Branch Target Buffer），并且学会用BTB来优化所给程序

利用BTB的特点，设计并了解在哪种状态下BTB无效

了解循环展开，并于BTB功能进行对比

对WinMIPS64的各个窗口和操作更加熟悉

## 二、实验内容

按照下面的实验步骤及说明，完成相关操作记录实验过程的截图：

首先，给出一段矩阵乘法的代码，通过开启BTB功能对其进行优化，并且观察流水线的细节，解释BTB在其中所起的作用；

其次，自行设计一段使得即使开启了BTB也无效的代码。

第三，使用循环展开的方法，观察流水因分支停顿的次数减少的现象，并对比采用BTB结构时流水因分支而停顿的次数。

（选做：在x86系统上编写C语言的矩阵乘法代码，用perf观察分支预测失败次数，分析其次数是否与你所学知识吻合。再编写前面第二部使用的令分支预测失败的代码，验证x86是否能正确预测，并尝试做解释）

## 三、实验环境

硬件：桌面PC

软件：Windows

## 四、实验步骤及说明

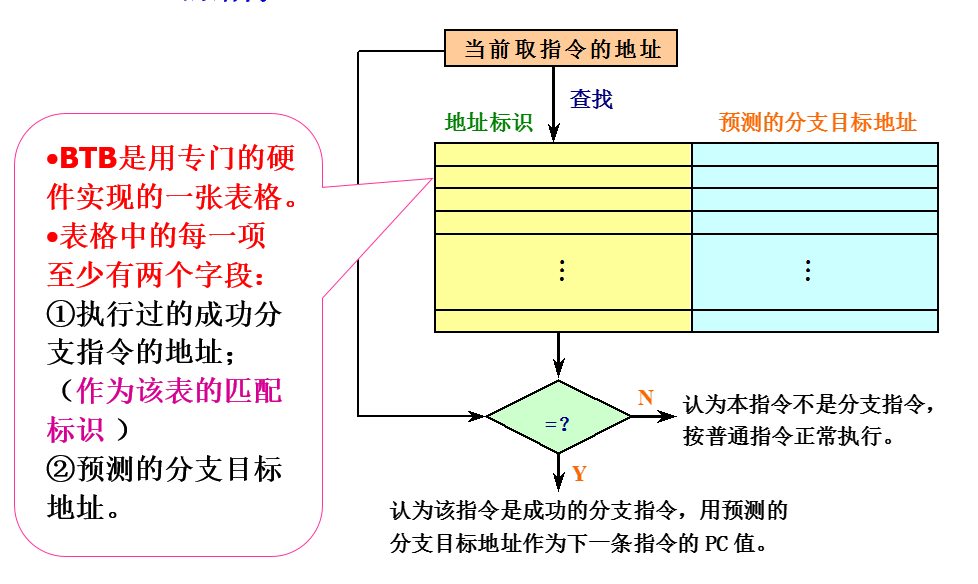
背景知识

在遇到跳转语句的时候，我们往往需要等到MEM阶段才能确定这条指令是否跳转（通过硬件的优化，可以极大的缩短分支的延迟，将分支执行提前到ID阶段，这样就能够将分支预测错误代价减小到只有一条指令），这种为了确保预取正确指令而导致的延迟叫控制冒险（分支冒险）。

