

# 中国科学院大学

## 《计算机组成原理(研讨课)》实验报告

姓名 宋俊仪 学号 2023K8009929018 专业 计算机科学与技术  
实验项目编号 5.2 实验名称 高速缓存(Cache)设计

注 1: 撰写此 Word 格式实验报告后以 PDF 格式保存 SERVE CloudIDE 的 `/home/serve-ide/cod-lab/reports` 目录下(注意: reports 全部小写)。文件命名规则: `prjN.pdf`, 其中 `prj` 和后缀名 `pdf` 为小写, `N` 为 1 至 4 的阿拉伯数字。例如: `prj1.pdf`。PDF 文件大小应控制在 5MB 以内。此外, 实验项目 5 包含多个选做内容, 每个选做实验应提交各自的实验报告文件, 文件命名规则: `prj5-projectname.pdf`, 其中“-”为英文标点符号的短横线。文件命名举例: `prj5-dma.pdf`。具体要求详见实验项目 5 讲义。

注 2: 使用 `git add` 及 `git commit` 命令将实验报告 PDF 文件添加到本地仓库 master 分支, 并通过 `git push` 推送到实验课 SERVE GitLab 远程仓库 master 分支(具体命令详见实验报告)。

注 3: 实验报告模板下列条目仅供参考, 可包含但不限定如下内容。实验报告中无需重复描述讲义中的实验流程。

## 一、Cache 设计分析

### 1 D-Cahce 的设计分析

#### • D\_cache 各信号分析

##### 1. CPU 接口 (CPU Interface)

这组接口负责 Cache 与 CPU 核心之间的直接通信。

##### 1.1 CPU 向 Cache 发起请求

这是 CPU 向 Cache 发出读/写命令的通道。

- `input from_cpu_mem_req_valid`: CPU 请求有效信号。o 当 CPU 想要进行一次内存访问(读或写)时, 它会把这个信号置为高电平。o 在 `valid` 为高期间, 下面几个相关的信号(地址、数据等)都是有效的。
- `input from_cpu_mem_req`: CPU 请求类型信号。o 0: 表示这是一个读请求。o 1: 表示这是一个写请求。
- `input [31:0] from_cpu_mem_req_addr`: CPU 请求地址。o CPU 想要访问的 32 位内存地址。代码注释提到地址是 4 字节对齐的, 这意味着地址的最低两位总是 00。
- `input [31:0] from_cpu_mem_req_wdata`: CPU 写入数据。o 如果是一个写请求(`from_cpu_mem_req` 为 1), 这个端口上是 CPU 想要写入的 32 位数据。o 如果是读请求, 这个端口的数据无效。
- `input [3:0] from_cpu_mem_req_wstrb`: CPU 写入选通(Write Strobe)。o 这是一个 4 位的信号, 用于指示 32 位数据中的哪些字节是有效的。`wstrb[i]` 对应第 `i` 个字节(从低到高)。o 例如: `4'b0001` 表示只写最低字节, `4'b1111` 表示写入全部 4 个字节。这允许 CPU 进行非对齐的或部分字节的写入。

• `output to_cpu_mem_req_ready`: Cache 就绪信号。o 这是 Cache 对 CPU 请求的“回应”。当 Cache 能够接收一个新的 CPU 请求时, 它会把这个信号置为高电平。o 如果 Cache 正在处理一个复杂的未命中(比如正在写回或填充), 它会拉低 `ready`, 告诉 CPU “请稍等, 我很忙”。

##### 1.2 Cache 向 CPU 返回响应

这是 Cache 向 CPU 返回读操作结果的通道。

- `output to_cpu_cache_rsp_valid`: Cache 响应有效信号。o 当 Cache 准备好了一个读请求的结果时, 它会把这个信号置高, 通知 CPU 可以来取数据了。
- `output [31:0] to_cpu_cache_rsp_data`: Cache 响应数据。o 当 `to_cpu_cache_rsp_valid` 为高时, 这个端口上是 Cache 返回给 CPU 的 32 位读取数据。

- `input from_cpu_cache_rsp_ready`: CPU 就绪信号。o CPU 通过这个信号告诉 Cache, 它已经准备好接收返回的数据了。这在高性能 CPU 中很重要, 例如 CPU 的流水线可能发生暂停(stall)。

