TLB Cache AXI 总线的五级流水 CPU

姓名：梁朝阳 2311561　**专业**：密码科学与技术

目录

摘要

本文在已有的 AXI 总线的五级流水 CPU 基础上，完成 TLB 与 I/D Cache 的最简实现与集成。TLB 采用 4 项全相联固定映射，支持 TLBL/TLBS/Modify 等异常；I-Cache 与 D-Cache 均为直映结构、行大小 16B，其中 I-Cache 只读，D-Cache 采用写直达 + 写分配并支持字节访问。通过在 MEM 阶段生成访存异常并在 WB 阶段统一仲裁写入 CP0，实现了地址异常与 TLB 异常的正确处理。最后仿真验证模块正确。

最后报告中没有展示完整的代码，所有的代码让我一起上传到 github 中了：`https://github.com/Zhaoyang-Liang/pipline_cpu_with_Cache-TLB` 里面包含了所有内容。

关键词：五级流水 CPU 设计图、TLB、I/DCache、github 代码

1 CPU 全局设计图

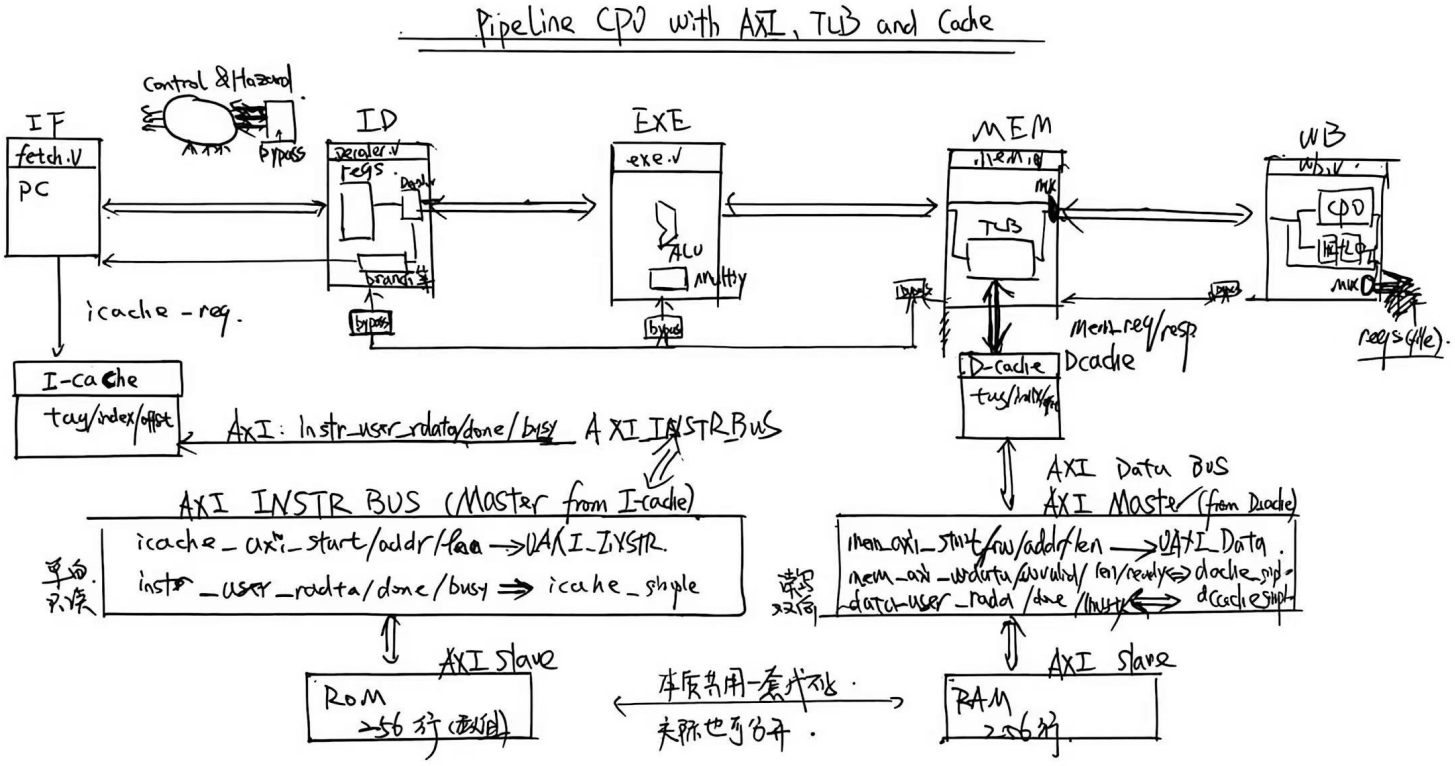


图 1.1: CPU 全局设计图

该图从左到右展示了五级流水主路径及关键访存链路，分点说明如下：

1. IF 阶段由 PC 选择逻辑产生取指地址，请求 I-Cache，命中则直接返回指令；未命中则通过指令 AXI 总线向指令存储器（ROM/AXI 从设备）发起 burst 读并回填 I-Cache。

