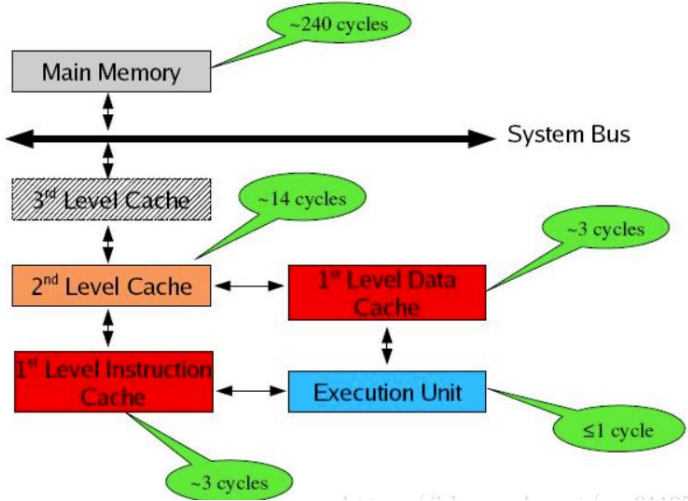
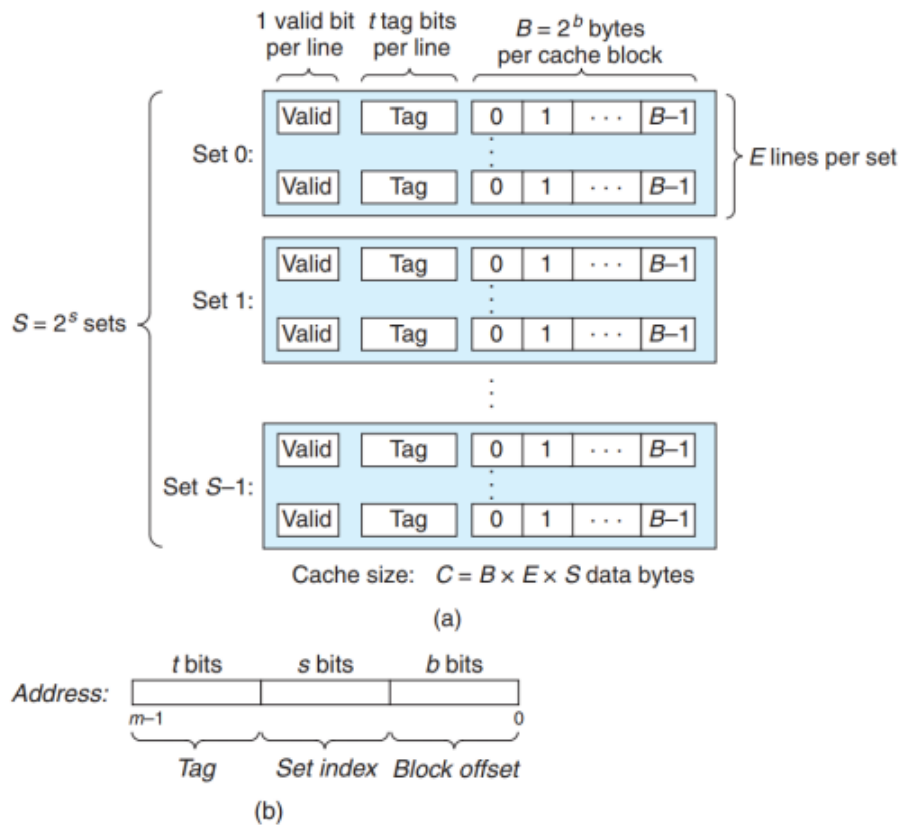