2. 内存/IO 读接口 (Memory/IO Read Interface) 这组接口用于当 Cache 需要从主内存(或 I/O 设备)读取数据时(例如, 发生读未命中或写未命中时)。

#### 2.1 Cache 向内存发起读请求

- `output to_mem_rd_req_valid`: Cache 读请求有效。o 当 Cache 需要从内存读数据时, 它会置高此信号。
- `output [31:0] to_mem_rd_req_addr`: Cache 读请求地址。o Cache 想要读取的内存地址。o I/O 读: 地址是 4 字节对齐的。o 缓存填充: 地址是 32 字节(一个缓存行)对齐的。
- `output [7:0] to_mem_rd_req_len`: Cache 读请求长度。o 表示这次请求需要读取多少个数据节拍(beat)。这是一个突发传输(burst)的长度。o 0: 表示只读 1 个数据节拍(32 位)。用于 I/O 读。o 7: 表示要连续读取 8 个数据节拍(总共 256 位)。用于缓存填充。
- `input from_mem_rd_req_ready`: 内存就绪信号。o 内存控制器通过此信号告知 Cache, 它已经准备好接收读请求了。

#### 2.2 内存向 Cache 返回读数据

- `input from_mem_rd_rsp_valid`: 内存读响应有效。o 内存每准备好一个 32 位的数据节拍, 就会将此信号置高。
- `input [31:0] from_mem_rd_rsp_data`: 内存返回的读数据。o 内存返回的 32 位数据。
- `input from_mem_rd_rsp_last`: 内存读响应最后节拍。o 当内存发送的是一次突发传输中的最后一个数据节拍时, 它会同时将此信号置高。这告诉 Cache, 这次读取操作完成了。
- `output to_mem_rd_rsp_ready`: Cache 就绪信号。o Cache 通过此信号告诉内存, 它已经准备好接收下一个数据节拍了。

3. 内存/IO 写接口 (Memory/IO Write Interface) 这组接口用于当 Cache 需要向主内存(或 I/O 设备)写入数据时(例如, 写回脏数据块或处理 Bypass 写操作)。

#### 3.1 Cache 向内存发起写请求

- `output to_mem_wr_req_valid`: Cache 写请求有效。o 当 Cache 需要向内存写数据时, 置高此信号。
- `output [31:0] to_mem_wr_req_addr`: Cache 写请求地址。o 要写入的内存地址。
- `output [7:0] to_mem_wr_req_len`: Cache 写请求长度。o 0: 写 1 个数据节拍。用于 I/O 写和部分写(write miss, 虽然代码注释有误, 但通常 write miss 只写一个字)。o 7: 写 8 个数据节拍。用于写回整个脏的缓存行。
- `input from_mem_wr_req_ready`: 内存就绪信号。o 内存控制器告知 Cache, 它已准备好接收写请求。

#### 3.2 Cache 向内存发送写数据

- `output to_mem_wr_data_valid`: Cache 写数据有效。o Cache 每准备好一个要写入的 32 位数据节拍, 就置高此信号。
- `output [31:0] to_mem_wr_data`: Cache 的写数据。o 要写入内存的 32 位数据。
- `output [3:0] to_mem_wr_data_strb`: Cache 的写选通。o 4'b1111: 在写回整个缓存行时, 总是写入全部字节。o 其他值: 在处理 I/O 写或 Bypass 写时, 会直接使用来自 CPU 的原始 wstrb。
- `output to_mem_wr_data_last`: Cache 写数据最后节拍。o 当 Cache 发送的是本次写操作的最后一个数据节拍时, 置高此信号。
- `input from_mem_wr_data_ready`: 内存就绪信号。o 内存告知 Cache, 它已准备好接收下一个写数据节拍。

