Cache 设计实验报告

张作柏
17300240035

2019 年 6 月 12 日

目录

1 缓存机制简介

计算机的主存一般由 DRAM 芯片构成。在过去的 30 年中，DRAM 的速度仅以每年 7% 增长，而处理器的性能则以每年 25%~ 50% 增长。在 1980 年前后，处理器和存储器的速度是一样的，但是性能从那时开始有了差别，存储器速度开始严重落后。DRAM 访问时间比处理器周期长一到两个数量级。

为了抵消这种趋势，计算机将最常用的指令和数据存储在更快但更小的存储器中，这种存储器称为*高速缓存* (cache)，其速度与处理器相近，但是容量较小。如果处理器需要的数据在高速缓存中可用，那么它可以快速返回，这成为缓存命中 (hit)，否则，处理器就需要从主存 (DRAM) 中获得数据，这称为缓存缺失 (miss)。如果在大部分情况下缓存命中，那么处理器就基本上不需要等待低速的主存。

1.1 高速缓存原理

高速缓存主要应用了*时间局部性*和*空间局部性*的原理。时间局部性意味着最近使用过的数据可能很快就会被再次使用。而空间局部性意味着当使用一个数据时，很可能会在之后使用与其相邻的数据。

高速缓存可以一次从主存中取许多相邻的数据，以此利用空间局部性。同时，还可以根据之前的数据使用情况，选择合适的预取策略。而 cache 对时间局部性的利用主要体现在保存之前使用的数据和数据替换策略的设计。我们将在接下来的两小节分别进行讨论。

1.2 高速缓存结构

高速缓存通常保存存储器数据，其存放数据字的数量称为*容量* (C)。

为了利用空间局部性，当处理器访问一块数据时，我们可以从内存中提取多个相邻的字。这样的一组字称为*高速缓存块* (cache block)，一个高速缓存块中的字数称为*块大小* (b)，容量为 C 的高速缓存包含了 $B = C/b$ 个块。

一个高速缓存可以组织成 S 组，其中每一组有一个或多个数据块。主存中数据的地址和高速缓存中数据的位置之间的关系称为映射 (mapping)。高速缓存按照组中块的数量进行分类：

- **直接映射**：每组内只有一块，所以组数 $S = B$。在这种情况下，数据替换策略是无意义的。
- **多路组相联**：每组提供 N 块，此时需满足 $S \times N = B$。每个内存地址可以映射到唯一的组中，但是它可以映射到一组中 N 块的任意一块。
- **全相联**：只包含一组，即 $S = 1$。这一组中包含了 B 路，存储器地址可以映射到这些路中的任何一块。

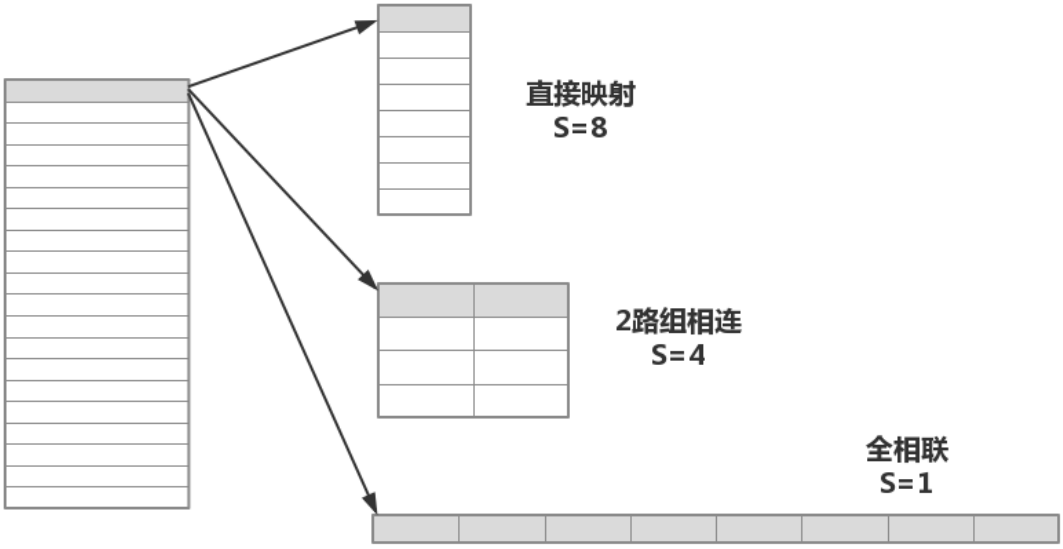


图 1: Cache 组织形式：直接映射、多路组相连和全相联

图 1简单地展示了三种组织形式，其中灰色的部分对应该内存地址可能映射到的块。注意到，以上三种组织形式是本质相同的，只是参数的设置不同而已，所以在书写代码时，可以将组数 S 和路数 N 设置为可调整的参数，以此方便的实现三种组织形式。

1.3 数据替换策略

在直接映射高速缓存中，每个地址映射到唯一的块和组上。如果装入新数据时，其所对应的组满了，那么组中的块就必须用新数据替换。在组相联和全相联的高速缓存中，高速缓存必须在组满时选择哪一个块被替换。常见的替换策略有：

