山东大学<u>　　计算机科学与技术　</u>学院

<u>　　计算机系统原理　</u>课程实验报告

<table>
<tr><td>学号：202200400053</td><td>姓名：王宇涵</td><td>班级：　2 班</td></tr>
<tr><td colspan="3">实验题目：设计 MIPS 五级流水线模拟器中的 Cache</td></tr>
<tr><td>实验学时：2</td><td colspan="2">实验日期：　　2023-12-05</td></tr>
</table>

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：

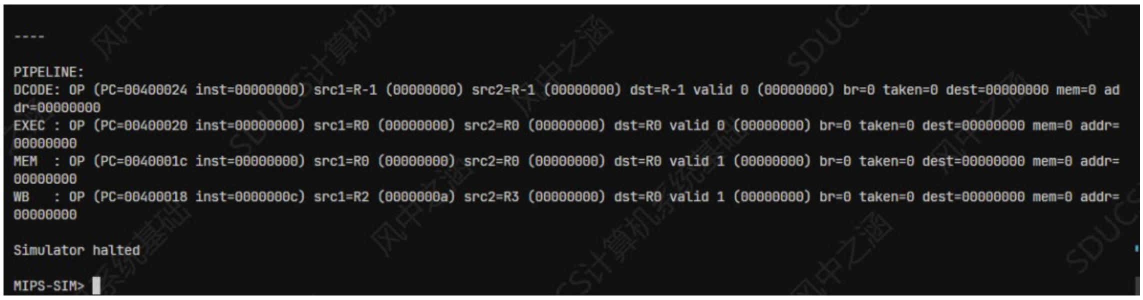
支持 MIPS 指令集的流水线计时模拟器

Linux 虚拟机进行 cache 扩展仿真

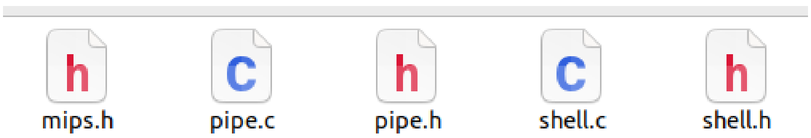
实验步骤与内容：

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 ;

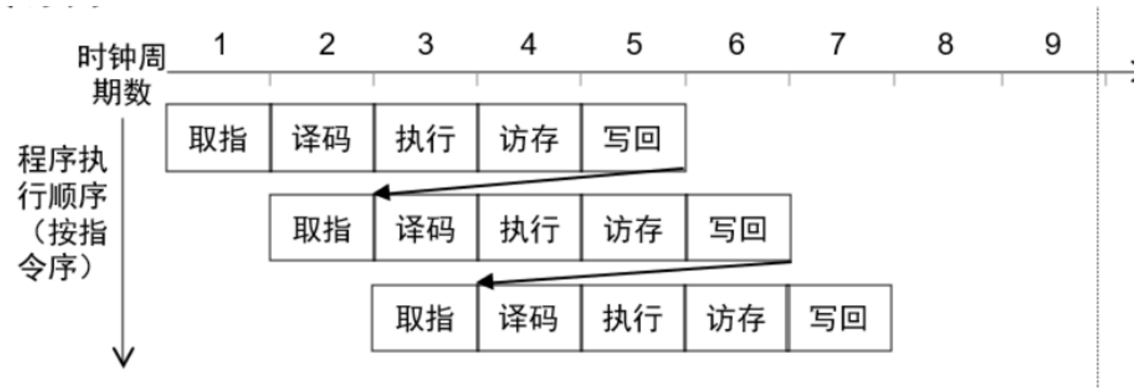
4.阅读 shell.c

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);
4. 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