## • D\_cache 状态转移设计

理解各项信号后，我们可以分析 D-Cache 的状态转移设计。Cache 的状态转移图如下（见图 1）：

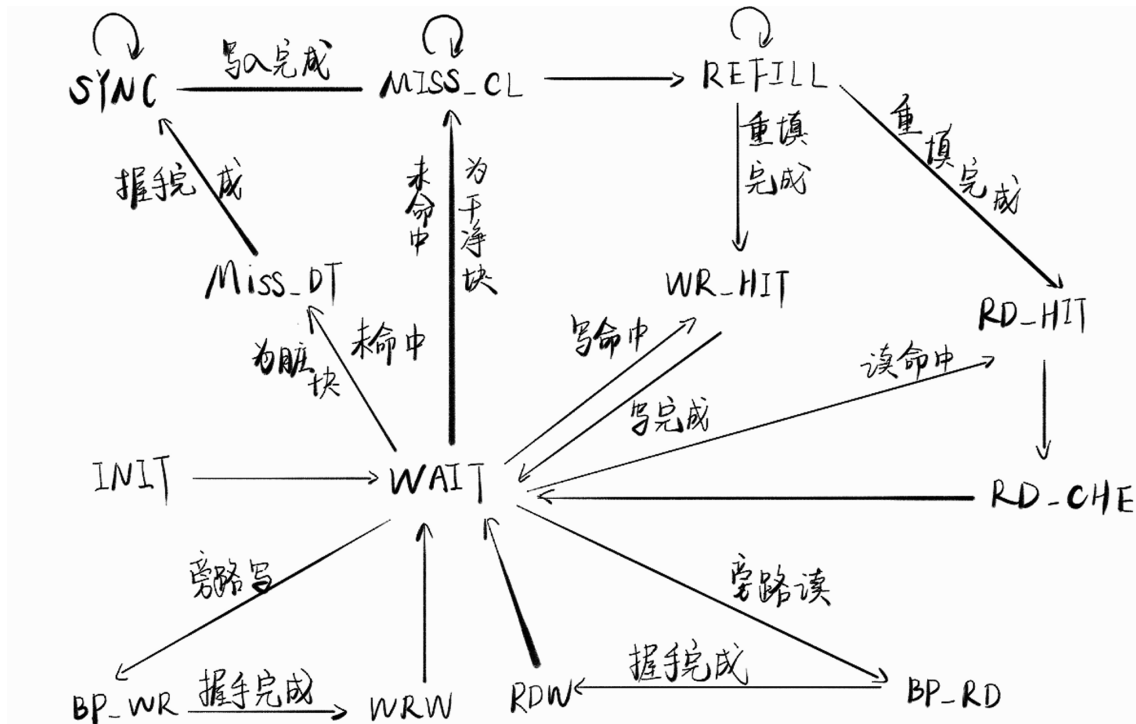


图 1: D-cache 状态转移图

根据上面设计的状态转移，具体的状态定义代码如下，各个状态的工作解释在代码注释中体现。然后依然与之前的实验保持一致，使用三段式实现状态转移，其中第二段的组合逻辑计算 next\_state 的代码逻辑如下图（见图 2、图 3）：

```
/* 有限状态机 (FSM) 状态定义 */
localparam INIT = 13'h0001; /* 初始化状态 */
localparam WAIT = 13'h0002; /* 等待CPU请求，并读取标签 */
localparam MISS_DT = 13'h0004; /* 未命中，且替换块为脏块 */
localparam SYNC = 13'h0008; /* 同步：将旧的脏块写回内存 */
localparam MISS_CL = 13'h0010; /* 未命中，且替换块为干净块 */
localparam REFILL = 13'h0020; /* 填充：从内存读取新块到Cache */
localparam WR_HIT = 13'h0040; /* 写命中 */
localparam RD_HIT = 13'h0080; /* 读命中 */
localparam RD_CHE = 13'h0100; /* 检查/送出：将Cache中的数据发送给CPU */
localparam BP_WR = 13'h0200; /* 旁路写：CPU -> 内存（发送请求） */
localparam BP_RD = 13'h0400; /* 旁路读：CPU -> 内存（发送请求） */
localparam WRW = 13'h0800; /* 旁路写等待：CPU -> 内存（等待数据传输完成） */
localparam RDW = 13'h1000; /* 旁路读等待：内存 -> CPU（等待数据传输完成） */
```