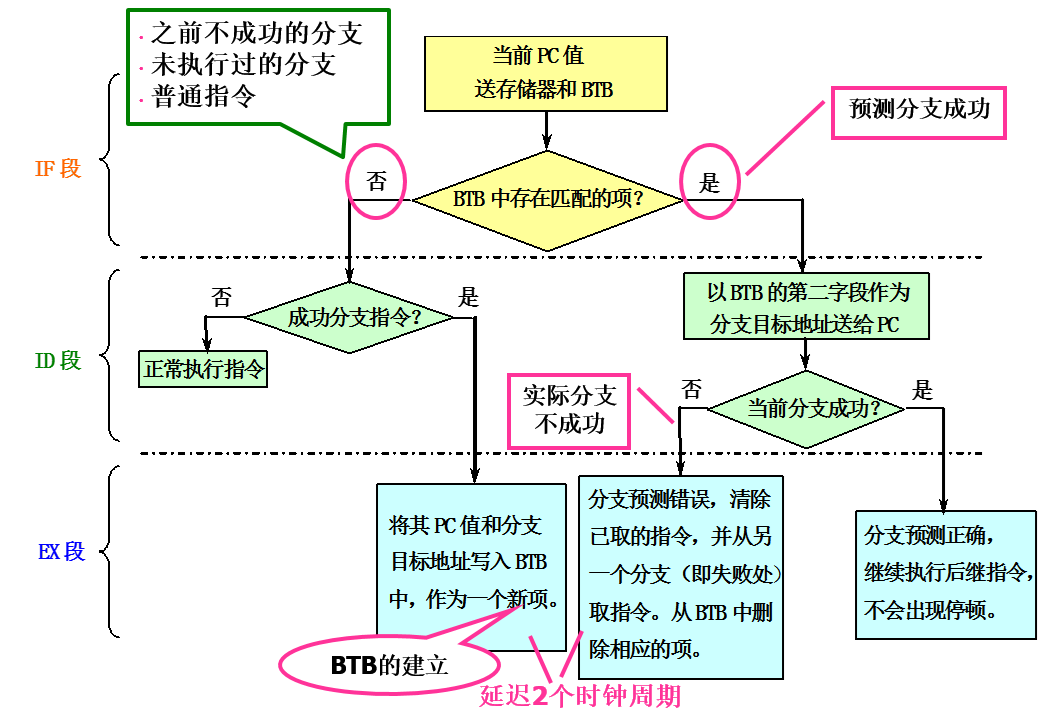
为了降低控制冒险所带来的性能损失，一般采用分支预测技术。分支预测技术包含编译时进行的静态分支预测，和执行时进行的动态分支预测。这里，我们着重介绍动态分支预测中的BTB（Branch Target Buffer）技术。

BTB即为分支目标缓冲器，它将分支指令（对应的指令地址）放到一个缓冲区中保存起来，当下次再遇到相同的指令（跳转判定）时，它将执行和上次一样的跳转（分支或不分支）预测。

一种可行的BTB结构示意图如下：



在采用了BTB之后，在流水线各个阶段所进行的相关操作如下：



注意，为了填写BTB，需要额外一个周期。

1. 矩阵乘法及优化

在这一阶段，我们首先给出矩阵乘法的例子，接着将流水线设置为不带BTB功能（configure->enable branch target buffer）直接运行，观察结果进行记录；然后，再开启BTB功能再次运行，观察实验结果。将两次的实验结果进行对比，观察BTB是否起作用，如果有效果则进一步观察流水线执行细节并且解释BTB起作用原因。

矩阵乘法的代码如下：

.data

str: .asciiz "the data of matrix 3:\n"

mx1: .space 512

mx2: .space 512

mx3: .space 512

.text

initial: daddi r22,r0,mx1 #这个initial模块是给三个矩阵赋初值

daddi r23,r0,mx2

daddi r21,r0,mx3

input: daddi r9,r0,64

daddi r8,r0,0

loop1: dsll r11,r8,3

dadd r10,r11,r22

dadd r11,r11,r23

daddi r12,r0,2

daddi r13,r0,3

sd r12,0(r10)

sd r13,0(r11)

daddi r8,r8,1

slt r10,r8,r9

bne r10,r0,loop1

mul: daddi r16,r0,8

daddi r17,r0,0

loop2: daddi r18,r0,0 #这个循环是执行for(int i = 0, i < 8; i++)的内容

loop3: daddi r19,r0,0 #这个循环是执行for(int j = 0, j < 8; j++)的内容

daddi r20,r0,0 #r20存储在计算result[i][j]过程中每个乘法结果的叠加值

loop4: dsll r8,r17,6 #这个循环的执行计算每个result[i][j]

