南開大學
Nankai University

南　　开　　大　　学

计　算　机　学　院

实验报告

计算机体系结构第五次实验报告

姓名：禹相祐

学号 :2312900

专业：计算机科学与技术

一、 实验目标与完成情况（TLB + Cache）

本次综合实验要求在**五级流水线 CPU**（IF/ID/EX/MEM/WB）基础上，新增 **TLB** 与 **Cache**，重点考察对二者工作方式与在流水线中“如何接入”的理解与实现能力。结合实验提示（不强制做到完全功能完备），本实现侧重于**可跑通、可验证、易调试**的版本。

(一) 完成范围说明

- **TLB 部分：**完成 TLB 表项结构、查找路径、无效化；完成 TLB 指令（TLBP/TLBR/TLBWI/TLBWR）的提交与 CP0 寄存器协同（Index/Random/EntryHi/EntryLo0/EntryLo1/PageMask）。
- **Cache 部分：** 完成 ICache 与 DCache；实现 blocking cache 的 miss 处理（busy + 请求/响应 + fill）；在 CPU 中接入 stall 控制，保证 miss 时流水线暂停、fill 后继续执行；DCache 写策略采用 write-through + no-write-allocate（便于调通与验证）。

二、 TLB 实现

(一) TLB 模块结构与接口信号

TLB 采用**定长表项数组**实现，表项内容包含：VPN、PPN、ASID、Flags、Valid。TLB 对外接口建议至少包含两类：**查找（lookup）接口**与**维护（write/invalidate）接口**。

查找接口（组合逻辑）

- `lookup_en`：查找使能
- `lookup_vpn`：待查虚拟页号（由虚拟地址高位得到）
- `lookup_asid`：当前 ASID（通常来自 CP0.EntryHi 的 ASID）
- `hit`：是否命中
- `hit_ppn`：命中时返回物理页号
- `hit_flags`：命中时返回页属性/权限

维护接口（时序逻辑）

- `op_tlbwi/op_tlbwr/op_tlbr/op_tlbp`：TLB 指令提交脉冲（WB 阶段产生）
- `w_index`：写入/读出索引（来自 CP0.Index 或 CP0.Random）
- `w_entryhi/w_entrylo0/w_entrylo1/w_pagemask`：写入表项的数据源（来自 CP0）
- `inv_all`：全局无效化
- `inv_match_en + inv_vpn + inv_asid`：精确无效化（VPN+ASID）

(二) TLB 查找流程

TLB 查找为组合逻辑：当 `lookup_en=1`，对所有有效表项做并行匹配：

$$\texttt{match}[i] = \texttt{valid}[i] \wedge (\texttt{vpn}[i] = \texttt{lookup\_vpn}) \wedge (\texttt{asid}[i] = \texttt{lookup\_asid})$$

若存在任意 `match[i]=1`，则命中并输出对应 `ppn/flags`，否则 miss。

1. TLB 组合查找示例代码

```
// Example: combinational lookup in TLB (pseudo-realistic)
integer i;
always @(*) begin
    hit       = 1'b0;
    hit_ppn   = {PPN_W{1'b0}};
    hit_flags = {FLAG_W{1'b0}};
    hit_idx   = {IDX_W{1'b0}}; // optional: return index for debug/probe

    if (lookup_en) begin
        for (i = 0; i < TLB_N; i = i + 1) begin
            if (!hit && tlb_valid[i] &&
                (tlb_vpn[i]  == lookup_vpn) &&
                (tlb_asid[i] == lookup_asid)) begin
                hit       = 1'b1;
                hit_ppn   = tlb_ppn[i];
                hit_flags = tlb_flags[i];
                hit_idx   = i[IDX_W-1:0];
            end
        end
    end
end
```

代码详细解释

- **先把输出全部置默认值**：避免组合逻辑推导出锁存器（latch），保证无论命中与否输出都是确定的。
- **用 `if (lookup_en)` 包住循环**：当流水线不需要查找时，不触发匹配，有助于减少无关切换与波形干扰。
- **用 `!hit` 保证“第一命中优先”**：当存在多个相同 VPN+ASID（正常情况下不应出现，但调试阶段可能出现），该写法保证输出稳定且可复现。
- **返回 `hit_idx`**：对调试很有用，便于在波形中定位到底命中了哪一项。

(三) CP0 中的 TLB 相关寄存器与写入优先级

CP0 提供 TLB 指令所需寄存器视图：

- **Index**：`[31]=P` 表示 probe miss，低位为表项索引；
- **Random**：TLBWR 使用的随机索引；
- **EntryHi**：VPN 与 ASID（probe 键）；
- **EntryLo0/EntryLo1**：PFN 与 flags；
- **PageMask**：本实现固定 4KB，恒为 0（简化）。

