中国科学院大学计算机组成原理（研讨课）

实 验 报 告

学号：2021K8009929001　　姓名：谢嘉楠　　专业：计算机科学与技术

实验序号：5.4　　实验名称：高速缓存（Cache）设计

一．实验目的

1. 理解 Cache 设计思想，实现指令 Cache（icache）和数据 Cache（dcache）。
2. 调整缓存替换策略，测试性能变化。

二．实验背景知识

在理论课上我们已经接触到一些在多周期处理器的基础上提高 CPU 性能的方法：

1. 提高处理器的**并行性**。一般可以开发三种层次的并行性。

a) 第一个层次是**指令级并行**。指令级并行分成两种，一种是时间并行，一种是空间并行。时间并行可以通过**指令流水线**来实现；空间并行则通过**超标量技术**加上指令的**乱序执行**来实现。同时还可以使用**超长指令字**（VLIW）指令级来更大限度地利用指令级并行性。

b) 第二个层次是**数据级并行**。主要指**单指令多数据流**（SIMD）的向量结构。

c) 第三个层次是**任务级并行**。它的代表是**多核处理器**以及**多线程处理器**，这是目前计算机体系结构提高性能最主要的方法。

2. 利用计算机中的**局部性原理**。利用访存局部性进行优化是体系结构提升访存指令性能的重要方法。访存局部性包括时间局部性和空间局部性两种。时间局部性指的是一个数据被访问后很有可能多次被访问。空间局部性指的是一个数据被访问后，它邻近的数据很有可能被访问。计算机体系结构使用访存局部性原理来提高性能的地方很多，如**高速缓存（cache）、TLB、指令预取**都利用了访存局部性原理。

3. 通过优化电路结构/采用高速器件，提高主频。

（以上资料来自张科老师计算机组成原理的课件以及胡伟武老师的《计算机体系结构基础》）

本实验我们要实现的是高速缓存，包括 icache 和 dcache。设计 cache 首先需要考虑的是 cache 的编址方式，以及读/写命中和缺失的策略。

cache 与主存的地址映射方式主要有 3 种。**直接映射**（每个缓存块可以和若干个主存块对应，而一个主存块只能和一个缓存块对应）；**全相联映射**（每个缓存块可以映射到任意主存块，每个主存块也可以映射到任意缓存块）；**组相联映射**（某一主存块 j 按模 Q 映射到缓存第 i 组的任意一块）。

cache 通常的写策略组合方式有两种。如果写命中时采用 write-through（写穿法），则写缺失时采用 no-write-allocate（write-around），即写命中时数据既写入 cache 又写入主存，而写缺失时数据绕过 cache 直接写入主存。采用这种策略时对于写入操作 cache 没有任何优势，因为不管写命中还是写缺失，都需要访问一次主存。如果写命中采用 write-back（写回法），则写缺失时采用 write-allocate，即写命中时写操作时只把数据写入 cache 而不写入主存（当 cache 数据被替换出去时才写回主存），而写缺失时要先将写入地址所在的主存块读入 cache 中，再将数据写入 cache。这种策略下我们希望后续的读写操作会落在写入位置的附近，这样就能减少写入主存的次数。

在这次实验中，我们采用独立的的指令和数据 cache（Harvard 结构）。icache 和 dcache 都使用 4 路组相联与主存地址映射，并且 dcache 采用 write-back（写回法）+write-allocate 的写策略组合。