2. ID 阶段完成指令译码、寄存器读取和分支判断，产生控制信号并通过旁路单元接收来自 EXE/MEM/WB 的前递数据以消除数据相关。

3. EXE 阶段执行 ALU/乘法运算并形成访存地址。

4. MEM 阶段首先对地址进行对齐检查，再经 TLB 完成虚拟地址到物理地址的转换，随后访问 D-Cache：命中直接返回数据或更新 cache 行；未命中则通过数据 AXI 总线访问外部数据存储器（RAM/AXI 从设备）并进行行填充。

5. WB 阶段将结果写回寄存器堆，同时处理 HI/LO 与 CP0 寄存器更新并进行异常仲裁。

6. 整体上，I-Cache 与 D-Cache 分别通过独立的 AXI 主接口访问指令/数据存储器，形成取指与访存的双通道结构。

2 实验原理与设计

2.1 TLB 模块

2.1.1 TLB 参数设计

设计采用全相联 TLB，表项数为 4。页大小固定为 4KB（**VPN=addr[31:12]，offset=addr[11:0]**），每条表项包含 VPN、PPN、valid、dirty 位。采用固定映射（PPN=VPN）并在复位/首次时钟初始化 valid/dirty 为 1，保证基本访问稳定可用。异常编码支持 TLBL/TLBS/Modify，优先级为 miss/invalid 优先于 modify。下图2.2主要是展示了一个地址转换的过程，左侧的 TLB 比较简单，就没有单独再绘制一个 TLB 的结构图了。（但是下面画了一个 Cache 的结构图）。

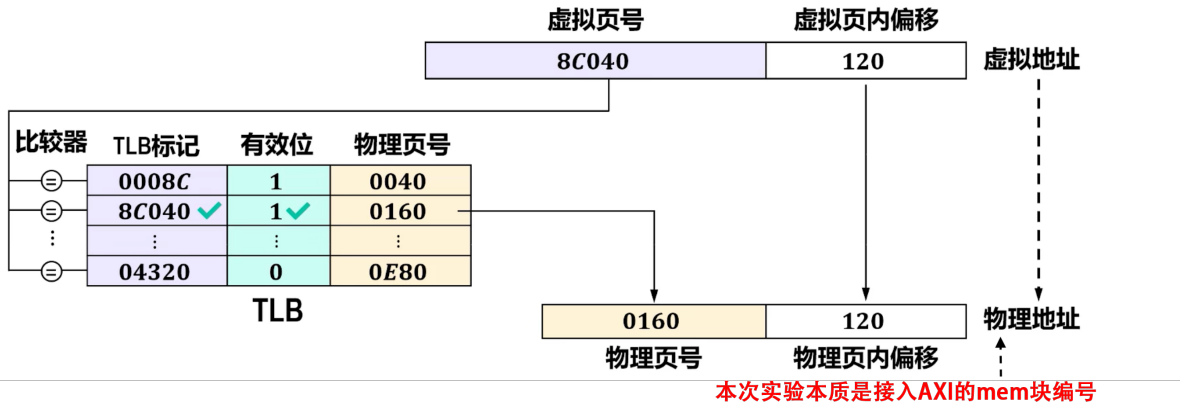


图 2.2: TLB 地址转换 (设计图)

