目录

实验二 Cache 设计与实现

Cache 是计算机体系结构中最热门的研究问题之一。通过 Cache，可以弥补 CPU 和内存间的速度鸿沟。而 Cache 的思想也可以被广泛应用于其它领域，如磁盘访问，网络访问等等。通过计组课程的学习，我们已经知道 Cache 有 3 种常见的结构——直接映射、组相联和全相联。

本次实验将引导同学们设计并实现一个基本的 Cache 模块，并介绍常见几种 Cache 的性能优化方法。本实验同时提供了一个 vivado 工程，同学们完成自己的 Cache 模块后，可以将其添加到一个提供好的 SoC 环境中。然后运行仿真，该环境会运行一个矩阵分块乘法程序，并输出运行花费的时钟周期数，同学们可以以此观察自己实现 Cache 的性能。

1 实验目的

1. 加深对 Cache 原理的理解
2. 通过使用 verilog 实现 Cache，加深对状态机的理解

2 实验设备

1. 装有 Xilinx Vivado 的计算机一台
2. [可选]Nexys4 DDR 实验开发板（用于上板观察运行时间）

3 实验任务

1. 最低要求：参考指导书中直接映射写直达 Cache 的实现，实现写回策略的 Cache
2. 替换实验环境中的 Cache 模块，并通过仿真测试
3. [选做] 性能优化，实现 2 路组相联的 Cache
4. [选做] 性能优化，使用伪 LRU 等替换策略实现 4 路以上组相联的 Cache
5. [选做] 实现其它 Cache 性能优化方法，如 axi burst 传输

4 前置知识

1. Cache 理论知识
2. 类 sram 接口协议

5 实验步骤

此次实验大概分为以下几个步骤

- 首先,阅读实验环境介绍,明白 Cache 模块所处的位置与作用。
- 然后,阅读示例 Cache 实现过程,了解设计并实现 Cache 的主要步骤。其中类 sram 接口部分可以参考文档《A12_ 类 SRAM 接口说明》
- 接着,阅读 Cache 性能提升章节写回 Cache 内容,并参照示例代码,实现写回 Cache。其中以下过程比较重要
  - 分析写回 Cache 的流程图
  - 分析写回 Cache 的状态机
  - 分析写回 Cache 输入输出波形图
- 接着,阅读调试章节的内容,用自己实现的 Cache 模块,替换掉示例 Cache 模块,然后运行仿真程序,进行调试。
- (选做) 阅读 Cache 性能提升章节其余内容,来提升 Cache 性能。

6 实验环境

6.1 硬件环境 (soc)

本次实验需要实现一个 Cache 模块，该模块接口已经定义。如图1，黄色部分即为需要实现的 Cache 模块，其它部分已给出。

图1显示了 CPU 顶层模块结构图。MIPS core 模块内部包含一个 5 级流水的 CPU 核，实现了 MIPS I 的 57 条基本指令。MIPS core 通过两个类 sram 接口对外进行指令访问和数据访问。

当 MIPS core 向 Cache 模块请求指令和数据时，Cache 模块如果命中，则可以马上返回数据，不用再访问内存，否则需要访问内存。而访问内存时，Cache 模块只需产生类 sram 信号，该类 sram 信号经过一个类 sram-axi 转换桥后，会被转换成 axi 信号，因此 CPU 顶层对外接口为 axi 接口。

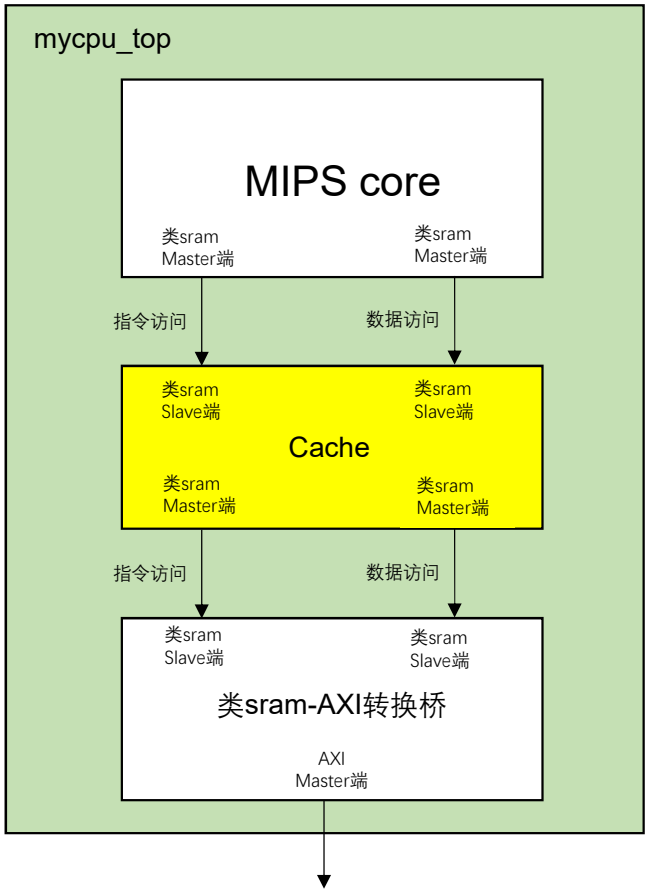

图 1: cpu 顶层结构图

如图2所示，在添加内存模块，和外设模块后，便可以构成一个完整的片上系统 (Soc)。CPU 运行的程序通过 coe 文件可以直接加载进内存模块。而外设模块主要用于控制板子上的 GPIO（如拨码开关，七段数码管，双色灯等）。AXI 1x2 bridge 为 ip 核，已经配置好了内存和外设的地址空间。从而可以根据访问的地址，决定是访问内存模块还是外设模块。注：此 Soc 为龙芯杯比赛提供，图中省略了部分细节。

总结，此次实验已经将 Cache 模块与其它模块交互部分统一封装为了类 sram 接口。同学们在对 Soc 整体结构有了一个认识后，便可以聚焦于 Cache 内部的设计，主要考虑 Cache 的结构，状态机，替换策略等

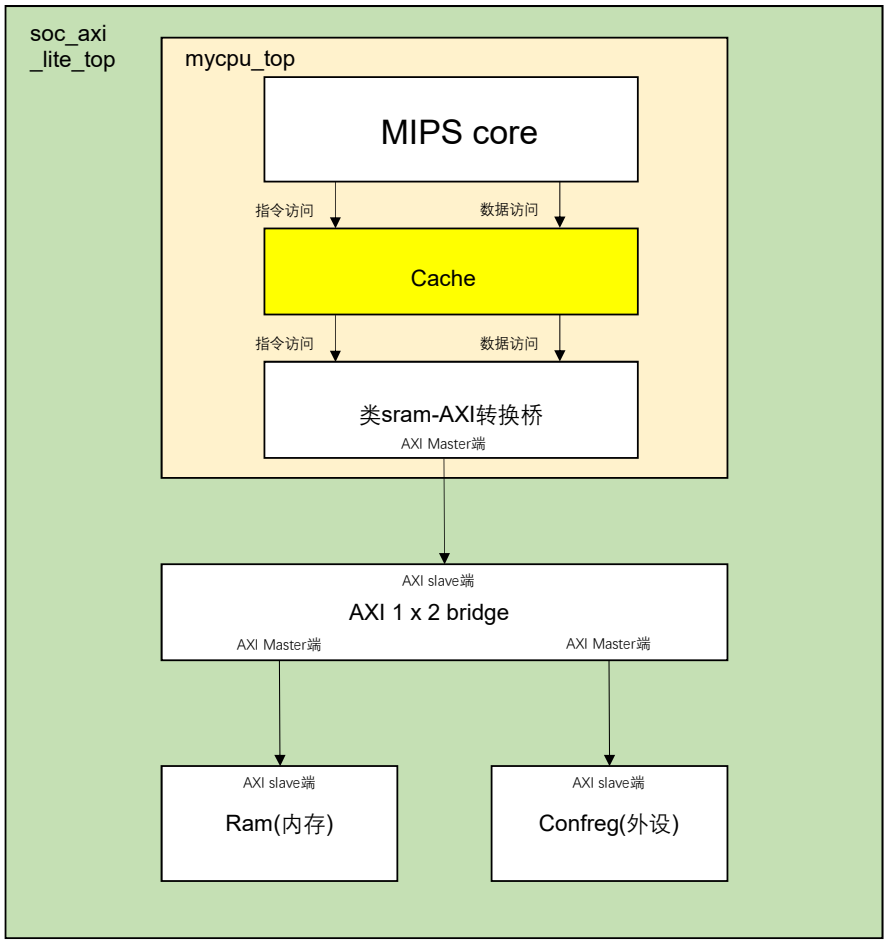

图 2: soc 顶层结构图

内容。

6.2 软件环境

本次实验的软件环境由龙芯杯性能测试的软件环境修改而来。因为本次实验已经提供编译好的结果(coe 文件),位于 obj 目录,故无需自己编译。

实验包中的 soft 目录为此次实验的软件环境,包含以下文件:

```
$ tree
‘-- cache_lab
    |-- Makefile
    |-- Readme_first.txt
    |-- bench
    |   ‘-- shell1
    |       |-- Makefile
    |       |-- get_array.py
    |       |-- matrix_mult.c
    |       ‘-- shell1.c
    |-- bin.lds
    |-- bin.lds.S
    |-- convert
```

```
|-- convert.c
|-- include/
|-- lib/
|-- obj
|   |-- allbench
|   |   |-- axi_ram.coe
|   |   |-- axi_ram.mif
|   |   |-- inst_data.bin
|   |   |-- main.elf
|   |   `-- test.s
|   `-- shell1
|       |-- axi_ram.coe
|       |-- axi_ram.mif
|       |-- inst_data.bin
|       |-- main.elf
|       `-- test.s
`-- start.S

9 directories, 84 files
```