dsll r9,r19,3

dadd r8,r8,r9

dadd r8,r8,r22

ld r10,0(r8) #取mx1[i][k]的值

dsll r8,r19,6

dsll r9,r18,3

dadd r8,r8,r9

dadd r8,r8,r23

ld r11,0(r8) #取mx2[k][j]的值

dmul r13,r10,r11 #mx1[i][k]与mx2[k][j]相乘

dadd r20,r20,r13 #中间结果累加

daddi r19,r19,1

slt r8,r19,r16

bne r8,r0,loop4

dsll r8,r17,6

dsll r9,r18,3

dadd r8,r8,r9

dadd r8,r8,r21 #计算result[i][j]的位置

sd r20,0(r8) #将结果存入result[i][j]中

daddi r18,r18,1

slt r8,r18,r16

bne r8,r0,loop3

daddi r17,r17,1

slt r8,r17,r16

bne r8,r0,loop2

halt

不设置BTB功能，运行该程序，观察Statistics窗口的结果截屏并记录下来，如图1所示。

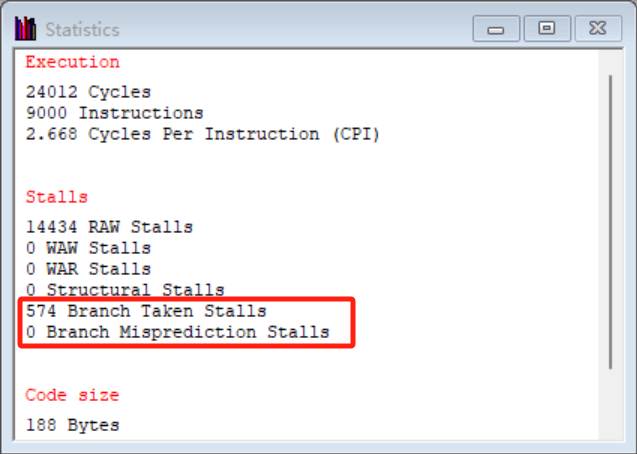


图1 未开启BTB

接着，设置BTB功能（在菜单栏处选择Configure项，然后在下拉菜单中为Enable Branch Target Buffer选项划上钩，如图2所示）。并在此运行程序，观察Statistics窗口的结果并截屏记录下来，如题3所示。

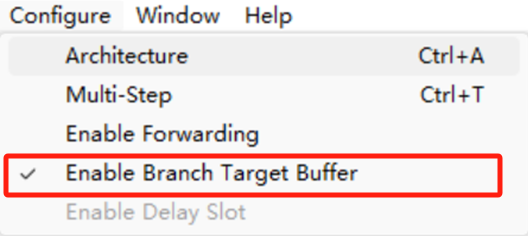


图2 开启BTB

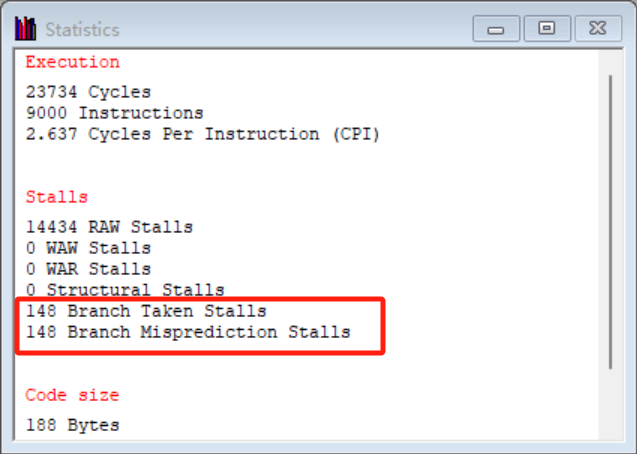


图3 开启BTB

在这里，我们仅仅观察比较Stalls中的最后两项------Branch Taken Stalls和Branch Misprediction Stalls。在未开启BTB时，Branch Taken Stalls是574次，Branch Misprediction Stalls是0次，开启BTB后的Branch Taken Stalls和Branch Misprediction Stalls都是148次。

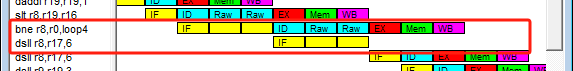
接下来，对比其结果。我们就结合流水线执行细节分析造成这种情况发生的原因。

在MIPS指令集中，bne指令用于判断两个寄存器的值是否不相等，如果不相等，则进行跳转。决策是否要跳转的阶段是在ID阶段因此，要阻塞一个时钟周期才可以得到要跳转到哪一条指令。

在没有开启BTB技术来进行动态分支预测时，需要等待bne指令作出决策之后，才知道需要哪一条指令，才可以去相应的地址进行取指。因此会阻塞一个时钟周期产生Branch Taken Stalls，如图4所示。因为没有进行预测，所以Branch Misprediction Stalls为0。