图 2: 各状态的工作解释

```

assign next_state = (
{13{state_INIT}} & WAIT | // 初始化后进入等待
{13{state_WAIT}} & (
{13{from_cpu_mem_req_valid}} & ( // 收到CPU请求
{13{hit}} & (from_cpu_mem_req ? WR_HIT : RD_HIT) | // 命中 -> 根据读写进入相应状态
{13{miss && !bypass}} & (dirty ? MISS_DT : MISS_CL) | // 未命中(非旁路) -> 根据脏位决定先写回还是直接填充
{13{bypass}} & (from_cpu_mem_req ? BP_WR : BP_RD) // 旁路 -> 根据读写进入旁路状态
) |
{13{!from_cpu_mem_req_valid}} & WAIT // 未收到CPU请求 -> 保持等待
)
{13{state_MISS_DT}} & (from_mem_wr_req_ready ? SYNC : MISS_DT) | // 脏未命中 -> 内存准备好后开始同步(写回)
{13{state_SYNC}} & (wr_last ? MISS_CL : SYNC) | // 写回 -> 写完后进入干净未命中状态, 否则继续写回
{13{state_MISS_CL}} & (from_mem_rd_req_ready ? REFILL : MISS_CL) | // 干净未命中 -> 内存准备好后开始填充
{13{state_REFILL}} & ( // 填充中
{13{rd_last && from_cpu_mem_req}} & WR_HIT | // 填充完毕且原请求是写 -> 转为写命中
{13{rd_last && !from_cpu_mem_req}} & RD_HIT | // 填充完毕且原请求是读 -> 转为读命中
{13{!rd_last}} & REFILL // 未填充完 -> 继续填充
)
{13{state_WR_HIT}} & WAIT | // 写命中完成 -> 返回等待
{13{state_RD_HIT}} & RD_CHE | // 读命中 -> 进入送出数据状态
{13{state_RD_CHE}} & WAIT | // 数据送出完成 -> 返回等待
{13{state_BP_WR}} & (from_mem_wr_req_ready ? WRW : BP_WR) | // 旁路写 -> 内存准备好后进入等待数据传输
{13{state_BP_RD}} & (from_mem_rd_req_ready ? RDW : BP_RD) | // 旁路读 -> 内存准备好后进入等待数据传输
{13{state_WRW}} & (wr_last ? WAIT : WRW) | // 旁路写等待 -> 写完后返回等待
{13{state_RDW}} & (rd_last ? WAIT : RDW) // 旁路读等待 -> 读完后返回等待
);

```

图 3: next\_state 的组合逻辑计算

#### • D\_cache 的存储设计

本次实验中,讲义要求实现四路 8 组的 D-cache,因此我设计了一个 4 路 8 组的 Cache 存储结构。但是我为了解决它可以更好地适应不同的 Cache 大小,我将 Cache 的存储结构设计成了一个参数化的模块,可以通过宏定义来设置 Cache 的大小。存储示意图如下(见图 4):

