中国科学院大学
《计算机组成原理(研讨课)》实验报告

姓名 韩初晓 学号 2023K8009908002 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。

1 电路设计及说明

本次实验的目标是设计并实现一个分离的指令 Cache (I-Cache) 和数据 Cache (D-Cache), 以提高 CPU 的访存性能。Cache 作为 CPU 与主存之间的高速缓冲存储器, 利用程序的局部性原理, 存储最近或最常访问的数据和指令, 从而减少 CPU 等待主存响应的时间。本设计中, I-Cache 和 D-Cache 均采用组相联映射方式, 并实现了相应的替换策略和写策略。

1.1 Cache 基本参数与地址映射

I-Cache 和 D-Cache 共享大部分结构参数定义, 如:

- `CACHE_SET = 8`: Cache 的组数, 即有 8 组。
- `CACHE_WAY = 6`: 每组的相联路数, 即 6 路组相联。
- `DATA_WIDTH = 32`: CPU 数据通路宽度, 即 32 位。
- `LINE_LEN = 256`: Cache 行(Line / Block)的大小, 即 256 位 (32 字节)。
- `TAG_LEN = 24`: Tag 字段的长度。
- `INDEX_WIDTH = 3`: Index 字段的长度 ($2^{\text{INDEX\_WIDTH}} = \text{CACHE\_SET}$)。
- `OFFSET_WIDTH = 5`: Offset 字段的长度 ($2^{\text{OFFSET\_WIDTH}} = \text{LINE\_LEN} / 8$ 字节)。

CPU 发出的 32 位地址被划分为三部分:

- **Tag**: `addr[31 : OFFSET_WIDTH + INDEX_WIDTH]`, 用于与 Cache 中存储的 Tag 进行比较, 判断是否命中。
- **Index**: `addr[OFFSET_WIDTH + INDEX_WIDTH - 1 : OFFSET_WIDTH]`, 用于选择 Cache 中的特定组。
- **Offset**: `addr[OFFSET_WIDTH - 1 : 0]`, 用于在命中的 Cache 行内选择具体的字节或字。

以上这些参数的定义通过宏定义来实现。

1.2 Cache 存储阵列

I-Cache 和 D-Cache 均由多个存储阵列构成,每个阵列通过自定义的参数化模块 `custom_array` 实例化。对于每一路(Way):

- **Valid Array**: 存储该 Cache 行是否有效。数据宽度为 1 位。
- **Tag Array**: 存储该 Cache 行的 Tag 值。数据宽度为 `TAG_LEN`。
- **Data Array**: 存储该 Cache 行的数据。数据宽度为 `LINE_LEN`。
- **Last Hit Time Array (LRU 信息)**: 存储该 Cache 行最近一次被命中的时间戳,用于 LRU 替换策略。数据宽度为 `TIME_WIDTH`。

D-Cache 额外包含:

- **Dirty Array**: 存储该 Cache 行是否为"脏"(即 Cache 中的数据被修改过但尚未写回主存)。数据宽度为 1 位。

为了简化电路的设计,我并没有使用框架里提前给定的模块,而是利用 Verilog 的参数声明特性来声明了一个统一的 `custom_array`,其定义如下:

```
`timescale 10 ns / 1 ns

`define TARRAY_ADDR_WIDTH 3

module custom_array #(
    // Define a parameter for the data width
    // Provide a default value for it
    parameter TARRAY_DATA_WIDTH = 24
) (
    input                           clk,
    input [`TARRAY_ADDR_WIDTH - 1 : 0] waddr,
    input [`TARRAY_ADDR_WIDTH - 1 : 0] raddr,
    input                           wen,
    input                           rst,
    input [TARRAY_DATA_WIDTH - 1 : 0] wdata,
    output [TARRAY_DATA_WIDTH - 1 : 0] rdata
);

    reg [TARRAY_DATA_WIDTH - 1 : 0] array [(1 << `TARRAY_ADDR_WIDTH) - 1 : 0];
    integer i;

    always @(posedge clk) begin
        if (rst) begin
          for (i = 0; i < (1 << `TARRAY_ADDR_WIDTH); i = i + 1)
            array[i] <= {TARRAY_DATA_WIDTH{1'b0}};
        end else if (wen)
            array[waddr] <= wdata;
    end

    assign rdata = array[raddr];
```

```
endmodule
```

于是,在 I-Cache 和 D-Cache 中,我们可以这样实例化它们:

```
// generate cache
genvar i;
generate
  for (i = 0; i < `CACHE_WAY; i = i + 1) begin
    custom_array #(
      .TARRAY_DATA_WIDTH(1)
    ) valid_array (
      .clk(clk),
      .waddr(index),
      .raddr(index),
      .wen(way_wen[i]),
      .rst(rst),
      .wdata(1'b1), // write valid bit
      .rdata(way_valids[i])
    );

    custom_array #(
      .TARRAY_DATA_WIDTH(1)
    ) dirty_array (
      .clk(clk),
      .waddr(index),
      .raddr(index),
      .wen(way_wen[i]),
      .rst(rst),
      .wdata(way_wen_at_hit[i]), // write dirty bit
      .rdata(way_dirty[i])
    );

   custom_array #(
      .TARRAY_DATA_WIDTH(`TAG_LEN)
    ) tag_array (
      .clk(clk),
      .waddr(index),
      .raddr(index),
      .wen(way_wen[i]),
      .rst(rst),
      .wdata(tag), // write tag
      .rdata(way_tags[i])
    );

    custom_array #(
      .TARRAY_DATA_WIDTH(`LINE_LEN)
    ) data_array (
      .clk(clk),
      .waddr(index),
      .raddr(index),
```

```
      .wen(way_wen[i]),
      .rst(rst),
      .wdata(way_wdata[i]), // write data
      .rdata(way_rdata[i])
    );
  end
endgenerate
```

1.3 指令 Cache (I-Cache) 设计

I-Cache 负责缓存 CPU 取指请求的指令。由于指令通常是只读的,I-Cache 的设计相对 D-Cache 简单。

1.3.1 I-Cache 主要模块与信号

- **存储阵列**: 包括 `valid_array`, `tag_array`, `data_array`, `last_hit_array`。
- **替换逻辑**: 实例化了 `replacement` 模块,实现 LRU 替换策略。该模块根据各路的时间戳决定替换哪一路。
- **LRU 时间戳计数器**: `lru_timestamp_counter`,在每次 Cache 命中时递增,并更新命中路的 `last_hit_array`。
- **命中/缺失判断**:
  - `way_hits[i] = way_valids[i] && (way_tags[i] == tag)`: 判断第 i 路是否命中。
  - `hit = way_hits[0] || ... || way_hits[5]`: 总体命中信号。
  - `miss = from_cpu_inst_req_valid && !hit`: 总体缺失信号。
  - `hit_way_index`: 命中路的索引。
- **数据选择**: 命中时, 根据 `offset` 从 `way_rdata[hit_way_index]` 中选择对应的 32 位指令返回给 CPU (`to_cpu_cache_rsp_data`)。