对于本代码来说，开始的输入矩阵是一个 64 次的循环，因此会产生 63 次Branch Taken Stalls。而在矩阵乘法的 8\*8\*8 的循环中，最内层计算 result[i][j]的部分，每次循环结束会产生 7 次 Branch Taken Stalls，而他会重复 64 次，因此会产生 64\*7=448次 Branch Taken Stalls。而中间层 j++的循环，同样每次循环结束会产生 7 次 Branch Taken Stalls。而它会重复 8 次，因此会产生 8\*7=56 次 Branch Taken Stalls。而最外一层 i++的循环，每次循环会产生 7 次 Branch Taken Stalls，而它只会重复 1 次，因此会产生 7 次Branch Taken Stalls 。 所 以 ， 当 不 开 启 BTB 代 码 时 ， 本 代 码 运 行 完 会 产 生63+448+56+7=574 次 Branch Taken Stalls。因此 Branch Taken Stall 产生的阻塞周期数为574。

图4 未开启BTB产生的阻塞



在开启BTB技术来进行动态分支预测后，在取指令阶段如果发现BTB中存在匹配的项，说明这在执行的这条指令是分支跳转指令并且之前已经记录过它的上一次跳转的地址。这样，在ID阶段还未结束没有做出决策时，流水线先按照预测进行下一条指令的IF、ID阶段等。若预测成功，则避免了Branch Taken Stall；若预测失败，重新执行正确的指令。如图5所示。在矩阵乘法的程序中，因为只有预测失败时，才会产生流水线阻塞。

当在循环开始时，第一次执行分支指令，会默认地按顺序取指令，因此此时要建立 BTB，就会产生 1 次 Branch Taken Stalls。而在循环结束时，此时已经不需要跳转了，因此会产生 1 次 Branch Misprediction Stalls。因此，一个循环不管会循环多少次，他都只会产生 1 个 Branch Taken Stalls 和 1 个 Branch Misprediction Stalls。而每次 Branch Taken Stall 和 Branch Misprediction Stalls 阻塞的周期数为 2 个。因此就会产生 2 个 Branch Taken Stall 和 Branch Misprediction Stalls 的阻塞周期。

而在代码中，在输入部分时有 1 个循环，在乘法部分有 64+8+1 个循环，总共有 74

次循环，因此总共会产生 Branch Taken Stalls 和 Branch Misprediction Stalls 的阻塞周期数都为 74\*2=148 个。

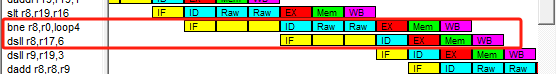


图5 开启BTB时预测正确

（二）设计使BTB无效的代码

在这个部分，我们要设计一段代码，这段代码包含了一个循环。根据BTB的特性，我们设计的这个代码将使得BTB的开启起不到相应的优化作用，反而会是的性能大大降低。

提示：一定要利用BTB的特性，即它的跳转判定是根据之前跳转成功与否来决定的。

给出所用代码以及设计思路，给出运行结果的截屏证明代码实现了目标。

BTB技术的原理是依照上一次的跳转结果来进行预测。要设计BTB失效的代码，也就是要设计一个每次跳转结果都不一样的代码。我们可以利用奇数和偶数的交替，为奇数时顺序执行，为偶数时跳转。代码如图6所示。

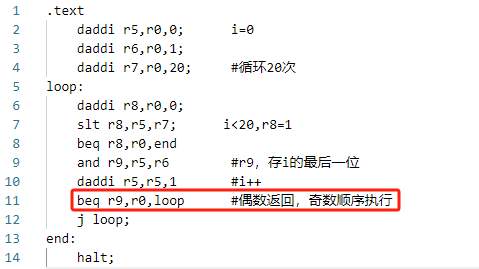


图6 BTB无效代码

取消勾选BTB，运行程序，运行结果如图7所示，可以看到，总时钟周期数为255，其中Branch Taken Stalls有21次。

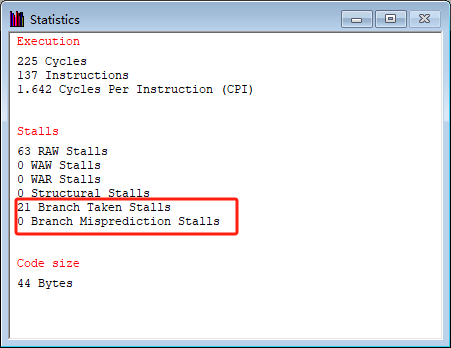


图7 未开启BTB

再开启BTB，运行程序，结果如图8所示，这次的总时钟周期数达到了248次，高于没开的225次，Branch Taken Stalls上升到24次，并且Branch Misprediction Stalls有20次。这与我们设想的一致，二十次循环中的beq，第一次没有预测，剩下十九次都预测失败。在加上指针i的最后预测的预测错误，总共20次分支预测错误。说明我们设计的这个代码将使得BTB的开启起不到相应的优化作用，反而会使得性能大大降低。