TLB（Translation Lookaside Buffer）用于加速虚拟地址到物理地址的转换。流水线访存时先用虚拟页号（VPN）在 TLB 中查找，如果命中则得到物理页号（PPN）并拼接页内偏移形成物理地址；若未命中或表项无效则触发 TLB 相关异常。

2.1.2 代码部分：TLB 接口与关键信号

TLB 作为 MEM 阶段前置地址转换单元，接口只保留最小必要信号，输入为访存请求与虚拟地址，输出为物理地址与异常信息：

```
module tlb_simple(
    input           clk, input          resetn,
    input           req_valid,
    input     [31:0] vaddr,
    input           is_store,
    output reg      hit,
    output reg [31:0] paddr,
    output reg      exc_valid,
    output reg [ 4:0] exc_code,
    output reg      badvaddr_valid,
    output reg [31:0] badvaddr
);
```

其中，**req_valid** 在 MEM 阶段且为 load/store 时拉高，**vaddr** 为 EXE 计算出的虚拟地址，**is_store** 用于区分 TLBL/TLBS 与 Modify 异常。TLB 命中输出 **paddr**，异常时输出 **exc_code** 并带上 **badvaddr**。

2.1.3 代码部分：TLB 查找输出

TLB 逻辑核心是“按 VPN 匹配并给出 paddr/异常码”，代码如下（精简版）：

```
// tlb_simple.v (simplified)
always @(*) begin
    hit = 1'b0; exc_valid = 1'b0; exc_code = 5'd0; paddr = vaddr;
    for (i = 0; i < TLB_ENTRIES; i = i + 1) begin
        if (tlb_valid[i] && tlb_vpn[i] == vaddr[31:12]) begin
            hit  = 1'b1;
            paddr = {tlb_ppn[i], vaddr[11:0]};
            if (!tlb_dirty[i] && is_store) begin
                exc_valid = 1'b1; exc_code = 5'd1; // Modify
            end
        end
    end
    if (!hit) begin
        exc_valid = 1'b1; exc_code = is_store ? 5'd3 : 5'd2; // TLBS/TLBL
    end
end
```

加了 TLB 后的一大改变 之前的 AXI 接到的 mem 我只写了为 256 个 word。实际访问时仅使用地址低位作为索引，高位地址会发生回绕（alias），因此逻辑地址范围大于物理存储容量时会映射到同一片存储区域。（简单来说就是直接把高位扔了，现在 TLB 可以在某种意义上实现一个简单的地址转换）。

2.2 Cache 模块

2.3 I/D-Cache 参数设计

两个 cache 虽然公能有区别，但是结构类似（设计的参数）：

共同点是直映结构、4 行、16B 行大小（byte offset = addr[1:0]，word offset = addr[3:2]，index = addr[5:4]，tag = addr[31:6]）