- include 和 lib 目录为程序提供了基本的 C 函数库，如 printf
- start.S 包含程序的入口点，通过该汇编代码来调用 bench 下的各个测试程序
- bench 目录包含若干测试程序（本次实验只有一个 shell1）
- obj 目录包含编译好的结果。位于 shell1 目录下的只运行 shell1 测试程序。而位于 allbench 目录下的会根据拨码开关的选择，来运行指定的测试程序，用于上板。
- Makefile 文件包含使用 make 自动化编译的规则。
- convert.c 用于将 elf 文件转换成可以导入 ram ip 核的 coe 文件
- bin.lds.S 包含链接脚本，可以观察到程序的代码段起始虚拟地址为 0x9fc0_0000

程序的执行过程如下：CPU PC 硬件复位为 0xbfc0_0000。由于 0xbfc0_0000 和 0x9fc0_0000 的物理地址都为 0x1fc0_0000，因此 CPU 实际上会运行 start.S。start.S 再调用 shell1 函数（定义在 shell1.c 中）。该函数主要作用是调用矩阵乘法函数 matrix_mult，并统计输出它的运行时间。

测试程序中的矩阵乘法采用了矩阵分块的方式来进行优化，可以参考课本软硬件接口第 5 版中的 5.4.5 节——通过分块进行软件优化。

7 示例 Cache 的实现

上面介绍了 Cache 模块所处的位置与作用，接下来我们将介绍如何实现一个简单的直接映射写直达的 Cache，并给出示例代码。

7.1 直接映射 Cache 结构

原始的内存数据可以看成一个一维的字节数组，内存地址作为一个索引，可以通过索引访问到对应的内存字节。地址和数据是一一对应的。而直接映射的 Cache 可以看成一个二维数组，内存地址被拆成 Tag，index 和 offset 三部分。其中 index 和 offset 构成这个二维数组的索引，通过 index 访问一个 cache line，通过 offset 确定 cache line 中数据块中对应的字。而由于 Cache 的容量远小于内存的容量，因此无法做到一一对应。当地址的 index 和 offset 都相同时，就会将两个地址映射到 cache 中的同一个位置造成冲突。因此此时就需要 tag 来表示两个不同的地址。

直接映射 Cache 结构如图3所示。

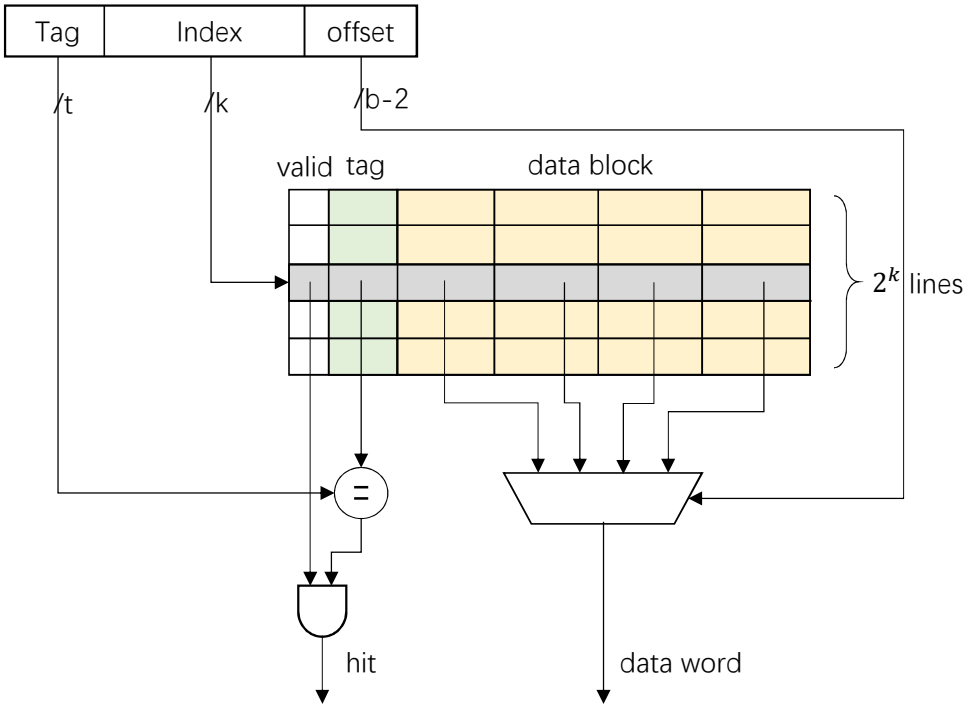

图 3: 直接映射 Cache 结构图

设计 Cache 时，我们首先确定 cache 块的大小和行数，也就是 index 和 offset 的位宽。这也同时确定了 Cache 的容量。本次示例程序将实现 4KB 的 Cache，且块大小为 1 个字。从而我们可以计算出行数为 $\frac{4KB}{1W} = 2^{10}$。因此 index 为 10 位，offset 为 2 位。tag 为 32 中剩下的 20 位。

说明：块大小为 1 字的缺点是不能利用访存的空间局部性，即访问一个地址，也会很有可能访问其周围的地址。对于指令 Cache 则更是如此。如果块大小为多个字，则每次 Cache 缺失，可以一次性读取多条相邻指令到 Cache 中，从而大大提高 Cache 的命中率。但块大小为多字需要能够一次访存，取得多个字的数据，而这涉及到 AXI 的 burst 传输。并且块多字也会增加 Cache 读，写缺失时的逻辑。因此为了简化，采用块大小 1 个字进行说明。

示例代码最终实现的 Cache 配置如表1所示。

表 1: 示例 Cache 配置

	指令 Cache	数据 Cache
结构	直接映射	直接映射
写策略	无	写直达
容量	4KB	4KB
块大小	1 word	1 word
行数	1024	1024
tag	20 bit	20 bit
index	10 bit	10 bit
offset	2 bit	2 bit

7.2 写直达-写不分配策略

当 CPU 执行 store 类指令时，对 Cache 应该如何操作呢？当 Cache 命中时，最简单的想法是将数据同时写入到 Cache 和内存。这样保证了 Cache 和内存中的数据都是修改后最新的数据，保持了一致性。但也导致了大量的写内存操作，因此效率非常低。而另一种想法是只写入 Cache，同时标记该 Cache line 为已修改，等到万不得已的时候（该 cache line 被替换时）再写入内存。这可以大大减少写内存的次数。这两种策略分别为写直达 (Write Through) 和写回 (Write Back)。本示例出于简单考虑，使用写直达策略。

上面讨论的是写命中的情况，当写缺失时。针对是否需要将数据写入 Cache，可以分为写分配 (Write Allocate) 和写不分配 (Non-Write Allocate)。写直达通常结合写不分配策略一起使用，即写缺失时，直接写入内存，而不写入 Cache。而写回通常结合写分配策略一起使用，即写缺失时，只写入 Cache，而不写入内存。

7.3 状态机的设计

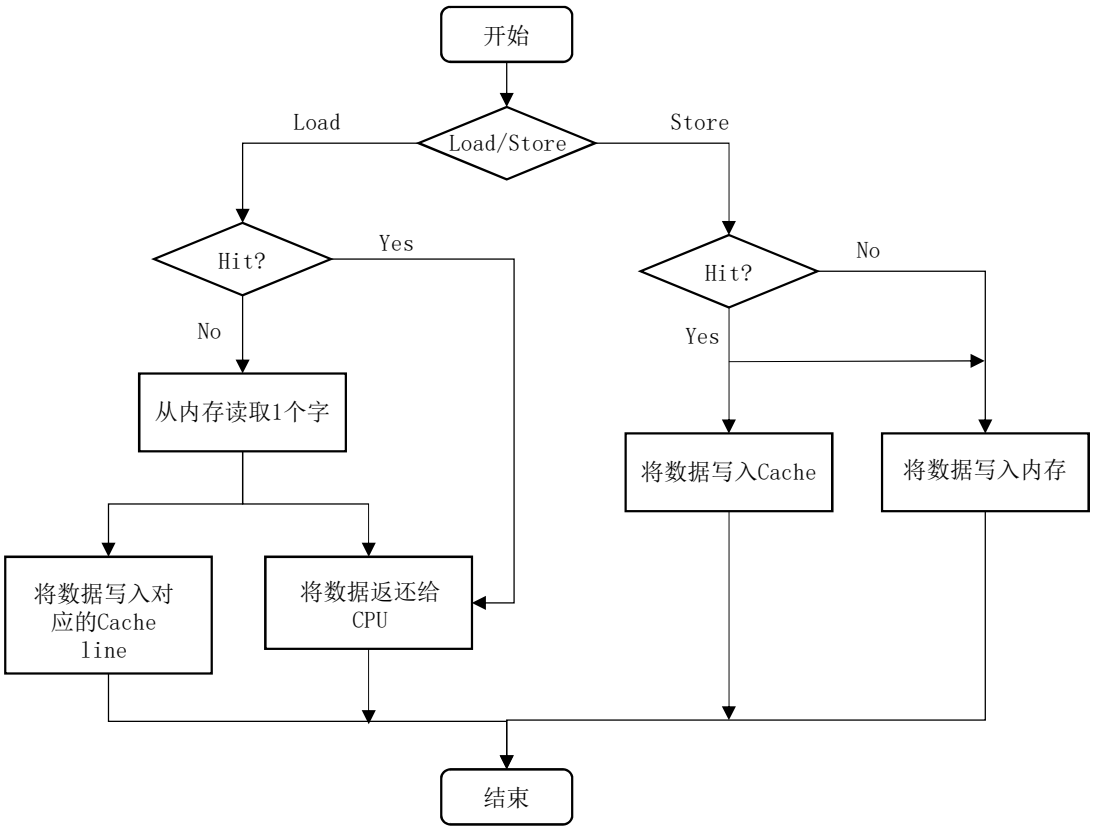

图 4: 示例数据 Cache 处理流程

由于指令 Cache 不包含写的情况，我们以数据 Cache 为例进行分析。经过上面的分析，我们可以画出 Cache 读写处理的流程图，如图4所示。

CPU 发来的访存请求可能是读 (load 类指令)，也可能是写 (store 类指令)。如果是读请求的话，判断是否命中。如果命中了的话，便立刻返回数据。而如果缺失了，则需要从内存读取数据。再将数据写入 Cache，并返回数据给 CPU，这两个操作可以同时进行。如果是写请求的话，判断是否命中。如果缺失的话，将数据写入内存。如果命中了的话，则既需要将数据写入内存，也需要写入 Cache。

