山东大学 计算机科学与技术 学院

计算机系统原理 课程实验报告

|  |  |  |  |
| --- | --- | --- | --- |
| 学号：202200400053 | 姓名：王宇涵 | | 班级： 2班 |
| 实验题目：设计MIPS五级流水线模拟器中的Cache | | | |
| 实验学时：2 | | 实验日期： 2023-12-05 | |
| 实验目的：  （1）Cache 结构及功能的设计  （2）了解指令流水线运行的过程  （3）探究 Cache 对计算机性能的影响 | | | |
| 硬件环境：  龙芯实验平台（MIPS） | | | |
| 软件环境：  支持MIPS 指令集的流水线计时模拟器  Linux虚拟机进行cache扩展仿真 | | | |
| 实验步骤与内容：   1. 进行环境的配置   编译必要组件,运行检查脚本, 编译一个启用了DEBUG宏的sim程序,并运行单个case进行调试(测试样例在input文件夹中,我们选用一个为lbu.x),终端显示了每个循环中流水线上Dcode,Exec,Mem,Wb阶段对应的PC,便于我们进行调试   2.配置好基础环境,观察源码   根据实验手册的提示,我们得知各个文件存放的分别是  mips.h: 指令的操作码和子操作码  pipe.c:实现了流水线具体的函数   * pipe\_init 初始化流水线 * pipe\_cycle 模拟流水线在一个cycle做的工作 * pipe\_recover 恢复流水线，当出现分支指令时会用上 * pipe\_stage\_fetch 取指阶段模拟 * pipe\_stage\_decode 译码阶段模拟 * pipe\_stage\_execute 执行阶段模拟 * pipe\_stage\_mem 访存阶段模拟 * pipe\_stage\_wb 写回阶段模拟   pipe.h**:** 定义了结构体指令pipe\_op , 流水线pipe\_state  shell.c**:** 实现了终端界面和mem\_read\_32和mem\_write\_32内存读写函数  shell.h:作了终端的声明 3.阅读pipe.h 看到第一个结构体pipe\_op,拥有以下我们实验会用到的重要成员变量  程序计数器PC,指令instruction,解码后的opcode  寄存器源reg\_src1和reg\_src2,寄存器目的地reg\_dst(value)  内存访问信息is\_mem, mem\_addr, mem\_write, mem\_value ; 4.阅读shell.c 阅读mem\_read\_32和mem\_write\_32内存读写函数,发现会根据传入的地址到初始化好的内存块中去寻找对应的内存块,进行读取或写入的操作 5.阅读核心的pipe.c 对整体流程有个大致的了解  观察pipe\_cycle(), 按照wb,mem,execute,decode,fetch的顺序执行,这不由得让我产生疑问:为什么需要倒叙执行呢?带着疑问接着往下看, 剩下的是处理分支恢复的函数,一次循环便结束了.  我们接着从fetch->wb的顺序观察5个阶段的函数,有几个值得注意的点:   * 每个阶段开始时(除了mem和wb外,因为wb不会产生延迟,因此没有判断的必要)都会判断该阶段的下一阶段是否正在执行(指令!=NULL),如果正在执行的话就取消执行,结合课上所学知识,我们不难想到,如果我们decode阶段正在延迟,我们执行fetch的时候,如果继续将op赋值给decode\_op,则会覆盖现在正在延迟的decode阶段,因此我们选择等待,直接return,这也解释了为什么我们需要倒叙执行,这样可以保证在某条阶段(对应op=m)延迟的时候,不影响op<m(在此之前)的阶段继续执行相应的操作,会使得op>m(在此之后)的阶段进入等待      * 实际上只有fetch和mem两个阶段对内存进行了操作,因此我们加入cache只需要修改这两个阶段和内存有关的读取操作,改为与cache进行交互即可. * 每个阶段结束时都会将当前阶段的op赋值为NULL,下一个阶段op赋值为当前的op,因此我们进入延迟之后,直接return的话,当前阶段便会显示一直正在执行,下一个阶段没有赋值,因此不会进入下一个阶段,和上方的倒叙执行相呼应.  6.建立cache.h 按照要求建立inst\_cache 和data\_cache两大类,每个大类包含组类sets,每个组类包含行类lines, 行类均设置有效位valid,替换策略LRU, data\_cache\_line还包含脏位dirty,来判断是否需要将数据写回内存. 7.inst\_cache具体函数 **inst\_cache\_hit函数**:传入内存地址,从cache中读取指令,判断是否命中   1. 先取出组号,也就是内存的倒数第6-11位: (address << 21) >> 26 2. 再根据组号遍历每一行,取得每一行的标志位(lines[i].tag),与实际标志位(address >> 11)做比较,如果有相等的且valid=1代表命中,返回cache中的数据,如果均未命中返回错误码   注 : 取cache中的数据操作: lines[i]. data[(address & 0x0000001f) >> 2], (address & 0x0000001f)代表取内存的后五位,也就是块内偏移量, >>2代表将偏移量改为4个字节,对应了data[8]的对应位元素  **inst\_cache\_miss函数**:如果没命中(miss),就将内存对应的块写入cache中   1. 首先我们需要判断内存的组成,对于inst\_cache, 6位组号(64块=2^6) ,5位块内偏移量(32字节=2^5),21位tag(32-6-5),因此内存组成为21~6~5 2. 再取出内存中对应的块(address & 0xffffffe0):代表找到了相应块的第一个字的地址 3. 再将内存中对应的指令地址取出(我们由mem\_read\_32函数可知,地址的偏移量为4个字节)   For (i = 0; i < 8; i++) mem[i] = mem\_read\_32(block\_address + i \* 4);   1. 我们需要找到cache对应的空行放进去,先遍历对应组的每一行,如果有空行(valid==0),则找到对应行,更新元素:LRU=0 ,valid=1,tag=address>>11,并将mem赋值给data,返回查找的指令地址(注意需要更新别的行LRU).   注 : 如果没有找到对应的空行,需要使用LRU替换算法,找到拥有最大LRU的一行进行替换,此时对别的行进行LRU++,替换数据后后同找到空行相同,返回查找的指令地址 7.data\_cache具体函数 整体思路和inst\_cache相同 :  首先我们需要判断内存的组成,对于inst\_cache, 8位组号(64块=2^6) ,5位块内偏移量(32字节=2^5),19位tag(32-8-5),因此内存组成为19~8~5.  **data\_cache\_read函数**: 传入内存地址,从cache中读取数据,如果没有命中将内存对应的块写入cache中.  整体与inst\_cache\_read类似, 不加以赘述.  **data\_cache\_write函数**: 传入内存地址和值, 将cache中对应的数据修改,如果没有命中将内存对应的块写入cache中,并修改对应的值,注意如果需要替换行的话,需要判断是否被替换的行dirty是否等于1,如果等于1需要将值写回内存.  注 : 将cache值写回内存操作 : 根据对应的被替换行的tag和组号相拼接,找到内存中对应块的第一个字节dCache.sets[sets\_index].lines[max\_LRU\_line].tag << 13) | sets\_index再写入内存, for (i = 0; i < 8; i++) mem\_write\_32(block\_address + i \* 4, dCache.sets[sets\_index].lines[max\_LRU\_line].data[i]); 8.延迟的实现 我们如果没有命中就需要从内存中读写,需要进入周期为50的延迟,因此我们对于inst\_cache,data\_cache,都设置延迟变量=50,每次需要进行miss操作的时候,便进行延迟变量减1,并return,直到延迟结束才进行操作,延迟结束后将延迟变量重新赋值为50. 9.拓展:改变cache的大小进行测试 Small\_cache :将inst\_cache和data\_cache的组数设为2,行数都设为2  Large\_cache:将inst\_cache和data\_cache的组数设为11,行数都设为2500  结果:两者测试不完全相同,对于少量数据来说对ipc结果的影响不是很大,但是large\_cache进行操作的时候具有卡顿,表示每一个任务执行的时间比较长,可能是由于行数设置比较多,每次都要遍历所有行的原因.     10.拓展:替换LFU算法进行测试 结果相同,可见两个算法对于少量数据差别不是很大 | | | |
| 结论分析与体会：  本次实验也具有一定的挑战性, 我设计了cache用来模拟与内存之间的数据交换,并深刻理解了流水线工作的机制, 设计了延迟,成功完成了实验目标.再完成了基础要求之后,我也修改了代码,对组数和行数进行了变量定义,进行了不同cache大小对于性能的测试,结果发现程序运行时间有了影响,但对于ipc来说影响不大.我还自己设计了LFU算法,结果发现和LRU算法没有明显的性能上的差距,也因此加深了对替换算法的认识. | | | |

附源代码

## 实验基础要求Cache

### Cache.h

#include"shell.h"

#include <string.h>

//指令cache

typedef struct

{

int valid;//有效位

int LRU;//判断如何选择替换的行

uint32\_t tag;//标志位

uint32\_t data[8];//数据区域，八个四字节

} inst\_cache\_line;

typedef struct//指令cache的一组

{

inst\_cache\_line lines[4];//一组四路

} inst\_cache\_sets;

typedef struct//指令cache

{

inst\_cache\_sets sets[64];//64组

} inst\_cache;