- **随机替换 (RAND)**：随机选择一个块被替换。
- **先进先出 (FIFO)**：选取最先被读入 Cache 的块进行替换。
- **最常使用 (MFU)**：该算法将一段时间内被访问次数最多的那个块替换出去。
- **最近最少使用 (LRU)**：把 CPU 近期最少使用的块替换出去。

2 实现细节

本次实验是在上次的流水线处理器上直接进行的改进，所以这里主要讨论 cache 模块的实现，对流水线的影响将在第 2.5节进行讨论。

2.1 缓存组织结构实现

首先，我们以 parameter 的形式设置 cache 的结构：块大小 (b)、组数 (S) 和路数 (N)。

```
module cache #(parameter BLOCK_SIZE=64, SET_CNT=2, LINE_CNT=2) (clk, ...);
```

在之后的实现中，我们将结合这些参数与 for 循环来模拟任意结构的 cache。

紧接着，我们分别计算出确定偏移量 (offset) 与组号所需要位数：

```
parameter OFFSET = $clog2(BLOCK_SIZE/8);
parameter SET_LEN = $clog2(SET_CNT);
```

这里我们使用了 Verilog 中自带的 clog2() 函数，其作用是求以 2 为底的对数。注意，这里因为在我的实现中，内存的每个地址对应 8 个 bit，而这里的 BLOCK_SIZE 代表的是一个块所占的 bit 数，所以要先除以 8。

之后，我们通过 Verilog 中的多维数组来模拟 cache 的结构：

```
reg [BLOCK_SIZE-1:0] data[SET_CNT-1:0][LINE_CNT-1:0];
reg [LINE_CNT-1:0] valid[SET_CNT-1:0];
reg dirty[SET_CNT-1:0][LINE_CNT-1:0];
reg [8 - SET_LEN - OFFSET:0] tag[SET_CNT-1:0][LINE_CNT-1:0];
```

如此，我们将 Cache 的基本结构构建完成，其示意图如下：

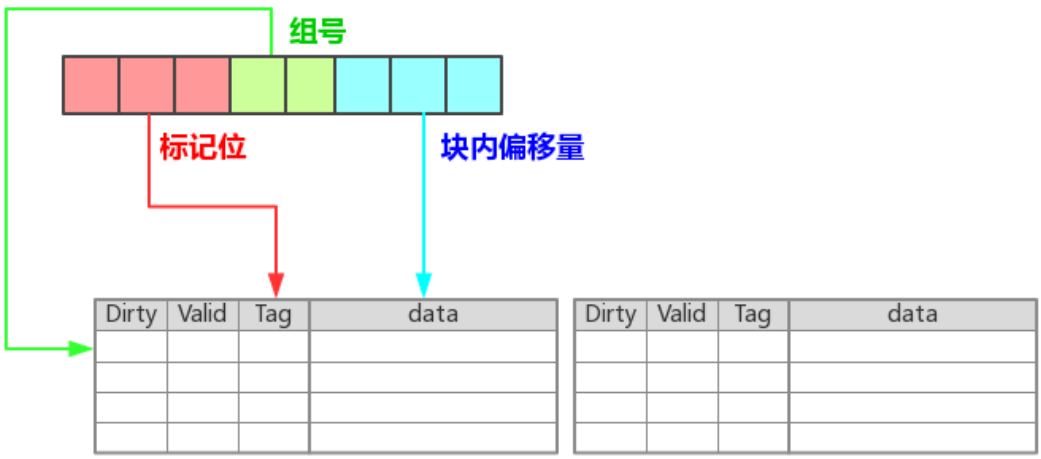


图 2: Cache 基本结构示例：块大小 $b = 2$、组数 $S = 4$、路数 $N = 2$

2.2 高速缓存读写

（1）如何判断 Cache 中是否存在我们需要的数据？

首先，我们先根据访存的地址确定其映射到的组号：

```
assign SET_ID = addr[OFFSET + SET_LEN - 1 : OFFSET];
```

之后，我们利用 for 循环，判断组内是否存在有效行的标记位与访存地址相同，若存在，则说明找到了命中的位置。

```
always @(*)
    for (i = 0; i < LINE_CNT; i = i + 1)
      hiti[i] = valid[SET_ID][i] & (tag[SET_ID][i] == addr[8:OFFSET +
   SET_LEN]);
assign hit = | hiti;
```

（2）如何控制 Cache 的读写？

当 Cache 命中时，我们可以根据刚刚计算出的 hiti 数组，对命中的组进行操作：

- 读：

```
for (i = 0; i < LINE_CNT; i = i + 1)
  if (hiti[i]) rdata = data[SET_ID][i][addr[OFFSET-1:2]*32 +: 32];
```

- 写：

```
for (i = 0; i < LINE_CNT; i = i + 1)
  if (hiti[i])
  begin
    dirty[SET_ID][i] = 1;
    data[SET_ID][i][addr[OFFSET-1:2]*32 +: 32] = wdata;
  end
```

（3）如何与 Memory 交互？

当 Cache 未命中时，我们需要从内存中取数据。这里我们直接套用了原来的 Dmem，通过改变写信号与访存地址来获取我们想要的数据。需要注意，我们这里每次访存取出的块大小不再是 32，而是给定的参数 BLOCK_SIZE。

```
Dmem #(BLOCK_SIZE) dmem(clk, reset, ADDR, WEN, WDATA, RDATA);
```