其中, 存储阵列的实例化通过和上面相同的 `generate for` 循环来实现, `way_hits` 等多位宽信号的每一位同样通过 `for` 循环来赋值,具体赋值逻辑如下:

```
// generate the hit, wen and wdata signals
generate
  for (i = 0; i < `CACHE_WAY; i = i + 1) begin
    assign way_hits[i] = way_valids[i] && (way_tags[i] == tag);
    assign way_wen_at_refill[i] = (replaced_way == i) && (current_state == REFILL) &&
        (from_mem_rd_rsp_valid); // only enable write when mem rsp is valid!
    assign way_wen[i] = way_wen_at_refill[i]; // write enable for the way that was hit or refilled
    assign way_wdata[i] = (way_wen_at_refill[i]) ? ({from_mem_rd_rsp_data, way_rdata[i][`LINE_LEN
        - 1 : `DATA_WIDTH]}) :
                          256'b0;
  end
endgenerate
```

在上述代码中,`way_wdata` 信号的定义使得我花费了很多的时间思考。事实上,对于 I-Cache,其写使能仅仅在充填时生效。重填时, 我们每次将自内存发送过来的数据写到选中的路的最左端, 在下次写的时候将该路中的数据右移之后再写入。这样,可以确保来自内存低地址数据存在该路的 LSB 的一端。

1.3.2 I-Cache 状态机 (FSM)

I-Cache 使用一个状态机来控制其操作流程:

- INIT: 初始化状态,复位后进入,然后跳转到 WAIT_CPU。
- WAIT_CPU: 等待 CPU 发出的指令请求 (from_cpu_inst_req_valid)。
  - 若请求有效且 Cache 命中 (hit), 则保持在 WAIT_CPU 状态, 并通过 to_cpu_cache_rsp_valid 和 to_cpu_cache_rsp_data 向 CPU 返回指令。同时更新 LRU 时间戳。
  - 若请求有效但 Cache 不命中 (!hit),则跳转到 MISS_CL 状态。
- MISS_CL (Cache Line Miss): 发生 Cache Miss。
  - 向主存发出读请求 (to_mem_rd_req_valid = 1, to_mem_rd_req_addr 为缺失块的行对齐地址)。
  - 等待主存就绪 (from_mem_rd_req_ready),若就绪则跳转到 REFILL 状态。
- REFILL: 从主存接收数据填充 Cache 行。
  - to_mem_rd_rsp_ready = 1,表示 Cache 准备好接收数据。
  - 当主存返回数据 (from_mem_rd_rsp_valid) 时,将数据写入被替换路 (replaced_way) 的 data_array, 并更新其 valid_array 和 tag_array。注意 way_wdata[i] 的赋值逻辑是部分更新,将新读入的 32 位数据与原 Cache Line 中剩余部分合并 (实际 I-Cache 是一次性读整个 line, 这里逻辑可能针对 burst 传输的第一个 beat)。
  - 当接收到最后一个数据包 (from_mem_rd_rsp_last),即 r_done 为真时,跳转回 WAIT_CPU 状态。否则保持在 REFILL。

I-Cache 的 to_cpu_inst_req_ready 信号在 WAIT_CPU 状态下为高,表示可以接收 CPU 请求。

该状态机的代码如下:

```
always @(*) begin
  case (current_state)
    INIT: begin
      next_state = WAIT_CPU; // Start in INIT state, then wait for CPU request
    end
    WAIT_CPU: begin
      if (from_cpu_inst_req_valid && hit) begin
        next_state = WAIT_CPU; // If CPU request is valid, go to lookup state
      end else if (from_cpu_inst_req_valid && !hit) begin
        next_state = MISS_CL;
      end
    end
    MISS_CL: begin
      next_state = (from_mem_rd_req_ready) ? REFILL : MISS_CL; // Wait for memory
    end
    REFILL: begin
      if (r_done) begin
        next_state = WAIT_CPU; // After refill, go to hit state
      end else begin
        next_state = REFILL; // Continue refilling if not done
      end
```

```
    end
    default: begin
      next_state = WAIT_CPU; // Default case to handle unexpected states
    end
  endcase
end
```

这个状态机的工作状态仅含有三个状态，可以配合流水线 RISC-V CPU 进行高效取值。经过我的实测，最佳情况下能够达到两拍一取指。实测发现，I-Cache 对 CoreMark 成绩影响巨大。我将状态机从四拍一取指改进为三排一取指之后，CoreMark 分数从 1341096 增长为 937016.20。

图 1: I-Cache 取指效率示意图

1.4 数据 Cache (D-Cache) 设计

D-Cache 负责缓存 CPU 的 Load/Store 指令访问的数据。D-Cache 需要处理读和写操作，并管理 Dirty 位以支持 Write-Back 策略，同时还要处理 Bypass 访问（如 I/O 空间）。

1.4.1 D-Cache 主要模块与信号

- **存储阵列**: 包括 `valid_array`, `dirty_array`, `tag_array`, `data_array`, `last_hit_array`。
- **替换逻辑**: 实例化了 `replacement_simple` 模块（总是替换第 0 路，实际应用中应为 LRU）。事实上，为了确保上板仿真成功和处理器工作的稳定性，我只能在 I-Cache 和 D-Cache 中任选其一使用 lru 替换策略，否则就会出现布线拥塞错误。
- **命中/缺失/Dirty 判断**:
  - `way_hits[i]`, `hit`, `miss`, `hit_way_index` 与 I-Cache 类似。
  - `dirty = way_dirty[replaced_way]`: 判断被替换路是否为脏。
  - 事实上，这些信号依然定义在一个 `generate for` 循环体内，具体代码如下：

```
// generate the hit, wen and wdata signals
generate
  for (i = 0; i < `CACHE_WAY; i = i + 1) begin
    assign way_hits[i] = way_valids[i] && (way_tags[i] == tag);
    assign way_wen_at_hit[i] = way_hits[i] && from_cpu_mem_req_valid && (current_state
        == W_HIT);
    assign way_wen_at_refill[i] = (replaced_way == i) && (current_state == REFILL) &&
        (from_mem_rd_rsp_valid); // only enable write when mem rsp is valid!
    assign way_wen[i] = way_wen_at_hit[i] || way_wen_at_refill[i];
    assign way_wdata[i] = (way_wen_at_hit[i]) ? (
                        (~({{(`LINE_LEN - `DATA_WIDTH){1'b0}}, mask} <<
                            {offset[`OFFSET_WIDTH - 1:2], 5'b0}) & way_rdata[i]) |
                            ({{(`LINE_LEN - `DATA_WIDTH){1'b0}}, (from_cpu_mem_req_wdata
                            & mask)} << {offset[`OFFSET_WIDTH - 1:2], 5'b0})
                        ) :
```

```
                        (way_wen_at_refill[i]) ? ({from_mem_rd_rsp_data,
                            way_rdata[i][`LINE_LEN - 1 : `DATA_WIDTH]}) :
                        256'b0;
  end
endgenerate
```

  - 这里，值得重点解析的仍然是 `way_wdata` 信号的赋值。这个信号用来自 CPU 的掩码处理好来自 CPU 的写数据，并通过字偏移写到 Cache 行对应的位置。

- **Bypass 判断**: `Bypass = ( |(from_cpu_mem_req_addr & `NO_CACHE_MASK)) || (|(from_cpu_mem_req_addr & `IO_SPACE_MASK))`: 判断访问地址是否属于不可缓存区域或 I/O 空间。