为避免同拍被覆盖，CP0 寄存器写入优先级：

TLBR > TLBP > MTC0

含义是：若某拍 WB 阶段提交的是 TLBR，则必须把“从 TLB 读出的表项”写回 CP0；若是 TLBP，则必须把 probe 结果写入 Index；最后才允许普通的 MTC0 写 CP0。

1. TLB 相关 CP0 寄存器写入

```
// ——— TLB CP0 register write (MTC0 and TLB instructions)
always @(posedge clk) begin
    if (!resetn) begin
        cp0_index    <= {1'b1, 31'd0};  // P=1 means "miss" by default
        cp0_random   <= 32'd0;
        cp0_entryhi  <= 32'd0;
        cp0_entrylo0 <= 32'd0;
        cp0_entrylo1 <= 32'd0;
        cp0_pagemask <= 32'd0;
    end
    // TLBR has highest priority: update EntryHi/EntryLo/PageMask from TLB read
        port
    else if (op_tlbr) begin
        cp0_entryhi  <= tlbr_entryhi_in;
        cp0_entrylo0 <= tlbr_entrylo0_in;
        cp0_entrylo1 <= tlbr_entrylo1_in;
        cp0_pagemask <= tlbr_pagemask_in;
    end
    // TLBP writes probe result into Index (P bit + index)
    else if (op_tlbp) begin
        cp0_index <= tlbp_index_in;
    end
    // Normal CP0 write by MTC0 (lowest priority)
    else if (mtc0_wen) begin
        case (cp0r_addr) // {rd, sel}
            {5'd0 ,3'd0}: cp0_index    <= mtc0_wdata;  // Index
            {5'd1 ,3'd0}: cp0_random   <= mtc0_wdata;  // Random
            {5'd10,3'd0}: cp0_entryhi  <= mtc0_wdata;  // EntryHi
            {5'd2 ,3'd0}: cp0_entrylo0 <= mtc0_wdata;  // EntryLo0
            {5'd3 ,3'd0}: cp0_entrylo1 <= mtc0_wdata;  // EntryLo1
            {5'd5 ,3'd0}: cp0_pagemask <= mtc0_wdata;  // PageMask
            default: ;
        endcase
    end
end
```

代码详细解释

- `op_tlbr/op_tlbp` **为什么不直接用** `tlbr/tlbp?` `op_tlbr/op_tlbp` 通常是“**WB 阶段有效提交**”的脉冲，已经包含 `WB_valid` 约束。这样 CP0 写口只在“真正提交”的那一拍动一次，避免被冲刷的指令产生副作用。

- **TLBR 为什么只写 EntryHi/EntryLo/PageMask，不写 Index?** TLBR 的体系结构语义是“读表项到 Entry 寄存器组”，它不应该改变 Index；Index 的改变应由软件显式写入或由 probe 指令产生。
- **TLBP 为什么只写 Index?** probe 的结果就是“是否命中 + 命中项 index”。写其他寄存器反而会破坏软件设置的查找键（EntryHi）。
- **MTC0 放最后的核心意义**：如果同一拍既有 TLB 指令结果写回，又有 MTC0 写寄存器，必须保证 TLB 指令的语义不被普通写覆盖。

2. TLB 指令脉冲生成

```
// Pulses are generated only when WB stage is valid
assign op_tlbp  = tlbp  & WB_valid;
assign op_tlbr  = tlbr  & WB_valid;
assign op_tlbwi = tlbwi & WB_valid;
assign op_tlbwr = tlbwr & WB_valid;
```

解释

- **为什么是“与 WB_valid 相与”而不是与 MEM_valid?** 因为真正的体系结构提交点在 WB：此时指令不会再被后续异常/冲刷撤销。把副作用放在 WB，可以显著减少“看起来偶发、实际是投机写入”的 bug。
- **为什么必须是脉冲?** TLBWI/TLBWR 是一次性动作：写一次表项即可。如果保持电平，可能连续多个周期重复写入（表项会被反复覆盖，波形也难读）。

(四) TLB 写表项与无效化

1. TLBWI/TLBWR 写表项示例代码

```
always @(posedge clk) begin
    if (!resetn) begin
        for (k = 0; k < TLB_N; k = k + 1) begin
            tlb_valid[k] <= 1'b0;
            tlb_vpn[k]   <= {VPN_W{1'b0}};
            tlb_asid[k]  <= {ASID_W{1'b0}};
            tlb_ppn[k]   <= {PPN_W{1'b0}};
            tlb_flags[k] <= {FLAG_W{1'b0}};
        end
    end else begin
        // global invalidate
        if (inv_all) begin
            for (k = 0; k < TLB_N; k = k + 1)
                tlb_valid[k] <= 1'b0;
        end

        // precise invalidate by VPN+ASID
        if (inv_match_en) begin
```

```
            for (k = 0; k < TLB_N; k = k + 1) begin
                if (tlb_valid[k] &&
                    tlb_vpn[k]  == inv_vpn &&
                    tlb_asid[k] == inv_asid) begin
                    tlb_valid[k] <= 1'b0;
                end
            end
        end

        // TLBWI: write at CP0.Index
        if (op_tlbwi) begin
            tlb_vpn[w_index]   <= w_entryhi_vpn;
            tlb_asid[w_index]  <= w_entryhi_asid;
            tlb_ppn[w_index]   <= w_entrylo0_ppn;    // simplified mapping
            tlb_flags[w_index] <= w_entrylo0_flags;
            tlb_valid[w_index] <= 1'b1;
        end

        // TLBWR: write at CP0.Random
        if (op_tlbwr) begin
            tlb_vpn[w_random]   <= w_entryhi_vpn;
            tlb_asid[w_random]  <= w_entryhi_asid;
            tlb_ppn[w_random]   <= w_entrylo0_ppn;
            tlb_flags[w_random] <= w_entrylo0_flags;
            tlb_valid[w_random] <= 1'b1;
        end
    end
end
```

