计算机科学与技术学院

计算机系统原理课程实验报告

实验题目： 设计 MIPS 五级流水线模拟器中的 Cache	**学号：** 202200130048
班级： 6	**姓名：** 陈静雯
Email： 1205037094@qq.com	

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。

实验软件和硬件环境：

Vm+ubuntu

实验原理和方法：

LRU，最近最少替换：

利用时间局限性，替换最近最少访问

实现办法：通过给每个 cache 行设定一个 LRU 计数器，根据计数值来记录这些主存块的使用情况。每一次被访问，则该行清零，其他行加 1；当 lru 队列满时，对 lru 队列中计数器最大行即最少使用的 cache 行进行替换。

写回策略（Write Back）：

①读：

若命中，直接返回其数据；

当未命中时，先在 cache 中根据映射关系找一个 cache 行，若 cache 组已满，找 lru 最大的进行替换；

判断替换的 cache 行是不是脏数据（是否修改）。

如果是脏数据，将该数据写回主存中，再读取需要的主存块；

如果不是脏数据，直接从主存中读需要的主存块到 cache 块中

返回数据。

②写：

若命中，直接将新数据写入缓存，并且标记 dirty 位为 1（被修改）；

若未命中，同读中的步骤一致，将数据读到 cache 后，将新数据写入缓存块，并标记 dirty=1。

实验步骤：

1. 读 shell.c 源码，发现核心函数是 mem_read32 和 mem_write_32

2. 根据 cache 所学知识，以及实验要求，确定 cache 的组成

由于块大小都为 32B，则块内偏移需要 5 位，即主存地址 0~4 位为块内偏移地址

InstructionCache 中为 4 路组相联，则 8KB/(32*4)B=64 组，即组号需要 6 位，所以主存地址 5~10 位为组号，剩余的高位为 Tag 位共 21 位，有效位 1 位，由 4 路组相联，则 lru 位需要表示 4 种状态，即为 2 位

DataCache 中为 8 路组相联，则 64KB/(32*8)B=256 组，即组号需要 8 位，所以主存地址 5~12 位为组号，剩余的高位为 Tag 位共 19 位，有效位 1 位，由 8 路组相联，则 lru 位需要表示 8 种状态，即为 3 位，采用写回策略，故增加一脏位 dirty 1 位

我们把 cache 的代码设计分为两部分：

取指阶段的 instructionCache 和访存阶段的 dataCache。

两个 cache 大体上结构相同，但 dataCache 涉及到写回，所以多了一个脏位（dirty）的判定，以及根据两个 cache 的 Size 要求对具体的位偏移做了不同的设计。

3.对于 instructionCache：

我们写了一个初始化函数 init，用于初始化 instructionCache，read 对指令地址进行读取，如果 cache 命中，则返回相应的数据，若 cache 未命中，则用写回策略，通过 write 去主存里面读数据，再返回。用 valid 有效位和 tag 位的比较来检测是否 cache 命中，并通过 LRU 对 cache 进行更新。

4.对于 dataCache：

同 instructionCache，我们也写了一个初始化函数 init，用于初始化 dataCache，read 对指令地址进行读数据，如果 cache 命中，则返回相应的数据，若 cache 未命中，则通过 load 将主存中的数据写入 cache 中。Write_val 函数用来实现写数据进 cache。用 valid 有效位和 tag 位的比较来检测是否 cache 命中，并通过 LRU 对 cache 进行更新。

dirty 位，用来判别写数据进 cache 时目标槽（slot）即 cache 行是否为脏，如果为脏，还需要先将之前的数据写回主存后再写入新数据。并在末尾对 valid、tag、lru 和 dirty 四个位进行更新，以同步此次操作的结果。

5.加入延迟机制：由于实验要求需要考虑延时设计（即在读取指令的时候若 cache 未命中，去主存里面找的时候要花 50 个 cycle），我们加入了 waiting_data 计数器，未命中的时候触发 waiting_data，并在该指令往后的 50 次 cycle 里面对 waiting_data 进行自减直到为 0，才把读到的指令释放出来。