因此，我们可以设计一个简单的状态机如图5。图中定义了三种状态，IDLE 表示空闲状态也就是初始状态，RM 表示读取内存的状态，WM 代表写内存的状态。以读为例，当读命中时，因为可以马上返回数据，因此仍然保持 IDLE 状态。当读缺失时，则进入 RM 状态，直到读取内存数据完成时（data_ok），Cache 返回数据给 CPU，写入 Cache，然后返回到 IDLE 状态。

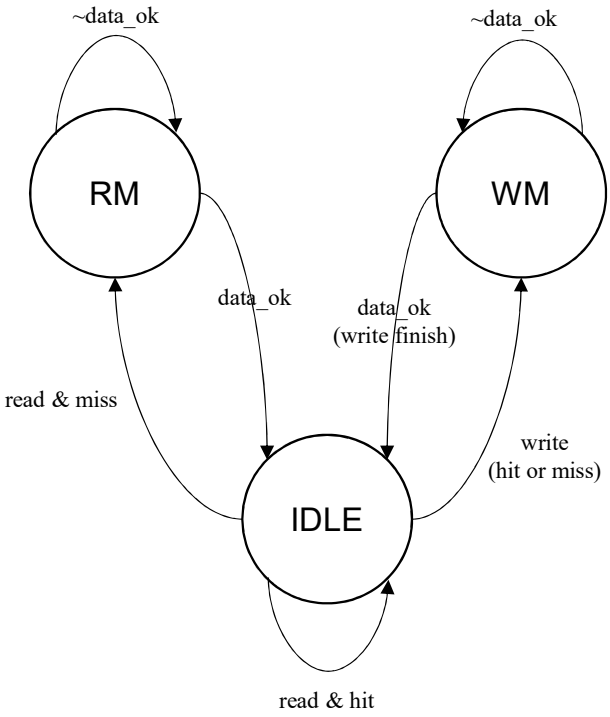

图 5: 示例数据 Cache 状态机

7.4 如何存储 Cache

在本次实验中，我们需要一些存储单元来存储 Cache。我们知道，一般使用 SRAM（静态随机存储器）来实现 Cache，因为 SRAM 是一种非常快的存储器，和 CPU 的速度接近。在 FPGA 上，可以使用两种方式组织存储器。

7.4.1 使用 LUT（查找表）

LUT 是 FPGA 中最重要的资源之一，因为 FPGA 通常使用 LUT 来实现任意的组合逻辑。简单来说，LUT 由一些 SRAM 比特和多选器组成，感兴趣的同学可以搜索具体原理。

我们可以采用 reg 定义二维数组的方式实现存储，如：

```
//Cache 配置
parameter  INDEX_WIDTH  = 10, OFFSET_WIDTH = 2;
localparam TAG_WIDTH    = 32 - INDEX_WIDTH - OFFSET_WIDTH;
localparam CACHE_DEEPTH = 1 << INDEX_WIDTH;

//Cache 存储单元
reg                  cache_valid [CACHE_DEEPTH - 1 : 0];
reg [TAG_WIDTH-1:0]  cache_tag   [CACHE_DEEPTH - 1 : 0];
reg [31:0]           cache_block [CACHE_DEEPTH - 1 : 0];
```

上面的代码根据 Cache 的配置，定义了 Cache 所需要的存储单元。再通过以下代码便可以实现对 Cache line 的访问。

```
//访问地址分解
```

```
wire [OFFSET_WIDTH-1:0] offset;
wire [INDEX_WIDTH-1:0] index;
wire [TAG_WIDTH-1:0] tag;

assign offset = addr[OFFSET_WIDTH - 1 : 0];
assign index = addr[INDEX_WIDTH + OFFSET_WIDTH - 1 : OFFSET_WIDTH];
assign tag = addr[31 : INDEX_WIDTH + OFFSET_WIDTH];

//访问Cache line
wire c_valid;
wire [TAG_WIDTH-1:0] c_tag;
wire [31:0] c_block;

assign c_valid = cache_valid[index];
assign c_tag   = cache_tag  [index];
assign c_block = cache_block[index];
```

上面的方式可以看作是“纯手工”打造了一个 ram。事实上，还可以使用 dram ip 核 (Distributed Memory Generator) 生成 ram。使用 ip 核的好处是接口比较统一，不必关心 ram 内部的组织细节。比可以很容易配置成有着读写双端口的 ram，而不必关心其内部是如何解决同时读写的冲突的。

7.4.2 使用 BRAM

上面实现 ram 的方式都使用的是 LUT 资源，其实 FPGA 中提供了专门用于实现 ram 的 bram 资源 (Block RAM)，有更高的速度。可以使用 Block Memory Generator ip 核生成各种类型的 ram。

一般来说，在容量较小，性能要求低的情况下使用 dram，在容量较大，性能要求高的情况使用 bram。对于我们的 Cache 来说，数据部分要求的容量还是比较大的，且对性能要求较高，因此采用 bram 比较合适。但使用 bram 的缺点是读数据时，无法像 dram 那样是组合逻辑，需要一个时钟的延迟。因此出于简化实验的目的，示例使用 reg 实现。但也鼓励同学们使用各种方式实现 Cache。

7.5 代码实现

7.5.1 指令 Cache

第一步，判断 Cache 命中 我们需要知道如何判断 cache 命中，参考图3直接映射 cache 的结构图，hit 判断逻辑如下：

```
//判断是否命中
wire hit, miss;
assign hit = c_valid & (c_tag == tag);  //cache line的valid位为1，且tag与地址中tag相等
assign miss = ~hit;
```

第二步，Cache 的状态机 分析状态机，能让我们对 Cache 的整体逻辑更加明确。同时状态机的状态变量也可以用于之后控制一些信号。

```
//FSM
parameter IDLE = 2'b00, RM = 2'b01; // i cache只有read
reg [1:0] state;
always @(posedge clk) begin
    if(rst) begin
        state <= IDLE;
    end
    else begin
        case(state)
            IDLE:   state <= cpu_inst_req & miss ? RM : IDLE;
            RM:     state <= cache_inst_data_ok ? IDLE : RM;
        endcase
    end
end
```

第三步，画出命中和缺失时的输入输出波形图 我们的 Cache 主要在和 CPU 和外界这两方打交道。这里对应两组类 sram 信号。分析状态机让我们从整体上了解 Cache 的内部逻辑，而分析输入输出信号的波形图则是具体到实现。

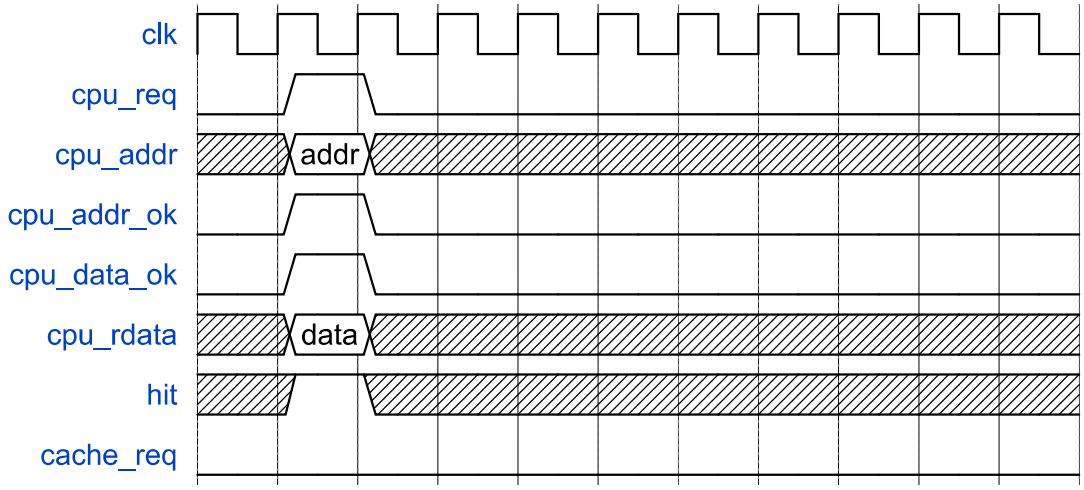

图 6: 指令 cache 命中时

如图6是指令 Cache 命中的情况，整个过程如下：刚开始 cpu_req 为 1，cpu_addr 为读指令地址。由于 hit 是组合逻辑，因此在这个周期产生 hit 信号。图中对应为 1，表示命中。由于 Cache 命中，因此我们不必发出 cache_req 请求，而可以直接给 cpu 返回数据。因此我们将 cpu_addr_ok 和 cpu_data_ok 都设置为 1，将 cpu_rdata 设置为从 cache 读取到的数据，表示 cpu 可以来取数据。