## 计算机科学与技术学院计算机系统基础课程实验报告

实验题目：实验四 设计 MIPS 五级流水线 模拟器中的 Cache	学号：202000130198
班级：20.4	姓名：隋春雨
Email: 202000130198@mail.sdu.edu.cn	
实验目的： ①Cache 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： 实验软件：ubuntu、Clion2020 硬件环境：多路处理器计算机教学实验系统 由四片四核龙芯 3A 处理器构成的 16 核 CC-NUMA 结构、内可配置 外可扩展结构的实验硬件平台。	
实验原理和方法： <b>1. 有关 Cache 的知识</b>  <p>The diagram illustrates the memory hierarchy and its access times in cycles. At the top is the Main Memory, connected to the System Bus, with an access time of approximately 240 cycles. Below the System Bus is the 3rd Level Cache, with an access time of approximately 14 cycles. The 3rd Level Cache is connected to the 2nd Level Cache, which has an access time of approximately 3 cycles. The 2nd Level Cache is connected to the 1st Level Data Cache, which has an access time of approximately 3 cycles. The 1st Level Data Cache is connected to the Execution Unit, which has an access time of less than or equal to 1 cycle. The 1st Level Instruction Cache is also connected to the Execution Unit, with an access time of approximately 3 cycles. The 1st Level Data Cache and 1st Level Instruction Cache are connected to each other.</p>	
<p><b>Cache 的出现是为了解决 CPU 越来越快、但是主存却没有跟上 CPU 的速度的矛盾的存在。</b></p> <p>Cache 存储器：电脑中为高速缓冲存储器，是位于 CPU 和主存储器 DRAM（Dynamic Random Access Memory）之间，规模较小，但速度很高的存储器，通常由 SRAM（Static Random Access Memory 静态存储器）组成。</p> <p>CPU 的速度远高于内存，当 CPU 直接从内存中存取数据时要等待一定时间周期，而 Cache 则可以保存 CPU 刚用过或循环使用的一部分数据，如果 CPU 需要再次使用该部分数据时可从 Cache 中直接调用，这样就避免了重复存取数据，减少了 CPU 的等待时间，因而提高了系统的效率。Cache 又分为 L1Cache（一级缓存）和 L2Cache（二级缓存），L1Cache 主要是集成在 CPU 内部，而 L2Cache 集成在主板上或是 CPU 上。</p> <p>Cache 的功能是提高 CPU 数据输入输出的速率。Cache 容量小但速度快，内存速度较低但容量大，通过优化调度算法，系统的性能会大大改善，仿佛其</p>	