6.初始化：在上文已经提到各个 cache 的初始化函数，最后将他们放入 pipe.c 里面的 init 函数内，实现 cache 启动的初始化。

结论分析与体会：

1.instructionCache.h

```
#ifndef CACHE
#define CACHE

#include "stdio.h"
#include "mips.h"
#include "stdbool.h"
#include "stdint.h"
typedef struct //数据区，相当于一个块
{
    uint8_t mem[32];//一个块有 32byte（uint8_t 表示别名，无符号八位整数）
}Data;

typedef struct //cache 行
{
    bool valid;//是否可用
    uint8_t lru;//lru 位,因为 4 路组相连，所以取后两位即可
    uint16_t line;//9 位行号
    Data data;//数据区
}Cache_line;
```

```
typedef struct //每组 cache
{
    Cache_line cache_line[4];//一组有四行
}Cache_set;

typedef struct
{
    Cache_set cache_set[64];//一共有 64 组
}Cache;

extern Cache cache;
void init();
void get_set(uint32_t address);
void cache_write(uint32_t address);//写入一个块
uint32_t cache_read(uint32_t address);//从给定本应读取的内存地址中读取数据
void cache_replace(Cache_line cache_line);//给定行，进行替换

extern int waiting;

#endif
```

2. instructionCache.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>

#define MEM_DATA_START  0x10000000
#define MEM_DATA_SIZE   0x00100000
#define MEM_TEXT_START  0x00400000
#define MEM_TEXT_SIZE   0x00100000
#define MEM_STACK_START 0x7ff00000
#define MEM_STACK_SIZE  0x00100000
#define MEM_KDATA_START 0x90000000
#define MEM_KDATA_SIZE  0x00100000
#define MEM_KTEXT_START 0x80000000
#define MEM_KTEXT_SIZE  0x00100000
Cache cache;

int waiting = 0;

void init(){
    int i =0, j = 0;
    for(i=0;i<64;i++)
    {
        for(j=0;j<4;j++)
        {
```

```
                cache.cache_set[i].cache_line[j].valid = 0;
                cache.cache_set[i].cache_line[j].lru = 0;
            }
        }
    }

    void get_set(uint32_t address)//给一个主存地址，取出对应的数
    {
        uint32_t tmp = address & 0xFFFFF;//取后 20 位
        uint32_t line = tmp >> 11 ;//取 12-20 位
        uint32_t set = (tmp >>5) &(0x6F);//取所对应的组号
        uint32_t begin = (address >> 5)<<5;
        printf("%d %d %d\n",tmp,line,set);
    }

    uint32_t cache_read(uint32_t address)
    {
        uint32_t tmp = address & 0xFFFFF;//取后 20 位
        uint32_t line = tmp >> 11 ;//取 12-20 位
        uint32_t set = (tmp >>5) &(0x3F);//取所对应的组号
        uint32_t inneraddress = address & 0x1f;//块内偏移量
    //    printf("inneraddress = %08x\n",inneraddress);
        uint32_t i = 0;
    //    printf("ok_read\n");
        for(i=0;i<4;i++)
        {
            if(cache.cache_set[set].cache_line[i].valid==1)
            {
                if(cache.cache_set[set].cache_line[i].line == line)//比较
                {
                    uint32_t j = inneraddress;
                    uint32_t word =
(cache.cache_set[set].cache_line[i].data.mem[j]<<0)  |

(cache.cache_set[set].cache_line[i].data.mem[j+1]<<8)|

(cache.cache_set[set].cache_line[i].data.mem[j+2]<<16) |

(cache.cache_set[set].cache_line[i].data.mem[j+3]<<24);
                    // printf("hit!");命中
                    //system("pause");
                    return word;

                }

            }
        }
        //printf("nohit!");
        waiting = 49;
        cache_write(address);
        return 0xffffffff;
```