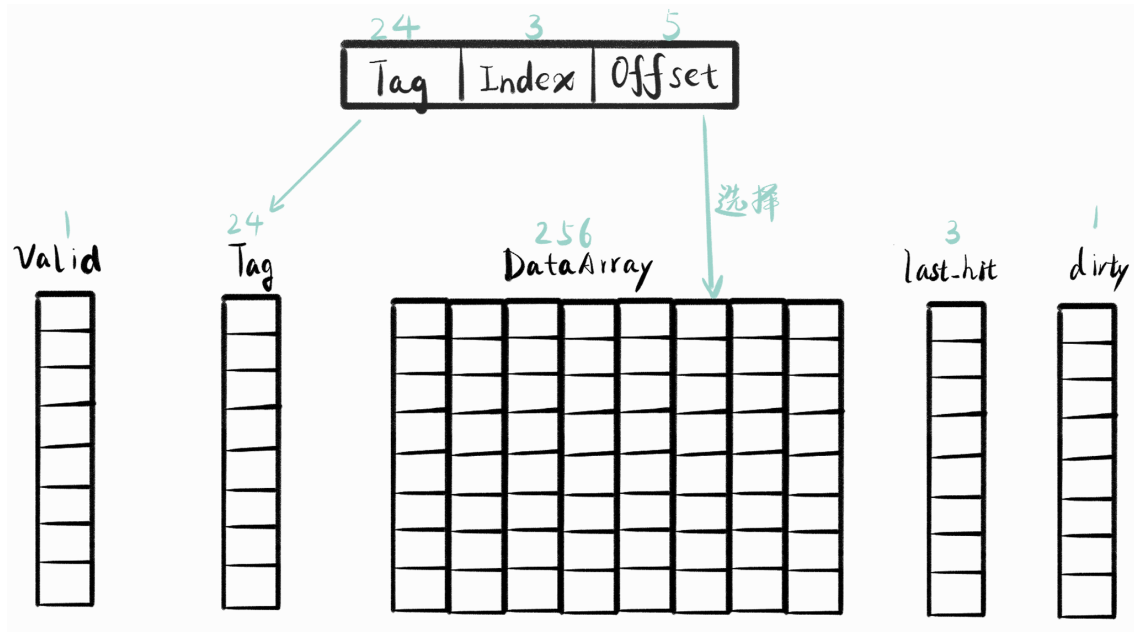


图 4: D-cache 存储结构示意图

除了框架中已有的 tag\_array、data\_array 以外,根据上面的 D-cache 存储结构,我还需要设计一个 one-bit 位的存储模块,用于存储 Dirty 位和 Valid 位。这个模块的设计如下(见图 5):

```

module one_bit_array (
    input                clk,
    input                rst,
    input  [`ONE_BIT_ADDR_WIDTH - 1 : 0] waddr, // 写地址
    input  [`ONE_BIT_ADDR_WIDTH - 1 : 0] raddr, // 读地址
    input                wen,    // 写使能
    input                wdata,  // 写入数据 (1位)
    output               rdata   // 读出数据 (1位)
);

reg array [(1 << `ONE_BIT_ADDR_WIDTH) - 1 : 0]; // 存储阵列

integer i; // for循环变量
always @(posedge clk) begin
    if (rst) begin // 复位逻辑
        for (i = 0; i < (1 << `ONE_BIT_ADDR_WIDTH); i = i + 1)
            array[i] <= 1'h0; // 将所有位清零
        end
    else if (wen) begin // 写入逻辑
        array[waddr] <= wdata;
    end
end

assign rdata = array[raddr]; // 异步读出

```

图 5: Dirty 位和 Valid 位 (one-bit 位) 的存储模块

## 2 I-Cahce 的设计分析

完成 D-Cache 的设计后, 接下的 I-Cache 就很简单了, 因为 I-cache 就是简单版的 D-cache。它不会有写操作, 也没有 Dirty 位和 Valid 位的存储。因此这里不再赘述 I-cache 的各个信号含义和 I-cache 的存储设计。

讲义中的 I-cache 的状态转移有 8 个状态, 但是可以进行化简, 我化简成了 6 个状态的 I-cache, 具体代码图如下(见图 6、图 7):

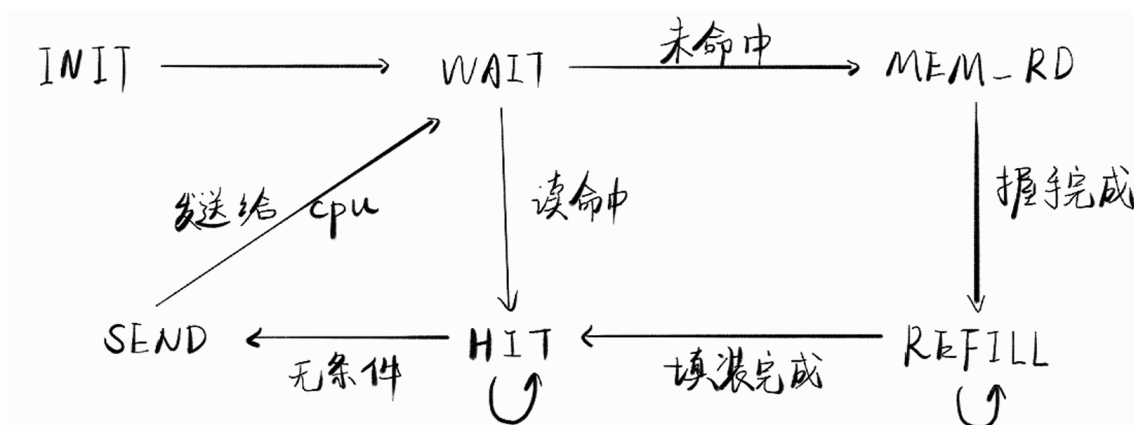


图 6: I-cache 状态转移图

```

//只有读，没有写，也就没有dirty与clear之分
localparam INIT = 6'b000001,/* 初始化状态 */
             WAIT = 6'b000010,/* 等待CPU请求，并读取标签 */
             MISS = 6'b000100,/* 未命中 */
             REFILL= 6'b001000,/* 填充：从内存读取新块到Cache */
             HIT = 6'b010000,/* 命中 */
             SEND = 6'b100000;/* 送出：将Cache中的数据发送给CPU */