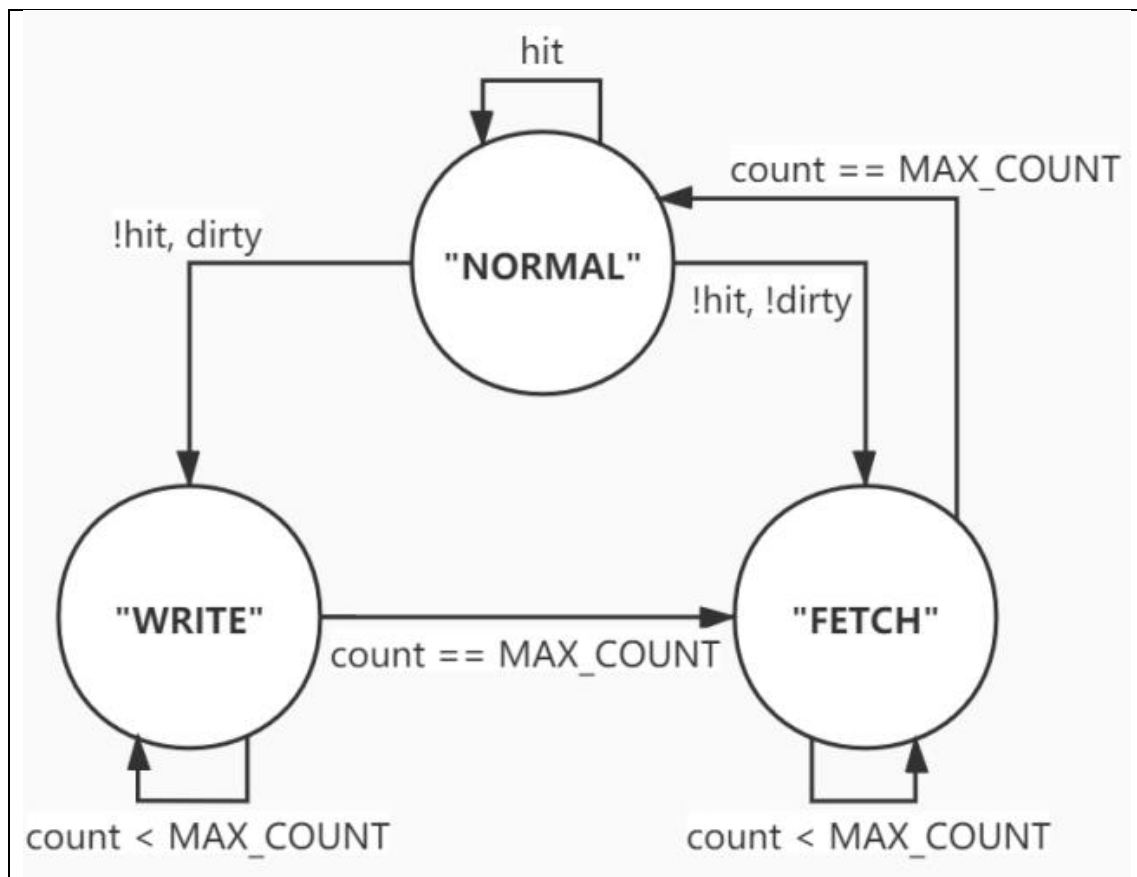
存储系统容量与内存相当而访问速度近似 Cache。

## 2. Cache 设计原理



32 位访存地址按位分为三个部分：标签(tag)、组索引(set index)、块位移(block offset)。我们定义每部分的宽度分别为 `CACHE_T`、`CACHE_S`、`CACHE_B`（在 `cache.svh` 中定义）。另外还定义相联度为常量 `CACHE_E`，其含义是每组包含 `CACHE_E` 个缓存行(line)。如果 `CACHE_B` 为 4 且 `CACHE_S` 为 2，则每个 cache 行存储 16 个字节或 4 字，且整个 cache 分为 4 个组。

## 3. Cache 工作原理



Cache 的工作原理如上图所示。给定 32 位地址时，高速缓存首先读取该地址中的组索引(set index)字段  $i$ ，然后检查第  $i$  组  $set[i]$  是否存在标记(tag)匹配的行(line)。如果存在则缓存命中(cache hit)，否则缓存未命中(cache miss)，我们需要从内存中载入一个块(block)到缓存行中。在现实世界中，一台 32 位计算机上的内存在一个时钟周期内最多可以提供 32 位数据。因此，若从内存中读取数据块，则需要大量的时钟周期才能获取所有数据。在高速缓存未命中的情况下，如果所选集合中充满了有效(valid)行，并且我们选择了要替换的脏(dirty)行，则应首先将数据写回(writeback)到内存中，然后将新数据加载到该行中。选择要替换的行的方式有多种，如最久未使用(LRU)、随机替换(RANDOM)、先进先出(FIFO)；

## 2、实验方法

- (1) 按照实验指导书要求，书写指令 Cache 和数据 Cache 的代码
- (2) 在 pipe.c 文件中将 Cache 文件 include，对读写策略优化，目标是在数据读写时优先考虑 Cache，再考虑主存。
- (3) 调用实验 run 程序，对比数值的变化。通过修改相关参数等以进一步探究不同因素对 Cache 速度的影响。

## 实验步骤:

### 一、环境配置

#### (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。

### 二、理解书写

#### (1) 阅读源码 pipe.c, pipe.h, shell.c, shell.h

```
pipe_stage_wb(); // 回写
pipe_stage_mem(); // 访存
pipe_stage_execute(); // 执行
pipe_stage_decode(); // 译码
pipe_stage_fetch(); // 取指
```

#### (2) 按照要求书写 Cache，明确替换策略

	Instruction Cache	Data Cache
Size	8 KB	64 KB
ways	4	8
Block size	32 bytes	32 bytes
Number of sets	64	256
Replacement (替换策略)	LRU	LRU
Sets index (组的索引)	PC的[10:5]	Address的[12:5]
Visit time (访问时期)	取指阶段	访存阶段

#### ①dataCache

```
typedef struct { // cache中的每一行: dataCacheLine
    uint8_t valid; // 是否已存入数据
    uint8_t dirty;
    uint8_t lru; // 记录存入时间 太大影响运行时间
    uint32_t line; // 行号
    uint8_t data[32]; // 32字节
} dataCacheLine;

typedef struct { // dataCache
    dataCacheLine theDataCache[256][8]; // 256组, 每组8行
} dataCache;
```

#### ②instructionCache

```
typedef struct { // cacheline
    uint8_t valid; // 是否已存入数据
    uint8_t lru; // 记录存入时间
    uint32_t line; // 行号
    uint8_t data[32]; // 32字节
} instructionCacheLine;

typedef struct { // instructionCache
    instructionCacheLine theInstructionCache[64][4]; // 64组, 每组4行
} instructionCache;
```

其中实现的功能有初始化，写入，按地址从主存数据等。

③策略：总是把最近最少用的那一块淘汰掉，即 LRU（Least Recently Used）算法。当数据所占内存达到一定阈值时，要移除掉最近最少使用的数据。当所需数据在 Cache 中时，将该数据移到 Cache 的最前端，其他数据后移。当所需数据不在 Cache 中时，从内存中寻找该数据，并将其插入 Cache 的最前端，其他数据后移。若此时有数据溢出，将最后端的元素写回主存（Cache 中最近最小

使用的），如果此位 dirty 位为 1，说明在 Cache 中被修改，写回主存时需要对其进行相应的修改（因为此时已经和主存中的不一致了）。当分块局部化范围（即：某段时间集中访问的存储区）超过了 Cache 存储容量时，命中率变得很低。极端情况下，假设地址流是 1,2,3,4,1 2,3,4,1,……，而 Cache 每组只有 3 行，那么，不管是 FIFO，还是 LRU 算法，其命中率都为 0。这种现象称为颠簸 (Thrashing / PingPong)。LRU 具体实现时，并不是通过移动块来实现的，而是通过给每个 cache 行设定一个计数器，根据计数值来记录这些主存块的使用情况。这个计数值称为 LRU 位。