```
}

void cache_write(uint32_t address)//写入一个块
{
//    printf("oknow\n");
    uint32_t tmp = address & 0xFFFFF;//取后 20 位
    uint32_t line = tmp >> 11 ;//取 12-20 位
    uint32_t set = (tmp >>5) &(0x3F);//取所对应的组号
    uint32_t begin = (address >> 5);//块头
//    printf("set = %08x\n",set);
    begin = begin << 5;
    int i = 0;
    int tobeset = -1;

    for(i=0;i<4;i++)
    {
        if(cache.cache_set[set].cache_line[i].valid==0)//找到一个空的块
        {
            cache.cache_set[set].cache_line[i].valid = 1;
            cache.cache_set[set].cache_line[i].line = line;//对应第几个群
                                //printf("%d %d\n",set,i);
            tobeset = i;//要写的行
            cache.cache_set[set].cache_line[i].lru = 0;
            break;
        }
    }
    for(i=0;i<4;i++)//处理 lru
    {
         if(tobeset==i)continue; //要写的 cache 行 lru 为 0
         else
            {
                if(cache.cache_set[set].cache_line[i].lru<3)
                  cache.cache_set[set].cache_line[i].lru++;//不是该行,lru++
            }
    }
    if(tobeset==-1){   //cache 满了，进行替换
        uint8_t maxn = 0;
        int i;
        for(i=0;i<4;i++)
        {
            if(cache.cache_set[set].cache_line[i].lru==0)continue;
            else
            {
                if(cache.cache_set[set].cache_line[i].lru<3)
                    cache.cache_set[set].cache_line[i].lru++;
                if(maxn<=cache.cache_set[set].cache_line[i].lru)
                {
                    maxn = cache.cache_set[set].cache_line[i].lru;
                                                        //找到 lru 最大的
                    tobeset = i;
                }
```

```
                }
            }

            cache.cache_set[set].cache_line[tobeset].valid = 1;
            cache.cache_set[set].cache_line[tobeset].line = line;//对应第几个群
            cache.cache_set[set].cache_line[tobeset].lru = 0;

        }

     //   printf("%08x",begin);
        for(i = 0;i<8;i++)//读取 32 个字节作为一块，替换
        {
            uint32_t word = mem_read_32(begin);
            cache.cache_set[set].cache_line[tobeset].data.mem[i*4]   =
(uint8_t)(word>>0)&0xff;
            cache.cache_set[set].cache_line[tobeset].data.mem[i*4+1] =
(uint8_t)(word>>8)&0xff;
            cache.cache_set[set].cache_line[tobeset].data.mem[i*4+2] =
(uint8_t)(word>>16)&0xff;
            cache.cache_set[set].cache_line[tobeset].data.mem[i*4+3] =
(uint8_t)(word>>24)&0xff;
            begin+=4;
        }

    }
```

3.datacache.h

```
#ifndef CACHEDATA
#define CACHEDATA

#include "stdio.h"
#include "mips.h"
#include "stdbool.h"
#include "stdint.h"
typedef struct //数据区，相当于一个块
{
    uint8_t mem[32];//一个块有 32byte
}Data_1;
typedef struct //每行 cache
{
    bool valid;//是否可用
    bool dirty;
    uint8_t lru;//lru 位,因为 8 路组相联，所以取后三位即可
    uint32_t line;//19 位行号 tag
    Data_1 data;//数据区
}Cache_data_line;
typedef struct //每组 cache
{
    Cache_data_line cache_data_line[8];//一组有八行
}Cache_data_set;
typedef struct
```

```
{
    Cache_data_set cache_data_set[256];//一共有 256 组
}Cache_data;
extern Cache_data cache_data;
void data_init();
void cache_data_write_val(uint32_t address,uint32_t write);//写入一个块
void cache_data_load(uint32_t address);//没有命中，要从主存读数据
uint32_t cache_data_read(uint32_t address);//从给定本应读取的内存地址中读取数据
extern int waiting_data;
#endif
```

4.cache_data