当发生 miss 时，我们通过改变地址，来从内存中获取数据：

```
ADDR = {addr[8:OFFSET], 'OFFSET'b0};
WEN = 0;
WDATA = 0;
```

注意，这里我们的 Dmem 是组合逻辑，所以不存在延时。而访存的延时，我们会通过状态机来设置，这将在下一小节详细讨论。

2.3 访存延时实现

为了模拟访存时的时间延时，在每次访存时我们需要让流水线等待几个周期，且根据 hit 与 miss 的情况等待不同的周期数。这一点我们可以通过状态机来实现。

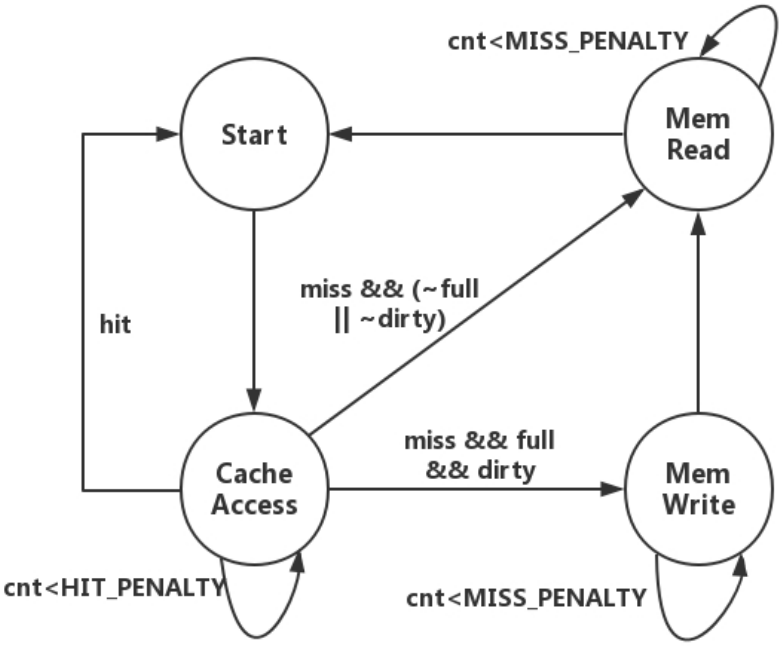


图 3: Cache 延时状态机

首先，我们需要设置两个延时参数：HIT_PENALTY 和 MISS_PENALTY，分别表示从 cache 中取数据和从 memory 中取数据需要的周期数。

之后，设计四个状态，含义如下：

1. **Start**：起始状态。若指令不需要访存，则始终停留在该状态；若需要访存时，从该状态起始，对计数器清零。

2. **CacheAccess**：访问 Cache 的状态，自带一个计数器。当计数器累加到 HIT_PENALTY 时，根据是否 hit，选择转移到的状态。
   (1) 若命中，则从 Cache 中将数据取出，并转移回 Start 状态，标记为取数据完成。
   (2) 若未命中且当前组已满且需要写入，则下一步需要根据替换策略进行替换，跳至 MemWrite 状态，并将计数器清零。
   (3) 若未命中且当前组仍有空位，则下一步从内存中取数据，跳至 MemRead 状态，并将计数器清零。

3. **MemWrite**：写内存状态，当需要进行替换时，进入该状态。在状态起始，更改访存的地址和写数据，设置写使能为 1。当计数器累加到 MISS_PENALTY 时，将其清零，跳至 MemRead 状态。

4. **MemRead**：读内存状态，从内存中读取需要的数据。在状态起始，将访存的地址设置为输入的地址信号。当计数器累加至 MISS_PENALTY 时，将其清零，输出读出的数据，并标记为取数据完成，跳至 Start 状态。

2.4 替换策略实现

随机替换 (RAND) 随机替换的实现借助了 Verilog 中自带的随机数，每次随机生成一个需要替换的组号。

```
ran = {$random} % LINE_CNT;
```

先进先出 (FIFO) 先进先出策略要求先进入 Cache 的块先被踢出，所以我们可以对每个组记录当前踢到哪一个块，每次需要替换时就踢出当前块，并使该记录加一。

```
ran = out[SET_ID];
out[SET_ID] <= (out[SET_ID] + 1) % LINE_CNT;
```

初始化时，将 out 数组全部初始化为 0。

最常使用 (MFU) 最不常用算法每次会踢出使用次数最多的块，我们需要记录每组每个块的使用次数，每次需要替换时，选出使用次数最多的进行替换。在新加入块后，将相应的次数计数清零。

```
ran = 0;
for (i = 1; i < LINE_CNT; i = i + 1)
  if (hit_num[SET_ID][i] > hit_num[SET_ID][ran]) ran = i;
```

最近最少使用 (LRU) 这里我们使用的是伪 LRU 算法，即每次替换时，替换掉上次使用时间最早的块。这需要我们记录每个块上一次使用的时间。一种方法是当一个块被使用时，将其时间戳设为 0，其余块的时间戳加一，最后选择出时间戳最大的块进行替换。

```
ran = 0;
for (i = 1; i < LINE_CNT; i = i + 1)
  if (dfn[SET_ID][i] > dfn[SET_ID][ran]) ran = i;
```