三．实验过程

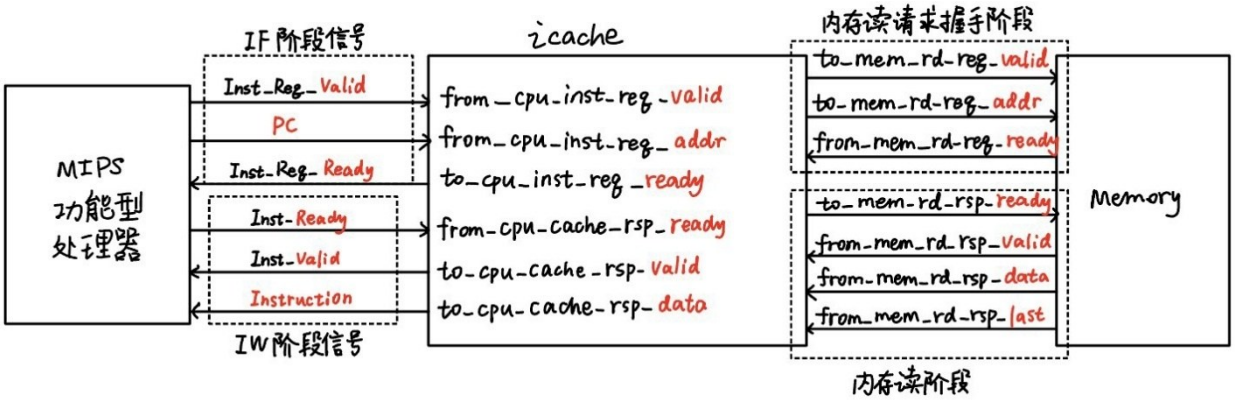
1. icache 的实现思路

icache 用于缓存内存中的指令，因为对于指令只需要读不需要写，因此 icache 只要考虑如何响应取指阶段的内存读请求即可。

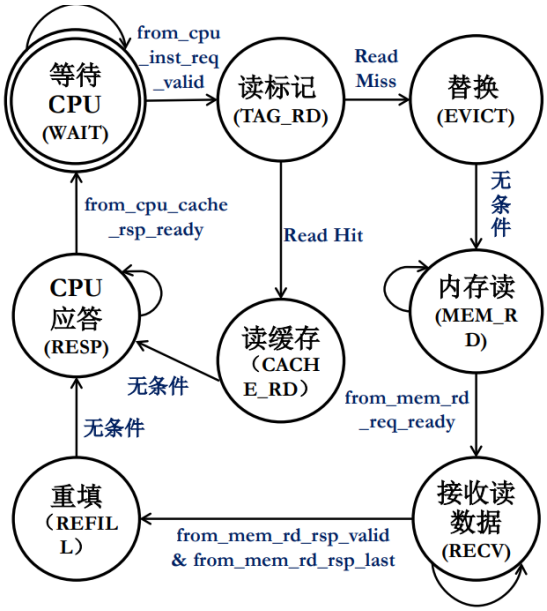
CPU、icache 和内存之间的接口如下图所示，CPU 的取指访存接口与 icache 相连，而 icache 的访存

接口与内存相连。CPU 在每条指令的取指（IF）阶段向 icache 发送取指的请求与地址（即 PC 的值），与 icache 进行第一次握手；然后 CPU 进入指令等待（IW）阶段等待 icache 返回指令。当 icache 找到目标指令后，与 CPU 进行第二次握手返回指令。

icache 访问内存时，也需要先发送内存读请求，以及内存读的起始地址和长度，等待内存握手；然握手成功后再进入数据传输阶段，从内存中读出一个 cache block 的指令数据，每读一个字都需要一次与内存的握手，直到最后一个字时内存会将 last 拉高。



icache 的状态转换图在课件中已经画出，如下所所示。icache 初始时处于 WAIT 状态，当接收到 CPU 的取指请求后进入 TAG_RD 状态，判断缓存中是否有需要的指令。如果读命中，则将 icache 中对应地址的数据读出（实际上读缓存的操作是异步的，不需要一个时钟周期来执行）；如果读缺失，则进入 EVICT 状态，根据替换算法选出一组中要被替换的 cache block。然后进入 MEM_RD 状态，向内存发出读请求。当内存握手后，进入 RECV 状态，接受从内存中读出的一个 cache block 数据，暂存在寄存器中。随后进入 REFILl 状态，将新的 cache block 数据写入被替换的位置（同时读出需要的指令）。不论读命中还是读缺失，最后都会有一个 RESP 状态，将读到的指令送回给 CPU，最后返回到 WAIT 状态。