代码详细解释

- **无效化优先级问题：** 如果同一拍既无效化又写入，建议明确优先级（例如先 invalidate 再 write），否则会出现“写入后马上被清掉”或“应该清掉但又被写回”的波形困惑。上面示例是先做 invalidate，再做写入。
- **精确无效化为什么要遍历整表？** 因为 TLB 是内容寻址（CAM-like）匹配：按 VPN+ASID 找到哪项无效，最直接的方式就是遍历比较。
- **为什么写入时要同时写 VPN/ASID/PPN/Flags/Valid？** 一次写入应该让表项“从无到有”变成可用状态：只写部分字段会造成短暂不一致（例如 valid=1 但 tag 还没更新）。

（五） 功能验证与波形截图插入位置

下面给出 TLBWI / TLBR / TLBP 的验证说明，并展示仿真截图。

1. TLBWI：写入行为验证

TLBWI 的验证要点是：在 WB 提交 TLBWI 的那一拍，TLB 写口拿到 CP0.EntryHi/EntryLo 的输入；下一拍（或同拍，取决于实现）对应索引表项的 `vpn/asid/ppn/flags/valid` 发生更新。

- 观察 `op_tlbwi` 拉高的那一拍:`w_index` 是否等于 CP0.Index;`w_entryhi_vpn/asid`、`w_entrylo_ppn/flags` 是否为软件写入值;
- 观察写入后：`tlb_valid[w_index]` 是否置 1，且对应 tag/data 与输入一致。

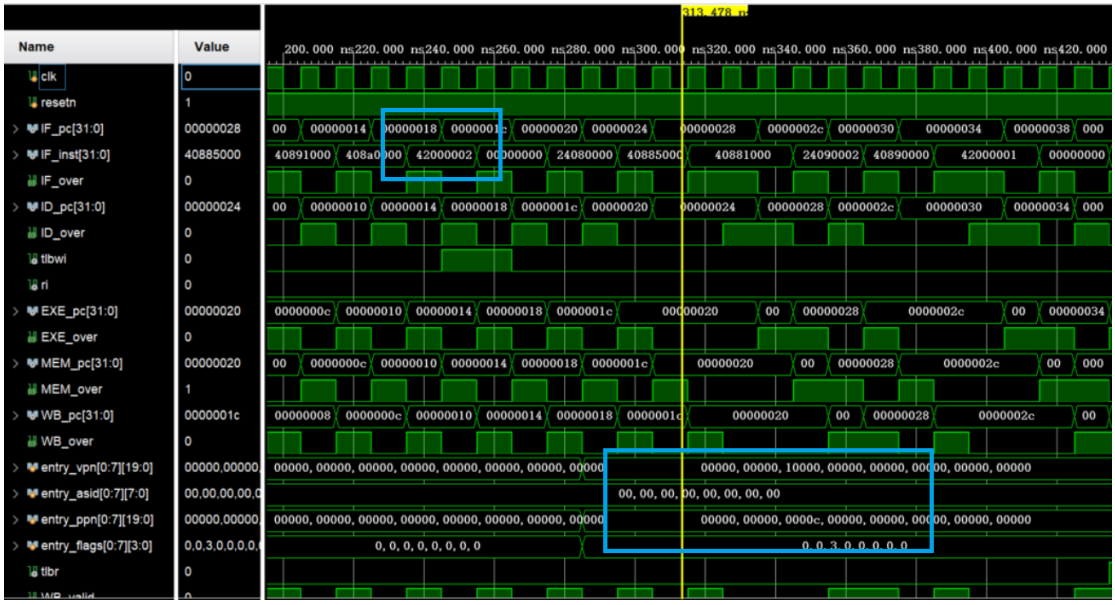

图 1: TLBWI 写入波形验证（写入 TLB[2]）

2. TLBR：读回行为验证

TLBR 的验证建议分两步：先把 CP0 的 EntryHi/EntryLo 清空（方便观察 TLBR 是否真正把 TLB 项读回），再执行 TLBR 看寄存器是否恢复。