//数据cache

typedef struct

{

int valid;

int LRU;

int dirty;// 脏位,判断数据是否需要写回内存

uint32\_t tag;

uint32\_t data[8];

}data\_cache\_line;

typedef struct

{

data\_cache\_line lines[8];

}data\_cache\_sets;

typedef struct

{

data\_cache\_sets sets[256];//256组

}data\_cache;

//判断指令是否命中cache

uint32\_t inst\_cache\_hit(uint32\_t address);

//将块从内存中放入cache

uint32\_t inst\_cache\_miss(uint32\_t address);

//从cache中读取数据

uint32\_t data\_cache\_read(uint32\_t address);

//到cache中写数据

uint32\_t data\_cache\_write(uint32\_t address, uint32\_t value);

### Cache.c

#include "cache.h"

#include "pipe.h"

#include "shell.h"

#include "mips.h"

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <assert.h>

//写延迟标志

int data\_cache\_write\_count = 50;

//读延迟标志

int data\_cache\_read\_count = 50;

//标志初始化

int inst\_cache\_count = 0;

int data\_cache\_count = 0;

//全局变量

inst\_cache iCache;

data\_cache dCache;

//指令cache是否命中，命中返回取得的指令,否则返回0x0381CD55

uint32\_t inst\_cache\_hit(uint32\_t address)//判断是否命中

{

//初始化指令cache

if (inst\_cache\_count == 0)

{

memset(&iCache, 0, sizeof(inst\_cache));

inst\_cache\_count++;

}

//此时内存的设置为 21位tag 6位组号 5位块内字节偏移量

uint32\_t sets\_index = (address << 21) >> 26;//得到六位的组号

int i;

for (i = 0; i < 4; i++)//检查对应组中的四行

{

uint32\_t tag = iCache.sets[sets\_index].lines[i].tag;//取得每一行的标志位

if (!(tag ^ (address >> 11)))//tag 位相同再检查 valid

{

if (iCache.sets[sets\_index].lines[i].valid == 1)//有效位为1，命中

{

return iCache.sets[sets\_index].lines[i].data[(address & 0x0000001f) >> 2];

//读取address后五位,右移两位缩小四倍，即为四个字节单元的偏移量,对应了data[]

}

}

}

return 0x0381CD55;//如果未命中

}

//指令cache缺失替换操作

uint32\_t inst\_cache\_miss(uint32\_t address)

{

//从内存中取出缺失的块

uint32\_t block\_address = address & 0xffffffe0; //找到对应块中第一个字的地址

uint32\_t mem[8];

int i;

//将内存中对应的指令存入mem

for (i = 0; i < 8; i++)

mem[i] = mem\_read\_32(block\_address + i \* 4);

//将取出的块放入cache中对应的组

uint32\_t sets\_index = (address << 21) >> 26; //计算组号

//找组中的空行

int j;

int beset=-1;//找到的对应的空行

for (j = 0; j < 4; j++)//有空行则将块放入

{

if (iCache.sets[sets\_index].lines[j].valid == 0)//空位

{

iCache.sets[sets\_index].lines[j].LRU = 0; //LRU初始化为0

iCache.sets[sets\_index].lines[j].valid = 1; //valid 置为有效

iCache.sets[sets\_index].lines[j].tag = address >> 11; // 更新tag

//将块放入数据区

int k;

for (k = 0; k < 8; k++)

iCache.sets[sets\_index].lines[j].data[k] = mem[k];

beset=j;break;

}

}

//找到了空行

if(beset!=-1)

{

//更新空行之外行的LRU++

for (int m = 0; m < 4; m++)

{

if(m==beset)

continue;

else

if(iCache.sets[sets\_index].lines[m].LRU<3)

iCache.sets[sets\_index].lines[m].LRU++;

}

return iCache.sets[sets\_index].lines[j].data[(address & 0x0000001f) >> 2];

}

//没有空行的话，替换LRU最大的行

int max\_LRU\_line = 0;

int max\_LRU = iCache.sets[sets\_index].lines[0].LRU;

int m;

for (m = 1; m < 4; m++) //寻找LRU最大的行

{

int theLRU = iCache.sets[sets\_index].lines[m].LRU;

if (theLRU >= max\_LRU)

{

max\_LRU = theLRU;

max\_LRU\_line = m;

}

}

//替换行更新LRU

for (m = 0; m < 4; m++) //寻找LRU最大的行

{

if(m==max\_LRU\_line)

continue;

else

if(iCache.sets[sets\_index].lines[m].LRU<3)

iCache.sets[sets\_index].lines[m].LRU++;

}

//替换该行内容

iCache.sets[sets\_index].lines[max\_LRU\_line].LRU = 0;

iCache.sets[sets\_index].lines[max\_LRU\_line].valid = 1;

iCache.sets[sets\_index].lines[max\_LRU\_line].tag = address >> 11;

int n;

for (n = 0; n < 8; n++)

iCache.sets[sets\_index].lines[max\_LRU\_line].data[n] = mem[n];

return iCache.sets[sets\_index].lines[max\_LRU\_line].data[(address & 0x0000001f) >> 2];

}

uint32\_t data\_cache\_read(uint32\_t address)

{

if (data\_cache\_count == 0)

{//初始化

memset(&dCache, 0, sizeof(data\_cache));

}

data\_cache\_count++;

uint32\_t sets\_index = (address << 19) >> 24;//计算组号

//检查组中有无tag相同的行

int i;

for (i = 0; i < 8; i++)//判断是否命中

{

uint32\_t tag = dCache.sets[sets\_index].lines[i].tag;

if (!(tag ^ (address >> 13)))//tag位相同

{

if (dCache.sets[sets\_index].lines[i].valid == 1)//命中

{

return dCache.sets[sets\_index].lines[i].data[(address & 0x0000001f) >> 2];

}

}

}

//没命中,进入延迟50周期

if (data\_cache\_read\_count != 0)

{

data\_cache\_read\_count--;

return 0x0381CD55;

}

data\_cache\_read\_count = 50;//将延迟计数复位

//miss操作

uint32\_t block\_address = address & 0xffffffe0;//找到对应块中第一个字的地址

uint32\_t mem[8];

int j;

for (j = 0; j < 8; j++)

mem[j] = mem\_read\_32(block\_address + j \* 4);

//寻找空行

int k;int beset=-1;

for (k = 0; k < 8; k++)

{

//找到空行装填入缺失块

if (dCache.sets[sets\_index].lines[k].valid == 0)

{

dCache.sets[sets\_index].lines[k].valid = 1;

dCache.sets[sets\_index].lines[k].LRU = 0;

dCache.sets[sets\_index].lines[k].dirty = 0;//初始dirty=0

dCache.sets[sets\_index].lines[k].tag = address >> 13;

int m;

for (m = 0; m < 8; m++)

dCache.sets[sets\_index].lines[k].data[m] = mem[m];

beset=k;

break;

}

}

//如果找到了空行,更新flag

if(beset!=-1)

{

//更新LRU

for (int m = 0; m < 8; m++)

{

if(m==beset)

continue;

else

if(dCache.sets[sets\_index].lines[m].LRU<7)

dCache.sets[sets\_index].lines[m].LRU++;

}

return dCache.sets[sets\_index].lines[beset].data[(address & 0x0000001f) >> 2];

}

//没有空行，找LRU最小的行

int max\_LRU\_line = 0;

int max\_LRU = dCache.sets[sets\_index].lines[0].LRU;

int n;

for (n = 1; n < 8; n++)

{

int theLRU = dCache.sets[sets\_index].lines[n].LRU;

if (theLRU >= max\_LRU)

{

max\_LRU = theLRU;

max\_LRU\_line = n;

}

}

//替换行更新LRU

for (int m = 0; m < 8; m++)

{

if(m==max\_LRU\_line)

continue;

else

if(dCache.sets[sets\_index].lines[m].LRU<7)

dCache.sets[sets\_index].lines[m].LRU++;

}

//根据最大LRU行的dirty确定它是否需要写回

if (dCache.sets[sets\_index].lines[max\_LRU\_line].dirty == 1)//需要写回

{

uint32\_t sets\_index\_13 = sets\_index;

//此时进行位的拼接

sets\_index\_13 << 5;//低5位是0，中间八位是组号，高位是0 与 高位是tag 其余为0 进行拼凑

uint32\_t block\_address = (dCache.sets[sets\_index].lines[max\_LRU\_line].tag << 13) | sets\_index\_13;

for (i = 0; i < 8; i++)

mem\_write\_32(block\_address + i \* 4, dCache.sets[sets\_index].lines[max\_LRU\_line].data[i]);

}

//将该块替换

dCache.sets[sets\_index].lines[max\_LRU\_line].valid = 1;

dCache.sets[sets\_index].lines[max\_LRU\_line].LRU = 0;

dCache.sets[sets\_index].lines[max\_LRU\_line].dirty = 0;

dCache.sets[sets\_index].lines[max\_LRU\_line].tag = address >> 13;

for (i = 0; i < 8; i++)

dCache.sets[sets\_index].lines[max\_LRU\_line].data[i] = mem[i];

return dCache.sets[sets\_index].lines[max\_LRU\_line].data[(address & 0x0000001f) >> 2];