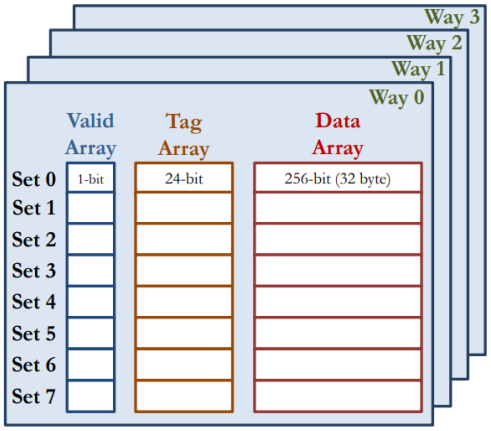


icache 的状态集如下所示。

```
localparam   WAIT     = 8'b00000001,  // 等待CPU
             TAG_RD   = 8'b00000010,  // 读标记
             CACHE_RD = 8'b00000100,  // 读缓存
             RESP     = 8'b00001000,  // CPU应答
             EVICT    = 8'b00010000,  // 替换
             MEM_RD   = 8'b00100000,  // 读内存
             RECV     = 8'b01000000,  // 接收读数据
             REFILL   = 8'b10000000;  // cache block重填
```

实验中实现的 icache 共分为 8 组，每一组中有 4 个 cache block，所以总共是 32 个 cache block。每个 cache block 的大小是 256bit（32B），也就是说可以存放 8 个指令字，而且每个 cache block 中数据在主存中的起始地址一定是 32byte 对齐的（即地址的低 5 位全为 0）。同时每个 cache block 还需要有一个 tag 域，用来存放其在主存中的地址高位。每个 ccache block 还有一个 valid 有效位，用来标识其中的数据是否可用，当某个 cache block 被选中要替换，但还没有完成替换时，valid 位应该置为 0。

icache 的内部结构以及主存地址的格式如下图所示。




```
// 在输入地址中截取相应位置的信号
wire [ 2:0] addr_index      = from_cpu_inst_req_addr[ 7:5]; // 指令地址的index域，表示cache组内地址
wire [ 4:0] addr_offset     = from_cpu_inst_req_addr[ 4:0]; // 指令地址的offset域，表示cache块内偏移地址
wire [23:0] addr_tag        = from_cpu_inst_req_addr[31:8]; // 指令地址的tag域，用于和cache block中的标签tag进行比较
```

当需要判断给定的指令地址是否在缓存中时，首先根据地址中的 cache 组地址字段选出对应的组，然后读出该组的 4 个 cache block 对应的 4 个 tag 域，与指令地址中的 tag 字段比较，若某一个 cache block 的 tag 与与指令地址的 tag 相同，则读命中；如果没有一个是匹配的，则读缺失。判断是否命中的结果可以用一个 4 位的信号 hit 来表示。

```
wire [3:0] read_hit;
assign read_hit[0] = valid_array[0] [addr_index] && (addr_tag == rtag_0);
assign read_hit[1] = valid_array[1] [addr_index] && (addr_tag == rtag_1);
assign read_hit[2] = valid_array[2] [addr_index] && (addr_tag == rtag_2);
assign read_hit[3] = valid_array[3] [addr_index] && (addr_tag == rtag_3);
```

实际在硬件实现时，icache 的数据域是存放在 4 个 Data_Array 中的，分别对应 4 路，每个 data_array 是一个 8*256 位的二维数组。标记（tag）域也是存放于 4 个 tag_array 中，每个 tag_array 是一个 8*24 位的二维数组。valid 有效位则存放于一个 4*8 位的二维数组中。如上图所示。

data_array 和 tag_array 的例化：

```
// icache的tag域与data域写使能控制信号
wire [ 3:0] cache_wen;
// icache每一组组内的读写地址
wire [ 2:0] raddr;
wire [ 2:0] waddr;
// icache每一组的tag域读写数据
wire [23:0] rtag_0, rtag_1, rtag_2, rtag_3;
wire [23:0] wtag;
// icache每一组的data域读写数据
wire [255:0] rdata_0, rdata_1, rdata_2, rdata_3;
wire [255:0] wdata;
```

```
// ichche中的valid_array
reg [7:0] valid_array [3:0];
// 例化icache中的4个way上的tag_array和data_array
tag_array tag_array_0(clk,waddr,raddr,cache_wen[0],wtag,rtag_0);
tag_array tag_array_1(clk,waddr,raddr,cache_wen[1],wtag,rtag_1);
tag_array tag_array_2(clk,waddr,raddr,cache_wen[2],wtag,rtag_2);
tag_array tag_array_3(clk,waddr,raddr,cache_wen[3],wtag,rtag_3);

data_array data_array_0(clk,waddr,raddr,cache_wen[0],wdata,rdata_0);
data_array data_array_1(clk,waddr,raddr,cache_wen[1],wdata,rdata_1);
data_array data_array_2(clk,waddr,raddr,cache_wen[2],wdata,rdata_2);
data_array data_array_3(clk,waddr,raddr,cache_wen[3],wdata,rdata_3);
```