2.5 流水线结构调整

整个流水线的变动并不大，主要调整在暂停信号。当数据未取完时，需要将流水线完全暂停住。我们可以通过增加两个信号来控制：LW 信号控制是否为 lw 或 sw 指令、DATA_COMPLETE 信号控制是否完成取数据操作。

在冒险处理 Hazard 模块中，若当前 Memory 阶段的指令为 lw 或 sw 指令，且未完成取数据操作，则令整个流水线暂停，设置所有状态寄存器的 Stall 信号为 1 即可。

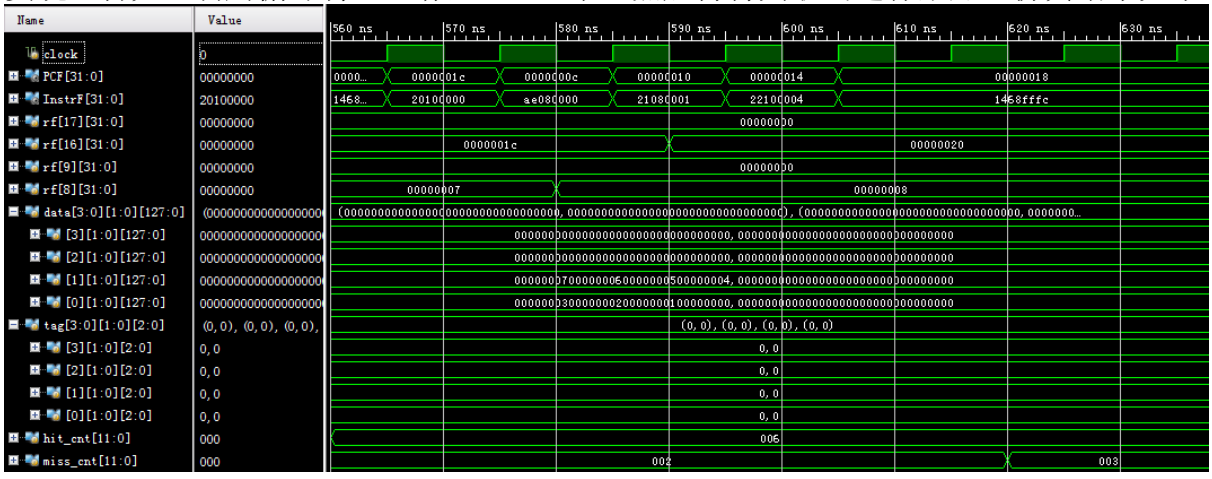
3 测试样例与结果

与单周期相同的测试样例均已通过，这里只列举几个关键的和新加入的测试样例。以下如未特殊声明，则均使用块大小为 $b = 4$，组数为 4，行数为 2 的高速缓存。

3.1 swlw.in

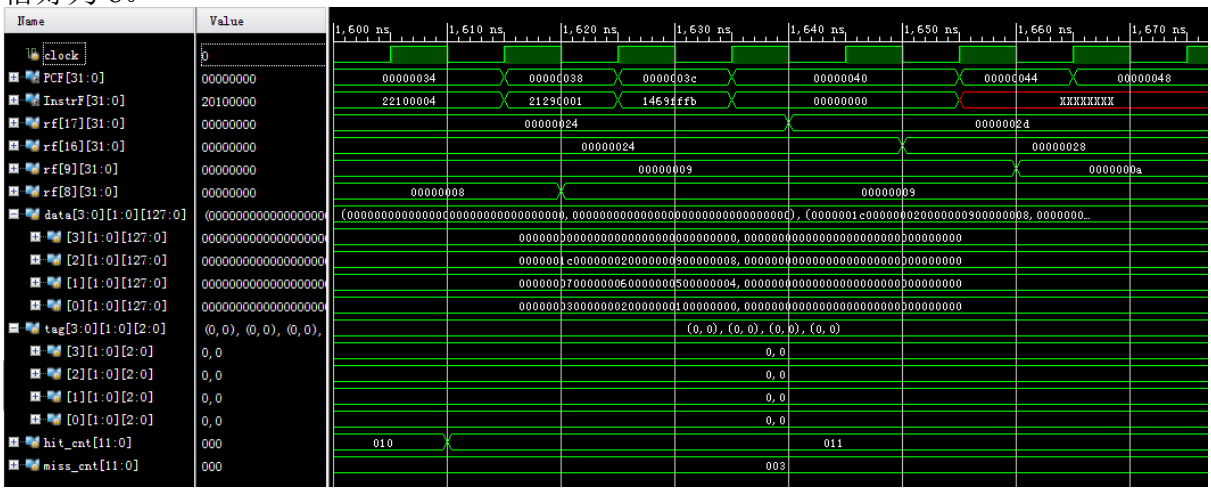
```
 0x0 : addi $s0, $0, 0       | 20100000
 0x4 : addi $t0, $0, 0       | 20080000
 0x8 : addi $v1, $0, 10      | 2003000a
 0xc : for:                  |
 0xc : sw $t0, 0($s0)        | ae080000
0x10 : addi $t0, $t0, 1      | 21080001
0x14 : addi $s0, $s0, 4      | 22100004
0x18 : bne $t0, $v1, for     | 1468fffc
0x1c : addi $s0, $0, 0       | 20100000
0x20 : addi $t0, $0, 0       | 20080000
0x24 : addi $t1, $0, 0       | 20090000
0x28 : addi $s1, $0, 0       | 20110000
0x2c : for2:                 |
0x2c : lw $t0, 0($s0)        | 8e080000
0x30 : add $s1, $s1, $t0     | 02288820
0x34 : addi $s0, $s0, 4      | 22100004
0x38 : addi $t1, $t1, 1      | 21290001
0x3c : bne $t1, $v1, for2    | 1469fffb
0x40 : nop                   | 00000000
0x44 : nop                   |
```