- **写回计数器与暂存器**:
  - `write_counts`: 用于在写回(SYNC)或 Bypass 写(WRW)时计数已发送的数据包。
  - `sync_reg`: 在 SYNC 状态下，暂存从 Cache 读出的待写回数据行，并逐个 32 位字发送给内存。

- **数据通路与写操作**:
  - `to_cpu_cache_rsp_data`: 读命中或 Bypass 读完成时，向 CPU 返回数据，也需要根据 `offset` 选择。事实上，这里需要特别注意，由于存在加载字节和加载半字的指令，在这些指令中，CPU 给出的地址不是字对齐的，但是，我们需要从内存中对齐的地址给出数据，再由 CPU 内部的掩码等机制来处理这个对齐的数据。因此，我们不应该直接使用 `offset`，而是应该以如下所示的方式实现：

```
assign to_cpu_cache_rsp_data = (current_state == RDW) ? from_mem_rd_rsp_data :
                               (current_state == SEND_CPU_DATA) ?
                                   way_rdata[hit_way_index][{offset[`OFFSET_WIDTH - 1 :
                                   2], 5'b0} +: `DATA_WIDTH] :
                               32'b0;
```

  - `way_wdata[i]`: 写命中时，根据 CPU 的写请求 (`from_cpu_mem_req_wdata`, `from_cpu_mem_req_wstrb`, `offset`) 更新对应 Cache 行中的部分字节。Refill 时，更新整个数据包。
  - `to_mem_wr_data`: 写回或 Bypass 写时，向内存发送的数据。
  - `to_mem_wr_data_strb`: 写回时为 `4'b1111`，Bypass 写时根据 CPU 请求。

1.4.2 D-Cache 状态机 (FSM)

D-Cache 的状态机更为复杂，以处理读、写、命中、缺失、写回和 Bypass 等多种情况：

- `INIT`: 初始化，然后跳转到 `WAIT_CPU`。
- `WAIT_CPU`: 等待 CPU 内存访问请求 (`from_cpu_mem_req_valid`)。
  - 若请求有效：
    * 若为 Bypass 访问 (`Bypass`)，根据是读 (`!from_cpu_mem_req`) 还是写 (`from_cpu_mem_req`)，跳转到 `R_BP` 或 `W_BP`。
    * 若非 Bypass 且 Cache 命中 (`hit`)，根据是读还是写，跳转到 `R_HIT` 或 `W_HIT`。
    * 若非 Bypass 且 Cache 不命中 (`miss`)，判断被替换路是否为脏 (`dirty`)。若脏，则跳转到 `MISS_DT` (Dirty Miss，先写回)；若不脏，则跳转到 `MISS_CL` (Clean Miss，直接 Refill)。

- W_HIT (Write Hit): CPU 写命中。更新命中路的 data_array 和 dirty_array,并更新 LRU。然后跳转回 WAIT_CPU。
- R_HIT (Read Hit): CPU 读命中。跳转到 SEND_CPU_DATA。
- SEND_CPU_DATA: 向 CPU 发送读出的数据。等待 CPU 接收就绪 (from_cpu_cache_rsp_ready) 后,跳转回 WAIT_CPU。
- MISS_DT (Dirty Miss / Write-Back Request): 被替换的 Cache 行是脏的,需要先写回主存。
  - 向主存发出写请求 (to_mem_wr_req_valid = 1, to_mem_wr_req_addr 为被替换脏块的行对齐地址, to_mem_wr_req_len = 8'd7 表示写回整个 Cache Line)。此时将 way_rdata[replaced_way] 锁存到 sync_reg。
  - 等待主存就绪 (from_mem_wr_req_ready),若就绪则跳转到 SYNC 状态。
- SYNC (Synchronize / Write-Back Data Transfer): 向主存逐个发送 sync_reg 中的 32 位数据。
  - to_mem_wr_data_valid = 1。
  - 每当主存接收数据就绪 (from_mem_wr_data_ready),write_counts 递增,sync_reg 右移 32 位。
  - 当所有数据包发送完毕 (w_done 为真,即 write_counts == 8),跳转到 MISS_CL 状态(因为写回后通常是为新的数据腾出空间)。
- MISS_CL (Cache Line Miss / Refill Request): Cache 不命中(或写回完成后的不命中)。
  - 向主存发出读请求 (to_mem_rd_req_valid = 1, to_mem_rd_req_addr 为缺失块的行对齐地址, to_mem_rd_req_len = 8'd7 表示读取整个 Cache Line)。
  - 等待主存就绪 (from_mem_rd_req_ready),若就绪则跳转到 REFILL 状态。
- REFILL: 从主存接收数据填充 Cache 行。
  - to_mem_rd_rsp_ready = 1。
  - 当主存返回数据时,更新被替换路的 data_array, valid_array, tag_array。Dirty 位在 Refill 后应为 clean (0),除非紧接着是写操作。
  - 当接收到最后一个数据包 (r_done),根据原始 CPU 请求是读还是写,跳转到 R_HIT 或 W_HIT(此时应为命中)。
- W_BP (Write Bypass Request): CPU 请求写 Bypass 区域。
  - 向主存发出写请求 (to_mem_wr_req_valid = 1, to_mem_wr_req_addr 为 CPU 请求地址, to_mem_wr_req_len = 8'd0 表示单次写)。
  - 等待主存就绪,若就绪则跳转到 WRW。
- R_BP (Read Bypass Request): CPU 请求读 Bypass 区域。
  - 向主存发出读请求 (to_mem_rd_req_valid = 1, to_mem_rd_req_addr 为 CPU 请求地址, to_mem_rd_req_len = 8'd0 表示单次读)。
  - 等待主存就绪,若就绪则跳转到 RDW。
- WRW (Write Bypass Word): 向主存发送 Bypass 写数据。
  - to_mem_wr_data_valid = 1。

  - 当写完成 (w_done,即 write_counts == 1 因为是单次写),跳转回 WAIT_CPU。

- RDW (Read Bypass Word): 从主存接收 Bypass 读数据。
  - to_mem_rd_rsp_ready = 1。
  - 当读完成 (r_done), 向 CPU 返回数据 (to_cpu_cache_rsp_valid, to_cpu_cache_rsp_data = from_mem_rd_rsp_data),并跳转回 WAIT_CPU。

D-Cache 的 to_cpu_mem_req_ready 信号在 W_HIT, R_HIT, WRW(完成时), R_BP(请求已发出时) 等状态下为高。

2 实验过程中遇到的问题、对问题的思考过程及解决方法

在 D-Cache 的设计和调试过程中,遇到了一些典型的问题,主要集中在 FSM 逻辑、数据通路和控制信号的正确性上。

1. **问题:Verilog 语法错误 - 数组归约赋值**
   - **错误的源代码:**在原始代码中,我使用了如下的逻辑:

```
wire array [5:0];
assign or_result = |array;
```

   - **思考:** Verilog 标准不允许直接对整个数组(packed or unpacked array of vectors)使用归约操作符并赋值给一个标量输出。归约操作符(如 |, &,^)通常用于向量的位操作。
   - **解决:**若意图是检查数组中所有位是否有至少一个为 1,应显式地将数组中的所有位(或所有向量的对应位)进行 OR 操作。例如,如果 array 是一个位向量数组,可能需要一个循环或 generate 块来组合。如果 array 是一个单一的 6 位向量,那么 assign or_result = |array; 是合法的。例如,在我的代码中需要对 hit 信号进行赋值,此时应使用 assign hit = way_hits[0] | way_hits[1] | ... ;。