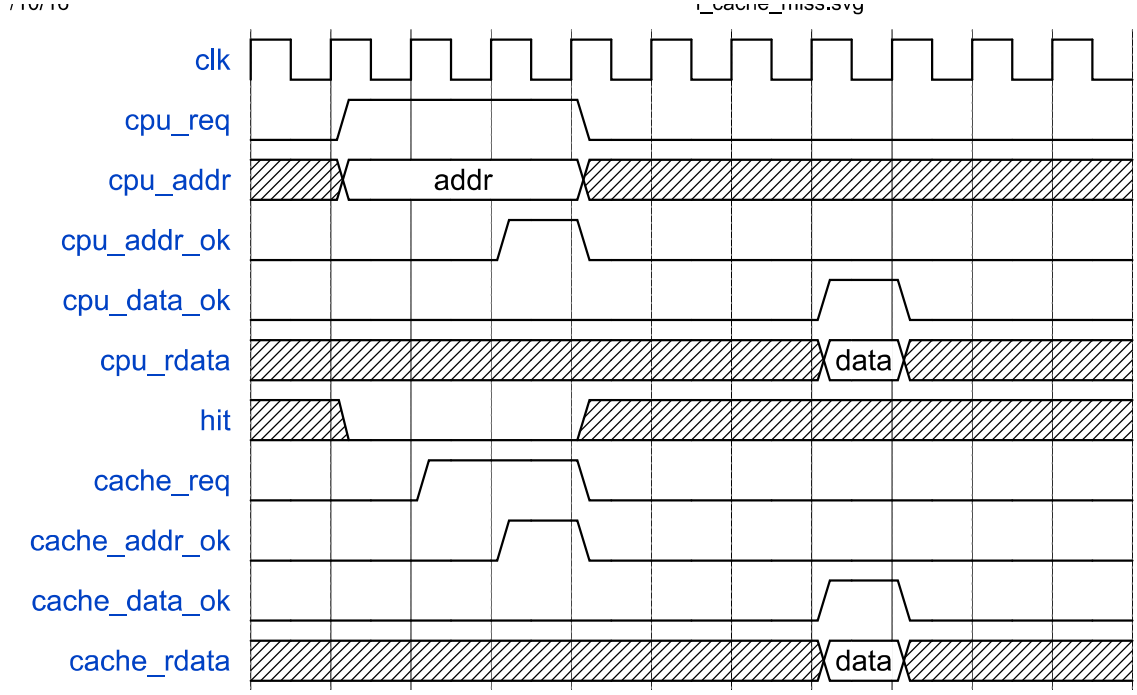

图 7: 指令 cache 缺失时

如图7是指令 Cache 缺失的情况，整个过程如下：同样刚开始 cpu_req 为 1，cpu_addr 为读指令地址。由于 Cache 缺失，对应图中 hit 信号为 0，因此我们下周期发出 cache_req 请求。之后的过程便是一次标准的类 sram 读事务。我们只需要在收到 cache_addr_ok 后，将 cpu_addr_ok 也设置为 1。并且在收到 cache_data_ok 后，向 cpu 返回数据。

第四步，根据波形图实现输入输出信号 代码中，使用了 read_req, addr_rcv, read_finish 变量来辅助生成类 sram 信号。read_req 代表读请求整个过程，addr_rcv 代表地址已经收到了，而 read_finish 代表读请求结束。根据类 sram 接口协议，req 信号在 addr_ok 后需要拉低，便可以通过以下代码来实现

```
cache_inst_req    = read_req & ~addr_rcv;
```

完整代码如下

```
//读内存
//变量read_req, addr_rcv, read_finish用于构造类sram信号。
wire read_req;      //一次完整的读事务，从发出读请求到结束
reg addr_rcv;       //地址接收成功(addr_ok)后到结束
wire read_finish;   //数据接收成功(data_ok)，即读请求结束
always @(posedge clk) begin
    addr_rcv <= rst ? 1'b0 :
                cache_inst_req & cache_inst_addr_ok ? 1'b1 :
                read_finish ? 1'b0 : addr_rcv;
end
assign read_req = state==RM;
```

```
assign read_finish = cache_inst_data_ok;

//output to mips core
assign cpu_inst_rdata   = hit ? c_block : cache_inst_rdata;
assign cpu_inst_addr_ok = cpu_inst_req & hit | cache_inst_req & cache_inst_addr_ok;
assign cpu_inst_data_ok = cpu_inst_req & hit | cache_inst_data_ok;

//output to axi interface
assign cache_inst_req   = read_req & ~addr_rcv;
assign cache_inst_wr    = cpu_inst_wr;
assign cache_inst_size  = cpu_inst_size;
assign cache_inst_addr  = cpu_inst_addr;
assign cache_inst_wdata = cpu_inst_wdata;
```

第五步，实现写入 Cache 等工作 最后一步，指令 cache 缺失后，将从内存读取到的数据写入 cache。由于只有在 cpu_inst_req 为 1 期间才能保证 cpu_inst_addr 的值是有效的，因此我们需要将地址存下来，防止之后发生变化（事实上提供的 CPU 在整个期间 addr 都会保持不变，这里是为了更加严谨）

```
//写入Cache
//保存地址中的tag，index，防止addr发生改变
reg [TAG_WIDTH-1:0] tag_save;
reg [INDEX_WIDTH-1:0] index_save;
always @(posedge clk) begin
    tag_save   <= rst ? 0 :
                  cpu_inst_req ? tag : tag_save;
    index_save <= rst ? 0 :
                  cpu_inst_req ? index : index_save;
end

integer t;
always @(posedge clk) begin
    if(rst) begin
        for(t=0; t<CACHE_DEEPTH; t=t+1) begin   //刚开始将Cache置为无效
            cache_valid[t] <= 0;
        end
    end
    else begin
        if(read_finish) begin //读缺失，访存结束时
            cache_valid[index_save] <= 1'b1;              //将Cache line置为有效
            cache_tag  [index_save] <= tag_save;
            cache_block[index_save] <= cache_inst_rdata; //写入Cache line
        end
    end
end
```

7.5.2 数据 Cache

数据 cache 读过程与指令 Cache 一致，需要额外处理写的逻辑。

第一步与指令 cache 一致

第二步添加额外的写入状态

```
//FSM
```

```
parameter IDLE = 2'b00, RM = 2'b01, WM = 2'b11;
reg [1:0] state;
always @(posedge clk) begin
    if(rst) begin
        state <= IDLE;
    end
    else begin
        case(state)
            IDLE:   state <= cpu_data_req & read & miss ? RM :
                             cpu_data_req & read & hit  ? IDLE :
                             cpu_data_req & write       ? WM : IDLE;
            RM:     state <= read & cache_data_data_ok ? IDLE : RM;
            WM:     state <= write & cache_data_data_ok ? IDLE : WM;
        endcase
    end
end
```

第三步,读命中和缺失的波形图也与指令 Cache 一致。对于 store 类指令来说,由于采用的写直达策略,无论命中或缺失都需要写入内存。波形图如图8

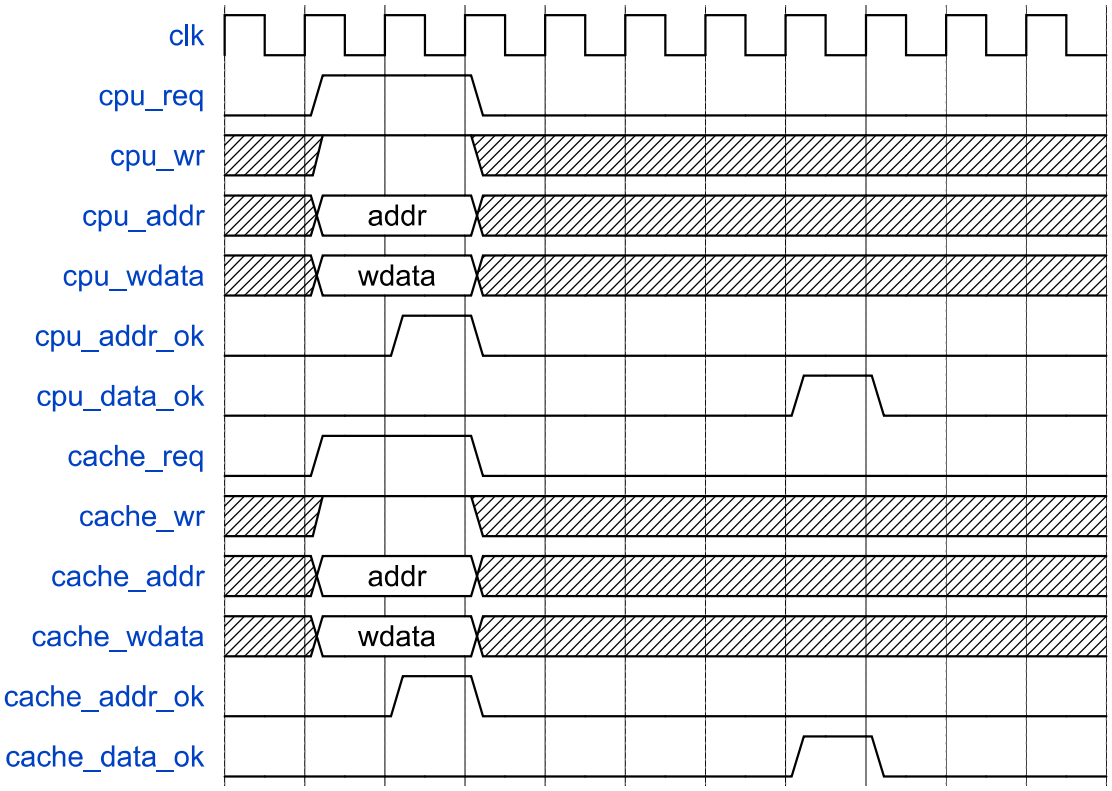

图 8: 数据 cache 写(命中或缺失)