}

uint32\_t data\_cache\_write(uint32\_t address, uint32\_t value)

{

if (data\_cache\_count == 0)//初始化

{

memset(&dCache, 0, sizeof(data\_cache));

data\_cache\_count++;

}

uint32\_t sets\_index = (address << 19) >> 24;//计算组号

//检查组中有无tag相同的行

int i;

for (i = 0; i < 8; i++)

{

uint32\_t tag = dCache.sets[sets\_index].lines[i].tag;

if (!(tag ^ (address >> 13)))//tag相同

{

if (dCache.sets[sets\_index].lines[i].valid == 1)//命中

{

dCache.sets[sets\_index].lines[i].valid = 1;

dCache.sets[sets\_index].lines[i].dirty = 1;

dCache.sets[sets\_index].lines[i].data[(address & 0x0000001f) >> 2] = value;

return 1;

}

}

}

//没命中,进入延迟

if (data\_cache\_write\_count != 0)

{

data\_cache\_write\_count--;

return 0;

}

data\_cache\_write\_count=50;

//miss 操作

//从内存中取出缺失块

uint32\_t block\_address = address & 0xffffffe0;

uint32\_t mem[8];

int j;

for (j = 0; j < 8; j++)

mem[j] = mem\_read\_32(block\_address + j \* 4);

//寻找空行

int k;int beset=-1;

for (k = 0; k < 8; k++)

{

//找到空行填入缺失块

if (dCache.sets[sets\_index].lines[k].valid == 0)

{

dCache.sets[sets\_index].lines[k].valid = 1;

dCache.sets[sets\_index].lines[k].LRU = 0;

dCache.sets[sets\_index].lines[k].dirty = 1;//脏位

dCache.sets[sets\_index].lines[k].tag = address >> 13;

int m;

for (m = 0; m < 8; m++)

dCache.sets[sets\_index].lines[k].data[m] = mem[m];

dCache.sets[sets\_index].lines[k].data[(address & 0x0000001f) >> 2] = value;

beset=k;

break;

}

}

//找到空行更新LRU

if(beset!=-1)

{

//更新LRU

for (int m = 0; m < 8; m++)

{

if(m==beset)

continue;

else

if(dCache.sets[sets\_index].lines[m].LRU<7)

dCache.sets[sets\_index].lines[m].LRU++;

}

return dCache.sets[sets\_index].lines[j].data

[(address & 0x0000001f) >> 2];

return dCache.sets[sets\_index].lines[k].data[(address & 0x0000001f) >> 2];

}

//没有空行找最大LRU

int max\_LRU\_line = 0;

int max\_LRU = dCache.sets[sets\_index].lines[0].LRU;

int n;

for (n = 1; n < 8; n++)

{

int theLRU = dCache.sets[sets\_index].lines[n].LRU;

if (theLRU >= max\_LRU)

{

max\_LRU = theLRU;

max\_LRU\_line = n;

}

}

//替换行更新LRU

for (int m = 0; m < 8; m++)

{

if(m==max\_LRU\_line)

continue;

else

if(dCache.sets[sets\_index].lines[m].LRU<7)

dCache.sets[sets\_index].lines[m].LRU++;

}

//根据最小LRU行的dirty确定它是否需要写回

if (dCache.sets[sets\_index].lines[max\_LRU\_line].dirty == 1)

{

uint32\_t sets\_index\_13 = sets\_index;

sets\_index\_13 << 5;//低5位是0，中间八位是组号，高位是0

uint32\_t block\_address = (dCache.sets[sets\_index].lines[max\_LRU\_line].tag << 13) | sets\_index\_13;

for (i = 0; i < 8; i++)

mem\_write\_32(block\_address + i \* 4, dCache.sets[sets\_index].lines[max\_LRU\_line].data[i]);

}

//将该块替换

dCache.sets[sets\_index].lines[max\_LRU\_line].valid = 1;

dCache.sets[sets\_index].lines[max\_LRU\_line].LRU = 0;

dCache.sets[sets\_index].lines[max\_LRU\_line].dirty = 1;

dCache.sets[sets\_index].lines[max\_LRU\_line].tag = address >> 13;

for (i = 0; i < 8; i++)

dCache.sets[sets\_index].lines[max\_LRU\_line].data[i] = mem[i];

dCache.sets[sets\_index].lines[max\_LRU\_line].data[(address & 0x0000001f) >> 2] = value;

return 1;//成功写入返回1

}

### Pipe.c

#include "pipe.h"

#include "shell.h"

#include "mips.h"

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <assert.h>

#include "cache.h"

int iCache\_read\_count = 50;

//#define DEBUG

/\* debug \*/

void print\_op(Pipe\_Op \*op)

{

if (op)

printf("OP (PC=%08x inst=%08x) src1=R%d (%08x) src2=R%d (%08x) dst=R%d valid %d (%08x) br=%d taken=%d dest=%08x mem=%d addr=%08x\n",

op->pc,

op->instruction,

op->reg\_src1,

op->reg\_src1\_value,

op->reg\_src2,

op->reg\_src2\_value,

op->reg\_dst,

op->reg\_dst\_value\_ready,

op->reg\_dst\_value,

op->is\_branch,

op->branch\_taken,

op->branch\_dest,

op->is\_mem,

op->mem\_addr);

else

printf("(null)\n");

}

/\* global pipeline state \*/

Pipe\_State pipe;

void pipe\_init()

{

memset(&pipe, 0, sizeof(Pipe\_State));

pipe.PC = 0x00400000;

}

void pipe\_cycle()//周期

{

#ifdef DEBUG

printf("\n\n----\n\nPIPELINE:\n");

printf("DCODE: "); print\_op(pipe.decode\_op);

printf("EXEC : "); print\_op(pipe.execute\_op);

printf("MEM : "); print\_op(pipe.mem\_op);

printf("WB : "); print\_op(pipe.wb\_op);

printf("\n");

#endif

pipe\_stage\_wb();

pipe\_stage\_mem();

pipe\_stage\_execute();

pipe\_stage\_decode();

pipe\_stage\_fetch();

/\* handle branch recoveries \*/

if (pipe.branch\_recover) {

#ifdef DEBUG

printf("branch recovery: new dest %08x flush %d stages\n", pipe.branch\_dest, pipe.branch\_flush);

#endif

pipe.PC = pipe.branch\_dest;

if (pipe.branch\_flush >= 2) {

if (pipe.decode\_op) free(pipe.decode\_op);

pipe.decode\_op = NULL;

}

if (pipe.branch\_flush >= 3) {

if (pipe.execute\_op) free(pipe.execute\_op);

pipe.execute\_op = NULL;

}

if (pipe.branch\_flush >= 4) {

if (pipe.mem\_op) free(pipe.mem\_op);

pipe.mem\_op = NULL;

}

if (pipe.branch\_flush >= 5) {

if (pipe.wb\_op) free(pipe.wb\_op);

pipe.wb\_op = NULL;

}

pipe.branch\_recover = 0;

pipe.branch\_dest = 0;

pipe.branch\_flush = 0;

stat\_squash++;

}

}

void pipe\_recover(int flush, uint32\_t dest)//覆盖

{

/\*如果已经计划了恢复，则它必须来自稍后的

\*阶段（执行较旧的指令），因此恢复覆盖

\*我们的恢复。在这种情况下，只需返回\*/

if (pipe.branch\_recover) return;

/\*安排恢复。这将在所有管道阶段模拟当前循环后完成\*/

pipe.branch\_recover = 1;

pipe.branch\_flush = flush;

pipe.branch\_dest = dest;

}

void pipe\_stage\_wb()//写回

{

/\*如果在这个流水线阶段没有指令，我们就完成了\*/

if (!pipe.wb\_op)

return;

/\*从我们的输入槽中取出op\*/

Pipe\_Op \*op = pipe.wb\_op;

pipe.wb\_op = NULL;

/\*如果此指令写入寄存器，请立即执行\*/

if (op->reg\_dst != -1 && op->reg\_dst != 0) {

pipe.REGS[op->reg\_dst] = op->reg\_dst\_value;

#ifdef DEBUG

printf("R%d = %08x\n", op->reg\_dst, op->reg\_dst\_value);

#endif

}

/\*如果这是系统调用，请执行操作\*/

if (op->opcode == OP\_SPECIAL && op->subop == SUBOP\_SYSCALL) {

if (op->reg\_src1\_value == 0xA) {

pipe.PC = op->pc; /\*fetch将执行pc+=4，然后我们停止使用正确的pc\*/

RUN\_BIT = 0;

}

}

/\*释放op\*/

free(op);

stat\_inst\_retire++;

}

void pipe\_stage\_mem()//取操作数，访存

{

/\*如果在这个流水线阶段没有指令，我们就完成了\*/

if (!pipe.mem\_op)

return;

/\*从我们的输入槽中取出op\*/

Pipe\_Op \*op = pipe.mem\_op;

uint32\_t val = 0;

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

if (op->is\_mem)

{

uint32\_t theVal = data\_cache\_read(op->mem\_addr & ~3);

if (theVal == 0x0381CD55)

return;

val = theVal;

}

switch (op->opcode) {

case OP\_LW:

case OP\_LH:

case OP\_LHU:

case OP\_LB:

case OP\_LBU:

{

/\*提取所需值\*/

op->reg\_dst\_value\_ready = 1;

if (op->opcode == OP\_LW) {

op->reg\_dst\_value = val;

}

else if (op->opcode == OP\_LH || op->opcode == OP\_LHU) {

if (op->mem\_addr & 2)

val = (val >> 16) & 0xFFFF;

else

val = val & 0xFFFF;

if (op->opcode == OP\_LH)

val |= (val & 0x8000) ? 0xFFFF8000 : 0;

op->reg\_dst\_value = val;

}

else if (op->opcode == OP\_LB || op->opcode == OP\_LBU) {

switch (op->mem\_addr & 3) {

case 0:

val = val & 0xFF;

break;

case 1:

val = (val >> 8) & 0xFF;

break;

case 2:

val = (val >> 16) & 0xFF;

break;

case 3:

val = (val >> 24) & 0xFF;

break;

}

if (op->opcode == OP\_LB)

val |= (val & 0x80) ? 0xFFFFFF80 : 0;

op->reg\_dst\_value = val;

}

}

break;

case OP\_SB:

{

switch (op->mem\_addr & 3) {

case 0: val = (val & 0xFFFFFF00) | ((op->mem\_value & 0xFF) << 0); break;

case 1: val = (val & 0xFFFF00FF) | ((op->mem\_value & 0xFF) << 8); break;

case 2: val = (val & 0xFF00FFFF) | ((op->mem\_value & 0xFF) << 16); break;

case 3: val = (val & 0x00FFFFFF) | ((op->mem\_value & 0xFF) << 24); break;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

int result1 = data\_cache\_write(op->mem\_addr & ~3, val);

if(result1==0)

return;

//mem\_write\_32(op->mem\_addr & ~3, val);

break;

}

case OP\_SH:

{

#ifdef DEBUG

printf("SH: addr %08x val %04x old word %08x\n", op->mem\_addr, op->mem\_value & 0xFFFF, val);

#endif

if (op->mem\_addr & 2)

val = (val & 0x0000FFFF) | (op->mem\_value) << 16;

else

val = (val & 0xFFFF0000) | (op->mem\_value & 0xFFFF);

#ifdef DEBUG

printf("new word %08x\n", val);

#endif

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

int result2 = data\_cache\_write(op->mem\_addr & ~3, val);

if (result2 == 0)

return;

//mem\_write\_32(op->mem\_addr & ~3, val);

break;

}

case OP\_SW:

{

val = op->mem\_value;

int result3 = data\_cache\_write(op->mem\_addr & ~3, val);

if (result3 == 0)

return;

//mem\_write\_32(op->mem\_addr & ~3, val);

break;

}

}

/\*清除阶段输入并转移到下一阶段\*/

pipe.mem\_op = NULL;

pipe.wb\_op = op;

}

void pipe\_stage\_execute()//执行

{

/\*如果正在进行乘法/除法，递减循环直到值准备就绪\*/

if (pipe.multiplier\_stall > 0)

pipe.multiplier\_stall--;

/\*如果下游失速，返回（并留下我们的任何输入）\*/

if (pipe.mem\_op != NULL)

return;

/\*如果没有要执行的操作，则返回\*/

if (pipe.execute\_op == NULL)

return;

/\*抓取操作和读取源\*/

Pipe\_Op \*op = pipe.execute\_op;

/\*读取寄存器值并检查旁路；必要时失速\*/

int stall = 0;

if (op->reg\_src1 != -1) {

if (op->reg\_src1 == 0)

op->reg\_src1\_value = 0;

else if (pipe.mem\_op && pipe.mem\_op->reg\_dst == op->reg\_src1) {

if (!pipe.mem\_op->reg\_dst\_value\_ready)

stall = 1;

else

op->reg\_src1\_value = pipe.mem\_op->reg\_dst\_value;

}

else if (pipe.wb\_op && pipe.wb\_op->reg\_dst == op->reg\_src1) {

op->reg\_src1\_value = pipe.wb\_op->reg\_dst\_value;

}

else

op->reg\_src1\_value = pipe.REGS[op->reg\_src1];

}

if (op->reg\_src2 != -1) {

if (op->reg\_src2 == 0)

op->reg\_src2\_value = 0;

else if (pipe.mem\_op && pipe.mem\_op->reg\_dst == op->reg\_src2) {

if (!pipe.mem\_op->reg\_dst\_value\_ready)

stall = 1;

else

op->reg\_src2\_value = pipe.mem\_op->reg\_dst\_value;

}

else if (pipe.wb\_op && pipe.wb\_op->reg\_dst == op->reg\_src2) {

op->reg\_src2\_value = pipe.wb\_op->reg\_dst\_value;

}

else

op->reg\_src2\_value = pipe.REGS[op->reg\_src2];

}

/\*如果旁路需要暂停（例如，在加载后立即使用），

\*无清除阶段输入的返回\*/

if (stall)

return;

/\*执行操作\*/

switch (op->opcode) {

case OP\_SPECIAL:

op->reg\_dst\_value\_ready = 1;

switch (op->subop) {

case SUBOP\_SLL:

op->reg\_dst\_value = op->reg\_src2\_value << op->shamt;

break;

case SUBOP\_SLLV:

op->reg\_dst\_value = op->reg\_src2\_value << op->reg\_src1\_value;

break;

case SUBOP\_SRL:

op->reg\_dst\_value = op->reg\_src2\_value >> op->shamt;

break;

case SUBOP\_SRLV:

op->reg\_dst\_value = op->reg\_src2\_value >> op->reg\_src1\_value;

break;

case SUBOP\_SRA:

op->reg\_dst\_value = (int32\_t)op->reg\_src2\_value >> op->shamt;

break;

case SUBOP\_SRAV:

op->reg\_dst\_value = (int32\_t)op->reg\_src2\_value >> op->reg\_src1\_value;

break;

case SUBOP\_JR:

case SUBOP\_JALR:

op->reg\_dst\_value = op->pc + 4;

op->branch\_dest = op->reg\_src1\_value;

op->branch\_taken = 1;

break;

case SUBOP\_MULT:

{

/\*我们立即设置结果值；但是，我们会

\*如果程序试图读取值，则模拟暂停

\*在准备就绪之前（或覆盖HI/LO）。此外，如果

\*稍后会有另一个乘法运算

\*更新值并重新设置失速循环计数

\*用于新操作。

\*/

int64\_t val = (int64\_t)((int32\_t)op->reg\_src1\_value) \* (int64\_t)((int32\_t)op->reg\_src2\_value);

uint64\_t uval = (uint64\_t)val;

pipe.HI = (uval >> 32) & 0xFFFFFFFF;

pipe.LO = (uval >> 0) & 0xFFFFFFFF;

/\*四周期乘数延迟\*/

pipe.multiplier\_stall = 4;

}

break;

case SUBOP\_MULTU:

{

uint64\_t val = (uint64\_t)op->reg\_src1\_value \* (uint64\_t)op->reg\_src2\_value;

pipe.HI = (val >> 32) & 0xFFFFFFFF;

pipe.LO = (val >> 0) & 0xFFFFFFFF;

/\*四周期乘数延迟\*/

pipe.multiplier\_stall = 4;

}

break;

case SUBOP\_DIV:

if (op->reg\_src2\_value != 0) {

int32\_t val1 = (int32\_t)op->reg\_src1\_value;

int32\_t val2 = (int32\_t)op->reg\_src2\_value;

int32\_t div, mod;

div = val1 / val2;

mod = val1 % val2;

pipe.LO = div;

pipe.HI = mod;

} else {

//实际上，这将是一个除以0的异常

pipe.HI = pipe.LO = 0;

}

/\*32周期分频器延迟\*/

pipe.multiplier\_stall = 32;

break;

case SUBOP\_DIVU:

if (op->reg\_src2\_value != 0) {

pipe.HI = (uint32\_t)op->reg\_src1\_value % (uint32\_t)op->reg\_src2\_value;

pipe.LO = (uint32\_t)op->reg\_src1\_value / (uint32\_t)op->reg\_src2\_value;

} else {

/\* really this would be a div-by-0 exception \*/

pipe.HI = pipe.LO = 0;

}

/\* 32-cycle divider latency \*/

pipe.multiplier\_stall = 32;

break;

case SUBOP\_MFHI:

/\*暂停直到值准备就绪\*/

if (pipe.multiplier\_stall > 0)

return;

op->reg\_dst\_value = pipe.HI;

break;

case SUBOP\_MTHI:

/\* stall to respect WAW dependence \*/

/\*暂停以尊重WAW依赖性\*/

if (pipe.multiplier\_stall > 0)

return;

pipe.HI = op->reg\_src1\_value;

break;

case SUBOP\_MFLO:

/\* stall until value is ready \*/

if (pipe.multiplier\_stall > 0)

return;

op->reg\_dst\_value = pipe.LO;

break;

case SUBOP\_MTLO:

/\* stall to respect WAW dependence \*/

if (pipe.multiplier\_stall > 0)

return;

pipe.LO = op->reg\_src1\_value;

break;

case SUBOP\_ADD:

case SUBOP\_ADDU:

op->reg\_dst\_value = op->reg\_src1\_value + op->reg\_src2\_value;

break;

case SUBOP\_SUB:

case SUBOP\_SUBU:

op->reg\_dst\_value = op->reg\_src1\_value - op->reg\_src2\_value;

break;

case SUBOP\_AND:

op->reg\_dst\_value = op->reg\_src1\_value & op->reg\_src2\_value;

break;

case SUBOP\_OR:

op->reg\_dst\_value = op->reg\_src1\_value | op->reg\_src2\_value;

break;

case SUBOP\_NOR:

op->reg\_dst\_value = ~(op->reg\_src1\_value | op->reg\_src2\_value);

break;

case SUBOP\_XOR:

op->reg\_dst\_value = op->reg\_src1\_value ^ op->reg\_src2\_value;

break;

case SUBOP\_SLT:

op->reg\_dst\_value = ((int32\_t)op->reg\_src1\_value <

(int32\_t)op->reg\_src2\_value) ? 1 : 0;

break;

case SUBOP\_SLTU:

op->reg\_dst\_value = (op->reg\_src1\_value < op->reg\_src2\_value) ? 1 : 0;

break;

}

break;

case OP\_BRSPEC:

switch (op->subop) {

case BROP\_BLTZ:

case BROP\_BLTZAL:

if ((int32\_t)op->reg\_src1\_value < 0) op->branch\_taken = 1;

break;

case BROP\_BGEZ:

case BROP\_BGEZAL:

if ((int32\_t)op->reg\_src1\_value >= 0) op->branch\_taken = 1;

break;

}

break;

case OP\_BEQ:

if (op->reg\_src1\_value == op->reg\_src2\_value) op->branch\_taken = 1;

break;

case OP\_BNE:

if (op->reg\_src1\_value != op->reg\_src2\_value) op->branch\_taken = 1;

break;

case OP\_BLEZ:

if ((int32\_t)op->reg\_src1\_value <= 0) op->branch\_taken = 1;

break;

case OP\_BGTZ:

if ((int32\_t)op->reg\_src1\_value > 0) op->branch\_taken = 1;

break;

case OP\_ADDI:

case OP\_ADDIU:

op->reg\_dst\_value\_ready = 1;

op->reg\_dst\_value = op->reg\_src1\_value + op->se\_imm16;

break;

case OP\_SLTI:

op->reg\_dst\_value\_ready = 1;

op->reg\_dst\_value = (int32\_t)op->reg\_src1\_value < (int32\_t)op->se\_imm16 ? 1 : 0;

break;

case OP\_SLTIU:

op->reg\_dst\_value\_ready = 1;

op->reg\_dst\_value = (uint32\_t)op->reg\_src1\_value < (uint32\_t)op->se\_imm16 ? 1 : 0;

break;

case OP\_ANDI:

op->reg\_dst\_value\_ready = 1;

op->reg\_dst\_value = op->reg\_src1\_value & op->imm16;

break;

case OP\_ORI:

op->reg\_dst\_value\_ready = 1;

op->reg\_dst\_value = op->reg\_src1\_value | op->imm16;

break;

case OP\_XORI:

op->reg\_dst\_value\_ready = 1;

op->reg\_dst\_value = op->reg\_src1\_value ^ op->imm16;

break;

case OP\_LUI:

op->reg\_dst\_value\_ready = 1;

op->reg\_dst\_value = op->imm16 << 16;

break;

case OP\_LW:

case OP\_LH:

case OP\_LHU:

case OP\_LB:

case OP\_LBU:

op->mem\_addr = op->reg\_src1\_value + op->se\_imm16;

break;

case OP\_SW:

case OP\_SH:

case OP\_SB:

op->mem\_addr = op->reg\_src1\_value + op->se\_imm16;

op->mem\_value = op->reg\_src2\_value;

break;

}

/\* handle branch recoveries at this point \*/

/\*此时处理分支恢复\*/

if (op->branch\_taken)

pipe\_recover(3, op->branch\_dest);

/\* remove from upstream stage and place in downstream stage \*//\*从上游阶段移除并放置在下游阶段\*/

pipe.execute\_op = NULL;

pipe.mem\_op = op;

}

void pipe\_stage\_decode()//指令解码，指令译码

{

/\*如果下游失速，返回（并留下我们的任何输入）\*/

if (pipe.execute\_op != NULL)

return;

/\*如果没有解码操作，则返回\*/

if (pipe.decode\_op == NULL)

return;

/\*抓取op并从舞台输入中移除\*/

Pipe\_Op \*op = pipe.decode\_op;

pipe.decode\_op = NULL;

/\*根据需要设置信息字段（source/dest regs、immediate、jump dest）\*/

uint32\_t opcode = (op->instruction >> 26) & 0x3F;

uint32\_t rs = (op->instruction >> 21) & 0x1F;

uint32\_t rt = (op->instruction >> 16) & 0x1F;

uint32\_t rd = (op->instruction >> 11) & 0x1F;

uint32\_t shamt = (op->instruction >> 6) & 0x1F;

uint32\_t funct1 = (op->instruction >> 0) & 0x1F;

uint32\_t funct2 = (op->instruction >> 0) & 0x3F;

uint32\_t imm16 = (op->instruction >> 0) & 0xFFFF;

uint32\_t se\_imm16 = imm16 | ((imm16 & 0x8000) ? 0xFFFF8000 : 0);

uint32\_t targ = (op->instruction & ((1UL << 26) - 1)) << 2;

op->opcode = opcode;

op->imm16 = imm16;

op->se\_imm16 = se\_imm16;

op->shamt = shamt;

switch (opcode) {

case OP\_SPECIAL:

/\*所有“SPECIAL”指令都是使用ALU和两个源的R类型

\*规则。设置源寄存器和立即值\*/

op->reg\_src1 = rs;

op->reg\_src2 = rt;

op->reg\_dst = rd;

op->subop = funct2;

if (funct2 == SUBOP\_SYSCALL) {

op->reg\_src1 = 2; // v0

op->reg\_src2 = 3; // v1

}

if (funct2 == SUBOP\_JR || funct2 == SUBOP\_JALR) {

op->is\_branch = 1;

op->branch\_cond = 0;

}

break;

case OP\_BRSPEC:

/\* branches that have -and-link variants come here \*/

/\*这里有-和链接变体的分支\*/

op->is\_branch = 1;

op->reg\_src1 = rs;

op->reg\_src2 = rt;

op->is\_branch = 1;

op->branch\_cond = 1; /\* conditional branch \*/

op->branch\_dest = op->pc + 4 + (se\_imm16 << 2);

op->subop = rt;

if (rt == BROP\_BLTZAL || rt == BROP\_BGEZAL) {

/\* link reg \*/

/\*链接寄存器\*/

op->reg\_dst = 31;

op->reg\_dst\_value = op->pc + 4;

op->reg\_dst\_value\_ready = 1;

}

break;

case OP\_JAL:

op->reg\_dst = 31;

op->reg\_dst\_value = op->pc + 4;

op->reg\_dst\_value\_ready = 1;

op->branch\_taken = 1;

/\* fallthrough通过 \*/

case OP\_J:

op->is\_branch = 1;

op->branch\_cond = 0;

op->branch\_taken = 1;

op->branch\_dest = (op->pc & 0xF0000000) | targ;

break;

case OP\_BEQ:

case OP\_BNE:

case OP\_BLEZ:

case OP\_BGTZ:

/\* ordinary conditional branches (resolved after execute) \*/

/\*普通条件分支（执行后解析）\*/

op->is\_branch = 1;

op->branch\_cond = 1;

op->branch\_dest = op->pc + 4 + (se\_imm16 << 2);

op->reg\_src1 = rs;

op->reg\_src2 = rt;

break;

case OP\_ADDI:

case OP\_ADDIU:

case OP\_SLTI:

case OP\_SLTIU:

/\* I-type ALU ops with sign-extended immediates \*/

/\*I型ALU操作，带符号扩展立即数\*/

op->reg\_src1 = rs;

op->reg\_dst = rt;

break;

case OP\_ANDI:

case OP\_ORI:

case OP\_XORI:

case OP\_LUI:

/\* I-type ALU ops with non-sign-extended immediates \*/

/\*I型ALU操作，具有非符号扩展立即数\*/

op->reg\_src1 = rs;

op->reg\_dst = rt;

break;

case OP\_LW:

case OP\_LH:

case OP\_LHU:

case OP\_LB:

case OP\_LBU:

case OP\_SW:

case OP\_SH:

case OP\_SB:

/\* memory ops \*/

/\*存储器操作\*/

op->is\_mem = 1;

op->reg\_src1 = rs;

if (opcode == OP\_LW || opcode == OP\_LH || opcode == OP\_LHU || opcode == OP\_LB || opcode == OP\_LBU) {

/\* load \*/

op->mem\_write = 0;

op->reg\_dst = rt;

}

else {

/\* store \*/

op->mem\_write = 1;

op->reg\_src2 = rt;

}

break;

}

/\*我们将在执行阶段处理reg读取和旁路\*/

/\*将op放置在下游插槽中\*/

pipe.execute\_op = op;

}

void pipe\_stage\_fetch()//取指令

{

/\*如果管道被暂停（我们的输出槽不是空的），返回\*/

if (pipe.decode\_op != NULL)

return;