```
    #include "datacache.h"
    #include "pipe.h"
    #include "shell.h"
    #include "mips.h"
    #include <stdio.h>
    #include <string.h>
    #include <stdlib.h>
    #include <assert.h>

    #define MEM_DATA_START  0x10000000
    #define MEM_DATA_SIZE   0x00100000
    #define MEM_TEXT_START  0x00400000
    #define MEM_TEXT_SIZE   0x00100000
    #define MEM_STACK_START 0x7ff00000
    #define MEM_STACK_SIZE  0x00100000
    #define MEM_KDATA_START 0x90000000
    #define MEM_KDATA_SIZE  0x00100000
    #define MEM_KTEXT_START 0x80000000
    #define MEM_KTEXT_SIZE  0x00100000
    Cache_data cache_data;
    int waiting_data = 0;
    void data_init(){
        int i =0, j = 0;
        for(i=0;i<256;i++)
        {
            for(j=0;j<8;j++)
            {
                cache_data.cache_data_set[i].cache_data_line[j].valid = 0;
                cache_data.cache_data_set[i].cache_data_line[j].lru = 0;
                cache_data.cache_data_set[i].cache_data_line[j].dirty = 0;
            }
        }
    }
    uint32_t cache_data_read(uint32_t address)//在 cache 读数据
    {
    //    printf("read address = 0x%08x\n",address);
        uint32_t line = address >> 13 ;//取前 19 位
        uint32_t set = (address >>5) &(0xFF);//取所对应的组号
```

```
        uint32_t inneraddress = address & 0x1f;
        uint32_t i = 0;
    //    printf("line = %d set = %d inneraddress
= %d\n",line,set,inneraddress);
        for(i=0;i<8;i++)
        {
            if(cache_data.cache_data_set[set].cache_data_line[i].valid==1)
                                                //当前位有效
            {
                if(cache_data.cache_data_set[set].cache_data_line[i].line ==
line)//当前行号和对应的主存行号相同，即命中
                {
                    uint32_t j = inneraddress;//内部地址
                    uint32_t word =
(cache_data.cache_data_set[set].cache_data_line[i].data.mem[j]<<0)    |

(cache_data.cache_data_set[set].cache_data_line[i].data.mem[j+1]<<8)  |

(cache_data.cache_data_set[set].cache_data_line[i].data.mem[j+2]<<16) |

(cache_data.cache_data_set[set].cache_data_line[i].data.mem[j+3]<<24);
                    return word;

                }

            }
        }
        waiting_data = 49;
        cache_data_load(address);
        return 0xffffffff;
    }

    void cache_data_load(uint32_t address)//没有命中
    //从主存地址读出一个块存到 cache 里
    {
        //内存地址 = 行号 19 位 + 组号 8 位 + 块内地址 5 位
        uint32_t line = address >> 13 ;//取前 19 位，要写入的 line
        uint32_t set = (address >>5) &(0xFF);//取所对应的组号
        uint32_t begin = (address >> 5);//块头
        begin = begin << 5;
        int i = 0;
        int tobeset = -1;//需要替换
        uint32_t getline = 0;
        uint32_t getset = 0;
        for(i=0;i<8;i++)
        {
            if(cache_data.cache_data_set[set].cache_data_line[i].valid==0)
                                        //不用替换，直接写入 cache
            {
                cache_data.cache_data_set[set].cache_data_line[i].valid = 1;
                cache_data.cache_data_set[set].cache_data_line[i].line = line;
```