(4) 加入延时，运行程序，观察性能。（部分截图如下）

把 cache 的逻辑嵌入到现提供的 MIPS 模拟器中即可，模拟器可以正确处理流水线中数据依赖问题，可以通过增加气泡的方式绕过并正确处理他们。我们的目标是让总的 cycle 尽可能的少，发挥 cache 应有的作用。这里用自检查批处理脚本，以检查编写的正确性。

```

Testing: inputs/random/random2.x
Stats: BaselineSim YourSim
R0: 0x00000000 0x00000000
R1: 0x7fffffff 0x7fffffff
R2: 0x0000000a 0x0000000a
R3: 0x00000000 0x00000000
R4: 0x10000000 0x10000000
R5: 0x00000000 0x00000000
R6: 0x00000000 0x00000000
R7: 0x00000000 0x00000000
R8: 0x00000001 0x00000001
R9: 0x7fffffff 0x7fffffff
R10: 0xab90c389 0xab90c389
R11: 0x7fffffff 0x7fffffff
R12: 0xfccca6b1 0xfccca6b1
R13: 0xab90c388 0xab90c388
R14: 0x7fffffff 0x7fffffff
R15: 0xffffffffc7 0xffffffffc7
R16: 0x00000000 0x00000000
R17: 0x00000000 0x00000000
R18: 0x00000000 0x00000000
R19: 0x00000000 0x00000000
R20: 0x00000000 0x00000000
R21: 0x00000000 0x00000000
R22: 0x00000000 0x00000000
R23: 0x00000000 0x00000000
R24: 0x00000000 0x00000000
R25: 0x00000000 0x00000000
R26: 0x00000000 0x00000000
R27: 0x00000000 0x00000000
R28: 0x00000000 0x00000000
R29: 0x00000000 0x00000000
R30: 0x00000000 0x00000000
R31: 0x00000000 0x00000000
HI: 0x142eb513 0x142eb513
LO: 0xab90c388 0xab90c388
Cycles 15479 14963
FetchedInstr 2057 2057
RetiredInstr 2053 2053
IPC 0.133 0.137
Flushes 0 0
REGISTER CONTENTS OK

```

### 三、修改 cache 的各项参数来探究 cache 对模拟器性能的影响

可以改变的参数有 sets, block size, ways 等，可以总结为 Cache 容量越大，命中率越高，Cycles 越小；Cache 的 block 在一定范围内越大命中率越高，若超出这个范围则恰恰相反。

```

R26: 0x00000000 0x00000000
R27: 0x00000000 0x00000000
R28: 0x00000000 0x00000000
R29: 0x00000000 0x00000000
R30: 0x00000000 0x00000000
R31: 0x00000000 0x00000000
HI: 0x04008008 0x04008008
LO: 0xfbff7ff7 0xfbff7ff7
Cycles 15560 15042
FetchedInstr 2065 2065
RetiredInstr 2061 2061
IPC 0.132 0.137
Flushes 0 0
REGISTER CONTENTS OK