每个 data_array 和 tag_array 的读操作是异步的，而写操作是同步的，由写使能信号和时钟信号共同控制。所以 4 个 data_array（tag_array）可以共用一个写入接口。只有在读缺失的重填（REFILL）状态时，icache 的 data_array 和 tag_array 才需要写入，所以它们的写使能可以用同一个 4 位信号控制（cache_wen）。

valid_array 需要在 EVICT 状态时将被选中的 cache block 有效位置 0，然后在 REFILL 状态再将它重新改为 1。如下所示。

```
// valid_array寄存器更新
always @(posedge clk)begin // valid_array寄存器的赋值
    if(rst == 1'b1)begin
        valid_array[0] <= 8'b0; // 初始全部设为0
        valid_array[1] <= 8'b0;
        valid_array[2] <= 8'b0;
        valid_array[3] <= 8'b0;
    end
    else begin
        case(current_state)
        EVICT:begin
            valid_array[way_counter][addr_index] <= 1'b0;
            end
        REFILL:begin
            valid_array[way_counter][addr_index] <= 1'b1;
            end
        endcase
    end
end
```

当 icache 读缺失时，需要将目标指令所在的内存块读出，替换掉 icache 中的某一个 cache block 的数据。而替换算法就是要在一组的 4 个 cache block 中选出一个来替换掉。在理论课上我们学过一些替换算法，如先进先出（FIFO）算法，近期最少使用（Least Recently Used,LUR）算法，以及随机法。我在实验中尝试了随机法和先进先出算法，这两种替换策略都比较简单，而且开销很小，但是它们都没有根据访存的局部性原理，所以不能提高 cache 的命中率。

随机法只需要另外增加一个 32 位的随机数寄存器 random_reg，它在每次读缺失时根据当前的值，用伪随机数生成算法更新为一个新的随机数，而被替换的 cache block 的编号就可以取 random_reg 的低 2 位（只需要在 00,01,10,11 中取值）。

先进先出（FIFO）法实现起来的开销更小，只要使用 2 位的寄存器 way_counter，在每次读缺失时将它的值+1 即可，这样它的值就在 00,01,10,11 之间循环（11 加 1 后进位丢失，又回到 00）。严格的 FIFO 应该需要给每一 cache 组都配上一个 way_counter，这样才能在每一组内的替换实现先进先出，而我实际上是所有的 cache 组共用一个 way_counter，不同组之间的替换都会改变它的值，就会相互影响。不过这种替换策略并没有出现什么问题，而且通过这种方法，可以使用非常小的寄存器开销来实现替换。下面是