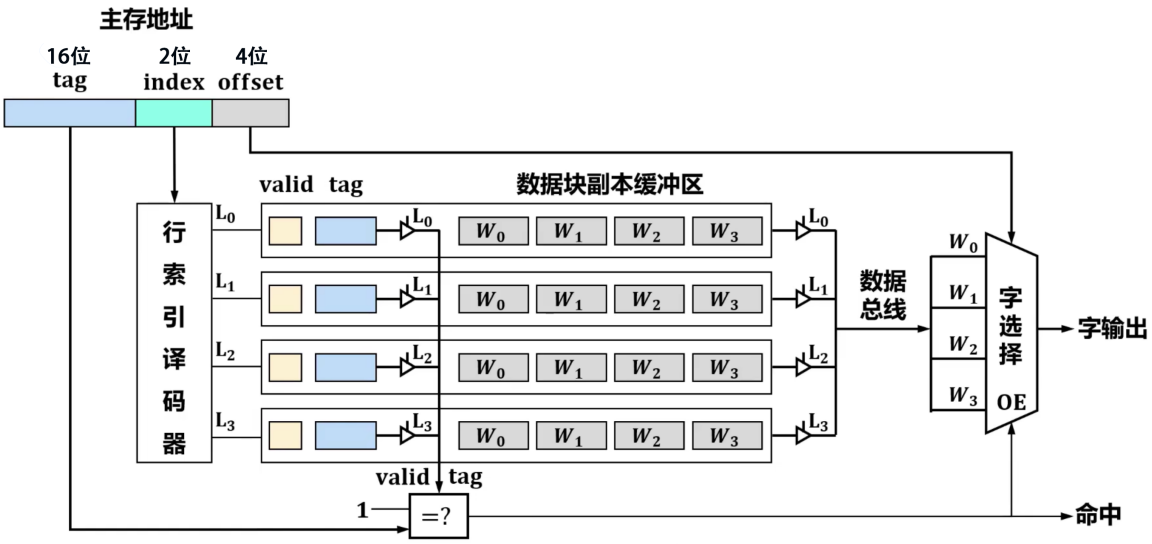


图 2.3: 直接映射 Cache 结构设计图

具体到每个 Cache 的具体实现区别：

1. I-Cache 采用直映结构，4 行，行大小 16B（4 个 word）。索引为 addr[5:4]，标记为 addr[31:6]，字偏移为 addr[3:2]。miss 时发起 burst 长度 4 的 AXI 读事务，填充后更新 tag 与 valid。接口采用 req/resp 握手，支持流水暂停等待。

2. D-Cache 采用直映结构，4 行，行大小 16B（4 个 word），索引与标记划分同 I-Cache。写策略为写直达 + 写分配：store 未命中先 refill，再更新 cache 行并发起单次 AXI 写；load 未命中则 refill 后返回数据。支持字/字节访问，字节写通过读-改-写合并实现，LB/LBU 按符号/无符号扩展返回。

2.3.1 I/D-Cache 区别

I-Cache 仅服务取指路径，属于只读缓存。取指地址先在 I-Cache 中查找，命中则直接返回指令；未命中则触发一次行填充（refill），从指令存储器按 cache 行大小进行 burst 读入，再返回所需指令。由于取指不涉及写操作，I-Cache 不需要脏位与写回策略，实现更简单、时序更稳定。

D-Cache 用于数据访存，支持读写操作。对 load 命中直接返回数据；对 store 命中先更新 cache 行，再根据写策略写到主存。未命中时需要行填充，之后再完成当前读/写。为了简化教学实现，本设计采用直映结构与写直达策略，避免复杂的替换与写回。总结可以为：

1. I-Cache 只读，只做取指；不需要写策略、字节合并、写直达

2. D-Cache 读写都要处理，包含写直达、写分配、LB/LBU/SB 的字节处理

2.3.2 代码部分：Cache 接口与关键信号

I-Cache 为只读接口，采用 req/resp 握手，并通过 AXI 读通道完成 refill：

```
module icache_simple(
    input           clk, resetn,
    input           req_valid,
    input    [31:0] req_addr,
    output          req_ready,
    output reg      resp_valid,
    output reg [31:0] resp_inst,
    output reg      axi_start,
    output reg [31:0] axi_addr,
    output reg [7:0] axi_len,
    input    [31:0] axi_rdata,
    input           axi_rvalid,
    input           axi_done,
    input           axi_busy
);
```

D-Cache 在此基础上增加写通道与访问类型控制（字/字节），并向 MEM 返回读结果：

```
module dcache_simple(
    input           clk, resetn,
    input           req_valid,
    input           req_is_store,
    input    [1:0] req_size,    // 2'b10=word, 2'b00=byte
    input    [31:0] req_paddr,
    input    [31:0] req_wdata,
    output          req_ready,
    output reg      resp_valid,
    output reg [31:0] resp_rdata,
    output reg      axi_start,
    output reg      axi_rw,       // 1=read, 0=write
    output reg [31:0] axi_addr,
    output reg [7:0] axi_len,
    output reg [31:0] axi_wdata,
    output reg      axi_wvalid,
    input           axi_wready,
    input    [31:0] axi_rdata,
    input           axi_rvalid,
    input           axi_done,
    input           axi_busy
);
```