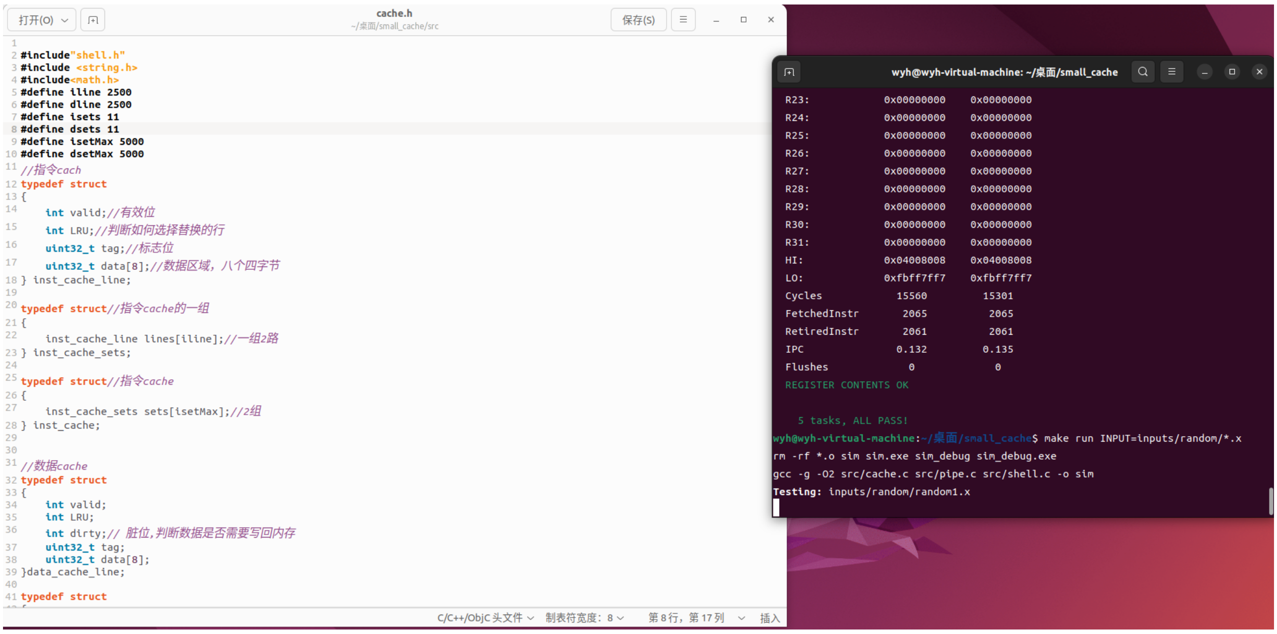
整体思路和 inst_cache 相同 :
首先我们需要判断内存的组成,对于 inst_cache, 8 位组号(64 块=2^6) ,5 位块内偏移量(32 字节=2^5),19 位 tag(32-8-5),因此内存组成为 19~8~5.
data_cache_read 函数: 传入内存地址,从 cache 中读取数据,如果没有命中将内存对应的块写入 cache 中.
整体与 inst_cache_read 类似, 不加以赘述.
data_cache_write 函数: 传入内存地址和值, 将 cache 中对应的数据修改,如果没有命中将内存对应的块写入 cache 中,并修改对应的值,注意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]);