2. **问题:LRU 时间戳更新时机错误**
   - **现象:**在 medium sum 测试中,寄存器写回数据错误 (rf_wdata)。定位发现是 LRU 时间戳 (last_hit_array) 的写使能 (wen) 条件不当,不应该在 Cache Refill 过程中更新 LRU。

```
================================================================
ERROR: at                50ns.
Yours:    PC = 0x00000068, rf_waddr = 0x05, rf_wdata = 0x00000001
Reference: PC = 0x00000060, rf_waddr = 0x0a, rf_wdata = 0x00000000
================================================================
```

   - **错误代码片段:** last_hit_array 的 wen 不应该包含与 Refill 相关的状态或信号。在下面的第一版代码中,我直接使用了 way_wen[i] 信号来作为 LRU 时间戳更新的使能信号。殊不知,这个信号在 REFILL 写使能的时候也会拉高,导致出错。

```
    custom_array #(
      .TARRAY_DATA_WIDTH(`TIME_WIDTH)
    ) last_hit_array (
      .clk(clk),
      .waddr(index),
```

```
      .raddr(index),
      .wen(way_wen[i]),
      .rst(rst),
      .wdata(lru_timestamp_counter), // write last hit time
      .rdata(way_last_hit[i])
    );
```

- **思考：** LRU（Least Recently Used）算法的核心是记录 Cache 行的最近访问情况。只有当 CPU 直接访问（读或写）并命中某 Cache 行时，该行才应被认为是“最近使用”的，其时间戳需要更新。在 Cache Miss 后从主存填充（Refill）数据到 Cache 行的过程，并不是 CPU 对该行的直接访问，因此不应更新其 LRU 时间戳。
- **解决：** 确保 `last_hit_array` 的写使能条件严格限定在 CPU 请求、Cache 命中且 FSM 处于表示 CPU 已完成对 Cache 访问的状态（如 D-Cache 中的 `WAIT_CPU` 状态下检测到命中，或 I-Cache 中 `WAIT_CPU` 状态下命中并送出数据时）。修改后，I-Cache 和 D-Cache 的 `last_hit_array` 的 `wen` 逻辑为 `(current_state == WAIT_CPU) && way_hits[i]`，这符合了仅在 `WAIT_CPU` 状态下发生命中时更新时间戳的要求。

3. **问题：Dirty 位判断逻辑错误**

- **现象：** 同样在 `medium sum` 测试中，寄存器写回数据错误，追溯到 D-Cache 的 `dirty` 信号判断逻辑不正确。发生如下的错误：

```
================================================================
ERROR: at                00ns.
Yours:    PC = 0x00000090, rf_waddr = 0x0f, rf_wdata = 0x00000000
Reference: PC = 0x00000090, rf_waddr = 0x0f, rf_wdata = 0x00000001
================================================================
```

- **错误代码片段：** 我在写第一版代码的时候犯了这样一个愚蠢的错误：

```
assign dirty = way_dirty[hit_way_index]; // dirty if the way that was hit is dirty
```

- **思考：** `dirty` 信号的目的是在发生 Cache Miss，需要选择一个 Cache Way 进行替换时，判断被选中的 `replaced_way` 是否是“脏”的。如果被替换路是脏的，则需要先将其内容写回主存，然后才能用新的数据覆盖它。因此，`dirty` 信号应该与 `replaced_way` 相关联，而不是在 Cache Hit 时根据 `hit_way_index` 来判断。
- **解决：** 将 `dirty` 信号的赋值修改为 `assign dirty = way_dirty[replaced_way];`。这样，当 FSM 进入 `MISS_DT` 或 `MISS_CL` 的决策点时，可以根据 `replaced_way` 对应的 `way_dirty` 值来正确判断是否需要写回。

4. **问题：D-Cache 响应 CPU 数据时未按字对齐**

- **现象：** 在 `advanced load-store` 测试中，D-Cache 返回给 CPU 的读数据（`to_cpu_cache_rsp_data`）没有根据请求地址的偏移量（offset）进行正确的字选择。
- **代码片段参考：** 原先，我的代码中采用了这样的逻辑，即直接根据内存地址来确定偏移量的值，进而筛选出数据：

```
assign to_cpu_cache_rsp_data = (current_state == RDW) ? from_mem_rd_rsp_data :
```

```
                                (current_state == SEND_CPU_DATA) ?
                                    way_rdata[hit_way_index][offset +: `DATA_WIDTH] :
                                32'b0; // Read data from memory or cache
```

- **思考：** CPU 通常以字（32 位）为单位进行访问。当 Cache 行（LINE_LEN = 256 位，即 32 字节或 8 个字）被读入 Cache 后，CPU 的 Load 指令可能请求该行内的任意一个字，半字或字节。Cache 需要根据原始请求地址的 offset 字段（特别是其中用于选择半字和字节的指令，应当使用 offset[4:2]）从 256 位的 way_rdata 中提取正确的 32 位数据。
- **解决：**确保 to_cpu_cache_rsp_data 在读命中（SEND_CPU_DATA 状态）时，使用正确的索引逻辑从命中的 Cache 行数据（way_rdata[hit_way_index]）中选取目标字。修改后，代码中使用了 way_rdata[hit_way_index][{offset[`OFFSET_WIDTH - 1 : 2], 5'b0} +: `DATA_WIDTH]（I-Cache）和类似的逻辑（D-Cache）。这里的 {offset[`OFFSET_WIDTH - 1 : 2], 5'b0} 计算了目标字在 256 位 Cache Line 中的起始比特位置（因为 OFFSET_WIDTH 是 5，所以 offset[4:2] 加上五个零就得到了包含所要求的地址的对齐字在行内的索引。回忆处理器处理 lb、lh 等指令的规则，我们发现，处理器首先先从内存中读取一个对齐字，随后，再根据内存地址的特征，从对齐字中筛选出它想要的字节或者半字。因此，提取出哪一个字节或是哪一个半字，是处理器的工作，我们实现的缓存只需要按要求提供包含数据的对齐字即可。

5. **问题：D-Cache FSM 在 Bypass 读（RDW）完成后跳转错误**

- **现象：** microbench sieve 程序在某个点卡死。调试发现是 D-Cache FSM 在 RDW 状态（Bypass 读数据传输）完成后，状态跳转逻辑错误。
- **错误代码片段：**在第一版代码中，我事实上是由于一个笔误造成了这个错误：

```
next_state = (r_done) ? SEND_CPU_DATA : RDW; // Wait for read to complete
```

- **思考：** RDW 状态用于处理对不可缓存区域的读操作。数据直接从主存返回给 CPU，不经过 Cache 的存储阵列。当数据传输完成（r_done 为真）后，该 Bypass 操作即告结束，Cache 应返回到空闲状态 WAIT_CPU，等待 CPU 的下一个访存请求。SEND_CPU_DATA 状态是用于 Cache 读命中后，从 Cache 阵列向 CPU 发送数据的，不适用于 Bypass 操作。
- **解决：**将 RDW 状态在 r_done 为真时的下一状态修改为 WAIT_CPU。即：RDW: next_state = (r_done) ? WAIT_CPU : RDW;。提供的 D-Cache 代码已按此方式修正。