```

### ①减小 cache 组数 (set)

结果：在实验所给数据中 cycle 数无变化，可能是因为所给数据的局限，并没有符合预想中的 miss 率上升，cycle 数增加。

### ②减小 block size

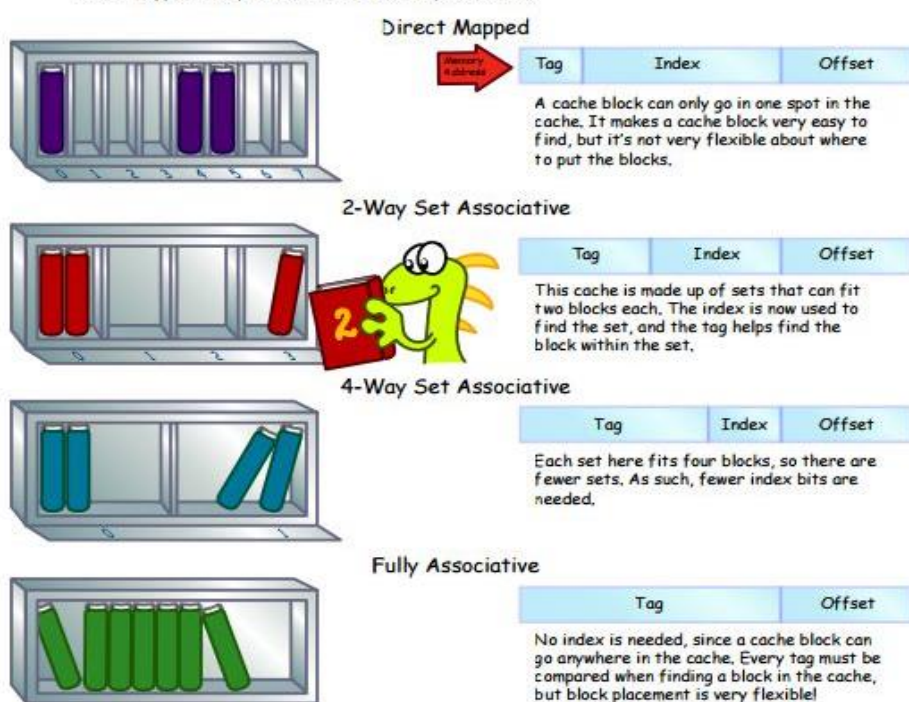
结果：每行存储的数据减小，运行减慢。

### ③减小 ways

每组 2 或 4 行（称为 2-路或 4-路组相联）较常用。通常每组 4 行以上很少用。在较大容量的 L2 Cache 和 L3 Cache 中使用 4-路以上。

结果：绝大部分测试点无变化，仅有一两个测试点缓慢减小的倍数。

Just as bookshelves come in different shapes and sizes, caches can also take on a variety of forms and capacities. But no matter how large or small they are, caches fall into one of three categories: direct mapped, n-way set associative, and fully associative.



## 四、改变替换策略——替换(Replacement)算法

### ①随机替换算法 (Random)

随机法是随机地确定替换的存储块。设置一个随机数产生器，依据所产生的随机数，确定替换块。这种方法简单、易于实现，但命中率比较低。

结果：顾名思义，弹出的数据是随机的，结果竟然也是“随机”的，随机算法有的测试点比 LRU 快，有的测试点比 LRU 慢。

### ②先进先出 FIFO (first-in-first-out)



° 总是把最先进入的那一块淘汰掉。

例：假定主存中的5块{1,2,3,4,5}同时映射到Cache同一组中，对于同一地址流，考察3行/组。

注：通常一组中含有 $2^k$ 行，这里3行/组主要为了简化问题而假设

CPU的访问数据涉  
及到的内存块  
访问序列

3行/组

1	2	3	4	1	2	5	1	2	3	4	5
1*	1*	1*	4	4	4*	5	5	5	5	5*	5*
	2	2	2*	1	1	1*	1*	1*	3	3	3
		3	3	3*	2	2	2	2	2*	4	4

不命中!

✓

✓

✓

命中率25%

这里是采取队列结构，最先进入 Cache 的元素将会从 Cache 中弹出，根据 dirty 位传回主存。

结果：绝大部分测试点无变化，仅有一两个测试点比 LRU 慢

③最近最少用 LRU (least-recently used)

即一开始选用的方法

° 总是把最近最少用的那一块淘汰掉。

例：假定主存中的5块{1,2,3,4,5}同时映射到Cache同一组中，对于同一地址流，考察3行/组的情况。

1	2	3	4	1	2	5	1	2	3	4	5
1	2	3	4	1	2	5	1	2	3	4	5
	1	2	3	4	1	2	5	1	2	3	4
		1	2	3	4	1	2	5	1	2	3