I/D-Cache 均采用直映结构：**index=addr[5:4]**、**tag=addr[31:6]**、**offset=addr[3:0]**。I-Cache 只读、无写策略；D-Cache 则包含写直达与写分配逻辑，并在 store 时触发 AXI 写事务。

2.4 AXI 与 Cache 连接代码说明

本设计将 I-Cache 与 D-Cache 分别连接到两路 AXI 主接口：I-Cache 只使用读通道，D-Cache 使用读 + 写通道。核心连接在 `pipeline_cpu.v` 中完成（精简版）：

```
// I-Cache -> AXI (instruction)
assign instr_user_start = icache_axi_start;
assign instr_user_addr = icache_axi_addr;
assign instr_user_len = icache_axi_len;
assign icache_axi_done = instr_user_done;

// D-Cache/MEM -> AXI (data)
assign data_user_start = mem_axi_start;
assign data_user_addr = mem_axi_addr;
assign data_user_len = mem_axi_len;
assign mem_axi_done  = data_user_done;
```

含义是：取指与访存形成双通道，互不干扰；I-Cache 只在 miss 时发起 burst 读，D-Cache 则在 miss 或写直达时发起相应的读/写事务。

2.5 TLB 与 Cache 例外支持

2.5.1 设计思路

为了让 TLB 与 Cache 接入后仍能产生正确的异常与中断处理，本设计在 MEM 阶段统一产生访存相关异常，并在 WB 阶段进行最终仲裁与送入 CP0。整体流程如下：

1. 地址对齐检查：LW/SW 需要 4 字节对齐，未对齐则产生 AdEL/AdES 异常；
2. TLB 查找：对 load/store 虚拟地址执行 TLB 查找，若 miss/invalid/modify 则产生 TLBL/TLBS/Modify 异常；
3. 异常优先级：TLB 异常优先于地址错异常（先保证地址转换合法性）；
4. Cache 访问：只有在“无异常”情况下才进入 D-Cache 或 AXI 访问；
5. 异常传递：MEM 将异常标志与 badvaddr 打包进 MEM_WB_bus，WB 阶段统一仲裁并写入 CP0。

2.5.2 代码实现

TLB 异常判定（tlb_simple.v） TLB 仅做最简命中判断与异常编码，支持 TLBL/TLBS/Modify：

```
if (req_valid) begin
    if (!match_found) begin
        exc_valid = 1'b1;
        exc_code = is_store ? 5'd3 : 5'd2; // TLBS/TLBL
    end else if (!match_valid) begin
        exc_valid = 1'b1;
        exc_code = is_store ? 5'd3 : 5'd2; // invalid -> TLBL/TLBS
```

```
    end else if (is_store && !match_dirty) begin
        exc_valid = 1'b1;
        exc_code = 5'd1; // Modify
    end
end
```

MEM 阶段异常生成与打包（mem.v） MEM 先做对齐检查，再检查 TLB 异常；若有异常则不再访问 D-Cache/AXI，并将异常打包到 MEM_WB_bus：

```
wire addr_unaligned = ls_word && (vaddr[1:0] != 2'b00);
wire mem_ex_adel = inst_load && addr_unaligned;
wire mem_ex_ades = inst_store && addr_unaligned;

wire mem_ex_tlbl = tlb_exc_valid && (tlb_exc_code == 5'd2);
wire mem_ex_tlbs = tlb_exc_valid && (tlb_exc_code == 5'd3);
wire mem_ex_mod = tlb_exc_valid && (tlb_exc_code == 5'd1);
wire mem_ex_any = mem_ex_adel | mem_ex_ades | mem_ex_tlbl | mem_ex_tlbs | mem_ex_mod;

assign MEM_over = (inst_load | inst_store) ?
                (mem_ex_any ? MEM_valid : cache_resp_valid) :
                MEM_valid;

assign MEM_WB_bus = {
    rf_wen, rf_wdest, MEM_result, lo_result, hi_write, lo_write,
    mfhi, mflo, mtc0, mfc0, cp0r_addr, syscall, brk, eret,
    mem_ex_adel, mem_ex_ades, mem_ex_tlbl, mem_ex_tlbs, mem_ex_mod,
    vaddr, pc
};
```