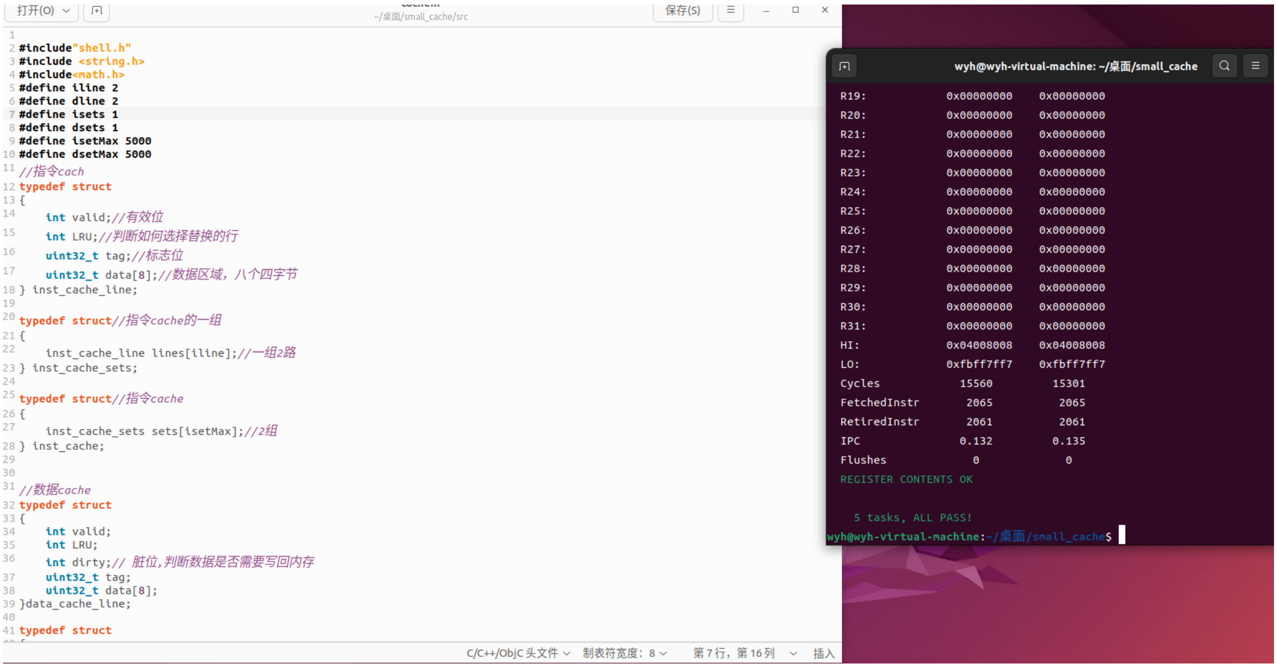
8.延迟的实现

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

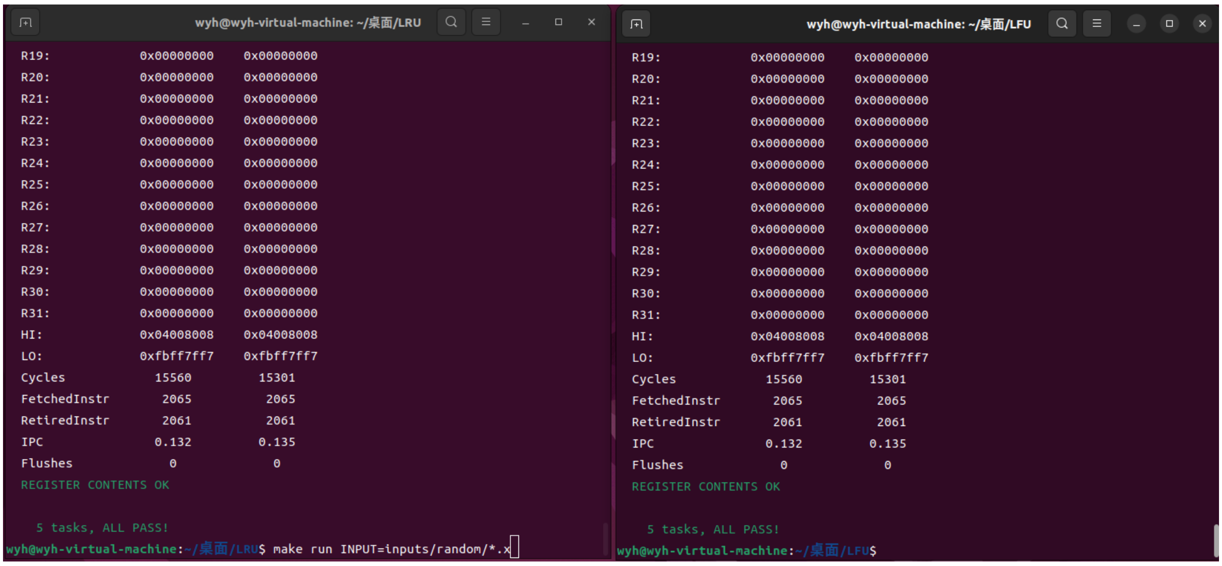
Small_cache :将 inst_cache 和 data_cache 的组数设为 2,行数都设为 2
Large_cache:将 inst_cache 和 data_cache 的组数设为 11,行数都设为 2500
结果:两者测试不完全相同,对于少量数据来说对 ipc 结果的影响不是很大,但是 large_cache 进行操作的时候具有卡顿,表示每一个任务执行的时间比较长,可能是由于行数设置比较多,每次都要遍历所有行的原因.





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
}
```