```

图 7: I-cache 状态定义

### 3 替换策略

讲义中提供了两种替换策略：LRU（最近最少使用）和随机替换。显然使用 LRU 的替换策略可以更好地利用 Cache，提高命中率。下面是我的设计：

为了实现 LRU 策略，我在 Cache 结构中增加了一个 last\_hit 域，用于记录最近一次的命中时间。每次访问缓存时，如果命中，则将该数据块的 last\_hit 置为当前时间；如果未命中，则将 last\_hit 最小的数据块替换，并将替换后新数据块的 last\_hit 置为当前时间。因为要从 6 个数据中找到最小的一个，所以我专门写了一个 replacement 模块，用于找到应该被替换的 last\_hit 最小的那个数据块。接口定义如下：（见图 8）

```

module replacement (
    input                clk,
    input                rst,
    input  [`TIME_WIDTH - 1 : 0] data_0, data_1, data_2, data_3, // 各路的时间戳输入
    // data_4, data_5,
    output [ 2 : 0] replaced_way // 输出被替换的路号
);

```

图 8: 替换策略模块接口定义

正常情况下的替换策略模块的工作为：通过比较各个块的时间戳，选出最久没有访问的块进行替换，具体代码如下（见图 9）：

```

// 两两比较所有时间戳
wire le_01, le_02, le_03, le_04, le_05,
    le_12, le_13, le_14, le_15, le_23,
    le_24, le_25, le_34, le_35, le_45;

// 标志哪一路的时间戳是最小的
reg  least_0, least_1, least_2,
    least_3, least_4, least_5;

assign le_01 = (data_0 < data_1);
assign le_02 = (data_0 < data_2);
assign le_03 = (data_0 < data_3);
assign le_12 = (data_1 < data_2);
assign le_13 = (data_1 < data_3);
assign le_23 = (data_2 < data_3);

```

图 9: 替换工作逻辑代码

但是, 实验中我发现, 如果只是这样简单地比较时间戳, 可能会导致一些问题。就是某些块, 长时间没有访问, 那么它的时间戳会一直增加, 可能会达到 INT\_MAX 的上限, 导致无法正确比较。因此我在替换策略模块中增加了一个复位功能, 当时间戳达到 INT\_MAX 时, 采用随机替换策略, 代码如下: (见图 10)

```

wire      full;          // 标志所有时间戳是否都已饱和
reg  [2 : 0] random_num; // 用于饱和状态下的随机替换

// 当所有时间戳都达到最大值时, full为1
assign full = (data_0 == `MAX_32_BIT) && (data_1 == `MAX_32_BIT) &&
    (data_2 == `MAX_32_BIT) && (data_3 == `MAX_32_BIT); // &&
    //(data_4 == `MAX_32_BIT) && (data_5 == `MAX_32_BIT);

// 一个简单的循环计数器, 生成0-5的伪随机数
always @(posedge clk) begin
    if (rst)
        random_num <= 3'b0;
    else if (random_num == 3'h5)
        random_num <= 3'h0;
    else
        random_num <= random_num + 1;
end

```

图 10: 当 data 块时间戳达到 INT\_MAX 时, 采用随机替换策略

## 二、实验中的难点及实验结果

### 1 实验中的难点

#### • LRU 替换策略的问题

在实现 LRU 替换策略时，我遇到了一个问题：当某个数据块长时间没有被访问时，它的 `last_hit` 会一直增加，可能会达到 `INT_MAX` 的上限，导致无法正确比较。这时我需要在替换策略模块中增加一个复位功能，当 `last_hit` 达到 `INT_MAX` 时，采用随机替换策略（这部分具体的设计已在上面图 8 图 10 说明）。