第四步,同样添加写逻辑

```
//读内存
//变量read_req, addr_rcv, read_finish用于构造类sram信号。
wire read_req;      //一次完整的读事务, 从发出读请求到结束
reg addr_rcv;       //地址接收成功(addr_ok)后到结束
wire read_finish;   //数据接收成功(data_ok), 即读请求结束
always @(posedge clk) begin
    addr_rcv <= rst ? 1'b0 :
```

```
                read & cache_data_req & cache_data_addr_ok ? 1'b1 :
                read_finish ? 1'b0 : addr_rcv;
end
assign read_req = state==RM;
assign read_finish = read & cache_data_data_ok;

//写内存
wire write_req;
reg waddr_rcv;
wire write_finish;
always @(posedge clk) begin
    waddr_rcv <= rst ? 1'b0 :
                 write & cache_data_req & cache_data_addr_ok ? 1'b1 :
                 write_finish ? 1'b0 : waddr_rcv;
end
assign write_req = state==WM;
assign write_finish = write & cache_data_data_ok;

//output to mips core
assign cpu_data_rdata   = hit ? c_block : cache_data_rdata;
assign cpu_data_addr_ok = read & cpu_data_req & hit | cache_data_req &
    cache_data_addr_ok;
assign cpu_data_data_ok = read & cpu_data_req & hit | cache_data_data_ok;

//output to axi interface
assign cache_data_req   = read_req & ~addr_rcv | write_req & ~waddr_rcv;
assign cache_data_wr    = cpu_data_wr;
assign cache_data_size  = cpu_data_size;
assign cache_data_addr  = cpu_data_addr;
assign cache_data_wdata = cpu_data_wdata;
```

最后一步,写入 Cache 的时机需要增加。当写命中时,需要写入 cache。

除此之外,代码中还有一个不同点。写命中时写入 cache 的数据 (write_cache_data) 经过了处理,而没有直接使用 cpu_data_wdata。这是因为对于 sb,sh 等指令,只修改原始内存中的部分字节,因而写 cache 前需要根据地址低两位和 cpu_data_size 信号进行处理。如果觉得复杂,可以去掉该部分逻辑,因为之后运行的软件主要进行矩阵乘法,不会用到 sb,sh 指令。

```
//写入Cache
//保存地址中的tag, index, 防止addr发生改变
reg [TAG_WIDTH-1:0] tag_save;
reg [INDEX_WIDTH-1:0] index_save;
always @(posedge clk) begin
    tag_save   <= rst ? 0 :
                  cpu_data_req ? tag : tag_save;
    index_save <= rst ? 0 :
                  cpu_data_req ? index : index_save;
end

wire [31:0] write_cache_data;
wire [3:0] write_mask;

//根据地址低两位和size, 生成写掩码 (针对sb, sh等不是写完整一个字的指令), 4位对应1个
    字 (4字节) 中每个字的写使能
assign write_mask = cpu_data_size==2'b00 ?
                        (cpu_data_addr[1] ? (cpu_data_addr[0] ? 4'b1000 : 4'b0100):
```

```
                                                (cpu_data_addr[0] ? 4'b0010 : 4'b0001))
                                                    :
                                (cpu_data_size==2'b01 ? (cpu_data_addr[1] ? 4'b1100 : 4'
                                    b0011) : 4'b1111);

//掩码的使用：位为1的代表需要更新的。
//位拓展：{8{1'b1}} -> 8'b11111111
//new_data = old_data & ~mask | write_data & mask
assign write_cache_data = cache_block[index] & ~{{8{write_mask[3]}}, {8{write_mask
    [2]}}, {8{write_mask[1]}}, {8{write_mask[0]}}} |
                            cpu_data_wdata & {{8{write_mask[3]}}, {8{write_mask[2]}},
                                {8{write_mask[1]}}, {8{write_mask[0]}}};

integer t;
always @(posedge clk) begin
    if(rst) begin
        for(t=0; t<CACHE_DEEPTH; t=t+1) begin   //刚开始将Cache置为无效
            cache_valid[t] <= 0;
        end
    end
    else begin
        if(read_finish) begin //读缺失，访存结束时
            cache_valid[index_save] <= 1'b1;              //将Cache line置为有效
            cache_tag  [index_save] <= tag_save;
            cache_block[index_save] <= cache_data_rdata; //写入Cache line
        end
        else if(write & cpu_data_req & hit) begin   //写命中时需要写Cache
            cache_block[index] <= write_cache_data;      //写入Cache line，使用index
                而不是index_save
        end
    end
end
```

完整的指令 Cache 和数据 Cache 模块代码参见实验提供的源码包。

8 提升 cache 性能

上面以给出一个示例的方式讲述了如何实现一个 cache。上面 cache 的结构是直接映射，采用的是写直达配合写不分配的策略。这是最容易实现的一种 cache，但是在性能方面有很大的提升空间。下面提出了一些提高 cache 性能的策略。

8.1 写回

本次实验要求，实现写回策略的 Cache。

我们在 CPU 中实现 cache 的目的是为了降低 CPU 访问内存的次数。在示例给出的直接映射的 cache 中，我们采用的是写直达配合写不分配的策略。我们可以简单分析其性能。在读命中的情况下，CPU 直接读取对应的 cache line 获取数据；在读缺失的情况下，CPU 直接访问内存获取数据；在写缺失的情况下，CPU 直接将数据写入到内存中；在写命中的情况，CPU 将数据同时写入对应的 cahce line 和内存中对应的地址。综上所述，也就是只有在读命中的情况下，CPU 不用访问内存，但是在其他情况下，CPU 都会访问内存。显然，这里还有很大的提升空间。

对于上面的情况，一种策略就是实现写回的 cache。写回给人一种不到万不得已不起床的感觉。写回策略是在写命中的情况下，并不直接将数据写入到下级存储器中，而是只将数据写入到索引到的 cache line 中，当且仅当一个脏的 cache line 要被替换的时候再将数据更新到内存中。写回策略一般是跟写分配策略配合的，写分配策略就是在写缺失的情况下，同时将数据写入到 cache 跟内存中。现在稍微总结一下采用写回--写分配策略、直接映射的 cache 的实现：

1 在读命中的情况下，CPU 直接读取对应的 cache line 的数据；
2 在读缺失的情况，如果索引到的 cache line 是干净的，那么发送读请求，从内存读取数据，然后返回给 CPU，同时将数据写入到索引到的 cache line 中；如果索引到的 cache line 是脏的，那么首先要发送写请求，将这个 cache line 的脏数据写入到内存中。等待写请求处理完成后，再发送读请求，从内存中读取对应的数据，然后再把数据返回给 CPU，同时将数据写入到索引的 cache line 中。
3 在写命中的情况下，如果索引到的 cache line 是干净的，那么直接将数据写入到对应的 cache line 中，并且将 dirty 位置为 1；如果索引到的 cache line 是脏的，直接把数据写入到 cache 中。
4 在写缺失的情况下，如果索引到的 cache line 是干净的，那么将数据写入到 cache line 中，覆盖掉原来的数据。如果索引到的 cache line 是脏的，那么首先发送写请求，将脏的 cache line 的数据更新到内存中；然后等待第一个写请求处理完成后，然后将数据写入到索引到的 cache line 中，并且将脏位标志位置为 1；

我们可能会多次将数据写入到某个相同的地址。针对这种情况，我们可以发现，写回策略仅需要在被替换出去的时候访问内存，而写通策略每次写操作都要访问内存。所以写回--写分配的策略有助于提升 cache 性能。

8.2 提高相联度

前置知识： 导致 cache 缺失的原因有三个：

1. Compulsory，即 cache 冷启动造成的缺失；
2. Capcity，即 cache 容量导致的缺失；
3. Conflict，多个数据映射到 cache 的同一个位置，导致频繁的替换。

这引起 cache 缺失的三个原因，我们称之为 3C 定理。对于 Compulsory 的情况，我们可以通过预取技术来减缓其影响。对于 Conflict 的情况，我们通常使用提高相连度或者使用 victim cache 技术来减缓其影响。

上面我们已经讲述了通过采用写回--写分配策略减少了 CPU 对内存的访问次数，从而提升 cache 的性能。但是对于直接映射的 cache，仍然有很大的提升空间。

对于直接映射，现在假设有这样一种情景，两条指令索引到 cache 中的同一个 cache line，那么频繁的使用这两条指令将导致对应位置的 cache line 一直是处于被替换状态中的，cache 一直缺失。这里我们将介绍组相联的 cache 来减缓这一个情况。

组相联是为了解决直接映射结构 cache 的不足而提出的，存储器中的一个数据不单单只能放在一个 cache line 中，而是可以放在多个 cache line 中，对于一个组相联的 cache 来说，如果一个数据可以放在 n 个位置，则称这个 cache 是 n 路组相联的 cache。两路组相联 cache 的结构示意图如下所示。