这些问题的解决过程通常涉及到仔细阅读代码、分析波形图、单步跟踪 FSM 状态转换以及对比预期行为与实际行为。对 Cache 的工作原理和各种特殊情况（如 Bypass、写回）的深刻理解是成功调试的关键。

3 实验结果

3.1 仿真结果与波形图分析

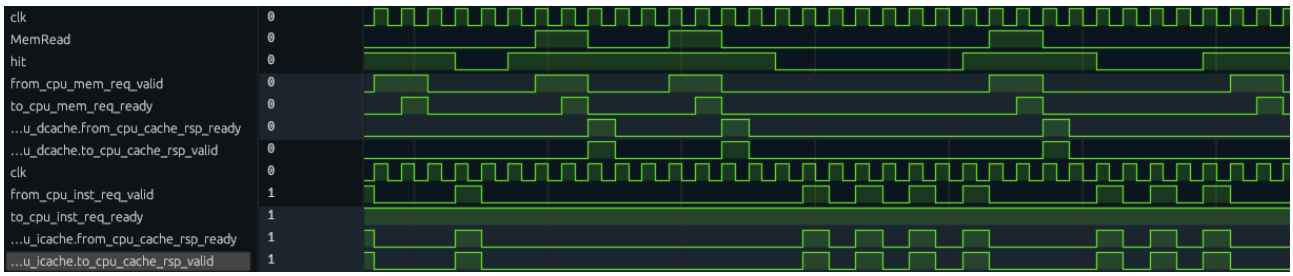

图 2: I-Cache 和 D-Cache 同 CPU 之间的握手

从上述图片可以看出，I-Cache 和 D-Cache 能够在缓存命中的时候正常和 CPU 进行交互。不过，D-Cache 取出一个数据所需的时钟周期数远大于 I-Cache。这是由于 D-Cache 的状态机比较复杂，这是一个值得着重优化的地方，如果能够精简 D-Cache 的状态机状态数量，使得 CPU 中的访存模块与 D-Cache 配合得更加紧密，那么性能将进一步得到提升。

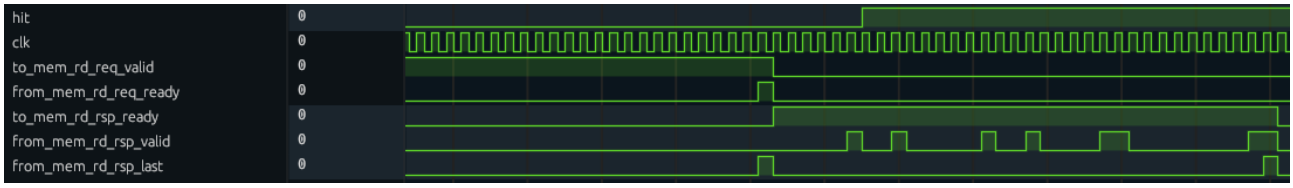

图 3: 未命中处理过程

上图展现了 I-Cache 在不命中时的处理流程：可以看到 `to_mem_rd_req_valid` 拉高，随后在 `REFILL` 状态接收数据。

最终，我的设计通过了全部的运行测试程序，我设计的 I-Cache 和 D-Cache 均能正确处理命中、不命中、数据填充、LRU 替换（I-Cache）、Dirty 位管理和写回（D-Cache）以及 Bypass 访问（D-Cache）等情况。例如，除上述图片中展示的外，在 D-Cache 发生 Dirty Miss 时，会先进入 `SYNC` 状态写回脏数据块，再进入 `MISS_CL` 和 `REFILL` 状态填充新数据块。Bypass 访问则直接与内存交互，不改变 Cache 内容，等等。

3.2 UART 控制器打印结果

UART 控制器的执行结果如下，cache 的工作符合要求：

```
RUNNER_CNT = 0
Completed FPGA configuration
Launching hello benchmark...
tggetattr: Inappropriate ioctl for device
reset: before MMIO access...
reset: MMIO accessed
axi_firewall_unblock: firewall error status: 00000000
main: before DDR accessing...
main: DDR accessed...
reset: before MMIO access...
reset: MMIO accessed
```

```
testing 1 2 0000003
faster and "cheaper"
deadf00d % DEADF00D
00000000100000000200000000300000004000000005
50 50 -50 4294967246
time 10748.91ms
reset: before MMIO access...
reset: MMIO accessed
./software/workload/ucas-cod/benchmark/simple_test/hello/riscv32/elf/hello passed
Hit good trap
pass 1 / 1
Job succeeded
```

3.3 性能计数器统计结果对比分析 (流水线 vs Cache)

为了直观地比较带有 Cache 的流水线处理器相对于基础流水线处理器(无 Cache)的性能提升,我们对两者在相同基准测试程序下的关键性能计数器数据进行了整理和分析。主要关注的指标包括总执行周期数 (Cycles)、CPI (Cycles Per Instruction)、数据内存访问相关的暂停周期 (MEM Access Stalls, RDW Stalls for No Cache; MEM Access Stalls for With Cache) 以及取指相关的暂停周期 (IW Stalls for No Cache; IF Stalls for With Cache)。

下表汇总了两个处理器版本在各个测试程序上的数据:

表 1: 流水线处理器 (无 Cache) vs 带 Cache 流水线处理器性能对比

Benchmark	Metric	Cycles		CPI		Data MEM Stalls	
		No Cache	With Cache	No Cache	With Cache	No Cache	With Cache
15pz	Cycles	522,031,244	50,765,643				
	Retired Instr.	5,224,497	5,224,497				
	CPI	99.92	9.72	99.92	9.72		
	Data MEM Stalls (Acc+RDW)	146,835,265	28,055,106			146,835,265	28,055,106
	Instr. Fetch Stalls	345,157,535	64,136				
bf	Cycles	38,802,291	3,187,934				
	Retired Instr.	452,870	452,870				
	CPI	85.68	7.04	85.68	7.04		
	Data MEM Stalls (Acc+RDW)	6,559,510	1,593,952			6,559,510	1,593,952
	Instr. Fetch Stalls	29,927,844	4,046				
dinic	Cycles	1,321,124	224,605				
	Retired Instr.	16,707	16,707				
	CPI	79.08	13.44	79.08	13.44		
	Data MEM Stalls (Acc+RDW)	289,366	167,633			289,366	167,633
	Instr. Fetch Stalls	1,102,620	7,160				
fib	Cycles	180,835,799	5,868,893				
	Retired Instr.	2,549,541	2,549,541				
	CPI	70.93	2.30	70.93	2.30		
	Data MEM Stalls (Acc+RDW)	312,648	139,735			312,648	139,735
	Instr. Fetch Stalls	170,564,736	29,367				
md5	Cycles	350,874	37,841				
	Retired Instr.	4,931	4,931				
	CPI	71.16	7.67	71.16	7.67		
	Data MEM Stalls (Acc+RDW)	39,383	23,070			39,383	23,070
	Instr. Fetch Stalls	328,329	2,905				
qsort	Cycles	675,607	38,941				
	Retired Instr.	9,496	9,496				
	CPI	71.15	4.10	71.15	4.10		
	Data MEM Stalls (Acc+RDW)	111,170	12,880			111,170	12,880
	Instr. Fetch Stalls	630,415	1,062				
queen	Cycles	5,798,766	284,616				
	Retired Instr.	81,506	81,506				
	CPI	71.14	3.50	71.14	3.50		
	Data MEM Stalls (Acc+RDW)	1,051,847	72,999			1,051,847	72,999
	Instr. Fetch Stalls	5,409,133	908				
sieve	Cycles	739,179	27,377				
	Retired Instr.	10,211	10,211				
	CPI	72.39	2.68	72.39	2.68		
	Data MEM Stalls (Acc+RDW)	24,145	730			24,145	730
	Instr. Fetch Stalls	682,075	1,576				
ssort	Cycles	44,418,521	1,721,513				
	Retired Instr.	619,061	619,061				
	CPI	71.75	2.78	71.75	2.78		
	Data MEM Stalls (Acc+RDW)	810,532	254,357			810,532	254,357
	Instr. Fetch Stalls	41,396,019	47,545				