#### • Cache 的取数逻辑

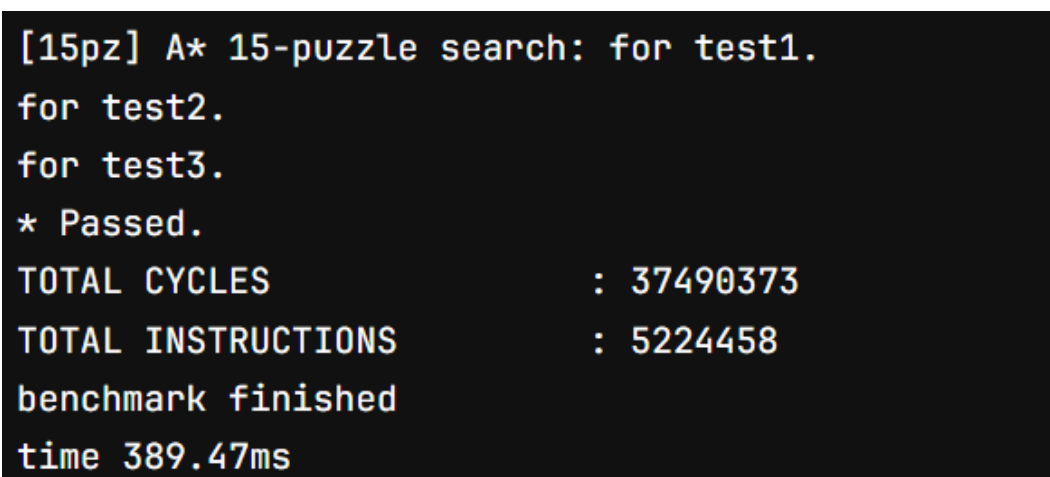
当 Hit 时，或者是 Miss 之后填充完毕，我们需要从 Cache 中取出目标数据，那么如何利用 `offset` 来从 Cache 的“字块”中准确的取出来那个数呢？最终实现的代码如下，这样的代码逻辑，离不开我们的 CPU 访存时的 4 字节对齐的前提：（见图 11）

```
// Cache向CPU返回的数据是什么
assign to_cpu_cache_rsp_data = (
    {`DATA_WIDTH{state_RD_CHE}} & block_all_way[hit_way][{offset, 3'b0} +: `DATA_WIDTH] |
    // 命中时，从数据阵列中根据偏移取数（从起始比特位取32位）
    /* assign Address = {ALU_result[31:2],2'b00}; 得益于CPU访存的地址一定是4字节对齐，
    所以offset只可能为0, 4, 8.....，所以offset*8就是块内某字节对应的比特起始位*/
    {`DATA_WIDTH{state_RDW}} & from_mem_rd_rsp_data // 旁路读时，直接转发内存来的数据
);
```

图 11: 从 Cache 字块中取数逻辑

### 2 实验结果

使用了 Cache 后，我的处理器在运行测试程序时，周期计数器和指令计数器的值都大大减少了，其性能提升是数量级上的提升——比如我的 Dhrystone 测试程序处理性能在没有加 Cache 时为 494（仅有流水线，在之前的实验报告中），此处加上 Cache 后，性能提升至 5464.6，提升了约 1105%。具体的结果如下（见图 12，图 13）：



```
[15pz] A* 15-puzzle search: for test1.
for test2.
for test3.
* Passed.
TOTAL CYCLES                : 37490373
TOTAL INSTRUCTIONS           : 5224458
benchmark finished
time 389.47ms
```

图 12: 运行结果

16 2023K8009929018

1e11031e34a7cf47039b8d56691e15a1ddcb0ac7

riscv32

5464.60

图 13: Dhrystone 测试程序处理性能

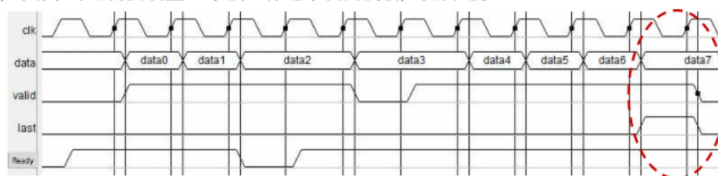


### 三、实验思考题与总结收获

#### 1 思考题

□ 基于之前实验中使用过的Valid/Ready握手信号，实现突发传输接口

- 每个数据的传输都需要Valid-Ready握手
- 添加last信号，标志本次突发传输最后一个有效数据
- 在发送读/写请求时还需要设置len字段，标记本次突发读/写的长度