- **TLBR 前：**软件执行 MTC0 将 `EntryHi/EntryLo0/EntryLo1` 清零;
- **TLBR 提交：**WB 阶段 `op_tlbr=1`，TLB 读口输出 `tlbr_entryhi_in/...`;
- **TLBR 后：**下一拍 CP0.EntryHi/EntryLo 被写回为 TLB[Index] 中的真实内容。

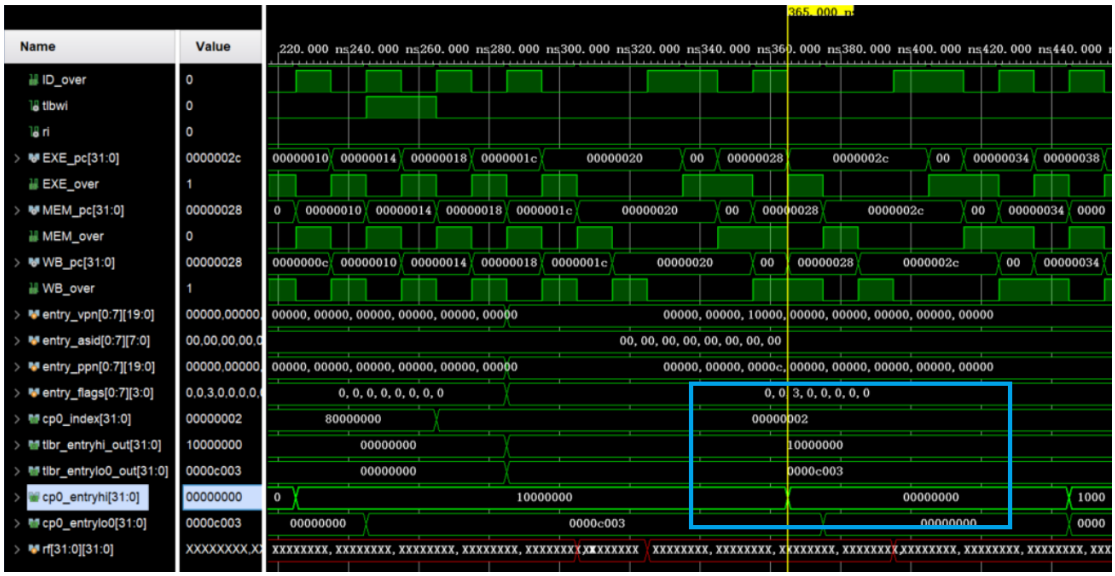

图 2: TLBR 执行前：CP0.EntryHi/EntryLo 清空波形

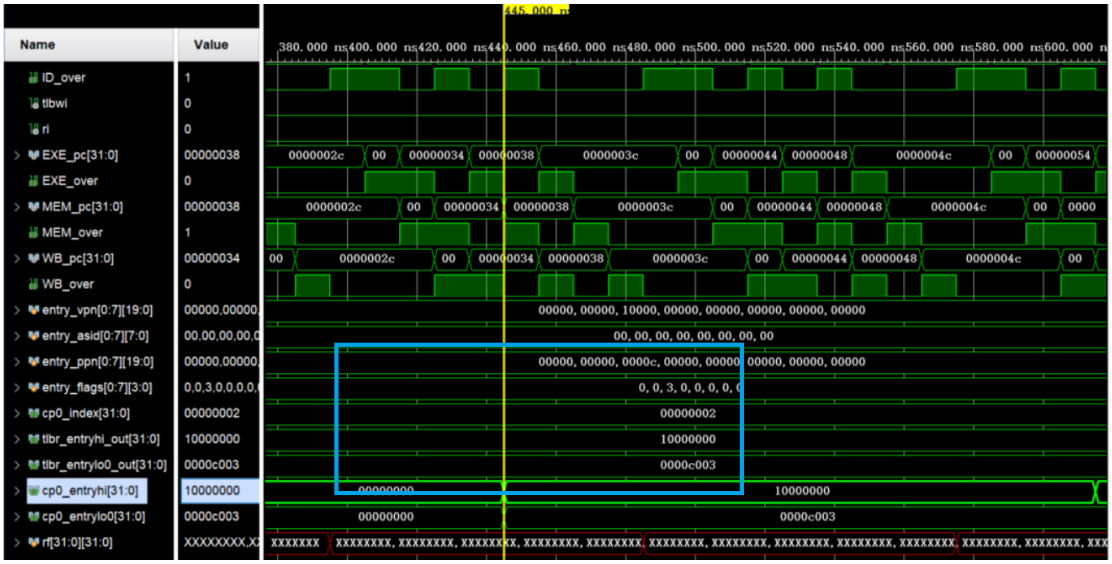

图 3: TLBR 执行后：CP0.EntryHi/EntryLo 恢复为 TLB[Index] 表项内容

3. TLBP：命中/未命中验证

TLBP 的验证要点是：以 `CP0.EntryHi` 中的 `VPN+ASID` 为查找键，对 TLB 做 probe，并把结果写入 `CP0.Index`：

- 命中时：`CP0.Index[31]=0`，并给出命中项索引；
- 未命中时：`CP0.Index[31]=1`（P 位为 1），索引字段可为 0 或保持不关心值（按实现约定）。