//延迟操作iCache\_read\_count为全局变量，初值为50

if (iCache\_read\_count > 0 && iCache\_read\_count < 50)

{//正在延迟

iCache\_read\_count--;

//stat\_inst\_fetch++;

//stat\_inst\_retire++;

return;

}

if (iCache\_read\_count==0)

{//延时结束，缺失替换

Pipe\_Op\* op = malloc(sizeof(Pipe\_Op));

memset(op, 0, sizeof(Pipe\_Op));

op->reg\_src1 = op->reg\_src2 = op->reg\_dst = -1;

op->instruction = inst\_cache\_miss(pipe.PC);//指令cache缺失替换操作，取出指令内容

op->pc = pipe.PC;

pipe.decode\_op = op;//将该指令放入流水线

//更新PC

pipe.PC += 4;

stat\_inst\_fetch++;

iCache\_read\_count = 50;//将延迟数字复位

return;

}

//正常先从cache中找

uint32\_t the\_inst = inst\_cache\_hit(pipe.PC);

if (the\_inst == 0x0381CD55)

{//没有命中

iCache\_read\_count--;

//stat\_inst\_fetch++;

//stat\_inst\_retire++;

return;//之后执行延迟

}

else

{//命中

Pipe\_Op\* op = malloc(sizeof(Pipe\_Op));

memset(op, 0, sizeof(Pipe\_Op));

op->reg\_src1 = op->reg\_src2 = op->reg\_dst = -1;

op->instruction = the\_inst;

op->pc = pipe.PC;

pipe.decode\_op = op;

//更新PC

pipe.PC += 4;

stat\_inst\_fetch++;

return;

}

///\* allocate an op and send it down the pipeline. \*/

//Pipe\_Op \*op = malloc(sizeof(Pipe\_Op));

//memset(op, 0, sizeof(Pipe\_Op));

//op->reg\_src1 = op->reg\_src2 = op->reg\_dst = -1;

//op->instruction = mem\_read\_32(pipe.pc);

//op->pc = pipe.pc;

//pipe.decode\_op = op;

///\* update pc \*/

//pipe.pc += 4;

//stat\_inst\_fetch++;

}

## 测试Cache大小代码(pipe.c相同)

### Cache.h

#include"shell.h"

#include <string.h>

#include<math.h>

#define iline 2

#define dline 2

#define isets 2

#define dsets 2

#define isetMax 100010

#define dsetMax 100010

//指令cache

typedef struct

{

int valid;//有效位

int LRU;//判断如何选择替换的行

uint32\_t tag;//标志位

uint32\_t data[8];//数据区域，八个四字节

} inst\_cache\_line;

typedef struct//指令cache的一组

{

inst\_cache\_line lines[iline];//一组2路

} inst\_cache\_sets;

typedef struct//指令cache

{

inst\_cache\_sets sets[isetMax];//2组

} inst\_cache;

//数据cache

typedef struct

{

int valid;

int LRU;

int dirty;// 脏位,判断数据是否需要写回内存

uint32\_t tag;

uint32\_t data[8];

}data\_cache\_line;

typedef struct

{

data\_cache\_line lines[dline];//一组两路

}data\_cache\_sets;

typedef struct

{

data\_cache\_sets sets[isetMax];//8组

}data\_cache;

//判断指令是否命中cache

uint32\_t inst\_cache\_hit(uint32\_t address);

//将块从内存中放入cache

uint32\_t inst\_cache\_miss(uint32\_t address);

//从cache中读取数据

uint32\_t data\_cache\_read(uint32\_t address);

//到cache中写数据

uint32\_t data\_cache\_write(uint32\_t address, uint32\_t value);

### Cache.c

#include "cache.h"

#include "pipe.h"

#include "shell.h"

#include "mips.h"

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <assert.h>

//写延迟标志

int data\_cache\_write\_count = 50;

//读延迟标志

int data\_cache\_read\_count = 50;

//标志初始化

int inst\_cache\_count = 0;

int data\_cache\_count = 0;

//全局变量

inst\_cache iCache;

data\_cache dCache;

//指令cache是否命中，命中返回取得的指令,否则返回0x0381CD55

uint32\_t inst\_cache\_hit(uint32\_t address)//判断是否命中

{

//初始化指令cache

if (inst\_cache\_count == 0)

{

memset(&iCache, 0, sizeof(inst\_cache));

inst\_cache\_count++;

}

//此时内存的设置为 24位tag 3位组号 5位块内字节偏移量

uint32\_t sets\_index = (address<<(27-isets))>>(32-isets);//得到3位的组号

int i;

for (i = 0; i < iline; i++)//检查对应组中的四行

{

uint32\_t tag = iCache.sets[sets\_index].lines[i].tag;//取得每一行的标志位

if (!(tag ^ (address>> (isets+5))))//tag 位相同再检查 valid

{

if (iCache.sets[sets\_index].lines[i].valid == 1)//有效位为1，命中

{

return iCache.sets[sets\_index].lines[i].data[(address & 0x0000001f) >> 2];

//读取address后五位,右移两位缩小四倍，即为四个字节单元的偏移量,对应了data[]

}

}

}

return 0x0381CD55;//如果未命中

}

//指令cache缺失替换操作

uint32\_t inst\_cache\_miss(uint32\_t address)

{

//从内存中取出缺失的块

uint32\_t block\_address = address & 0xffffffe0; //找到对应块中第一个字的地址

uint32\_t mem[8];

int i;

//将内存中对应的指令存入mem

for (i = 0; i < 8; i++)

mem[i] = mem\_read\_32(block\_address + i \* 4);

//将取出的块放入cache中对应的组

uint32\_t sets\_index = (address<<(27-isets))>>(32-isets); //计算组号

//找组中的空行

int j;

int beset=-1;//找到的对应的空行

for (j = 0; j < iline; j++)//有空行则将块放入

{

if (iCache.sets[sets\_index].lines[j].valid == 0)//空位

{

iCache.sets[sets\_index].lines[j].LRU = 0; //LRU初始化为0

iCache.sets[sets\_index].lines[j].valid = 1; //valid 置为有效

iCache.sets[sets\_index].lines[j].tag = address>> (isets+5); // 更新tag

//将块放入数据区

int k;

for (k = 0; k < 8; k++)

iCache.sets[sets\_index].lines[j].data[k] = mem[k];

beset=j;break;

}

}

//找到了空行

if(beset!=-1)

{

//更新空行之外行的LRU++

for (int m = 0; m < iline; m++)

{

if(m==beset)

continue;

else

if(iCache.sets[sets\_index].lines[m].LRU<iline-1)

iCache.sets[sets\_index].lines[m].LRU++;

}

return iCache.sets[sets\_index].lines[j].data[(address & 0x0000001f) >> 2];

}

//没有空行的话，替换LRU最大的行

int max\_LRU\_line = 0;

int max\_LRU = iCache.sets[sets\_index].lines[0].LRU;

int m;

for (m = 1; m < iline; m++) //寻找LRU最大的行

{

int theLRU = iCache.sets[sets\_index].lines[m].LRU;

if (theLRU >= max\_LRU)

{

max\_LRU = theLRU;

max\_LRU\_line = m;

}

}

//替换行更新LRU

for (m = 0; m < iline; m++) //寻找LRU最大的行

{

if(m==max\_LRU\_line)

continue;

else

if(iCache.sets[sets\_index].lines[m].LRU<iline-1)

iCache.sets[sets\_index].lines[m].LRU++;

}

//替换该行内容

iCache.sets[sets\_index].lines[max\_LRU\_line].LRU = 0;

iCache.sets[sets\_index].lines[max\_LRU\_line].valid = 1;

iCache.sets[sets\_index].lines[max\_LRU\_line].tag = address>> (isets+5);

int n;

for (n = 0; n < 8; n++)

iCache.sets[sets\_index].lines[max\_LRU\_line].data[n] = mem[n];

return iCache.sets[sets\_index].lines[max\_LRU\_line].data[(address & 0x0000001f) >> 2];

}

uint32\_t data\_cache\_read(uint32\_t address)

{

if (data\_cache\_count == 0)

{//初始化

memset(&dCache, 0, sizeof(data\_cache));

}

data\_cache\_count++;

uint32\_t sets\_index = (address<<(27-dsets))>>(32-dsets);//计算组号

//检查组中有无tag相同的行

int i;

for (i = 0; i < dline; i++)//判断是否命中

{

uint32\_t tag = dCache.sets[sets\_index].lines[i].tag;

if (!(tag ^ (address>> (dsets+5))))//tag位相同

{

if (dCache.sets[sets\_index].lines[i].valid == 1)//命中

{

return dCache.sets[sets\_index].lines[i].data[(address & 0x0000001f) >> 2];

}

}

}

//没命中,进入延迟50周期

if (data\_cache\_read\_count != 0)

{

data\_cache\_read\_count--;

return 0x0381CD55;

}

data\_cache\_read\_count = 50;//将延迟计数复位

//miss操作

uint32\_t block\_address = address & 0xffffffe0;//找到对应块中第一个字的地址

uint32\_t mem[8];

int j;

for (j = 0; j < 8; j++)

mem[j] = mem\_read\_32(block\_address + j \* 4);