3行/组

✓

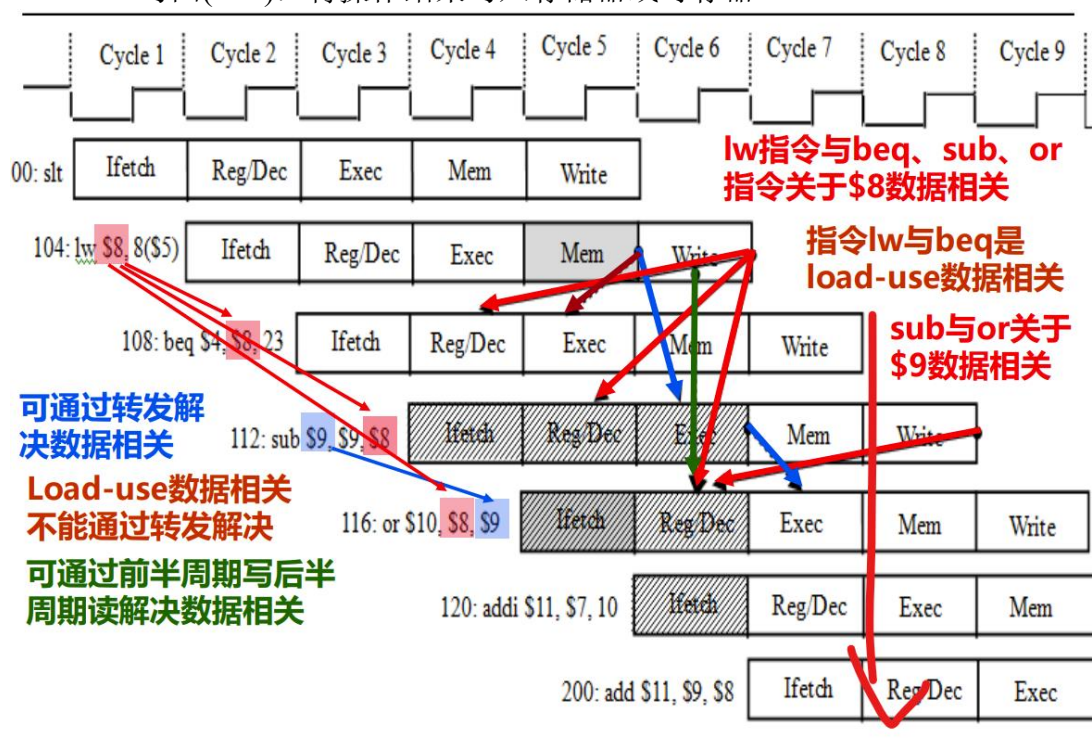
✓

结论分析与体会：

### 1、通过实验理解了五级流水线

流水线是将计算机指令处理过程拆分为多个步骤，并通过多个硬件处理单元并行执行来加快执行速度。在三级流水“取指-译码-执行”的基础上，引入 Cache，对读取速度进一步提高；此外，因为 Load/Store 指令很有可能会超过一个时钟周期，所以，我们加了访存和回写，用五级流水线的方式来解决。

- 取指令(IF)：根据 PC 的值从存储器取出指令。
- 指令译码(ID)：产生指令执行所需的控制信号。
- 取操作数(OF)：读取存储器操作数或寄存器操作数。
- 执行(EX)：对操作数完成指定操作。
- 写回(WB)：将操作结果写入存储器或寄存器。



### 2、通过实验进一步理解了 Cache 的相关知识

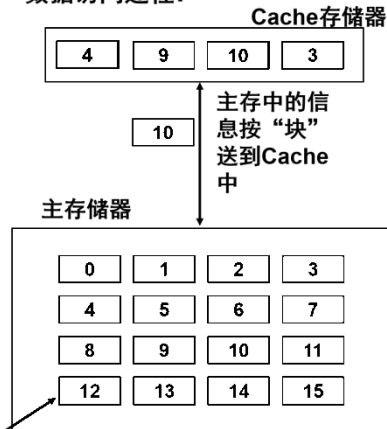
前提：大量典型程序的运行情况分析结果表明在较短时间间隔内，程序产生的地址往往集中在一个很小范围内，这种现象称为程序访问的局部性：空间局部性、时间局部性。

- 指令：指令按序存放，地址连续，循环程序段或子程序段重复执行
- 数据：连续存放，数组元素重复、按序访问

在 CPU 和主存之间设置一个快速小容量的存储器 Cache，其中总是存放最活跃（被频繁访问）的程序和数据，由于程序访问的局部性特征，大多数情况下，CPU 能直接从这个高速缓存中取得指令和数据，而不必访问主存，如果没有找到则再去主存中寻找，并根据所选策略对 Cache 进行更新等操作。如果没有 Cache，就需要到主存中寻找，耗费较多时间。



#### 数据访问过程:



#### 块 (Block)

3、解决了配置运行过程中的系列问题:

①要注意 Python 版本, 且须配置环境变量和工具包, 否则程序无法运行

②go 之后一直在跑, 就是不 halted

这时候可能是从 Cache 中取指有误, debug 一下, 打印 Cache 中的返回值与 memread 的返回值对比一下。

#ifdef DEBUG // 仅在 Debug 时被编译

③递归栈过深

```
RecursionError: maximum recursion depth exceeded while calling a Python object
```

使用 python 写的递归程序如果递归太深, 那么极有可能因为超过系统默认的递归深度限制而出现的错误。这是需要人为设定。

4、在 Cache 的设计中体会到多级结构设计的重要性

使用了结构体设计多级结构, Cache 由 set 组成, set 由 line 组成, line 由 valid, tag, data 组成……这样子定义结构体层次清晰直观, 在对参数和替换策略修改的时候也比较方便。

5、/\* each of these functions implements one stage of the pipeline 以下各个函数分别实现流水线的各个阶段 \*/

```
void pipe_stage_fetch();
```

```
void pipe_stage_decode();
```

```
void pipe_stage_execute();
```

```
void pipe_stage_mem();
```

```
void pipe_stage_wb();
```

但为什么代码要“倒着写”?