WB 阶段异常仲裁与 CP0 写入（wb.v） WB 阶段将 TLB 异常与地址错等异常进行优先级仲裁，再送入 CP0：

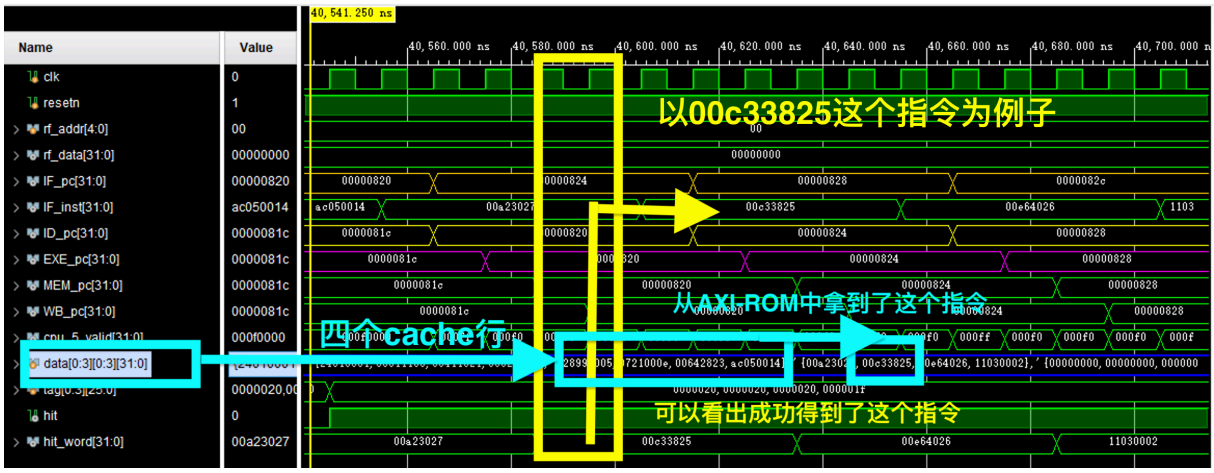
```
assign wb_ex_valid_no_int = (mem_ex_tlbl_wb | mem_ex_tlbs_wb | mem_ex_mod_wb |
                          mem_ex_adel_wb | mem_ex_ades_wb | brk_wb | syscall) ? WB_valid : 1'b0;
assign wb_ex_code_no_int = mem_ex_tlbl_wb ? 5'd2 :
                        mem_ex_tlbs_wb ? 5'd3 :
                        mem_ex_mod_wb ? 5'd1 :
                        mem_ex_adel_wb ? 5'd4 :
                        mem_ex_ades_wb ? 5'd5 :
                        brk_wb ? 5'd9 : 5'd8;
```