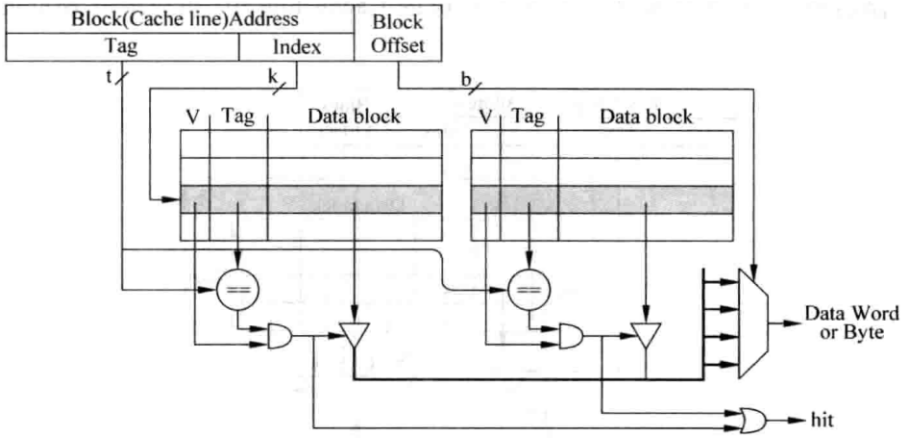

图 9: 两路组相联 cache

在这种结构的 cache 中，我们同样是通过 Index 值取索引 cache。不过不同于直接映射的是，每次索引到的 cache line 有两个。我们需要将 PC 中的 Tag 值跟索引到的两个 cache line 的值进行比对，判断是否命中跟命中哪一个 cache line。此外，当 cache 发生缺失时，需要替换 cache line 时，我们需要从对应的 cache set 中选择一个 cache line 进行替换。在两路组相联中，我们可以用一位标记位记录下最近使用过的 cache line，则可以通过标志位选择另一个 cache line 进行替换。在更多路的组相联 cache 中，我们使用一些替换算法来选择将要被替换的 cache line。

8.2.1 LRU 替换算法

LRU (Least Recently Used) 是使用最为广泛的一种替换算法。LRU 的意思是“最近最少被使用”，即 LRU 替换算法不是“展望未来 "，而是“回顾过去 "。就像新招一个队员同时让一个“板凳队员“卷铺盖走人，实现 LRU 算法要求对每个“队员”做记录，看谁在板凳上坐的时间最长。

以组相联映射 Cache 为例。假设一组有八路，则每路要有一个三位计数器，通过各路的计数值来区分哪一路最近老没被教练派上场。规则如下：

1. 替换掉计数值为 0 的块（如果有多千一个块的计数值为 0，随便替换哪一个）。设置该块的计数器为最大值 7，其他块的计数器都减 1。但若计数值是 0 就不再减了。我们称这样的计数器为饱和计数器 (Saturated Counter)。
2. 命中时，假设命中路的计数值为 k，则把计数值大于 k 的所有其他块的计数器减 1，并把命中块的计数器设置为最大值 7（此步操作完成的是排序操作）。

3. 计算机刚启动时，把所有的 Cache 块的计数器和有效位全部清零。这意味着该队连一个队员也还没有呢。

如果一组有四路，则需为每路安排一个两位计数器。如果一组只有两路，按推理需为每路安排一个一位计数器。但是，如果其中的一路是最近最少被使用的话，不言而喻，另一路一定是最近最多被使用的，因此只用一个一位计数器就足够了。

实现 LRU 算法需要消耗比较多的资源，对于 8 路组相联的结构来说，一个 cache set 需要 8*3=24 位，这是一笔不小的开销，因此常使用“伪 LRU”的算法实现。

8.2.2 伪 LRU 替换算法

如上所述，采用 LRU 算法需要消耗过多的资源，因此常使用伪 LRU 算法替代。一种伪 LRU 算法通过，对 cache 的路进行分组，每组使用一个位来表示最近是否使用过。对于有 2^n 路的 cache，每个 cache set 需要 $2^n - 1$bit 位来存储最近访问信息。

图10展示了伪 LRU 算法的过程。对于 4 路的结构，每一个 cache set 需要 3bit 来存储最近使用信息。将这 3 个比特组织成树状结构，每一 bit 可以表示其左子树或右子树是否被访问过。为了方便说明，对每个 bit 节点进行编号，按照从上往下，从左往右的顺序分别编号为 1, 2, 3。

伪 LRU 算法确定替换哪一路的算法类似于一个二分查找，每次通过 1bit 的信息可以将范围缩小到原本的一半，若干次迭代后便可以确定到具体的某一路。具体算法如下：从根节点（1 号节点）开始，如果其值为 0，则表示其右子树（对应的路，即 0 路，1 路）最近没有访问过。反之，则表示其左子树最近没有被访问。假设 1 号节点为 0，则我们通过判断 3 号节点的值来判断替换哪一路：如果为 0，替换 0 路，如果为 1，替换 1 路。假设 1 号节点为 1，我们便通过 2 号节点来判断替换哪一路，过程类似。

当访问了 cache 的某一路后，需要对 bit 信息进行更新。更新的算法一句话来说就是保持每一 bit 的语义不变，即每一 bit 代表其左右子树最近是否被访问过。具体来说其实只需要更新访问路径（从根节点到某一路的连线）上的节点的状态值即可。（并非直接取反，而是保持语义）

如图10所示，根据上述的确定替换路的算法，刚开始该 cache set 的状态表示应该替换 way0，但在访问了一次 way0 后，需要更新访问 way0 路径上的节点也就是 0 号和 3 号节点的状态。对 0 号节点来说，由于访问了其右半子树，因此更新为 1，对 3 号节点同理更新为 1。在更新之后，该 cache set 的状态变为应该替换 way2。

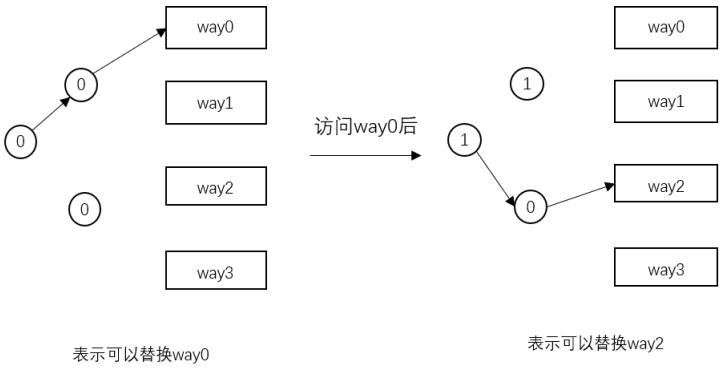

图 10: 伪 LRU 替换算法过程

8.2.3 随机替换算法

LRU 算法是通过总结历史经验来预测将来的发展方向。这种算法需要“投资" ’ 即为每一块配备一个计数器。

如果你想省钱,随机 (Random) 算法倒是一个相当不错的选择。随机算法完全不管 Cache 块过去、现在及将来的使用情况,简单地根据一个随机数,选择一块替换掉。注意,整个 Cache 只需一个随机数产生器,所以比 LRU 省钱。随机数可由硬件产生,例如设置一个计数器,由系统时钟进行计数。是不是被替换掉就看你的运气了。虽然有些 Cache 设计者讨厌这种方法,但模拟结果证明,它的性能还是相当不错的,至少把钱先省了。

还有一种先进先出 FIFO (First-In First-Out) 算法: 不管你在场上的表现如何, 谁先入队谁先走人。与 LRU 算法一样,也需投资。虽然还有其他一些替换算法,但实现起来过于复杂。在大多数 Cache 设计中,二路和四路组相联映像 Cache 用 LRU 替换算法,其他的用随机替换算法(直接映像 Cache 用不着替换算法)。

组相联实现方式的 cache 在现实的处理器应用中最为广泛。我们可以通过它可以减少由冲突产生的缺失, 但是我们也要考虑它的命中时间过长, 从而对主频产生影响。有兴趣的同学可以去了解一下 cache 的并行访问跟 cache 的串行访问。

8.3 块多字

cache 之所以能够发挥作用,是因为程序的时间局部性跟空间局部性。上面我们同时提高相连度,减少冲突引起的缺失,进一步的利用时间局部性。接下来我们将考虑实现块多字, 以利用程序的空间局部性来提高 cache 的性能。

在上面我们所讲述的 cache 结构中, 无论是直接映射, 还是组相联结构的 cache, 他们的数据块大小都是一个字。那么我们每次访问内存都是只能取一个字的数据。但往往在程序中,我们可能会访存一块连续的地址空间。若数据块大小为一个字, 那么 CPU 需要多次访存内存, 对性能有严重影响。因此, 如果我们能够一次性将这一块数据取出来, 就只用访问一次内存, 大幅减少访问内存的次数。一个典型的例子就是我们的指令 cache。考虑指令 cache, 假设数据块大小为 8 字,那么我们只用访问一此内存,就能将接下来的 8 条指令全部取出来。假设不发生跳转, 汇编程序顺序执行, 接下来的 8 次指令 cache 访问都将命中, 从而大幅减少访问内存的次数。

在本次实验环境中,类 sram 并不支持多字传输,如果要实现块多字的 cache,需要调整 cache 的输出端口,同时在类 sram 转 axi 模块中实现 burst 传输机制,或者将 cache 接口改为 axi 接口,然后实现仲裁模块将 2 个 axi 接口合并成一个 axi 接口。有兴趣的同学可以去了解下类 sram 接口跟 AXI(AXI-FULL) 协议。

8.4 其他相关 cache 优化性能方法

除了上述方法能够提升 cache 性能以外, 还有其他或是从硬件层面, 或是从软件层面优化 cache 性能的方法。有兴趣的同学可以参考《计算机体系结构--量化研究方法》相关章节。