Note: "Data MEM Stalls" for No Cache = MEM Access Stalls + RDW Stalls. For With Cache = MEM Access Stalls + RDW Stalls (as per data). "Instr. Fetch Stalls" for No Cache corresponds to its IW Stalls; for With Cache, it's IF Stalls. Retired Instructions are nearly identical across versions. CPI = Cycles / Retired Instr.

分析与讨论:

1. **总执行周期数 (Cycles) 和 CPI 大幅降低:** 最显著的改进是,在所有测试程序中,带有 Cache 的流水线处理器的总执行周期数远低于无 Cache 的流水线处理器。相应地,CPI(每条指令平均执行周期数)也大幅下降。

- 例如,在 `15pz` 测试中,周期数从约 5.22 亿降至约 5076 万,CPI 从 99.92 降至 9.72。

- 在 `fib` 测试中,周期数从约 1.8 亿降至约 587 万,CPI 从 70.93 降至 2.30。

这充分证明了 Cache 通过缓存常用指令和数据,有效减少了 CPU 对慢速主存的直接访问次数,从而极大地提升了处理器的执行效率。

2. **指令获取暂停周期 (Instr. Fetch Stalls) 的显著减少:** 通过将无 Cache 版本中的 `IW Stalls` (Instruction Wait Stalls, 代表等待指令存储器) 与带 Cache 版本中的 `IF Stalls` (Instruction Fetch Stalls, 主要源于 I-Cache Miss) 进行对比,我们可以清晰地看到 I-Cache 带来的巨大性能提升。

- 在所有测试程序中,无 Cache 版本的 `IW Stalls` 都非常高,占据了总周期数的很大一部分,这反映了每次取指都直接访问慢速主存的巨大开销。
- 引入 I-Cache 后,取指相关的暂停 (`IF Stalls`) 急剧下降。例如,在 `15pz` 中,从约 3.45 亿周期降至仅约 6.4 万周期;在 `fib` 中,从约 1.7 亿周期降至约 2.9 万周期。

这表明 I-Cache 极大地提高了指令获取的效率,绝大多数指令能够从高速的 I-Cache 中获取,避免了漫长的主存访问等待。

3. **数据内存访问暂停周期 (Data MEM Stalls) 的显著减少:** 我们将无 Cache 版本中的 `MEM Access Stalls + RDW Stalls` 之和作为其数据内存访问的总暂停, 与带 Cache 版本中的 `MEM Access Stalls + RDW Stalls` (主要是 D-Cache Miss 和相关操作) 进行比较。

- 同样地,在所有测试中,带 Cache 版本的数据内存访问暂停远低于无 Cache 版本。例如,在 `15pz` 中,从约 1.47 亿周期降至约 2800 万周期。对于 `sieve` 这样数据访问相对较少的程序,效果更为明显,从约 2.4 万周期降至仅 730 周期。
- 这体现了 D-Cache 在缓存程序数据、减少 Load/Store 操作对主存的直接依赖方面的有效性。带 Cache 版本中的 `MEM Access Stalls` 主要反映了 D-Cache Miss 和写回操作的开销。
- 不过,在这里,我们可以明显感受到 D-Cache 对访存暂停周期的优化效果不如 I-Cache 好。这是因为,我为了能在上板阶段避免布线拥塞,在 D-Cache 中采用了 `replacement_simple` 逻辑,即每次替换都选择第零路。这种设计使得真正工作的 D-Cache 只有一路。导致频繁替换的发生,从而使得 D-Cache 性能下降。

4. **对整体性能的贡献:** Cache 的引入,通过同时优化指令获取和数据访问的效率,使得流水线能够更流畅地运行,减少了因等待存储器而造成的"空转"周期。这直接导致了总执行周期数和 CPI 的显著降低。

5. **其他计数器:**

- 指令级的计数器(如 `Retired Instructions`, `Loads`, `Stores`, `Branches` 等)在两个版本间基本一致,再次确认了 Cache 主要影响性能而非程序执行的逻辑路径。
- `Total MEM Ops Issued`(在 Cache 版本中,指 Cache 与主存的交互)远小于无 Cache 版本中 CPU 与主存的交互次数(即所有取指 +Load/Store),直观地显示了 Cache 的过滤作用。

结论:

通过对比分析,性能计数器数据清晰地揭示了 Cache 对流水线处理器性能的决定性提升。分离的 I-Cache 和 D-Cache 通过大幅减少对主存的直接访问,有效地缓解了存储墙问题,使得处理器在指令获取和数据操作两方面都获得了显著的加速。取指暂停和数据访存暂停的急剧减少是总执行周期和 CPI 降低的主要原因。实验结果有力地证明了存储层次结构(特别是 Cache)在现代高性能处理器设计中的核心地位和不可或缺性。

4 对讲义中思考题(如有)的理解和回答

本实验讲义中未明确指定思考题,但结合 Cache 设计实践,可以思考以下相关问题:

1. **Cache 替换策略的选择及其影响？**本实验 I-Cache 使用了 LRU (Least Recently Used) 替换策略，通过 `replacement` 模块实现，它试图替换掉最长时间未被访问的 Cache 行，理论上能较好地利用时间局部性，提高命中率。D-Cache 使用了简化的 `replacement_simple` 模块(总是替换第 0 路)，这在实际中性能较差。具体表现为 I-Cache 对性能的提升效果远大于 D-Cache。其他常见策略包括：

   - **Random**: 随机选择替换路。实现简单，硬件开销小，平均性能尚可，事实上，我在本次实验中也尝试过随即替换策略，其性能与 LRU 策略类似。
   - **FIFO (First-In, First-Out)**: 替换最早进入 Cache 的行。实现也较简单，但可能替换掉仍被频繁访问的旧行。

   LRU 通常性能最好，但硬件实现最复杂，需要跟踪每行的访问历史。对于高相联度的 Cache，完全 LRU 的实现成本很高，常采用伪 LRU (PLRU) 等近似算法。