该测试样例主要是为了检查 Cache 的基本存取操作，以及流水线的功能能否正常实现。首先，利用循环将 0~9 存入 cache 中，然后再将其取出进行累加，仿真结果如下：



第一幅图截取的是前半段的某一时刻，分别展示了 Fetch 阶段的 PC 值和取出的指令 InstrF，和四个寄存器 $s1,$s0,$t1,$t0 的值。这里恰好正在存入第 8 个值，所以 $t0 等

于 8，$s0 等于 0x20，而 data 表示的是 cache 中的信息。可以注意到，缓存中已经存好了前 8 个数，因为块大小为 4，所以每行存储着 4 个数字。最下方记录的是 hit 和 miss 的次数，因为只取了两个块出来，所以说明只 miss 了 2 次，而 hit 的次数为 6，加起来恰好为 8。



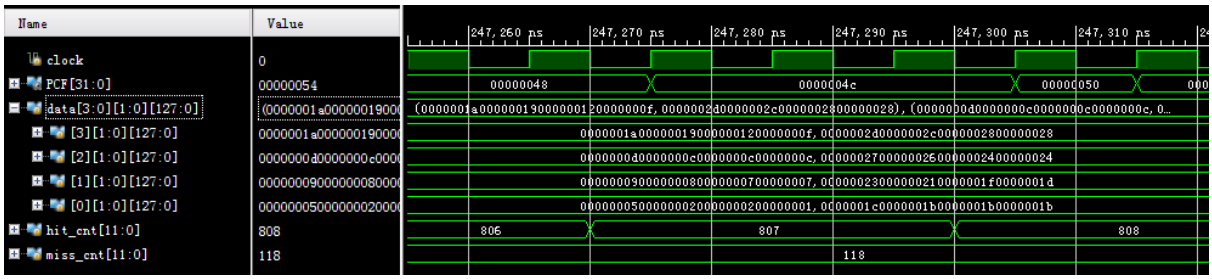
第二幅图截取的是后半段的结束。注意到，寄存器 $s1 的值已累加到 0x2d=45，等于 0~9 的和。Cache 中取出的是前三块的内容，前十项都已被赋值为正确的值，第十一项和第十二项是 RAM 预初始化的值，与本测试样例无关。最终，miss 的次数为 3，hit 的次数为 17，总共 20 次。

3.2 bubble_sort.txt

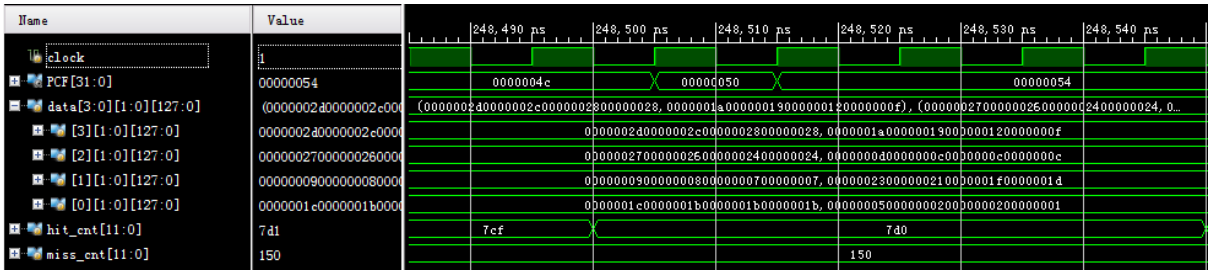
```
 0x0 : add $t0, $0, $0       | 00004020
 0x4 : addi $t2, $0, 64      | 200a0040
 0x8 : addi $t3, $0, 256     | 200b0100
 0xc : For:                  |
 0xc : add $sp, $0, $t0      | 0008e820
0x10 : lw $t4, 0($sp)        | 8fac0000
0x14 : add $t5, $0, $sp      | 001d6820
0x18 : for:                  |
0x18 : lw $t6, 0($sp)        | 8fae0000
0x1c : slt $t7, $t6, $t4     | 01cc782a
0x20 : bne $t7, $0, ret      | 140f0002
0x24 : add $t4, $0, $t6      | 000e6020
0x28 : add $t5, $sp, $0      | 03a06820
0x2c : ret:                  |
0x2c : addi $sp, $sp, 4      | 23bd0004
0x30 : bne $sp, $t3, for     | 157dfff9
0x34 : addi $t3, $t3, -4     | 216bfffc
0x38 : lw $t8, 0($t3)        | 8d780000
0x3c : sw $t4, 0($t3)        | ad6c0000
0x40 : sw $t8, 0($t5)        | adb80000
```

```
0x44 : bne $0, $t3, For      | 1560fff1
```