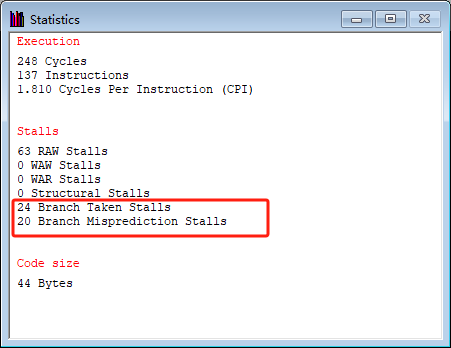


图8 开启BTB性能下降

（三）循环展开与BTB的效果比对

首先，我们需要对循环展开这个概念有一定的了解。

什么是循环展开呢？所谓循环展开就是通过在每次迭代中执行更多的数据操作来减小循环开销的影响。其基本思想是设法把操作对象线性化，并且在一次迭代中访问线性数据中的一个小组而非单独的某个。这样得到的程序将执行更少的迭代次数，于是循环开销就被有效地降低了。

接下来，我们就按照这种思想对上述的矩阵乘法程序进行循环展开。要求将上述的代码通过循环展开将最里面的一个执行迭代8次的循环整个展开了，也就是说，我们将矩阵相乘的三个循环通过代码的增加，减少到了两个循环。只需要将原本最内层的代码重复8次，每次将r19的值加一即可，代码如图9所示。

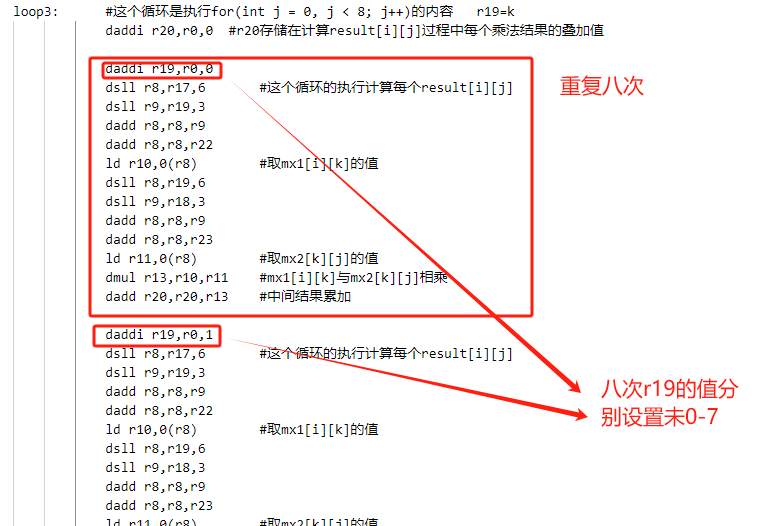


图9 循环展开代码

比较，通过对比循环展开（未启用BTB）、使用BTB（未进行循环展开）以及未使用BTB且未作循环展开的运行结果。比较他们的Branch Tanken Stalls和Branch Misprediction Stalls的数量，并尝试给出评判。

使用循环展开未开启BTB的Branch Taken Stalls为126次（如图10所示）。因为循环展开后最内层循环就消失了，而 n 次循环会产生 n-1 次 Branch Taken Stall，而每次 Branch Taken Stall 会阻塞 1 个周期。因此Branch Taken Stall 产生阻塞的周期数为 63+8\*7+1\*7=126。

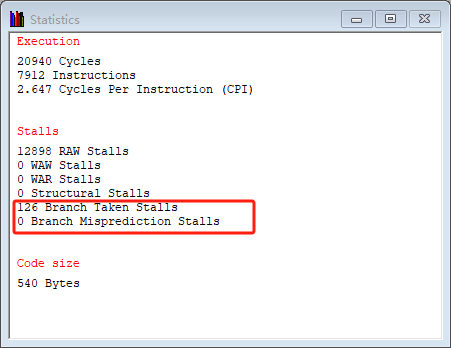


图10 循环展开（未启用BTB）

开启BTB未进行循环展开的次数为148次Branch Take Stalls+148次Branch Misprediction Stalls（如图11所示），因为循环的个数为 1+64+8+1=74 次，而每次循环都会产生 1 次 Branch Taken Stall 和 1 次 Branch Misprediction Stalls，而每次 Branch Taken Stall 和 Branch Misprediction Stalls 阻塞的周期数为 2 个。因此 Branch Taken Stall 和Branch Misprediction Stalls 产生阻塞的周期数都等于 148。