```
                                                    //对应第几个群

                tobeset = i;//要写的列

                cache_data.cache_data_set[set].cache_data_line[i].lru = 0;
                break;
            }
        }

        for(i=0;i<8;i++) //更新没有被使用的 cache 行的 lru
        {
             if(tobeset==i)continue;
             else
                {
                    if(cache_data.cache_data_set[set].cache_data_line[i].lru<7)
                    cache_data.cache_data_set[set].cache_data_line[i].lru++;
                }
        }

        if(tobeset==-1){ //需要替换，找 lru 最大的进行替换
            uint8_t maxn = 0;
            int i;
            for(i=0;i<8;i++)
            {

if(cache_data.cache_data_set[set].cache_data_line[i].lru==0)continue;
                else
                {
                    if(cache_data.cache_data_set[set].cache_data_line[i].lru<7)

cache_data.cache_data_set[set].cache_data_line[i].lru++;

if(maxn<=cache_data.cache_data_set[set].cache_data_line[i].lru)
                    {
                        maxn =
cache_data.cache_data_set[set].cache_data_line[i].lru;
                        tobeset = i;//将要替换的行
                        getline =
cache_data.cache_data_set[set].cache_data_line[i].line;//将要替换的行的行号
                        getset = set;
                    }
                }
            }

            //写回主存
            if(cache_data.cache_data_set[set].cache_data_line[tobeset].dirty ==
0)goto ok;//待替换行 dirty 为 1，即进行过修改，需要将数据写回主存。
            uint32_t readdress = 0;
            readdress = (getline<<13)|(set<<5);//将要替换的行的源地址块头
            for(i=0;i<8;i++)//一个块是 32 byte 要读八次
            {
```

```
                uint32_t word = cache_data_read(readdress);//根据地址从 cache 中
读出一个字
                mem_write_32(readdress,word);//将这一个字写回到主存
                readdress+=4;
            }
            ok:;
            cache_data.cache_data_set[set].cache_data_line[tobeset].valid = 1;
            cache_data.cache_data_set[set].cache_data_line[tobeset].line =
line;//对应第几个群
            cache_data.cache_data_set[set].cache_data_line[tobeset].lru = 0;
        }

        for(i = 0;i<8;i++)//读取 32 个字节作为一块
        {
            uint32_t word = mem_read_32(begin);

cache_data.cache_data_set[set].cache_data_line[tobeset].data.mem[i*4]   =
(uint8_t)(word>>0)&0xff;

cache_data.cache_data_set[set].cache_data_line[tobeset].data.mem[i*4+1] =
(uint8_t)(word>>8)&0xff;

cache_data.cache_data_set[set].cache_data_line[tobeset].data.mem[i*4+2] =
(uint8_t)(word>>16)&0xff;

cache_data.cache_data_set[set].cache_data_line[tobeset].data.mem[i*4+3] =
(uint8_t)(word>>24)&0xff;
            begin+=4;
        }
    }



    void cache_data_write_val(uint32_t address,uint32_t write)//写到从主存读取
数据后 cache 对应的块里
    {
        if(cache_data_read(address)==0xffffffff){ //不命中，先从主存里读
            cache_data_load(address);
        }
        uint32_t line = address >> 13 ;//取前 19 位
        uint32_t set = (address >>5) &(0xFF);//取所对应的组号
        uint32_t inneraddress = address & 0x1f;
        uint32_t i = 0;

        for(i=0;i<8;i++)
        {
            if(cache_data.cache_data_set[set].cache_data_line[i].valid==1)//当
前位有效的时候
            {
                if(cache_data.cache_data_set[set].cache_data_line[i].line ==
line)//当前行号和对应的主存行号相同
                {
```

```
                    uint32_t j = inneraddress;//内部地址
                    cache_data.cache_data_set[set].cache_data_line[i].dirty =
1;

cache_data.cache_data_set[set].cache_data_line[i].data.mem[j]   =
(uint8_t)(write>>0)&0xff;

cache_data.cache_data_set[set].cache_data_line[i].data.mem[j+1] =
(uint8_t)(write>>8)&0xff;

cache_data.cache_data_set[set].cache_data_line[i].data.mem[j+2] =
(uint8_t)(write>>16)&0xff;

cache_data.cache_data_set[set].cache_data_line[i].data.mem[j+3] =
(uint8_t)(write>>24)&0xff;
                    return;
                }
            }
        }
    }
```