对 way_counter 的赋值。

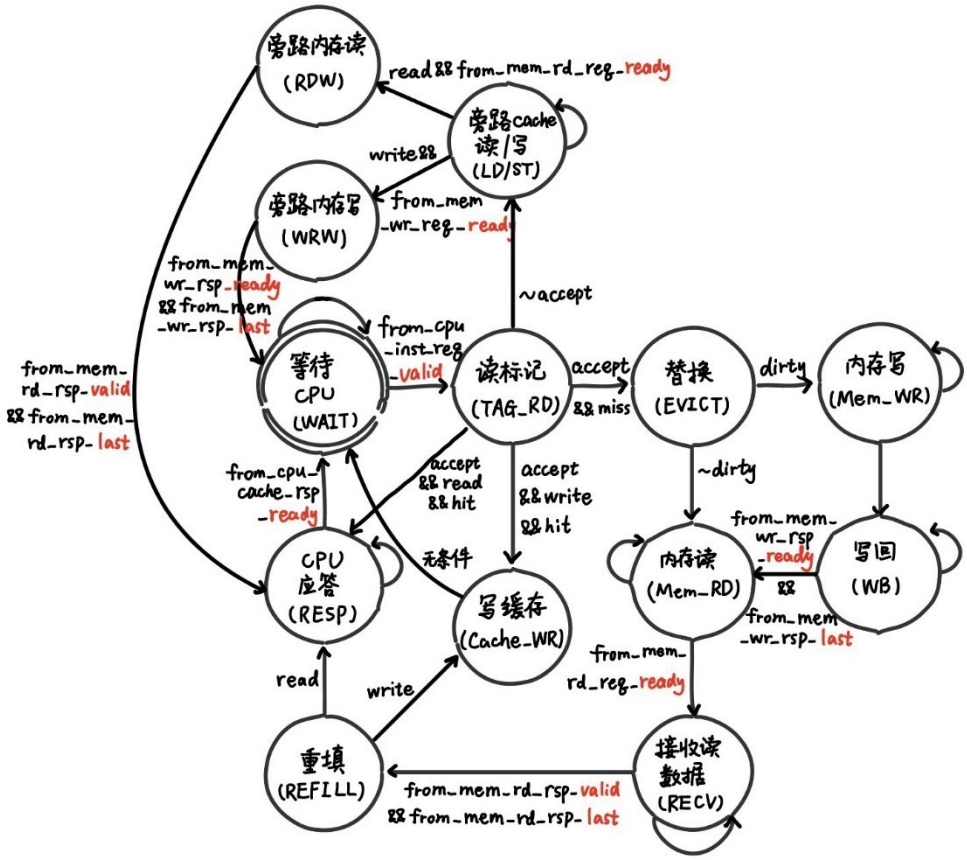
```
reg [1:0] way_counter;

always @(posedge clk)begin
    if(rst == 1'b1)
        way_counter <= 2'b0;
    else if(current_state == TAG_RD && read_hit == 4'b0) // 读缺失时更新
        way_counter <= way_counter + 2'b1;
end
```

2. dcache 的实现思路

dcache 的读操作与 icache 时相同的，但是它还需要支持写操作，所以它的控制逻辑要比 icache 麻烦上不少。

除此之外，还要考虑不可缓存的情况。内存空间中地址 0x00 - 0x1F 以及 0x4000_0000 以上的地址范围（I/O 空间）在 dcache 中是不可缓存的。由此我在 icache 状态机的基础上增加了一些状态，组成了如下的状态机。

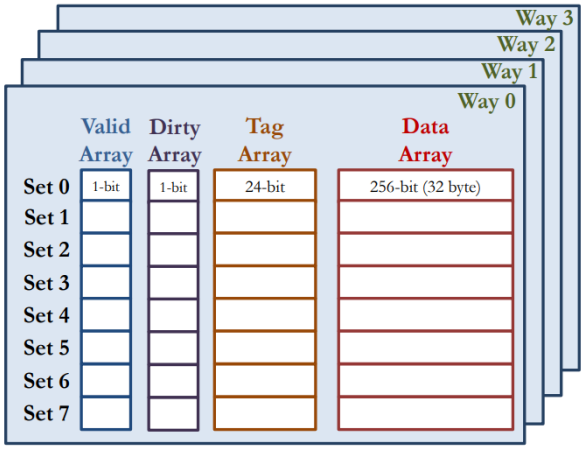


当 load 型指令进入到 LOAD 阶段，或者 store 型指令进入到 STORE 阶段时，CPU 会向 dcache 发送数据访存请求（拉高 from_cpu_inst_req_valid 信号），dcache 从 WAIT 状态进入到 TAG_RD 状态。在 TAG_RD 状态，dcache 需要根据访存地址判断地址是否是可缓存的，以及数据是否在缓存中。如果地址本身属于不可缓存范围，则进入 LDST（旁路 cache 读写）状态。如果地址是可缓存的，但是读缺失或者写缺失，则进入 EVICT 状态。如果地址可缓存，且读命中，则进入 RESP 状态应答 CPU（这里省去了 icache 中的 CHACHE_RD 状态，因为读 cache 全部是组合逻辑，不需要专门用一个时钟周期）；如果地址可缓存，且写命中，则进入 CACHE_WR 状态，将数据写入 dcache。

在 LDST 状态，如果 CPU 发送的是读请求，则向内存发送内存读请求；如果 CPU 发送的是写请求，则向内存发送写请求。等待内存的握手。然后分别进入 RDW 状态和 WRW 状态，从内存中读出或者写入一个字。需要注意的是，写入内存后 dcache 就可以直接回到 WAIT 状态了，不需要向 CPU 返回信息；而从内存读出数据后，dcache 需要进入 RESP 状态，将数据传给 CPU，并与 CPU 握手后才能回到 WAIT 状态。