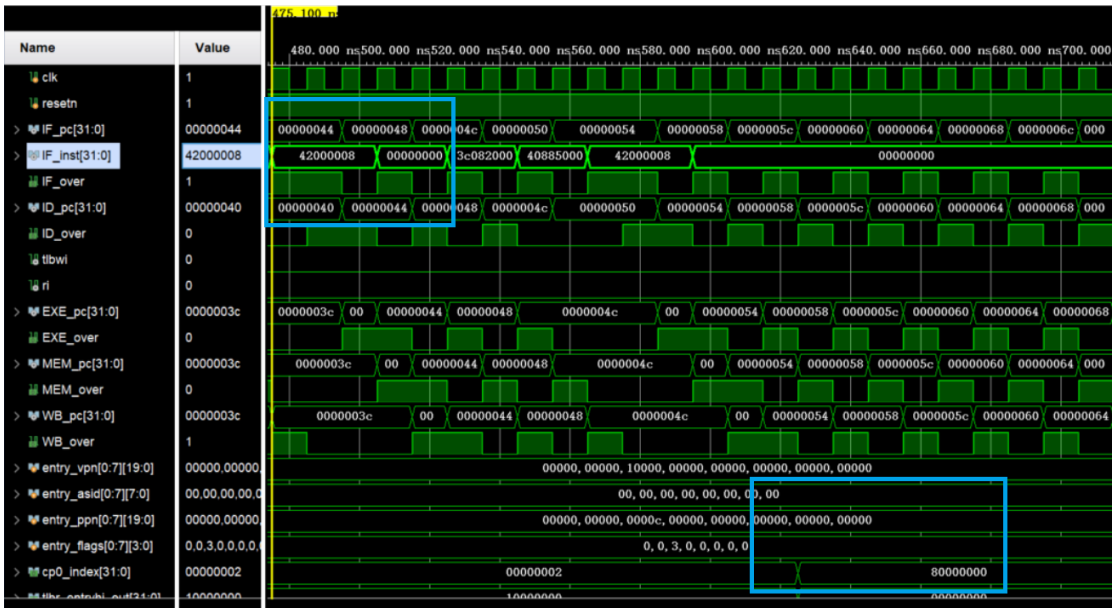

图 4: TLBP 命中与未命中波形验证

三、 Cache 实现

(一) 总体设计与参数选择

Cache 部分至少需要 ICache 与 DCache。本实现采取：

- **Split I/D Cache**：取指与数据访问分离，降低端口冲突与控制复杂度；
- **Line size = 16B**：即 4 个 32-bit word，一次 fill 128-bit；
- **2-way set associative**：每组两路，命中比较简单，冲突率明显优于直映；
- **替换策略：1-bit LRU**：每个 set 记录“最近使用的 way”，miss 时替换另一路；
- **Blocking cache**：miss 时 busy=1，冻结流水线，fill 完成后继续。

(二) 地址拆分（set/tag/offset）

以 32-bit 地址为例，设行大小为 16B，则块内偏移 offset 为 4 bit（最低 4 位）；其中 word offset 为 2 bit（选择 4 个 word），byte offset 为 2 bit（选择 word 内字节）。其余高位再按 cache 容量决定 set 与 tag 的位宽（工程中固定参数）。

(三) ICache：命中/未命中 + FSM + LRU

1. 命中比较与数据选择

```
// tag compare (2-way)
assign way0_hit = valid[0][set] && (tag_arr[0][set] == req_tag);
assign way1_hit = valid[1][set] && (tag_arr[1][set] == req_tag);
assign hit      = way0_hit | way1_hit;

// data mux
always @(*) begin
    if (way0_hit)      cpu_rdata = data_arr[0][set][word];
    else if (way1_hit) cpu_rdata = data_arr[1][set][word];
    else               cpu_rdata = 32'd0;
end
```

解释

- `valid && (tag==)` 是最基本的命中判定；valid=0 的行即使 tag 恰好相同也不能算命中。
- 两路命中理论上不应同时为 1；若出现，多半是写入/无效化控制问题，波形可据此定位 bug。
- `cpu_rdata` 的选择必须与 `word` 对齐，避免“命中了但取错行内字”的错误。

2. 替换路选择（空行优先 + LRU）

```
// victim select: prefer invalid line, else use LRU
always @(*) begin
    if (!valid[0][set])      victim = 1'b0;
    else if (!valid[1][set]) victim = 1'b1;
    else                     victim = ~lru[set]; // lru=1 means way1 recently used
end
```