//寻找空行

int k;int beset=-1;

for (k = 0; k < dline; k++)

{

//找到空行装填入缺失块

if (dCache.sets[sets\_index].lines[k].valid == 0)

{

dCache.sets[sets\_index].lines[k].valid = 1;

dCache.sets[sets\_index].lines[k].LRU = 0;

dCache.sets[sets\_index].lines[k].dirty = 0;//初始dirty=0

dCache.sets[sets\_index].lines[k].tag = address>> (dsets+5);

int m;

for (m = 0; m < 8; m++)

dCache.sets[sets\_index].lines[k].data[m] = mem[m];

beset=k;

break;

}

}

//如果找到了空行,更新flag

if(beset!=-1)

{

//更新LRU

for (int m = 0; m < dline; m++)

{

if(m==beset)

continue;

else

if(dCache.sets[sets\_index].lines[m].LRU<dline-1)

dCache.sets[sets\_index].lines[m].LRU++;

}

return dCache.sets[sets\_index].lines[beset].data[(address & 0x0000001f) >> 2];

}

//没有空行，找LRU最小的行

int max\_LRU\_line = 0;

int max\_LRU = dCache.sets[sets\_index].lines[0].LRU;

int n;

for (n = 1; n < dline; n++)

{

int theLRU = dCache.sets[sets\_index].lines[n].LRU;

if (theLRU >= max\_LRU)

{

max\_LRU = theLRU;

max\_LRU\_line = n;

}

}

//替换行更新LRU

for (int m = 0; m < dline; m++)

{

if(m==max\_LRU\_line)

continue;

else

if(dCache.sets[sets\_index].lines[m].LRU<dline-1)

dCache.sets[sets\_index].lines[m].LRU++;

}

//根据最大LRU行的dirty确定它是否需要写回

if (dCache.sets[sets\_index].lines[max\_LRU\_line].dirty == 1)//需要写回

{

uint32\_t sets\_index\_13 = sets\_index;

//此时进行位的拼接

sets\_index\_13 << 5;//低5位是0，中间八位是组号，高位是0 与 高位是tag 其余为0 进行拼凑

uint32\_t block\_address = (dCache.sets[sets\_index].lines[max\_LRU\_line].tag << (dsets+5)) | sets\_index\_13;

for (i = 0; i < 8; i++)

mem\_write\_32(block\_address + i \* 4, dCache.sets[sets\_index].lines[max\_LRU\_line].data[i]);

}

//将该块替换

dCache.sets[sets\_index].lines[max\_LRU\_line].valid = 1;

dCache.sets[sets\_index].lines[max\_LRU\_line].LRU = 0;

dCache.sets[sets\_index].lines[max\_LRU\_line].dirty = 0;

dCache.sets[sets\_index].lines[max\_LRU\_line].tag = address>> (dsets+5);

for (i = 0; i < 8; i++)

dCache.sets[sets\_index].lines[max\_LRU\_line].data[i] = mem[i];

return dCache.sets[sets\_index].lines[max\_LRU\_line].data[(address & 0x0000001f) >> 2];

}

uint32\_t data\_cache\_write(uint32\_t address, uint32\_t value)

{

if (data\_cache\_count == 0)//初始化

{

memset(&dCache, 0, sizeof(data\_cache));

data\_cache\_count++;

}

uint32\_t sets\_index = (address<<(27-dsets))>>(32-dsets);//计算组号

//检查组中有无tag相同的行

int i;

for (i = 0; i < dline; i++)

{

uint32\_t tag = dCache.sets[sets\_index].lines[i].tag;

if (!(tag ^ (address>> (dsets+5))))//tag相同

{

if (dCache.sets[sets\_index].lines[i].valid == 1)//命中

{

dCache.sets[sets\_index].lines[i].valid = 1;

dCache.sets[sets\_index].lines[i].dirty = 1;

dCache.sets[sets\_index].lines[i].data[(address & 0x0000001f) >> 2] = value;

return 1;

}

}

}

//没命中,进入延迟

if (data\_cache\_write\_count != 0)

{

data\_cache\_write\_count--;

return 0;

}

data\_cache\_write\_count=50;

//miss 操作

//从内存中取出缺失块

uint32\_t block\_address = address & 0xffffffe0;

uint32\_t mem[8];

int j;

for (j = 0; j < 8; j++)

mem[j] = mem\_read\_32(block\_address + j \* 4);

//寻找空行

int k;int beset=-1;

for (k = 0; k < dline; k++)

{

//找到空行填入缺失块

if (dCache.sets[sets\_index].lines[k].valid == 0)

{

dCache.sets[sets\_index].lines[k].valid = 1;

dCache.sets[sets\_index].lines[k].LRU = 0;

dCache.sets[sets\_index].lines[k].dirty = 1;//脏位

dCache.sets[sets\_index].lines[k].tag = address>> (dsets+5);

int m;

for (m = 0; m < 8; m++)

dCache.sets[sets\_index].lines[k].data[m] = mem[m];

dCache.sets[sets\_index].lines[k].data[(address & 0x0000001f) >> 2] = value;

beset=k;

break;

}

}

//找到空行更新LRU

if(beset!=-1)

{

//更新LRU

for (int m = 0; m < dline; m++)

{

if(m==beset)

continue;

else

if(dCache.sets[sets\_index].lines[m].LRU<dline-1)

dCache.sets[sets\_index].lines[m].LRU++;

}

return dCache.sets[sets\_index].lines[j].data

[(address & 0x0000001f) >> 2];

return dCache.sets[sets\_index].lines[k].data[(address & 0x0000001f) >> 2];

}

//没有空行找最大LRU

int max\_LRU\_line = 0;

int max\_LRU = dCache.sets[sets\_index].lines[0].LRU;

int n;

for (n = 1; n < dline; n++)

{

int theLRU = dCache.sets[sets\_index].lines[n].LRU;

if (theLRU >= max\_LRU)

{

max\_LRU = theLRU;

max\_LRU\_line = n;

}

}

//替换行更新LRU

for (int m = 0; m < dline; m++)

{

if(m==max\_LRU\_line)

continue;

else

if(dCache.sets[sets\_index].lines[m].LRU<dline-1)

dCache.sets[sets\_index].lines[m].LRU++;

}

//根据最小LRU行的dirty确定它是否需要写回

if (dCache.sets[sets\_index].lines[max\_LRU\_line].dirty == 1)

{

uint32\_t sets\_index\_13 = sets\_index;

sets\_index\_13 << 5;//低5位是0，中间八位是组号，高位是0

uint32\_t block\_address = (dCache.sets[sets\_index].lines[max\_LRU\_line].tag << (dsets+5)) | sets\_index\_13;

for (i = 0; i < 8; i++)

mem\_write\_32(block\_address + i \* 4, dCache.sets[sets\_index].lines[max\_LRU\_line].data[i]);

}

//将该块替换

dCache.sets[sets\_index].lines[max\_LRU\_line].valid = 1;

dCache.sets[sets\_index].lines[max\_LRU\_line].LRU = 0;

dCache.sets[sets\_index].lines[max\_LRU\_line].dirty = 1;

dCache.sets[sets\_index].lines[max\_LRU\_line].tag = address>> (dsets+5);

for (i = 0; i < 8; i++)

dCache.sets[sets\_index].lines[max\_LRU\_line].data[i] = mem[i];

dCache.sets[sets\_index].lines[max\_LRU\_line].data[(address & 0x0000001f) >> 2] = value;

return 1;//成功写入返回1

}

## LFU的cache.c代码

#include "cache.h"

#include "pipe.h"

#include "shell.h"

#include "mips.h"

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <assert.h>

int data\_cache\_write\_count = 50;

int data\_cache\_read\_count = 50;

//标志初始化

int inst\_cache\_count = 0;

int data\_cache\_count = 0;

inst\_cache iCache;

data\_cache dCache;

//指令cache是否命中，命中返回指令

uint32\_t is\_inst\_cache\_hit(uint32\_t address)//判断是否命中

{

if (inst\_cache\_count == 0)

{

memset(&iCache, 0, sizeof(inst\_cache));

inst\_cache\_count++;

}

uint32\_t sets\_index = (address << 21) >> 26;//得到六位的组号

int i;

for (i = 0; i < 4; i++)//检查对应组中的四行

{

uint32\_t tag = iCache.sets[sets\_index].lines[i].tag;//取得每一行的标志位

if (!(tag ^ (address >> 11)))//tag 位相同再检查 valid

{

if (iCache.sets[sets\_index].lines[i].valid == 1)//有效位为一，命中

{

if (iCache.sets[sets\_index].lines[i].LFU < 3)//需要更新LFU位

iCache.sets[sets\_index].lines[i].LFU++;

return iCache.sets[sets\_index].lines[i].data[(address & 0x0000001f) >> 2];

//address & 0x0000001f为address后五位（块内偏移量）

//右移两位缩小四倍，即为四个字节单元的偏移量

}

}

}

return 0x0381CD55;//如果未命中

}

//指令cache缺失替换操作

uint32\_t inst\_cache\_miss(uint32\_t address)

{

//从内存中取出缺失的块

uint32\_t block\_address = address & 0xffffffe0; //计算块地址，块中第一个字的地址

uint32\_t mem[8];

int i;

for (i = 0; i < 8; i++)

mem[i] = mem\_read\_32(block\_address + i \* 4);