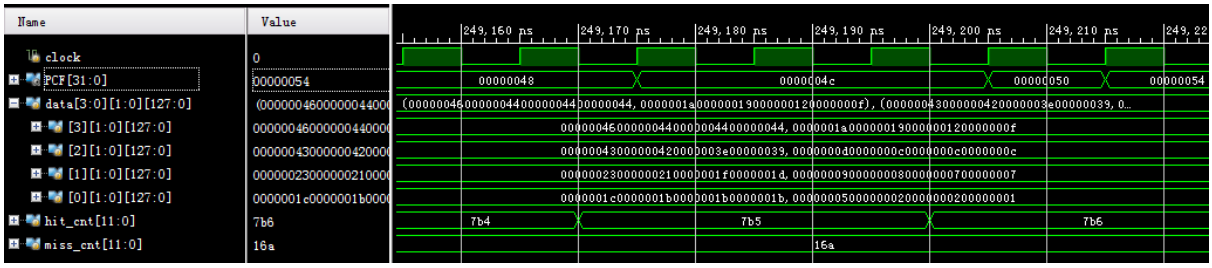
该样例是由罗翔同学构造的冒泡排序代码，此处借来检验 Cache 的正确性。该段代码会对预存在 Dmem 中的 64 个数升序排序，最终将结果写入 Memory。我实现了四种不同的替换策略，其仿真结果如下图所示：



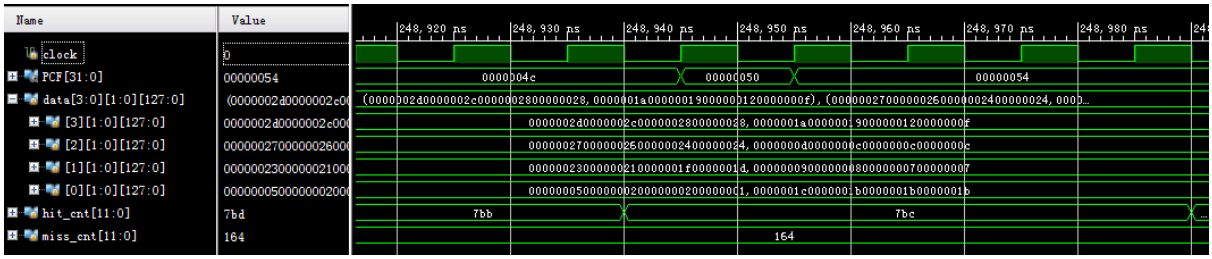
(a) RAND



(b) FIFO



(c) MFU



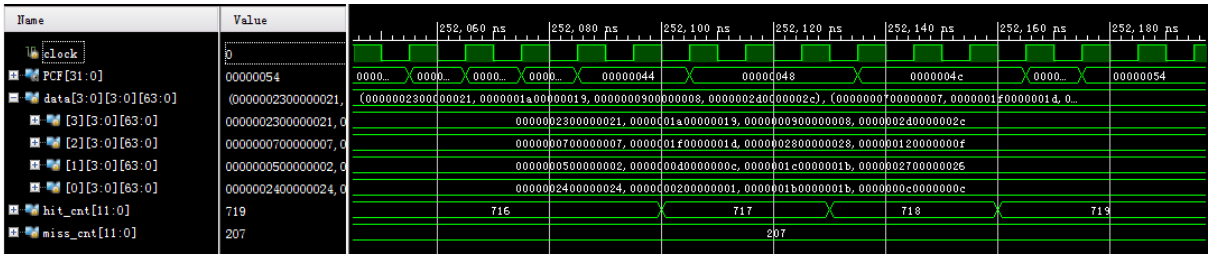
(d) LRU

图 4: 冒泡排序仿真结果 (128 位 4 组 2 路)

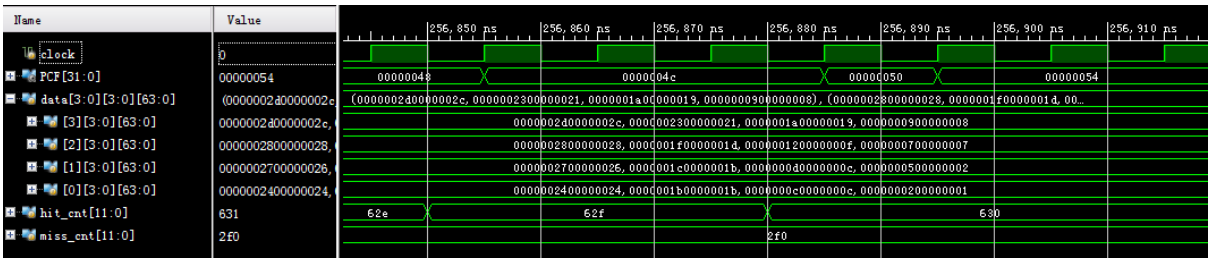
因为内存中的值太多不好显示，所以这里可以只看 Cache 里的值。这里使用的是 4 组 2 路，每个块 128 位，四个字的 Cache。注意到，四种策略中最终 Cache 内同一个 block 内的数据都是升序排序的，可以部分说明算法的正确性。而四种策略中最终留在