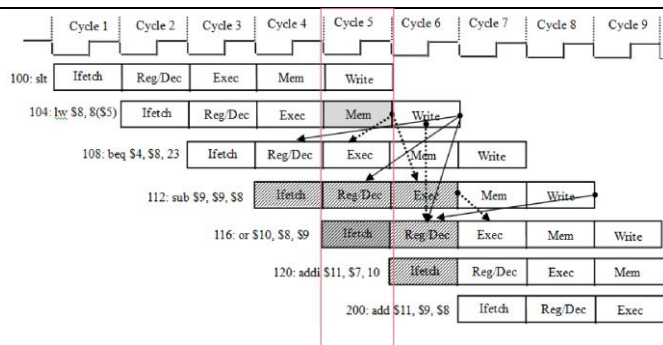
```
pipe_stage_wb(); // 回写
```

```
pipe_stage_mem(); // 访存
```

```
pipe_stage_execute(); // 执行
```

```
pipe_stage_decode(); // 译码
```

```
pipe_stage_fetch(); // 取指
```



由上图中的 cycle5 可以清晰的看出，在这个 cycle 内其实是按着第一步的回写，第二步的访存，.....这样子执行的。可以说是阶段上倒置，指令上正置。

#### 附录：data\_Cache.h

```

1.  #ifndef DATA_CACHE
2.  #define DATA_CACHE
3.
4.  #include "stdio.h"
5.  #include "mips.h"
6.  #include "stdbool.h"
7.  #include "stdint.h"
8.
9.  typedef struct { // cache 中的每一行
10.     uint8_t valid; // 是否已存入数据
11.     uint8_t dirty;
12.     uint8_t lru; // 记录存入时间 太大影响运行时间
13.     uint32_t line; // 行号
14.     uint8_t data[32]; // 32 字节
15. } data_cache_line;
16.
17. typedef struct { // 数据 cache
18.     data_cache_line data_cache_set[256][8]; // 256 组，每组 8 行
19. } data_cache;
20.
21. void data_cache_init(); // 初始化
22. void mem_write_data_cache(uint32_t address, uint32_t value); // 修改数据
23. void data_cache_load(uint32_t address); // 将数据写入 cache
24. uint32_t data_cache_read(uint32_t address); // 从 cache 中读取数据
25.
26. extern int data_pause;
27.
28. #endif

```

Data\_cache.c

```

1. #include "mips.h"
2. #include "pipe.h"
3. #include "shell.h"
4. #include <assert.h>
5. #include <stdio.h>
6. #include <stdlib.h>
7. #include <string.h>
8. #include "data_cache.h"
9.
10. data_cache cache;
11. int data_pause = 0;
12.
13. // 初始化
14. void data_cache_init() {
15.     int i, j;
16.     for (i = 0; i < 256; i++) {
17.         for (j = 0; j < 8; j++) {
18.             cache.data_cache_set[i][j].valid = 0;
19.             cache.data_cache_set[i][j].lru = 0;
20.             cache.data_cache_set[i][j].dirty = 0;
21.         }
22.     }
23. }
24.
25. // 从 cache 中读取数据
26. uint32_t data_cache_read(uint32_t address) {
27.     uint32_t line = address >> 13;
28.     uint32_t set = (address >> 5) & (0xFF); // 组号
29.     uint32_t inneraddress = address & 0x1F;
30.
31.     // printf("data_cache_address: %x", address);
32.     int i = 0;
33.     for (i = 0; i < 8; i++) {
34.         if (cache.data_cache_set[set][i].valid == 1 &&
35.             cache.data_cache_set[set][i].line == line) { // 当前行号和对应的主存
行号相同
36.             uint32_t tmp = address & 0x1F; // 内部地址
37.             // uint32_t a = (cache.data_cache_set[set][i].data[tmp] << 0) |
38.             // (cache.data_cache_set[set][i].data[tmp+1] << 8) |
39.             // (cache.data_cache_set[set][i].data[tmp+2] << 16) |
40.             // (cache.data_cache_set[set][i].data[tmp+3] << 24);
41.             // printf("res: %x", a);
42.
43.             return
44.                 (cache.data_cache_set[set][i].data[tmp] << 0) |
45.                 (cache.data_cache_set[set][i].data[tmp+1] << 8) |
46.                 (cache.data_cache_set[set][i].data[tmp+2] << 16) |
47.                 (cache.data_cache_set[set][i].data[tmp+3] << 24);
48.         }
49.     }
50.     data_pause = 49; // 停顿 50 个周期