   事实上，我为了提升性能，在不同的提交中频繁尝试了各种各样的算法。一开始，我便遇到了多路 D-Cache 因为布线拥塞无法综合出电路的问题。一开始，我认为是由于 `replacement` 模块的规模过大导致无法综合出电路。于是，我便着手优化 `replacement` 模块的逻辑。我试图通过维护一个队列来选出最后一个写入路来替换。具体方法是，初始化时，将第 j 路的 `last_hit[j]` 初始化成 j，随后，每当向一路中写入数据，那一路的 `last_hit` 值就变为零，其他路的 `last_hit` 值按一定规则递增。这样，每次只需要选出 `last_hit` 值为 5 的路，就对应着需要被替换的路。以上思路的实现代码如下：

```
// prepare init value for last_hit_array for replacement
integer j;
always @(posedge clk) begin
  if (rst) begin
    for (j = 0; j < `CACHE_WAY; j = j + 1) begin
      last_hit[0][j] <= j;
      last_hit[1][j] <= j;
      last_hit[2][j] <= j;
      last_hit[3][j] <= j;
      last_hit[4][j] <= j;
      last_hit[5][j] <= j;
      last_hit[6][j] <= j;
      last_hit[7][j] <= j;
    end
  end else if (current_state == WAIT_CPU && hit) begin
    for (j = 0; j < `CACHE_WAY; j = j + 1) begin
      last_hit[index][j] <= way_hits[j]                                    ? 0                    :
                            (last_hit[index][j] < last_hit[index][hit_way_index]) ?
                                last_hit[index][j] + 1 :
                            last_hit[index][j];
    end
  end
end

genvar i;
generate
    for (i = 0; i < `CACHE_WAY; i = i + 1) begin
        // For each element 'i' of the way_last_hit array,
        // assign it the corresponding element from the selected row of the last_hit memory.
        assign way_last_hit[i] = last_hit[index][i];
```

```
    end
  endgenerate

module replacement_simple (
    input                    clk,
    input                    rst,
    input [`TIME_WIDTH - 1 : 0] data_0, data_1, data_2,
                             data_3, data_4, data_5,
    output [           2 : 0] replaced_way
);

assign replaced_way = (data_0 == 5) ? 3'h0 :
                  (data_1 == 5) ? 3'h1 :
                  (data_2 == 5) ? 3'h2 :
                  (data_3 == 5) ? 3'h3 :
                  (data_4 == 5) ? 3'h4 :
                  (data_5 == 5) ? 3'h5 : 3'b0; // Default to 0 if no way has last hit time of
                      5
endmodule
```

此版本的优化理论上能够显著降低所需面积，实现一个较高性能的 LRU 排队器。但是在生成比特流的过程中，仍然遇到了布线阻塞的情况：

```
INFO: [Route 35-449] Initial Estimated Congestion
 ----------------------------------------------------------------------------
|          | Global Congestion | Long Congestion | Short Congestion |
|          |___________________|_________________|__________________|
| Direction | Size | % Tiles | Size | % Tiles | Size | % Tiles |
|___________|________|_________|________|_________|________|_________|
|     NORTH|  32x32|    1.05|    8x8|     0.21|  32x32|     1.88|
|___________|________|_________|________|_________|________|_________|
|     SOUTH|  16x16|    0.65|    4x4|     0.07|  32x32|     1.48|
|___________|________|_________|________|_________|________|_________|
|      EAST|  32x32|    0.74|    2x2|     0.03|  32x32|     1.94|
|___________|________|_________|________|_________|________|_________|
|      WEST|  32x32|    1.19|    2x2|     0.07|  64x64|     2.26|
|___________|________|_________|________|_________|________|_________|
```

可见，还是出现了非常严重的布线阻塞。究其原因，我认为产生布线阻塞的原因并是因为 `replacement` 模块的逻辑复杂，而是，当我的排队器只选择第零路时，综合器就只会生成第零路的电路，而将剩下的五路优化掉。这样，当我在 I-Cache 上使用 LRU 排队器而在 D-Cache 上使用仅替换第零路的排队器时，实际综合出来的电路中，I-Cache 有六路，而 D-Cache 只有一路。这样的电路规模是可以接受的；但是，当我对 I-Cache 和 D-Cache 都使用 LRU 策略时，就会导致综合出来的电路是双六路，进而导致严重的布线淤塞。因此，我下一步需要查看究竟是我的哪些语句造成了综合器布线规模如此巨大。(因为我替换学长代码后发现，虽然我们的代码逻辑类似，但是学长生成的电路布线拥塞程度远低于我的设计)

2. **写策略 (Write-Through vs Write-Back) 的优缺点和适用场景？**本实验 D-Cache 采用了 Write-Back (写回)策略。

   - **Write-Through (写直通)**: CPU 写操作同时更新 Cache 和主存。

  - **优点**: 实现简单,Cache 和主存数据总是一致,对多处理器 Cache 一致性友好。
  - **缺点**: 写操作速度受限于主存速度,频繁写操作会占用大量内存总线带宽,性能较低。通常配合写缓冲(Write Buffer)来缓解性能瓶颈。
- **Write-Back (写回)**: CPU 写操作只更新 Cache,并将对应 Cache 行标记为"脏"(Dirty)。脏行只在被替换出 Cache 时才写回主存。
  - **优点**: 写操作速度快(只写 Cache),多次写同一行只会产生一次主存写(在替换时),显著减少主存写次数,节省总线带宽,整体性能较高。
  - **缺点**: 实现复杂(需要 Dirty 位和写回控制逻辑),Cache 和主存数据可能不一致,若发生电源故障可能丢失未写回的脏数据,对 Cache 一致性协议要求更高。

Write-Back 通常用于对性能要求较高的场景,Write-Through 则在简单性或强一致性要求下使用。

3. **写不命中策略 (Write-Allocate vs No-Write-Allocate) 的优缺点？**本实验 D-Cache 在写不命中时,会先判断是否需要写回被替换的脏块, 然后从内存读取新的数据块 (Refill), 之后再进行写操作 (表现为 `REFILL` 后进入 `W_HIT` 状态),这符合 Write-Allocate 的行为。

   - **Write-Allocate (写分配)**: 当发生写不命中时,先把对应的内存块读入 Cache,然后再对 Cache 行执行写操作。通常与 Write-Back 策略配合使用。
     - **优点**: 利用了空间局部性。如果写入后很快会再次访问(读或写)该块或其附近数据,则后续访问可能命中 Cache。
     - **缺点**: 每次写不命中都需要一次额外的读主存操作(调块),开销较大,尤其当写入后不再访问该块时。
   - **No-Write-Allocate (非写分配 / Around-Write)**: 当发生写不命中时,数据直接写入主存,而不将对应块调入 Cache。通常与 Write-Through 策略配合使用。
     - **优点**: 避免了不必要的 Cache 块调入,减少了主存读带宽的占用,尤其适用于那些写入后很少或不再被访问的数据。
     - **缺点**: 如果写入后立即有对该块的读或再次写操作,则会再次发生 Cache Miss。