Cache 内的数据不同，也说明四种策略的效率确实有差距。

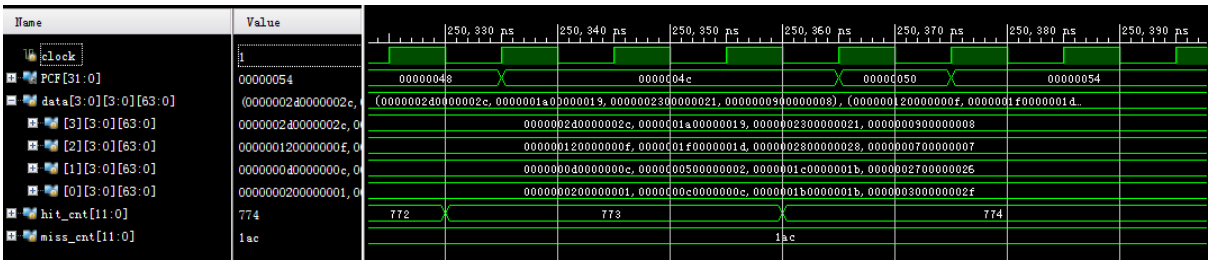
为了对比 Cache 的大小对效率的影响，我在这里还使用了另一种大小的 Cache(64 位 4 组 4 路) 来执行冒泡排序的代码，结果如下：



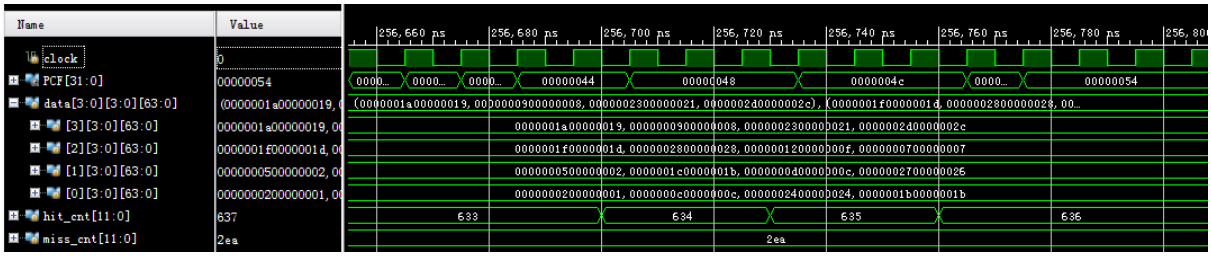
(a) RAND



(b) FIFO



(c) MFU



(d) LRU

图 5: 冒泡排序仿真结果 (64 位 4 组 4 路)

对于块大小为 128 位，4 组 2 路的 Cache，四种策略的 hit 和 miss 次数分别为：

- **RAND:** 0x808 次 hit，0x118 次 miss
- **FIFO:** 0x7d0 次 hit，0x150 次 miss
- **MFU:** 0x7b6 次 hit，0x16a 次 miss

- **LRU:** 0x7bc 次 hit，0x164 次 miss

在此处，随机替换策略是最好的，而 FIFO 是最差的。四种策略的访存总次数均为 $0x920 = (65 * 64)/2 + 4 * 64 = 2336$，正好对应冒泡排序的次数。

而对于块大小为 64 位，4 组 4 路的 Cache，四种策略的 hit 和 miss 次数分别为：

- **RAND:** 0x719 次 hit，0x207 次 miss
- **FIFO:** 0x630 次 hit，0x2f0 次 miss
- **MFU:** 0x774 次 hit，0x1ac 次 miss
- **LRU:** 0x636 次 hit，0x2ea 次 miss

结果分析：容易发现，对于相同容量的两个 Cache，增大 block 的大小可以显著增加 hit 的次数，而增加路数后的性能受替换策略影响较大。若替换策略不合理，则可能会导致之后需要的块被踢出去了。MFU 策略受 block_size 的变化的影响最小，也说明了 MFU 策略的合理性。在本测试中，被交换的次数越多就越可能在合适的位置了，因此保留访问次数较少的块也就显得合理了。

4 Cache 性能实验

4.1 测试代码

```
 0x0 : addi $v0, $0, 512    | 20020200
 0x4 : addi $v1, $0, 28     | 2003001c
 0x8 : add $s0, $0, $0      | 00008020
 0xc : add $t1, $0, $0      | 00004820
0x10 : addi $s4, $0, 1      | 20140001
0x14 : Fori:                |
0x14 : add $t0, $0, $s0     | 00104020
0x18 : Forj:                |
0x18 : lw $s1, 0($t0)       | 8d110000
0x1c : add $t1, $t1, $s1    | 01314820
0x20 : add $t0, $t0, $v1    | 01034020
0x24 : slt $at, $t0, $v0    | 0102082a
0x28 : beq $at, $s4, Forj   | 1281fffb
0x2c : addi $s0, $s0, 4     | 22100004
0x30 : bne $s0, $v1, Fori   | 1470fff8
0x34 : nop                  | 00000000
0x38 : nop                  | 00000000
0x3c : nop                  | 00000000
```

这段代码是由我编写的，过程是利用二重循环遍历数组中的元素。此处，数组中一共有 128 个元素，每次遍历的步长由寄存器 $v1 的值控制。例如，这里 $v1 的值为 28，表示内循环一次跨越 7 个数，而外层循环则在 0 到 6 之间循环，等价于先访问下标模 7 余数为 0 的元素，再访问余数为 1 的元素... 可通过调节 $v1 的值，控制循环的步长。