解释

- **空行优先**：避免把仍有用的数据替换掉，且初始化阶段更稳定。
- **LRU 位含义要写清楚**：例如约定 `lru[set]=1` 表示 way1 最近用过，那么替换就选 `lru`。
- LRU 更新要在“命中”与“fill”两处都做，否则会出现替换不符合预期。

3. 两态 FSM 与 busy 控制

```
localparam IDLE = 1'b0;
localparam MISS = 1'b1;

always @(posedge clk) begin
    if (!resetn) state <= IDLE;
    else         state <= state_n;
end

always @(*) begin
    state_n   = state;
    mem_req   = 1'b0;
    cache_busy= 1'b0;

    case (state)
    IDLE: begin
        if (cpu_req && !hit) begin
            state_n    = MISS;
            mem_req    = 1'b1;
            cache_busy = 1'b1;
        end
    end
    MISS: begin
        cache_busy = 1'b1;
        mem_req    = ~mem_ready; // keep requesting until ready
        if (mem_ready) state_n = IDLE;
    end
    endcase
end
```

解释

- **IDLE 命中不拉 busy**：命中应走最快路径，不阻塞流水线。
- **MISS 必须一直 busy**：miss 持续多个周期时，必须冻结前端，防止 CPU 地址变化导致填充写错 set/tag。
- **mem_req 的保持方式**：用 `mem_ready` 是最容易读懂的写法：ready 到来后自然撤销请求。

（四） DCache：读 miss 阻塞、写直达 + no-write-allocate

1. 写路径（命中更新 + 主存写直达）

```
// store path (write-through)
if (cpu_req && cpu_wen) begin
    mem_req   = 1'b1;
    mem_wen   = 1'b1;
    mem_addr  = cpu_addr;
    mem_wdata = cpu_wdata;

    if (hit) begin
        data_arr[hit_way][set][word] = cpu_wdata; // update cache line word
        // LRU update is also needed here
    end
    // miss: no-write-allocate -> do NOT fill cache
end
```

解释

- **写直达**保证主存始终最新，因此可以弱化 dirty 管理（或维持为 0）。
- **no-write-allocate** 的关键是：写 miss 不进入 MISS 状态机、不请求整行、不拉 busy，只把写直接送主存。

2. 读 miss：锁存地址 + busy 冻结 + fill 恢复

```
if (cpu_req && !cpu_wen && !hit && !miss_active) begin
    miss_active <= 1'b1;
    miss_addr   <= cpu_addr; // MUST latch
    cache_busy  <= 1'b1;

    mem_req  <= 1'b1;
    mem_wen  <= 1'b0;
    mem_addr <= {cpu_addr[31:OFFSET_W], {OFFSET_W{1'b0}}}; // line aligned
end

if (mem_ready && miss_active) begin
    // fill line
    tag_arr[victim][miss_set]    <= miss_tag;
    valid[victim][miss_set]      <= 1'b1;
    data_arr[victim][miss_set][0] <= mem_rdata[31:0];
    data_arr[victim][miss_set][1] <= mem_rdata[63:32];
    data_arr[victim][miss_set][2] <= mem_rdata[95:64];
    data_arr[victim][miss_set][3] <= mem_rdata[127:96];

    lru[miss_set] <= victim; // mark as recently used
    miss_active   <= 1'b0;
    cache_busy    <= 1'b0;
end
```

解释

- **`miss_addr` 必须锁存**：因为 miss 期间流水线被阻塞并不意味着所有组合信号都不变，锁存能确保 fill 对应到正确行。
- **busy 冻结粒度**：DCache busy 一般要冻结 MEM 以及整个流水线（通过 allow_in 反压），否则 load 指令可能被后续指令覆盖。
- **fill 后 LRU 更新**：刚填充的路应视为最近使用，否则下一次冲突可能立即把刚填的行替换掉，影响命中率与可解释性。

（五） Cache 与流水线 stall 集成

1. IF：ICache busy 时 PC 暂停

```
always @(posedge clk) begin
    if (!resetn) pc <= STARTADDR;
    else if (next_fetch && !icache_busy)
        pc <= next_pc;
end
```

2. MEM：DCache busy 时压低 MEM_over

```
assign MEM_over = dcache_busy ? 1'b0 :
                  (inst_load ? MEM_valid_r : MEM_valid);
```

3. allow_in：把 busy 融入反压链条

```
assign IF_allow_in  = ((IF_over & ID_allow_in) | cancel) & ~icache_busy;
assign MEM_allow_in = (~MEM_valid | (MEM_over & WB_allow_in)) & ~dcache_busy;
```

解释

- **ICache busy 直接阻止 PC 推进**：保证 miss 期间取指地址稳定，fill 完成后从同一 PC 继续。
- **DCache busy 通过 MEM_over 反压全流水线**：load miss 必须暂停，直到数据被 fill 并可正确写回。