在 EVICT 状态，还是通过替换算法选出被替换的 cache block，但是此时还需要考虑这个 cache block 中的数据是否 dirty。如果数据不是 dirty 的，说明它在 dcache 中没有被改动过，则可以将它直接替换掉；但如果它是 dirty 的，说明它在 dcache 中内改动过，需要先把它写回内存的对应位置（将对应的内存块更新），才能将它替换掉。因此，如果 dirty，还要增加 MEM_WR 和 WB 两个状态，才能进入 MEM_RD 状态读出新的 cache block。读出新的 cache block 后，进入 REFILL 状态重填。此时如果 CPU 是读请求，则进入 RESP 状态（此时一定已经读命中）；如果 CPU 是写请求，则进入 CACHE_WR 状态，在新的 cache block 中写入数据。

dcache 的内部结构如下图，它增加了一个 dirty_array，它与 valid_array 类似，是一个 4*8bit 的二维数组，用来记录对应的 cache block 是否 dirty。dirty_array 的更新如下所示。




```
// dirty_array 寄存器的赋值
always@(posedge clk)begin
    if(rst == 1'b1)begin
        dirty_array[0] <= 8'b0;
        dirty_array[1] <= 8'b0;
        dirty_array[2] <= 8'b0;
        dirty_array[3] <= 8'b0;
    end
    else begin
        if(current_state == CACHE_WR)
            dirty_array[hit_index][addr_index] <= 1'b1;
        else if(current_state == REFILL)
            dirty_array[way_counter][addr_index] <= 1'b0;
    end
end
```

在 REVCT 阶段的替换策略与 icache 相同，这里就不再重复了。dcache 的信号赋值明显比 icache 要复杂一些，最恐怖的莫过于写命中时对命中的 cache block 进行修改的操作。它需要先把命中的 cache block 中的 256 位数据读出，然后根据块内地址取出要写的一个字，接下来根据 strb 的值（MIPS 指令集中有 wb 和 wh 指令，需要写入一个字节或者半个字）修改这个字，最后还要把改好的这个字和 cache block 中的另外 7 个字拼接起来，再写入 cache block。整个操作除了写入全部都是组合逻辑完成，给人的感觉就是非常耗费组合逻辑资源，因为 cache 的读写位宽只能是一整个 cache block，它不能只读写一个字，所以在写入一个字时就挺需要额外的很多截取和拼接操作，而且这些都要在 data_array 的外部完成。（实际上这些额外的操作如果在 data_array 内部完成开销还要大，因为每个 data_array 中都要增加，就相当于使用了 4 倍的资源）反正这一块我还没有想到什么更方便的办法。

3. 在 MIPS 处理器中实现指令提交的信息封装

最后需要添加的是 MIPS 处理器中 inst_retire 信号的赋值，它用于仿真测试时与金标准输出的对比。inst_retire 信号寄存器写使能 RF_wen，程序计数器 PC，寄存器写地址 RF_waddr 以及寄存器写数据 RF_wdata 拼接起来，在指令的写回阶段与金标准对比。这里需要注意的是它输出的 PC 值是执行完的这

条指令的 PC，而不是当前的 PC（在写回阶段 PC 肯定已经更新到下一条指令的 PC 了），所以比较方便的做法是用寄存器 PC_retired 来存放执行完的指令的 PC，它在每次 PC 更新的同时将旧的 PC 值存下来，如下所示。

```
// PC_retired寄存器的赋值
always @(posedge clk)begin
    if(current_state == INIT)
        PC_retired <= 32'b0;
    else if(current_state == IW && Inst_Valid)
        PC_retired <= PC_reg;       // PC更新之前，将退休指令的PC保存下来
end
```

四．实验总结

1. 性能分析

经过 FPGA_run 阶段的 benchmark 测试，其中 coremark 和 dhrystone 性能测试的结果如下表所示，分为了 4 种情况：同时使用 icache 和 dcache；只使用 icache；只使用 dcache；以及不使用 cache。

Benchmark 性能测试结果	Dhrystones per second	coremark: time
icache+dcache	4796	38760.87ms
icache alone	3413	39836.66ms
dcache alone	508	401031.21ms
no cache	495	402137.12ms