4.2 测试结果

步长	$128 \times 4 \times 4$				$128 \times 8 \times 2$				$128 \times 2 \times 8$				$64 \times 4 \times 8$			
	RAND	FIFO	MFU	LRU	RAND	FIFO	MFU	LRU	RAND	FIFO	MFU	LRU	RAND	FIFO	MFU	LRU
1	0x20	0x20	0x20	0x20	0x20	0x20	0x20	0x20	0x20	0x20	0x20	0x20	0x40	0x40	0x40	0x40
2	0x38	0x40	0x35	0x40	0x3b	0x40	0x39	0x40	0x38	0x40	0x33	0x40	0x76	0x80	0x65	0x80
3	0x52	0x60	0x49	0x60	0x57	0x60	0x4f	0x60	0x55	0x60	0x43	0x60	0x69	0x7a	0x56	0x80
4	0x6a	0x80	0x60	0x80	0x74	0x80	0x69	0x80	0x6c	0x80	0x60	0x80	0x72	0x80	0x65	0x80
5	0x68	0x7a	0x5b	0x80	0x66	0x69	0x62	0x74	0x64	0x7d	0x56	0x80	0x5b	0x55	0x40	0x4e
6	0x55	0x60	0x56	0x80	0x5c	0x59	0x4e	0x66	0x5c	0x65	0x56	0x80	0x4e	0x4a	0x40	0x4e
7	0x53	0x4a	0x53	0x5d	0x55	0x4d	0x51	0x57	0x59	0x4a	0x45	0x63	0x4c	0x4e	0x40	0x4a
8	0x71	0x80	0x60	0x80	0x76	0x80	0x69	0x80	0x6e	0x80	0x60	0x80	0x71	0x80	0x65	0x80

表 1: 实验结果 (miss 次数)

我对四种大小的 Cache 分别进行了实验，其中步长的设置为 1 到 8，最终结果如上表所示，其中表中数值代表 miss 的总次数。

4.3 结果分析

通过分析表格中的数据，可得到以下结论：

1. 综合所有情况，NFU 是表现最好的替换策略，在几乎所有的情况中都给出了最优解。RAND 虽不及 MFU 效果好，但是 miss 次数也与 MFU 相差不大，因为其实现代价较小，所以也不失为一种合理的策略。而 FIFO 与伪 LRU 算法则时常表现不佳。

2. 横向比较，当步长较小时，block_size 的设置起到了很大的作用。当步长为 1 时，128 位的块明显效率更高，主要是因为所有的访存都是连续的，所以一次取数越多越好。而四种策略的性能差异不大，则是因为连续读取导致同一块中的元素会被连续访问，被访问后即可被踢出 Cache，所以替换策略作用不大。

3. FIFO 策略基本优于伪 LRU 算法，分析其本质是因为这里每个块被访问的次数总是常数，也就是说越近被访问的块之后被访问的概率越低，而这些块恰好是 LRU 保存的，所以也就解释了为什么 LRU 算法在这里的效率如此低下。

4. 观察步长对性能的影响。当步长为 2 的次幂时，FIFO 与 LRU 的性能总是最差，可能原因是步长为 2 的时候会一跳就是一整块，块内的元素很少同时在一次内循环中访问，而所需要的块经常被替换掉，所以导致 miss 次数非常多。

5. 因为缓存中的总块数是相同的，所以缓存大小对 miss 次数的影响并不大。

6. MFU 之所以效果好，是因为每个块被访问的次数总是固定的，所以访问次数越多就说明之后被访问的概率越小。

综合本节与上一节的实验结果，RAND 算法是比较稳定的，很少出现极端的情况，实现代价也比较小，所以是很合适的替换策略。

本实验中未考虑读入 block 所耗费的代价，实际上，读取不同大小的 block 所花费的时间必然是不同的。若想考虑该因素，可通过设置时钟延时来实现。

5 注意事项

5.1 遇到的问题

1. 对 Dmem 的初始化如果放在 initial 中好像会有些问题，我把它调整为 reset 为 1 时进行初始化就好了。
2. 仿真的结果和在板子上的结果可能会差异很大，在同一个时钟沿，仿真时两个信号会同时变化，而在板子上就不是了。可以通过将信号分别调整为上升沿和下降沿触发来解决该问题。
3. 综合时，可能会报一些权限不足的错误，也无法找到对应的 log 文件。此时应检查安装目录和项目目录中是否包含空格，如果不包含，可尝试重新综合，第二次说不定就好了。

5.2 注意点

1. 流水线的汇编代码需要在最后加入几句 nop，不然可能会出现奇怪的 xxx 信号
2. 在实现 Cache 时，应优先使用参数化的方法，虽然代码写起来要花更多脑筋，但是可以减少大量的代码量 (Cache 总共 150 行左右)，且便于复用。
3. 注意字对齐，访存地址必须是 4 的倍数。

6 申 A 理由

- 实现了参数化的高速缓存，复用性强
- 设计检验 Cache 正确性与性能的汇编代买
- 进行了大量实验，对比不同大小与替换策略 Cache 的性能

7 致谢

感谢罗翔同学与我进行深入的讨论，对实验的设计提出了宝贵的建议！