```

```

51. data_cache_load(address); // 写入 data_cache
52. return data_cache_read(address); // 再次读取并返回
53. }
54.
55. // 修改数据
56. void mem_write_data_cache(uint32_t address, uint32_t value) { //把本应写在
    主存 address 里的 write 写到 cache 里
57.
58. data_cache_read(address); // 保证修改数据在 cache 中
59.
60. uint32_t line = address >> 13; //取前 19 位
61. uint32_t set = (address >> 5) & (0xFF); //取所对应的组号
62. // printf("mem_address: %x", address);
63. int i;
64. for (i = 0; i < 8; i++) {
65.     if (cache.data_cache_set[set][i].line == line) { // 找到对应的主存块
66.
67.         uint32_t tmp = address & 0x1F;
68.
69.         cache.data_cache_set[set][i].dirty = 1;
70.         cache.data_cache_set[set][i].data[tmp] = (uint8_t)(value >> 0) & 0xFF;
71.         cache.data_cache_set[set][i].data[tmp+1] = (uint8_t)(value >> 8) & 0xFF;
72.         cache.data_cache_set[set][i].data[tmp+2] = (uint8_t)(value >> 16) & 0xFF;
73.         cache.data_cache_set[set][i].data[tmp+3] = (uint8_t)(value >> 24) & 0xFF;
74.
75.         return;
76.     }
77. }
78. }
79.
80. // 将数据写入 cache 中
81. void data_cache_load(uint32_t address) {
82.     uint32_t line = address >> 13; //取前 19 位, 要写入的 line
83.     uint32_t set = (address >> 5) & (0xFF); //取所对应的组号
84.     uint32_t begin = (address >> 5) << 5; //块头
85.
86.     int i = 0, pos = -1;
87.     for (i = 0; i < 8; i++) {
88.         if (cache.data_cache_set[set][i].valid == 0) {
89.             cache.data_cache_set[set][i].valid = 1;
90.             cache.data_cache_set[set][i].line = line;
91.             cache.data_cache_set[set][i].lru = 0;
92.
93.             pos = i;
94.             break;
95.         }
96.     }
97.
98.     // lru cycle

```

```

99. // 2、3 925848
100. // 4 926721
101. // 5 926527
102. // 6 926430
103. // 7 926333
104.
105. for (i = 0; i < 8; i++) { // 更新 lru
106.     if (pos != i && cache.data_cache_set[set][i].lru < 7) {
107.         cache.data_cache_set[set][i].lru++;
108.         // printf("更新 lru");
109.         // printf("lru%d: %x %x", i, cache.data_cache_set[set][i].lru, cache.d
ata_cache_set[set][i].dirty);
110.     }
111.     // printf("\n");
112. }
113.
114. // printf("%d\n", pos);
115. if (pos == -1) {
116.     uint8_t max = 0;
117.     // printf("-----max-----\n");
118.     for (i = 0; i < 8; i++) {
119.         if (max <= cache.data_cache_set[set][i].lru) { // 找到替换行
120.             max = cache.data_cache_set[set][i].lru;
121.             pos = i;
122.             printf("%d %d %d\n", max, pos, i);
123.         }
124.     }
125.     // printf("-----max-----\n");
126.     // printf("-----%d-----\n", pos);
127.
128.     if (cache.data_cache_set[set][pos].dirty != 0) { // 将脏数据写回
129.         // printf("写回脏数据\n");
130.         uint32_t tmp = (cache.data_cache_set[set][pos].line << 13) | (set << 5);
        // 替换位置
131.         for (i = 0; i < 8; i++) {
132.             mem_write_32(tmp, data_cache_read(tmp));
133.             tmp += 4;
134.         }
135.     }
136.     cache.data_cache_set[set][pos].valid = 1;
137.     cache.data_cache_set[set][pos].lru = 0;
138.     cache.data_cache_set[set][pos].line = line;
139. }
140.
141. for (i = 0; i < 8; i++) { // 写入 cache
142.     uint32_t tmp = mem_read_32(begin);
143.     cache.data_cache_set[set][pos].data[i*4] = (uint8_t)(tmp >> 0) & 0xFF;
144.     cache.data_cache_set[set][pos].data[i*4+1] = (uint8_t)(tmp >> 8) & 0xFF;
        F;
145.     cache.data_cache_set[set][pos].data[i*4+2] = (uint8_t)(tmp >> 16) & 0xFF;
        FF;

```

```
146.         cache.data_cache_set[set][pos].data[i*4+3] = (uint8_t)(tmp >> 24) & 0x
           FF;
147.
148.         begin += 4;
149.     }
150. }
```