（六） Memory Adapter（简化为 1-cycle ready，便于验证）

```
// ICache adapter: request -> next cycle ready (line data)
always @(posedge clk) begin
    if (!resetn) begin
        ic_mem_ready <= 1'b0;
        ic_mem_rdata <= 128'd0;
    end else begin
        ic_mem_ready <= ic_mem_req;
        ic_mem_rdata <= {4{inst_rom_word}}; // pack to 16B line
    end
```

```
end

// DCache adapter: only for read miss
always @(posedge clk) begin
    if (!resetn) begin
        dc_mem_ready <= 1'b0;
        dc_mem_rdata <= 128'd0;
    end else begin
        dc_mem_ready <= dc_mem_req & ~dc_mem_wen;
        dc_mem_rdata <= {4{data_ram_dout}};
    end
end
```

解释

- 真实存储系统的延迟与握手更复杂；adapter 的目的就是把验证重点聚焦在“cache 的控制逻辑是否正确”上。
- 采用固定 1-cycle ready，能够稳定复现 miss 的全过程：`req` → `busy` → `ready` → `fill` → `unbusy`。

四、 实验现象、问题与收获

(一) 验证时最关键的观察点

为了让验证结论更可信，本次实验的波形检查不是只看“结果对不对”，而是尽量在波形中建立“**指令出现** → **控制信号提交** → **状态更新生效**”的完整分析。因此验证时重点观察点如下。

- **TLB：**
  - **指令识别是否正确：**在 `IF_inst` 或 `ID/EXE` 相关信号中确认四条 TLB 指令码确实被取到并进入流水线（例如 `TLBWI=42000002`、`TLBP=42000008` 等），避免出现“以为执行了但其实没有进入流水线”的误判。
  - **提交点是否正确（WB_valid 约束）：**重点检查 `op_tlbwi/op_tlbr/op_tlbp/op_tlbwr` 是否只在 `WB_valid=1` 时出现单周期脉冲，保证被异常/冲刷取消的指令不会产生 CP0/TLB 的副作用。
  - **CP0 写入优先级是否符合语义：**

    $$\texttt{TLBR} > \texttt{TLBP} > \texttt{MTC0}$$

    在波形中表现为:同一拍若存在 `op_tlbr`,则 `cp0_entryhi/entrylo` 必须更新为 `tlbr_entry*_out`；若存在 `op_tlbp`，则 `cp0_index` 必须更新 probe 结果；普通 `MTC0` 只能在无 TLB 指令写回时生效。
  - **功能验证的四个关键现象：**
    - **TLBWI：**写入后 `entry_vpn/entry_ppn/entry_flags` 中对应表项由 0 变为目标值，并且 `valid` 置 1；
    - **TLBR 前：**人为清空 `cp0_entryhi/entrylo`，证明“此时 CP0 视图是空的”；
    - **TLBR 后：**`cp0_entryhi/entrylo` 恢复，并与 `tlbr_entry*_out` 一致；

  - **TLBP：** `cp0_index` 的 P 位（`Index[31]`）能正确反映命中/未命中，且命中时 index 字段对应到正确表项。

- **ICache：**
  - **命中路径是否保持单拍返回：** 命中时 `cpu_hit` 拉高且 `cpu_data` 正确，`icache_busy=0`，PC 连续递增不出现停顿。
  - **miss 时 PC 是否真正停止：** 当 `cpu_req=1` 且 `hit=0`, `icache_busy` 必须及时拉高, 随后 `IF_pc` 在 busy 期间保持不变，避免出现“miss 期间 PC 仍推进导致取指地址飘移”的错误。
  - **fill 的行是否写对 set/way/word：** 观察 `mem_req/mem_ready` 握手后，tag/valid 与 data 阵列写入是否同步发生；填充完成后紧接着同一条 PC 能命中并返回正确指令。
  - **LRU 更新是否合理：** 命中与填充两处都应更新 LRU, 否则容易出现“刚填进去下一次就被替换”的现象。
- **DCache：**
  - **load miss 是否冻结全流水线：** 当 MEM 阶段访问为读且未命中时，`dcache_busy` 拉高并迫使 `MEM_over=0`，进而通过 allow_in 反压冻结 EX/ID/IF，直到 `mem_ready` 返回完成 fill。
  - **fill 后 load 的返回数据是否对齐：** miss 对应地址必须锁存（`miss_addr`），fill 完成后 load 返回值应来自正确的 word offset，避免“填充对了但取字错位”的问题。
  - **store 策略是否符合 write-through + no-write-allocate：** 写命中既更新 cache 行也写主存；写未命中不进入 miss 流程、不拉 busy、直接写主存，流水线继续执行。
  - **忙信号不应误触发：** 写 miss 在 no-write-allocate 下不应造成全流水线暂停；若出现不必要的 busy，说明写路径与 miss 状态机耦合过深或条件判断不严谨。

(二) 调试过程中的典型问题与解决

本次实验属于综合性较强的探索实验，调试时间主要花在“控制时序”和“副作用提交点”上。实际过程中遇到的问题不止一次出现“现象像随机错误”的情况，但通过逐步缩小范围、对齐波形关键拍，最终都能定位到具体原因。