9 关于调试

在我们的计算机组成原理实验 4 中，我们提供了一个非常简单的测试程序，仅测试了十几条指令，并在注释中给出了正确的执行顺序。同学们通过观察波形图，然后跟正确执行顺序进行比对，从而定位错误。本次实验主要着眼于 cache 的正确实现，所以我们提供了一个实现了 57 条指令的、封装完整的 mips core，同学们不用考虑 CPU 数据通路内部实现。为了更能直观体现出 cache 的作用，同时能够完整的测试 cache 模块的正确性，提供了一个相对于计算机组成原理 4 复杂很多的测试程序，相对于计组实验 4 测试程序更加完整。在这种情况，计组实验 4 的调试方法麻烦且不太现实。

9.1 使用 trace 进行调试

不同于计组实验 4 的测试程序，cache 测试程序包含的指令数目有一千多条。在这种情况下，我们同样可以跟正确执行顺序比对从而调试，但是存在两个问题，一是对于包含一千多条汇编指令的测试程序，人为的推理指令的执行序列不现实；二是人为的比对一千条指令的执行序列不现实。一种思想是关注那些会对 CPU 状态产生影响的指令，也就是会将数据写入到通用寄存器的指令，比如 lw，add 等。在这种思想的指导下，我们可以使用一个完全正确的 Soc 去仿真测试程序，当发现要写入通用寄存器的时候，将 PC 值，指令等定位信息写入到文本文件 trace.txt 中。第二步，我们使用自己实现的 Soc 环境仿真测试程序，当发现要写入到通用寄存器的时候，将 PC 值，指令等信息跟文本文件 trace.txt 中的信息比对，如果相等，则仿真继续，如果不相等，则仿真中断。

这就是此次体系实验 2 的调试方法，同时也是硬件综合设计的调试方法，希望同学熟练掌握。

9.2 测试程序汇编代码

本次实验运行的程序位于 soft/cache_lab 目录下，该程序主要由 start.S, bench/shell1/下的 shell.c 和 matrix_mult.c 组成。程序入口为 start.S，该汇编程序会调用 shell1.c 中的 shell1 函数，shell1 函数再调用 matrix_mult.c 中的 matrix_mult 函数。整个工程通过 Makefile 进行编译。提供的实验包中，程序已经编译，编译结果位于 soft/cache_lab/obj/shell1 文件夹下。其中 main.elf 为编译生成的可执行文件，axi_ram.coe 为生成的用于导入 ram 的 coe 格式文件，axi_ram.mif 和 inst_data.bin 不用管。重点需要了解 test.s 文件，该文件是 cpu 实际执行的汇编代码，通过 main.elf 反汇编得到 (objdump 命令)。

当我们知道 CPU 运行错误的地址后，我们通过查找 test.s 文件，可以知道 cpu 的执行上下文，从而可以分析出当前 cpu 的错误情况。

9.3 调试方法示例

当我们的 cache 实现错误时，在终端会输出报错信息。其中 PC 代表出错指令地址，wb_rf_num 表示写寄存器地址，wb_rf_wdata 表示写数据。其中 reference 那一行为从 trace 文件中读取出来的信息，为正确执行结果。mycpu 为我们自己实现的 Soc 仿真的结果。

在图中我们发现，mycpu 跟 reference 的 PC 值和 wb_rf_num 的值是一样的，但是 wb_rf_wdata 是不一样的。PC 值相同代表我们的指令执行序列是一样的，wb_rf_num 相同代表我们写入的通用存储器地址是正确的。wb_rf_wdata 不同则代表我们写入通用寄存器的数据错了。综上所述，即我们取指正确，但是指令执行结果错误，即写入通用寄存器错误。

```
------------------------------------------------------------
[  23889 ns] Error!!!
    reference: PC = 0x9fc012f4, wb_rf_wnum = 0x10, wb_rf_wdata = 0x00000073
    mycpu    : PC = 0x9fc012f4, wb_rf_wnum = 0x10, wb_rf_wdata = 0x00000000
------------------------------------------------------------
$finish called at time : 23928500 ps : File "D:/Depository/NSCSCC/Lab2/cache_lab_v0.
```

图 11: trace 比对，发现错误，输出信息

1. 定位错误。由 reference.pc=0x9fc012f4，我们在 test.s 中检索，定位到错误指令 lb，此指令从内存加载一个字节的数据到通用寄存器中。

```
3    9fc012f0: afb00010  sw   s0,16(sp)
4    /home/rain/loongson/cache_lab/lib/puts.c
5    9fc012f4:→80900000·→lb→ s0,0(a0)
6    9fc012f8: 00000000  nop
7    9fc012fc: 12000013  beqz  s0,9fc0134c <p
```

图 12: 在 test.s 中检索定位错误指令

2. 追溯 wb_rf_wdata 的来源。由于此条指令为 load 指令，所以 wb_rf_wdata 是从 cache 返回给 CPU 的。最后溯源到 cpu_data_rdata，最后发现是我们将 cpu_data_rdata 恒置为 0 导致的错误。

```
    .cache_inst_data_ok (cache_inst_data_ok )
);

assign cpu_data_rdata = 32'b0;

d_cache d_cache(
    .clk(clk), .rst(rst),
    .cpu_data_req     (cpu_data_req     ),
    .cpu_data_wr      (cpu_data_wr      ),
    .cpu_data_size    (cpu_data_size    ),
    .cpu_data_addr    (cpu_data_addr    ),
    .cpu_data_wdata   (cpu_data_wdata   ),
    // .cpu_data_rdata   (cpu_data_rdata   ),
    .cpu_data_addr_ok (cpu_data_addr_ok ),
```

图 13: 定位到错误原因

9.4 波形图的导入与导出、波形图的调试小技巧

在仿真调试过程中，我们需要从波形图获取有效信息。为了避免重复拉取波形图信号，我们可以将已经配置好的波形图保存下来。在工程的后续调试中或者小组合作时，可以导入已经配置好的波形图文件，当然，需要在同一个工程项目中。vivado 所有版本都提供这种功能，以下是在 vivado_v2019.2 中的操作步骤。

1. 导出配置好的波形图文件。快捷键 ctrl+s，或者点击 file–simulation waveform–save configuration as……

图 14: 导出波形图配置

2. 导入波形图文件。点击 file–simulation waveform–open configuration。

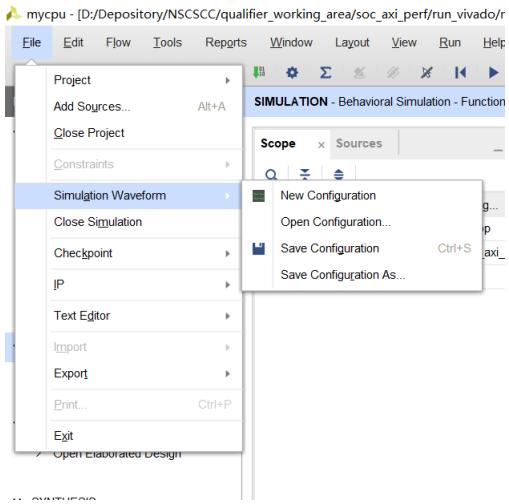

图 15: 导入波形图配置文件

3. 当我们导入好波形图后，我们需要 restart 以加载导入的波形图，然后点击 run all 命令，开始仿真。注意区分 relaunch 与 restart 用法。当修改设计代码、测试代码和设置 ip 核参数后，只用 relaunch 仿真，才能生效。对于加载波形图跟修改波形图，只用 restart 并且 run all 即可。

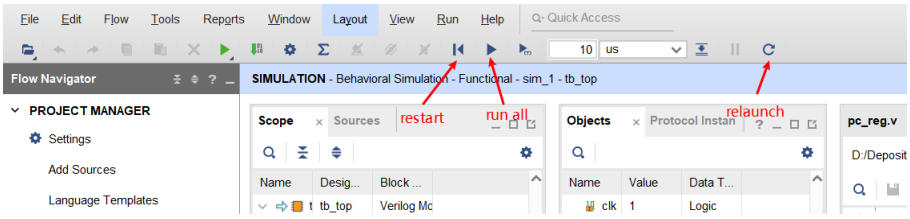

图 16: 导入波形图后如何运行仿真

同时，我们提供一些波形图配置的小技巧。

1. 信号量分组。多选住信号量后，右键可进行分组。对于有很多变量的分组，可以按住 shift 键，然后鼠标左键确定起点和终点，即可多选。
2. 善用搜索功能，选中一条波形，可通过数值索引到对应的位置。
3. 当信号量过多时，可能发生左侧信号量跟波形错位的情况，同学们应注意。
4. 当信号量过多时，可通过配置波形颜色，减轻用眼负担。

9.5 示例波形图

我们将 cache 抽取成功一个完全独立的、类 sram 接口的模块。为了便于大家调试，同时我们提供一个基础的示例波形图。在此波形图中，我们拉取了相关的基础信号量，我们认为这些基础信号量有助于同学们分析时序。当然，在进行实验的过程中，同学们要拉取新的信号量，配合这些基础信号进行调试。下面是对这些基础信号量的语义翻译。

基础信号量大致可以分为五组，第一组是全局复位信号；第二组是 DEBUG 组，提供 trace 比对信息；第三组是 Mipscore，有关 CPU 数据通路内部信号量；第四组是指令 cache 的输入、输出接口信号量；第五组是数据 cache 的输入、输出接口的信号量。