对于32-bit位宽的内存访问接口，一次Cache Block数据交互  
就是突发读/写8个数据（一般使用最后一个数据的下标值来标记len）

**问题：**如果用上述接口只传输32-bit数据（len = 0），last在哪个位置拉高？

分析：

确定传输长度：根据实际传输数量 = len + 1 的规则，当 len = 0 时，实际传输的数据数量是 0 + 1 = 1 个。

确定 last 的含义：last 信号用于标记一次突发传输中的“最后一个”数据。

逻辑推导：当整个突发传输只有一个数据时，那么这唯一的一个数据既是第一个数据，也是最后一个数据。因此，last 信号必须在传输这个唯一的数据时就被拉高，以表示“这次传输开始了，并且马上就结束”。

结论：对于 len=0（即只有一次数据传输）的情况，last 信号应该在传输这唯一一个数据（我们称之为 data0）的同时被拉高。

#### 2 实验收获

本次高速缓存实验，我通过亲手设计从直接映射到四路组相联的缓存，深刻理解了缓存是提升系统性能、降低访存延迟的核心。通过用不同负载进行测试，我具体分析了全相联、直接映射和组相联等策略在灵活性、成本和冲突率上的权衡，并认识到组相联设计是兼顾性能与成本的理想方案。

除此之外，我还思考了这个问题：为什么 Cache 的效率并不是组数和路数越多越好呢？

通过学习相关理论课知识点和查阅相关资料，我得出了如下结论：

方面	优点	缺点
增加路数	<p><b>1. 显著降低冲突未命中 (Conflict Miss):</b> 这是最主要的好处。多个地址映射到同一组时, 有更多“车位”可用, 不易被踢出。</p> <p><b>2. 提高缓存命中率:</b> 由于冲突减少, 整体命中率得到提升。</p> <p><b>3. 对程序访存模式的适应性更强:</b> 灵活性更高。</p>	<p><b>1. 增加命中延迟 (Hit Latency):</b> 最关键的缺点。需要同时比较组内所有路的 Tag, 硬件 (比较器、多路选择器) 更复杂, 导致命中所需时间变长。</p> <p><b>2. 增加硬件成本和芯片面积:</b> 需要更多的比较器和更复杂的选择逻辑。</p> <p><b>3. 增加功耗:</b> 每次访问, 所有比较器都要工作, 能耗更高。</p> <p><b>4. 边际效益递减:</b> 从 8 路增加到 16 路带来的命中率提升, 远小于从 2 路增加到 4 路。</p>
增加组数	<p><b>情况一: 总容量固定 (路数必须减少)</b></p> <p><b>1. 更细粒度的地址映射:</b> 由于用于索引的地址位数增加, 内存地址被更分散地映射到不同的组。</p> <p><b>2. 硬件相对简单:</b> 因为路数减少, 比较和选择逻辑更简单, 命中延迟可能更低。</p> <p><b>情况二: 路数固定 (总容量增加)</b></p> <p><b>1. 显著降低容量未命中 (Capacity Miss):</b> 因为缓存总容量变大了, 能装下更多的数据。</p> <p><b>2. 提高整体命中率:</b> 能容纳的工作集更大。</p>	<p><b>情况一: 总容量固定 (路数必须减少)</b></p> <p><b>1. 增加冲突未命中的风险:</b> 每个组内的字块变少, 更容易在组内发生冲突。</p> <p><b>情况二: 路数固定 (总容量增加)</b></p> <p><b>1. 急剧增加成本、面积和功耗:</b> 这是最直接的代价。</p> <p><b>2. 可能增加访问延迟:</b> 更大的存储阵列 (SRAM) 物理尺寸更大, 地址解码和数据线路更长, 访问时间会增加。</p>

## 四、 实验所耗时间

在课后, 你花费了大约 30 (代码完成 15 小时, Debug 完成 10 小时, 完成实验报告 5 小时) 小时完成此次实验。

致谢: 感谢蓝宇舟学长在我的指令译码表绘制上的帮助和朱徐塬助教在实验 debug 过程中给予的指导