在 coremark 性能测试的结过中，每一种情况下的 Iterations/Sec 的跑下来都是 0（可能是因为处理器的性能实在太弱了），而且我也不清楚它的 tatal us 代表了什么，所以只能看它完成测试的总时长时间来定性地对比一下。

在 Dhrystones per second 的数据上，不使用 cache 时是最低的；只使用 dcache 时只比没有 cache 时稍微高了一些，但提升时分不明显；只使用 icache 时性能提升非常明显；而 icache 和 dcache 组合使用是性能提升会更明显，几乎达到了没有 cache 时速度的 10 倍之多。而且，很神奇的是，没有 icache 时加入 dcache 对性能提升的效果很微薄（从 4095 到 508），但是在 icache 的基础上加入 dcache 对于性能的提升却还是比较明显的（从 3413 到 4796）。

上面的结果可以在一定程度上说明，icache 相对于 dcache 对性能的提升更加显著。这可能是因为所有指令都需要取指，而并不是所有指令都需要数据访存，所以说 dcache 的效果会随着程序中访存指令的比例改变而改变。而且还有一个可能的因素是指令访问的时间和空间局部性会更强一些，而数据访问的时间和空间局部性相对而言没有那么强，所以 icache 的命中率可能会高于 dcache（不过我还没有统计过 icache 和 dcache 的命中率）。

总体上来看，加入 cache（尤其是 icache）对于 CPU 性能的提升效果真的非常明显，这也与之前多周期处理器的性能统计结果相匹配。之前在多周期处理器中，测试程序跑下来计算出的 CPI 差不多都在 70~100 之间，也就是说每条指令平均都需要 70~100 个时钟周期才能完成。但是在指令周期中涉及到的实际的运算在执行（EX）阶段只需要一个时钟周期就可以完成了，所以说指令周期的大多数（超过 90%）的时间都花在取指和访存上了，也就是说访存的速度就是制约 CPU 性能的一个瓶颈。而 cache 就是针对这个瓶颈而设计的，所以对于指令执行速度的提升效果就非常明显。

2. 调试过程总结

在写 cache 的过程中我一直有一个矛盾，就是到底是应该按照状态转换的顺序来写，还是应该按照 cache 的输入输出接口的顺序来写。一开始写的时候我感觉按照状态转换图的顺序来写感觉比较顺，于是就一个状态一个状态地写，但是写完之后我发现这样写导致整体看起来非常乱，同一个信号的赋值在不同的两个状态会有不同的情况，这样在排查错误的时候就需要上下翻来翻去，很容易出错。所以后来我还是改成了按照定义接口的顺序来写，考虑一个信号在不同状态下的赋值情况。这样写会使得代码看起来更加清晰，也容易看出自己的错误。

dcache 的调试可能是我整个计组实验课中最头疼的一次调试了。最后那一个错误我花了 2 天才找出

来，因为它只在一个仿真样例中出错，而且出错的时间非常晚，在 1400 万 ns 左右。关键是这个出错的时间只是代表了将数据读出时出错，而实际上可能在很久之前写入时就错了。最终我还是找到了问题，结果是 dirty_array 的赋值出错了（是因为我在 dirty_array 更新时直接将 4 位的 hit 信号赋给了相应的 dirty_array 的位置，导致之前 dirty 的位置可能被抹除）。

最后是一个在 bit_gen 阶段出现的问题。因为一开始我考虑到每次 dcache 的写入实际上只是写一个字，但是却需要写入一整个 cache block，非常不方便，所以在 data_array 模块的输入中有增加了一些信号，用来处理只写入一个字的情况（一个 32 bit 的 wword 表示写入的字，一个 1bit 的 flag 用于标志写入字还是块，一个 5 位的 offset 表示块内偏移量），这样虽然在代码上可以使用“+:”来拼接更加简便，但是相当于将写命中时的截取、拼接操作放到了每一个 data_array 中，实际上的组合逻辑开销更大了。结果就是在 bit_gen 阶段会因为板卡资源不足而生成不下去。最终我尝试把这些截取、拼接操作放到 data_array 外面（不改变 data_array 内部结构），就可以生成电路了。

总之，这个实验是非常有意义的，它让我初次体验到了提高 CPU 性能的重要方法，理解了 cache 的运行原理，也看到了 cache 的加入对于 CPU 性能的提升之巨大。