   选择哪种策略取决于应用的访问模式和所配合的写策略。

4. **Cache 一致性问题在本实验中是如何处理的？如果扩展到多核环境，会有什么挑战？**本实验是单 CPU 单 Cache 设计, 不涉及多核 Cache 一致性问题。数据的一致性仅指 Cache 与主存之间的一致性, 通过 Write-Back 策略和 Dirty 位来管理。如果扩展到多核环境,每个核有自己的 Cache,它们共享主存。当一个核修改了其 Cache 中某共享数据块的副本后,其他核 Cache 中该数据块的副本就会失效。为保证数据一致性,需要实现 Cache 一致性协议,如 MESI (Modified, Exclusive, Shared, Invalid) 或其变种。这些协议通过监听总线事务和 Cache 间的通信来维护各个副本的状态,确保任何读操作都能获取最新值,并正确处理写操作的传播。挑战包括协议的复杂性、总线/网络带宽的占用、以及可能引入的额外延迟。

5. **Cache 的组织参数(如 Cache 大小、相联度、块大小)如何影响性能？**

   - **Cache 大小 (Size)**: 通常情况下,Cache 越大,能存储的数据越多,发生容量不命中 (Capacity Miss) 的概率越低,命中率越高。但成本也越高,且过大的 Cache 可能导致访问延迟增加。
   - **相联度 (Associativity / Ways)**: 相联度越高,一个内存块可以映射到 Cache 中的位置越多,发生冲突不命中 (Conflict Miss) 的概率越低。例如,从直接映射 (1-way) 到 N 路组相联再到全相联,冲突不命中逐渐减少。但相联度增加会导致比较逻辑更复杂、硬件成本更高、功耗更大,访问延迟也可能略微增加。本实验是 6 路组相联。

- **块大小 (Line/Block Size)**: 较大的块可以更好地利用空间局部性，一次调入更多相邻数据，从而减少强制不命中 (Compulsory Miss) 的后续影响。但如果块太大而程序空间局部性差，则可能载入很多无用数据，浪费 Cache 空间和内存带宽，且 Miss 惩罚(从主存取块的时间)也更大。

这些参数的选择需要在命中率、成本、延迟和功耗之间进行权衡，在实验过程中，我是通过大量仿真和应用场景分析来确定最佳配置的。由于本设计中的所有数组都使用了 `reg` 型变量，似乎无需考虑延迟问题。因此，我尝试了调整 Cache 的路数，并评估路数对性能的影响。结果如下：

Listing 1: 四路组相联 I-Cache CoreMark 性能报告

```
test start
computation done
2K performance run parameters for coremark.
CoreMark Size  : 666
Total us : 1263908
```

Listing 2: 六路组相联 I-Cache CoreMark 性能报告

```
test start
computation done
2K performance run parameters for coremark.
CoreMark Size  : 666
Total us : 937097
```

通过对比 CoreMark 我们可以看出，改变路数带来的性能提升十分巨大。这也说明，抛开无法综合出电路的困难不谈，我的 D-Cache 还有巨大的性能提升潜力。

6. **为何要将 I-Cache 和 D-Cache 分开设计(哈佛 Cache 结构)？**

   - **提高带宽和并行性**: 分离的 I-Cache 和 D-Cache 允许 CPU 在同一个时钟周期内同时进行取指令和数据访存(如果流水线支持)，从而提高了处理器的指令级并行度和整体吞吐率。如果使用统一 Cache (冯·诺依曼 Cache 结构)，取指和数据访问需要竞争 Cache 端口，可能产生结构冒险。
   - **针对性优化**: 指令和数据的访问模式有差异。例如，指令通常是只读的，I-Cache 可以设计得更简单(如不需要 Dirty 位和写回逻辑)。而数据访问有读有写，D-Cache 需要处理更复杂的情况。分开设计允许针对各自特点进行优化。
   - **简化流水线控制**: 在流水线中，取指和数据访存通常在不同阶段，分离 Cache 可以使这些阶段的硬件设计更独立。
   - **缺点**: 可能会导致 Cache 整体利用率不如同样总大小的统一 Cache，因为可能出现 I-Cache 空闲而 D-Cache 满载(或反之)的情况。同时，需要两条独立的到内存的通路或一个更复杂的仲裁器。

5 实验所耗时间

在课后，我花费了大约 35 小时完成此次 I-Cache 和 D-Cache 的设计、Verilog 代码实现、仿真调试及实验报告的撰写工作。其中，D-Cache 的状态机设计和写回逻辑的调试占用了较多时间。

6 实验总结与心得体会

本次实验通过设计和实现分离的指令 Cache (I-Cache) 和数据 Cache (D-Cache)，我对 Cache 的工作原理、组成结构以及其在现代计算机系统中的重要性有了更为深刻和具体的理解。Cache 作为弥合 CPU 与主存速度差

异的关键部件,其设计直接影响到整个系统的性能。

在设计过程中,我首先明确了 Cache 的基本参数,如组数、相联度、行大小,并据此完成了地址到 Tag、Index、Offset 的映射逻辑。I-Cache 和 D-Cache 均采用了组相联映射,并通过参数化的 `custom_array` 模块构建了 Valid、Tag、Data 等存储阵列。I-Cache 实现了 LRU 替换策略,而 D-Cache(根据代码)使用了一个简化的替换策略,并额外管理了 Dirty 位以支持 Write-Back 写策略。

I-Cache 的设计相对直接,主要处理 CPU 的取指请求,发生不命中时从主存 Refill 数据。其 FSM 主要包含等待 CPU 请求、处理不命中、从内存填充等状态。LRU 时间戳的更新逻辑是保证替换策略有效性的关键。

D-Cache 的设计则复杂得多,因为它需要处理 CPU 的读写请求,并支持 Bypass 访问不可缓存区域(如 I/O 空间)。其 FSM 状态繁多,需要精确控制写命中(更新数据和 Dirty 位)、读命中、写不命中(可能涉及写回被替换的脏块,然后 Refill 新块,再执行写)、读不命中(Refill)以及 Bypass 读写等多种流程。写回(SYNC)过程中的数据暂存 (`sync_reg`) 和多周期传输计数 (`write_counts`) 是确保数据正确完整写回主存的保障。

调试是本次实验最具挑战性的环节。遇到的问题包括 Verilog 语法细节、LRU 更新时机、Dirty 位判断逻辑的偏差、CPU 响应数据未对齐以及 FSM 状态转换错误等。例如,D-Cache FSM 在处理 Bypass 读操作 (`RDW` 状态) 完成后,错误地跳转到为 Cache 命中设计的 `SEND_CPU_DATA` 状态,而不是返回 `WAIT_CPU`,导致程序卡死。这些问题的解决依赖于对 Cache 工作流程的细致理解、对代码逻辑的反复推敲以及对仿真波形图的耐心分析。每一次成功定位并修复 bug,都加深了我对 Cache 内部机制和时序控制的认识。

通过本次实验,我不仅掌握了 Cache 的基本设计方法,包括地址映射、数据存储、命中/缺失判断、替换策略、写策略等,还实践了复杂状态机的设计与调试。我体会到,Cache 的设计充满了各种权衡(如命中率与成本、性能与复杂度),模块化设计和清晰的接口定义对于管理这种复杂性至关重要。这次实验极大地锻炼了我的硬件设计和问题解决能力,为后续学习更高级的存储体系结构和计算机系统性能优化打下了坚实的基础。

7 致谢

感谢宁彦祯同学和宋俊仪同学,他们的真知灼见给了我很大的启发;感谢朱徐塬学长及各位助教,您们的帮助对我有非常大的指导意义。