//将取出的块放入cache中对应的组

uint32\_t sets\_index = (address << 21) >> 26; //计算组号

//找组中的空行

int j;

for (j = 0; j < 4; j++)//有空行则将块放入

{

if (iCache.sets[sets\_index].lines[j].valid == 0)//空位

{

iCache.sets[sets\_index].lines[j].LFU = 0; //LFU初始化为0

iCache.sets[sets\_index].lines[j].valid = 1; //valid 置为有效

iCache.sets[sets\_index].lines[j].tag = address >> 11; // 更新tag

//将块放入数据区

int k;

for (k = 0; k < 8; k++)

iCache.sets[sets\_index].lines[j].data[k] = mem[k];

return iCache.sets[sets\_index].lines[j].data[(address & 0x0000001f) >> 2];//返回数据

}

}

//没有空行的话，替换LFU最小的行

int min\_LFU\_line = 0;

int min\_LFU = iCache.sets[sets\_index].lines[0].LFU;

int m;

for (m = 1; m < 4; m++) //寻找LFU最小的行

{

int theLFU = iCache.sets[sets\_index].lines[m].LFU;

if (theLFU < min\_LFU)

{

min\_LFU = theLFU;

min\_LFU\_line = m;

}

}

//替换该行内容

iCache.sets[sets\_index].lines[min\_LFU\_line].LFU = 0;

iCache.sets[sets\_index].lines[min\_LFU\_line].valid = 1;

iCache.sets[sets\_index].lines[min\_LFU\_line].tag = address >> 11;

int n;

for (n = 0; n < 8; n++)

iCache.sets[sets\_index].lines[min\_LFU\_line].data[n] = mem[n];

return iCache.sets[sets\_index].lines[min\_LFU\_line].data[(address & 0x0000001f) >> 2];

}

uint32\_t data\_cache\_read\_32(uint32\_t address)

{

printf("address1:%x\n", address);

if (data\_cache\_count == 0)

{//初始化

memset(&dCache, 0, sizeof(data\_cache));

}

data\_cache\_count++;

uint32\_t sets\_index = (address << 19) >> 24;//计算组号

//检查组中有无tag相同的行

int i;

for (i = 0; i < 8; i++)//判断是否命中

{

uint32\_t tag = dCache.sets[sets\_index].lines[i].tag;

if (!(tag ^ (address >> 13)))

{//tag位相同

if (dCache.sets[sets\_index].lines[i].valid == 1)

{//命中

//更新LFU

if (dCache.sets[sets\_index].lines[i].LFU < 7)

dCache.sets[sets\_index].lines[i].LFU++;

return dCache.sets[sets\_index].lines[i].data[(address & 0x0000001f) >> 2];

}

}

}

//没命中

if (data\_cache\_read\_count != 0)

{

data\_cache\_read\_count--;

return 0x0381CD55;

}

data\_cache\_read\_count = 50;//将延迟计数复位

//miss操作

uint32\_t block\_address = address & 0xffffffe0;//从内存中取出缺失的块

uint32\_t mem[8];

int j;

for (j = 0; j < 8; j++)

mem[j] = mem\_read\_32(block\_address + j \* 4);

//寻找空行

int k;

for (k = 0; k < 8; k++)

{

//找到空行装填入缺失块

if (dCache.sets[sets\_index].lines[k].valid == 0)

{

dCache.sets[sets\_index].lines[k].valid = 1;

dCache.sets[sets\_index].lines[k].LFU = 0;

dCache.sets[sets\_index].lines[k].dirty = 0;

dCache.sets[sets\_index].lines[k].tag = address >> 13;

int m;

for (m = 0; m < 8; m++)

dCache.sets[sets\_index].lines[k].data[m] = mem[m];

return dCache.sets[sets\_index].lines[k].data[(address & 0x0000001f) >> 2];

}

}

//没有空行，找LFU最小的行

int min\_LFU\_line = 0;

int min\_LFU = dCache.sets[sets\_index].lines[0].LFU;

int n;

for (n = 1; n < 8; n++)

{

int theLFU = dCache.sets[sets\_index].lines[n].LFU;

if (theLFU < min\_LFU)

{

min\_LFU = theLFU;

min\_LFU\_line = n;

}

}

//根据最小LFU行的dirty确定它是否需要写回

if (dCache.sets[sets\_index].lines[min\_LFU\_line].dirty == 1)

{

uint32\_t sets\_index\_13 = sets\_index;

sets\_index\_13 << 5;//低5位是0，中间八位是组号，高位是0

uint32\_t block\_address = (dCache.sets[sets\_index].lines[min\_LFU\_line].tag << 13) | sets\_index\_13;

for (i = 0; i < 8; i++)

mem\_write\_32(block\_address + i \* 4, dCache.sets[sets\_index].lines[min\_LFU\_line].data[i]);

}

//将该块替换

dCache.sets[sets\_index].lines[min\_LFU\_line].valid = 1;

dCache.sets[sets\_index].lines[min\_LFU\_line].LFU = 0;

dCache.sets[sets\_index].lines[min\_LFU\_line].dirty = 0;

dCache.sets[sets\_index].lines[min\_LFU\_line].tag = address >> 13;

for (i = 0; i < 8; i++)

dCache.sets[sets\_index].lines[min\_LFU\_line].data[i] = mem[i];

return dCache.sets[sets\_index].lines[min\_LFU\_line].data[(address & 0x0000001f) >> 2];

}

uint32\_t data\_cache\_write\_32(uint32\_t address, uint32\_t value)

{

if (data\_cache\_count == 0)//初始化

{

memset(&dCache, 0, sizeof(data\_cache));

data\_cache\_count++;

}

uint32\_t sets\_index = (address << 19) >> 24;//计算组号

//检查组中有无tag相同的行

int i;

for (i = 0; i < 8; i++)

{

uint32\_t tag = dCache.sets[sets\_index].lines[i].tag;

if (!(tag ^ (address >> 13)))

{//tag相同

if (dCache.sets[sets\_index].lines[i].valid == 1)

{//命中

//更新LFU

if (dCache.sets[sets\_index].lines[i].LFU < 7)

dCache.sets[sets\_index].lines[i].LFU++;

dCache.sets[sets\_index].lines[i].valid = 1;

dCache.sets[sets\_index].lines[i].dirty = 1;

dCache.sets[sets\_index].lines[i].data[(address & 0x0000001f) >> 2] = value;

return 1;

}

}

}

//没命中

if (data\_cache\_write\_count != 0)

{

data\_cache\_write\_count--;

return 0;

}

data\_cache\_write\_count=50;

//miss 操作

//从内存中取出缺失块

uint32\_t block\_address = address & 0xffffffe0;

uint32\_t mem[8];

int j;

for (j = 0; j < 8; j++)

mem[j] = mem\_read\_32(block\_address + j \* 4);

//寻找空行

int k;

for (k = 0; k < 8; k++)

{

//找到空行填入缺失块

if (dCache.sets[sets\_index].lines[k].valid == 0)

{

dCache.sets[sets\_index].lines[k].valid = 1;

dCache.sets[sets\_index].lines[k].LFU = 0;

dCache.sets[sets\_index].lines[k].dirty = 1;//脏位

dCache.sets[sets\_index].lines[k].tag = address >> 13;

int m;

for (m = 0; m < 8; m++)

dCache.sets[sets\_index].lines[k].data[m] = mem[m];

dCache.sets[sets\_index].lines[k].data[(address & 0x0000001f) >> 2] = value;

return 1;

}

}

//没有空行找最小

int min\_LFU\_line = 0;

int min\_LFU = dCache.sets[sets\_index].lines[0].LFU;

int n;

for (n = 1; n < 8; n++)

{

int theLFU = dCache.sets[sets\_index].lines[n].LFU;

if (theLFU < min\_LFU)

{

min\_LFU = theLFU;

min\_LFU\_line = n;

}

}

//根据最小LFU行的dirty确定它是否需要写回

if (dCache.sets[sets\_index].lines[min\_LFU\_line].dirty == 1)

{

uint32\_t sets\_index\_13 = sets\_index;

sets\_index\_13 << 5;//低5位是0，中间八位是组号，高位是0

uint32\_t block\_address = (dCache.sets[sets\_index].lines[min\_LFU\_line].tag << 13) | sets\_index\_13;

for (i = 0; i < 8; i++)

mem\_write\_32(block\_address + i \* 4, dCache.sets[sets\_index].lines[min\_LFU\_line].data[i]);

}

//将该块替换

dCache.sets[sets\_index].lines[min\_LFU\_line].valid = 1;

dCache.sets[sets\_index].lines[min\_LFU\_line].LFU = 0;

dCache.sets[sets\_index].lines[min\_LFU\_line].dirty = 1;

dCache.sets[sets\_index].lines[min\_LFU\_line].tag = address >> 13;

for (i = 0; i < 8; i++)

dCache.sets[sets\_index].lines[min\_LFU\_line].data[i] = mem[i];

dCache.sets[sets\_index].lines[min\_LFU\_line].data[(address & 0x0000001f) >> 2] = value;

return 1;//成功写入返回1

}