**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：202100130122** |
| **班级：计机21.6** | **姓名： 吕鹏洋** | |
| **Email：2656678633@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：**  **Ubuntu虚拟机**  **Macbook** | | |
| **实验原理和方法：**  **1.准备工作：**  本实验已经提供了支持 MIPS 指令集的流水线计时模拟器，通过这个模拟器你将可以更好的了解计算机是如何运行的。这台计算机主要包括两个部分： CPU 和内存。计算机要执行的程序（包括代码和数据）都存储在内存中， CPU 会将指令从内存中取出，进行解码并且执行代码所表示的操作（包括算术运算、逻辑运算以及存储器控制操作）。要注意的是，这里的 CPU 所采用的指令集是 MIPS 。   1. **编译和运行:**   代码逻辑写好后执行命令 make ,即可在当前目录下生成名为 sim 的可执行程序，可以运行 sim 可执行程序,进行调试.这是一个功能十分强大的 MIPS 流水线计时模拟器，它将一条 MIPS 指令的生命周期分为 5 个模块 : 取指->译码->执行->访存->回写。整个过程为取指令，指令译码，将译码出的指令放到算术逻辑运算部件 ALU 上执行，根据 ALU 算得的结果进行访存和将访存的结果写回寄存器。相关的代码实现在 pipe.c 中。 | | |
| **实验步骤：**   1. **设计cache：**    1. **头文件：**   #include"shell.h"  #include <string.h>  **typedef** **struct**//指令cache行  {  **int** valid;//有效位  **int** LFU;//LFU计数      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;  **typedef** **struct**//数据cache的一行  {  **int** valid;  **int** LFU;  **int** dirty;      uint32\_t tag;//低19位为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;  uint32\_t is\_inst\_cache\_hit(uint32\_t address);  uint32\_t inst\_cache\_miss(uint32\_t address);  uint32\_t data\_cache\_read\_32(uint32\_t address);  uint32\_t data\_cache\_write\_32(uint32\_t address, uint32\_t value);   * 1. **功能实现：**   #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;      }      //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;              data\_cache\_write\_count = 50;//将延迟计数器复位  **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;      data\_cache\_read\_count = 50;//延迟计数复位  **return** 1;//成功写入返回1  }  **1.3修改pipe.c:**  **在pipe.c中将一些直接从主存读、写的操作改为通过cache读、写。**  **在mem函数中：**  **int** result1 = data\_cache\_write\_32(op->mem\_addr & ~3, val);  **if**(result1==0)  **return**;              //mem\_write\_32(op->mem\_addr & ~3, val);  **if** (op->is\_mem)      {          uint32\_t theVal = data\_cache\_read\_32(op->mem\_addr & ~3);  **if** (theVal == 0x0381CD55)  **return**;          val = theVal;      }  **特别是fetch函数，将其重写为：**  **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 = is\_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**;      }   1. **设计延时：**   **根据实验要求，一次访存需要50个cycle，为模拟延时，定义了全局变量waiting(Data cache中为waiting\_data);进行cache写操作则令waiting=50;在模拟流水线运行过程中若某一阶段对应waiting不为0则waiting--，return，无法将正确的op指针传入下一阶段，即该指令后面的阶段无法执行，直至waiting=0;这样不会影响其他指令在该时间周期的阶段的进行。** | | |
| **结论分析与体会：**  **（1）学习并设计了指令cache和数据cache的 结构及功能**  **（2）对指令流水线运行的过程的了解更加深入**  **（3）探究了 cache 对计算机性能的影响通过本次实验，我了解了cache和五级流水线的工作原理和实现方法，体会到cache在CPU与主存交互中的作用。** | | |