1 全局复位信号量。全局复位，我们主要通过此信号量来定位波形图中 CPU 启动的位置。

2 DEBUG。上面我们讲到，我们通过比对 trace.txt 文件来定位错误。DEBUG 信号量组中给出了比对的信息。其中 debug_ 开头的信号量表示 CPU 内部的信号量，ref_ 开头的信号表示为读取 trace.txt 文件中对应列的信号量，例如，ref_wb_pc 信号量对应 trace.txt 文件的第二列。下面对各个信号进行解释。

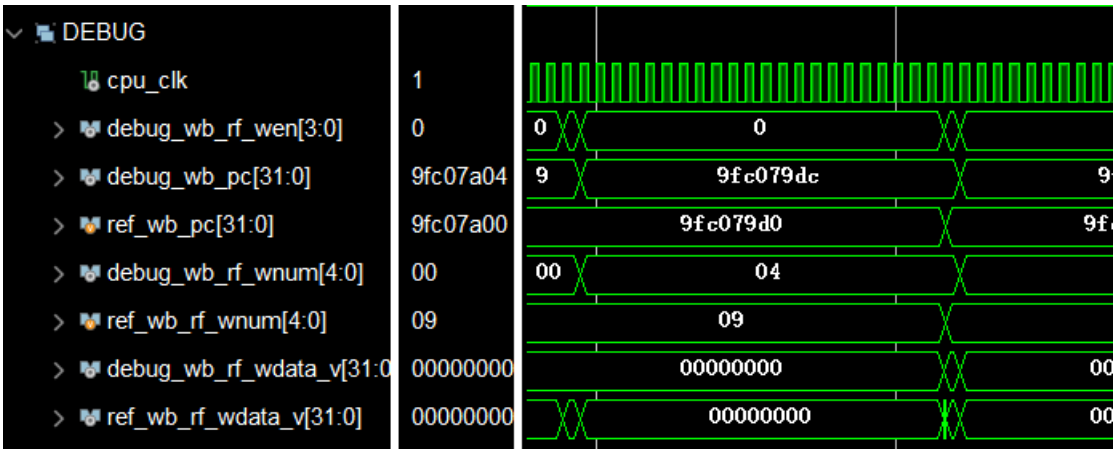

图 17: DEBUG 信号量

(a). debug_wb_rf_wen。为 CPU 内部访存阶段写通用寄存器使能。
(b). debug_wb_pc。为 CPU 内部指令访存阶段的指令地址。
(c). ref_wb_pc。trace 文件中写通用寄存器时的指令地址，对应 trace 文件第二列。
(d). debug_wb_rf_wnum。CPU 访存阶段写通用寄存器地址，即写哪一个通用寄存器。
(e). ref_wb_rf_wnum。trace 文件中写通用寄存器地址，对应 trace 文件第三列。
(f). debug_wb_rf_wdata。CPU 访存阶段写通用寄存器数据。
(g). ref_wb_rf_wdata。trace 文件中写通用寄存器数据，对应 trace 文件第四列。

3 MipsCore。主要有两个信号量，为 CPU 数据通路部分(datapath)的信号量

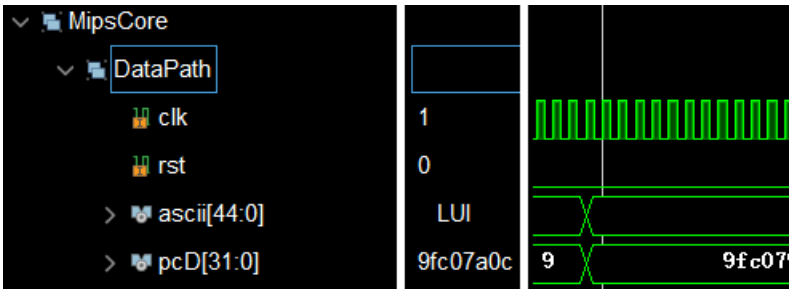

图 18: MIPSCORE 信号量

(a). ascii。译码阶段的指令。我们使用一个转换模块，将 32 位指令编码翻译成指令，方便大家调试。
(b). pcD。译码阶段的 PC 值。

4 I_Cache。主要是指令 cache 的输入、输出接口信号量，后续同学们自己实现的 cache 的信号量可以添加到此处。

(a). cpu_inst_req。Mipscore–>Cache 接口。MipsCore 发起读写请求。
(b). cpu_inst_addr。Mipscore–>Cache 接口。MipsCore 读写地址。
(c). cpu_inst_rdata。Cache–>Mipscore 接口。Cache 返回给 MipsCore 的数据。
(d). cpu_inst_addr_ok。Cache–>Mipscore 接口。Cache 返回给 Mipscore 的地址握手成功。
(e). cpu_inst_data_ok。Cache–>Mipscore 接口。Cache 返回给 Mipscore 的数据成功。
(f). cache_inst_req。Cache–>axi_interface 接口。Cache 发送的读写请求，可因为 cache 缺失或者脏的 cacheline 被替换产生的写请求。

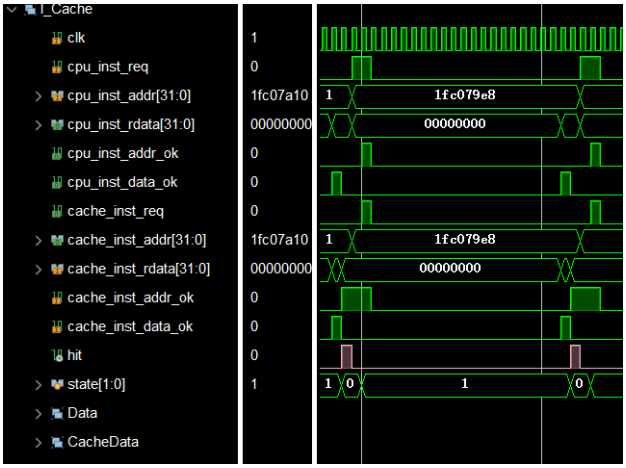

图 19: 指令 cache 信号量

(g). cache_inst_rdata。从内存返回给指令 cache 的数据。

(h). cache_inst_addr_ok。从内存中返回，如果是读请求，表示成功收到地址，地址握手成功；如果是写请求，则表示成功收到地址数据。

(i). cache_inst_data_ok。从内存中返回，如果是读请求，表示从返回 cache 的数据。如果是写请求，代表写入数据成功。

(j). hit。hit 为 1，表示 cache 命中。为 0，表示缺失。

(k). state。cache 状态机所处状态。

5 D_Cache。主要是跟数据 cache 有关，跟指令 cache 几乎一样，不再赘述。对上面未涉及的四个信号量进行解释。

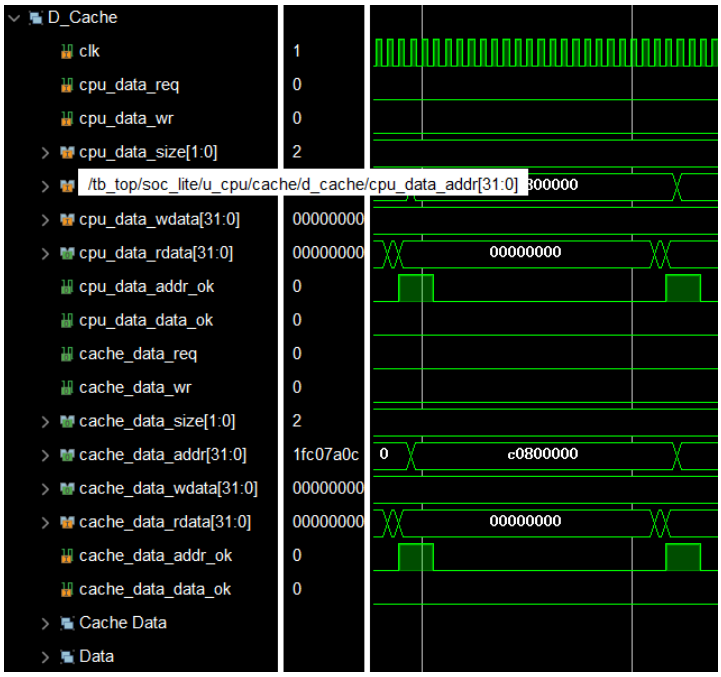

图 20: 数据 cache 信号量

(a). cpu_data_wr。MipsCore–>Cache。代表当前请求是否是写请求。

(b). cpu_data_size。Mipscore–>Cache。结合地址最低两位，确定数据的有效字节，详情请参考类 sram 接口相关文档。

(c). cache_data_wr。Cache 输出接口。代表当前请求是否是写请求。

(d). cpu_data_size。Cache 输出接口。结合地址最低两位，确定数据的有效字节。详情请参考类 sram 接口相关文档。

9.6 调试建议

1. 本次实验提供通过测试的 Soc 环境。实验内容是要求同学们自实现 cache。因此，如果测试出现问题，极大概率是 cache 实现错误。
2. verilog 是并行编程，要求较复杂的时序分析能力。调试时请关注类 sram 接口的时序逻辑，以及 cache 在读命中、读缺失、写命中、写缺失等情况下的时序逻辑。
3. 在调试中，如果出现写数据错误，在这种情况下，可以默认 debug_wb_rf_wdata == cpu_data_rdata。因为 Mipscore 这部分是没有错的，如果是写数据错误，那么意味着从 cache 返回来的数据有误，即同学们自己的实现有误。

9.7 常见错误

在使用 trace 比对机制调试的情况下，由于 mipscore 是正确的，所以写寄存器地址不会错误。即有可能是 PC 值和写寄存器数据不一样的错误。如果是 PC 值错误，那么应该是指令 cache 实现错误。如果是写寄存器数据错误，那么应该是数据 cache 实现错误。

欢迎同学们补充！