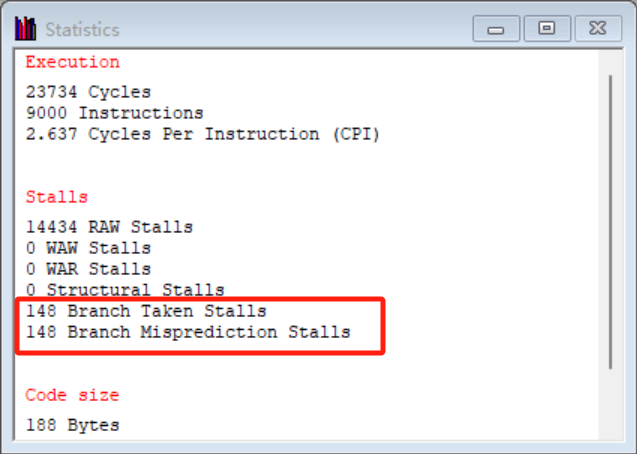


图11 用BTB（未进行循环展开）

未开启BTB且未作循环展开的Branch Taken Stalls为574次（如图1所示），因为一个 n 次循环都会产生 n-1 次 Branch Taken Stall，而每次 Branch Taken Stall 会阻塞 1 个周期，因此 Branch Taken Stall 阻塞的周期数等于 63+64\*7+8\*7+1\*7=574。

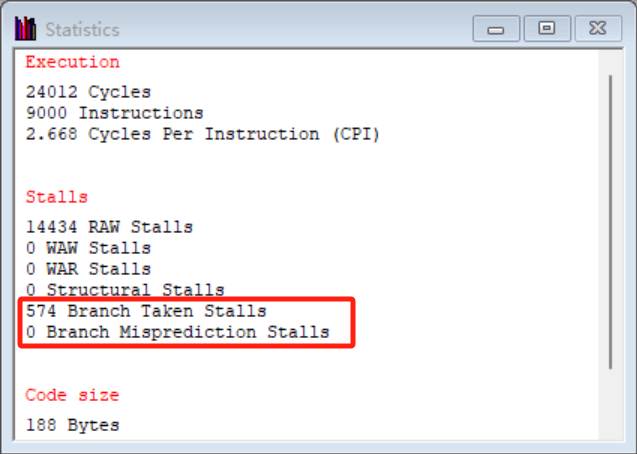


图 未使用BTB且未作循环展开

可以总结出，循环展开可以通过减少迭代次数来大大减少分支判断的开销，循环展开对于矩阵乘法性能的提高比 BTB 对矩阵乘法性能的提高还更好。但是循环展开会使得代码变得冗长，可读性下降和维护困难。

## 五、实验结果

1. 给出一段矩阵乘法的代码，通过开启 BTB 功能对其进行优化，并且观察流水线的细节，解释 BTB 在其中所起的作用；

2. 自行设计一段使得即使开启了 BTB 也无效的代码。

3. 对矩阵乘法的代码使用循环展开的方法，观察流水因分支停顿的次数减少的现象，并对比采用 BTB 结构时流水因分支而停顿的次数。

## 六、实验总结与体会

通过参本次实验，我更加深刻的理解了控制冒险、分支预测以及BTB等概念。特别是对动态分支预测和BTB的机制有了更为深入的理解，通过对给定矩阵乘法代码的优化，我实际体验了开启BTB功能对流水线性能的提升。观察流水线的细节，特别关注了BTB在程序中所起的作用，对于分支的预测和流水线的优化有了更为清晰的认识。然后，自行设计无效的BTB代码，这一部分的设计挑战让我更深刻地理解了BTB的特点以及在何种状态下它会失效。最后，使用循环展开方法观察流水线因分支停顿的次数减少的现象，同时与BTB结构时的流水线停顿次数进行对比。这一步骤让我进一步认识了不同优化手段对程序性能的影响，对于选择合适的优化策略具有指导意义。

总的来说，通过这次实验，我不仅深入了解了计算机体系结构中与分支预测相关的重要概念和机制，而且通过实际操作对于流水线的优化有了更加深刻的认识。

|  |
| --- |
| **指导教师批阅意见：**  **成绩评定：**  指导教师签字： 王毅  2023年 12 月 15 日 |
| 备注： |