- **问题 1：miss 未锁存地址导致 fill 写错行 现象：** 程序偶尔能跑通，但运行一段时间后取指或读数据出现明显错误；波形上表现为 miss 发生时 PC/访存地址在 busy 期间发生变化，最终 fill 写入的 set/tag 与最初 miss 的地址不一致。**定位方法：** 在波形中对齐 `cpu_addr`、`miss_addr`、`mem_addr`，重点观察 miss 当拍到 `mem_ready` 到来的整个时间窗，确认 fill 使用的是哪个地址。**解决：** 在进入 MISS 的第一拍锁存 `miss_addr`，后续 set/tag/word 全部从 `miss_addr` 派生；同时确保 busy 期间上层地址即使变化也不会影响本次 fill。
- **问题 2：busy 未正确接入 allow_in, 流水线在 miss 期间仍推进 现象：** cache miss 时看似 busy 拉高, 但流水线某些级仍在走, 导致 WB 写回错位、指令乱序的假象；严重时出现“load 的数据写回到另一条指令”的情况。**定位方法：** 同时观察 `IF_allow_in/MEM_allow_in` 与 `IF_pc/MEM_pc/WB_pc`, 如果 busy=1 时 PC 仍递增或各级 PC 仍持续推进, 说明反压链没有真正生效。**解决：** 将 `icache_busy`、`dcache_busy` 显式并入 allow_in 逻辑，并在 MEM 阶段用 `MEM_over=0` 强制阻塞后级，保证 miss 期间全流水线冻结。
- **问题 3：TLB 指令副作用未绑定 WB 提交点 现象：** 在出现异常/冲刷或分支取消时，偶尔仍能看到 CP0/TLB 被改写, 随后 probe 或 tlbr 行为异常, 且难以复现。**定位方法：** 检查 `tlbwi/tlbr/tlbp` 控制信号是否来自译码级直接输出；对齐 `WB_valid` 观察在 `WB_valid=0` 时是否仍发生写入。**解决：**

所有 TLB 指令均改为由 `op_tlb* = tlb* & WB_valid` 产生单周期脉冲驱动，确保只有“最终提交”的指令才会更新 CP0/TLB。

- **问题 4：CP0 写入优先级不当导致 TLBR/TLBP 结果被 MTC0 覆盖 现象：** TLBR 明明读口输出正确，但 CP0.EntryHi/EntryLo 没有恢复，或者 probe 后 Index 的 P 位不稳定。**定位方法：** 检查同一拍是否既发生了 `mtc0` 写入，又发生 `op_tlbr/op_tlbp`；若是，则需要确认优先级是否满足语义。**解决：** 实现 `TLBR>TLBP>MTC0` 的顺序，保证指令结果不会被普通写覆盖。

- **问题 5：LRU/valid 更新时序不一致导致命中率异常或出现假命中 现象：** 刚 fill 的行下一次冲突访问立即被替换；或者 valid 先置 1 而 data 尚未写全导致短暂假命中。**定位方法：** 对齐 fill 的写入拍，观察 tag/valid/data/LRU 的更新是否在同一拍完成；检查命中时是否也更新 LRU。**解决：** 确保 fill 时 tag+data+valid 同拍更新，并将“fill 的 way”标记为最近使用；命中路径同样更新 LRU。

(三) 收获

通过本次综合实验，将“地址转换（TLB）”与“缓存访问（Cache）”真正落地到五级流水线的控制逻辑中，最大的收获不只是实现了功能，更重要的是对复杂硬件模块的调试方法与工程约束有了更清晰的理解。

- **对提交点与副作用的理解更扎实：**以前对“WB 才是提交点”更多停留在概念层面,本次在 TLB/CP0 这种强副作用模块上真实踩坑后，明确了：**只要副作用没有绑定 WB_valid，就一定会在异常/冲刷下引入难复现错误**。

- **掌握了 blocking cache 的完整闭环：**不仅理解了 busy 的含义，还把它真正贯穿到：`miss` 检测 → `mem_req` → `mem_ready` → `fill` → 清 `busy` → 流水线恢复。这使得对 cache 与流水线交互的认识从“结构图”变成了“可用的控制逻辑”。

- **形成了波形驱动的调试方法：**遇到“偶发错误”时不再盲目改代码，而是先确定**关键观察点**（地址是否锁存、busy 是否冻结、提交脉冲是否正确、写回是否覆盖），通过对齐关键拍逐步缩小范围，最终能把问题定位到明确的条件或时序更新点。

- **对模块边界与简化策略有了更现实的把握：**在综合实验时间有限的情况下，选择 write-through + no-write-allocate、PageMask 固定 4KB、adapter 固定 1-cycle ready 等策略，可以显著降低复杂度，把精力集中在“核心机制正确”上；这也让实现更容易验证、结论更可信。