上述实现保证了 TLB 异常与 Cache 访问解耦：只要出现地址转换异常，D-Cache 与 AXI 访问就被抑制，并在 WB 阶段统一上报 CP0 处理。

3 实验结果

本实验主要通过仿真日志验证 TLB 与 Cache 功能，覆盖如下测试场景：

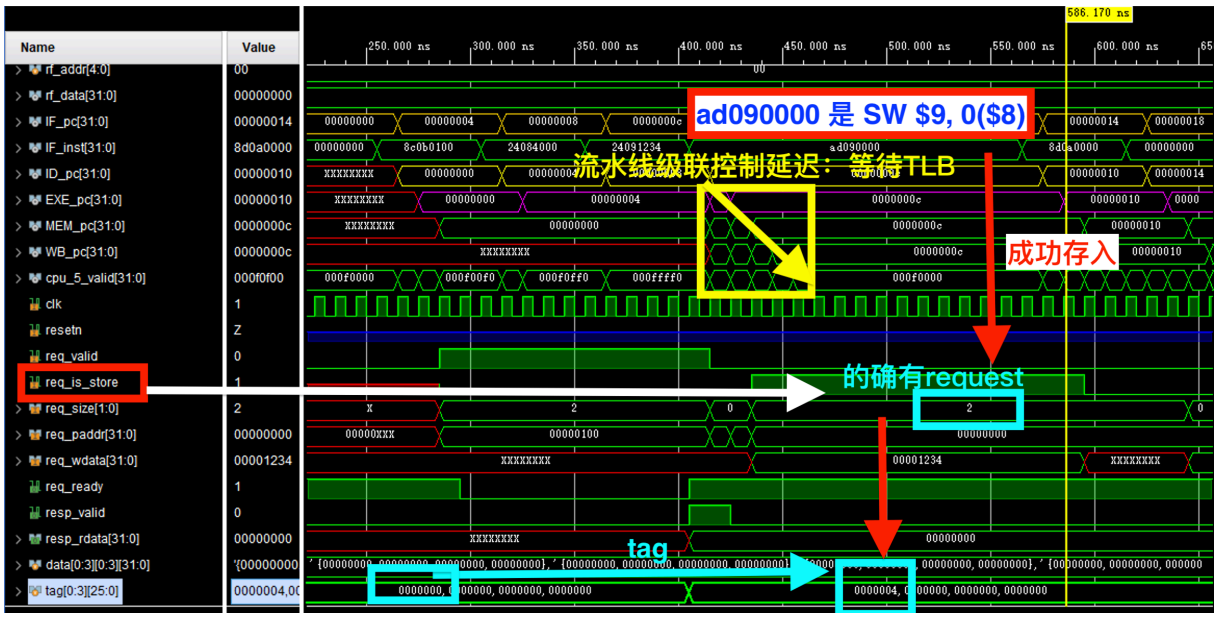
3.1 I-Cache 命中与未命中示例



这里主要分析 I-Cache 中的这几个信号 `data`、`hit`、`hit_word`：以图中波形为例：在 PC 从 0x00000820 递增到 0x0000082C 的窗口内，`hit` 保持为 0，说明这些取指均处于 **miss** 状态；此时 `hit_word` 仍会随 offset 变化给出候选字（如 00a23027、00c33825、00e64026、11030002），但它只是 data 阵列的直读结果，并不代表最终命中输出。待 refill 完成后，`hit` 拉高，I-Cache 才会真正命中并输出稳定指令。该现象与 I-Cache 的命中/回填逻辑一致。

3.2 D-Cache 写直达示例（store）

D-cache 和 TLB 我都用 ad090000 这个例子（一个 store 指令），可以先看图“



图中其实写的比较清楚，store 后，tag 变化，re_is_store 变为 1，然后开始写直达。最后 tag 变化就是写成功的证明

另一个角度可以看 display 出来的详细信息，我在写代码的时候增加了 display 的调错信息，例如以对地址 0x00000014 的 store 为例，D-Cache 命中后先更新 cache 行，再通过 AXI 写直达主存：

```
TLB_LOOKUP: vaddr=00000014 ... hit=1 ... exc=0
D$ REQ: addr=00000014 ... hit=1 is_store=1 size=10
D$ HIT-STORE-W: new=0000000d
D$ WRITE-START: addr=00000014 wdata=0000000d
PIPE_AXI_DATA: start rw=0 addr=00000014 len=1
```

该过程说明 store 命中时不会等待 refill，直接进入写直达阶段；AXI 写完成后出现 WRITE-DONE。

最后效果可以看到从 ROM 中拿到了这个指令。

那个图中等待 TLB 的意思准确来说是等待 TLB、Dcache、AXI 都处理完。

3.3 TLB 命中与异常示例

大致流程是 TLB 命中时 TLB_LOOKUP 显示 hit=1 且 exc=0；若映射无效或越界，则出现 TLBL/TLBS/Modify 异常并在 WB 仲裁后写入 CP0（日志含异常码与 badvaddr）。

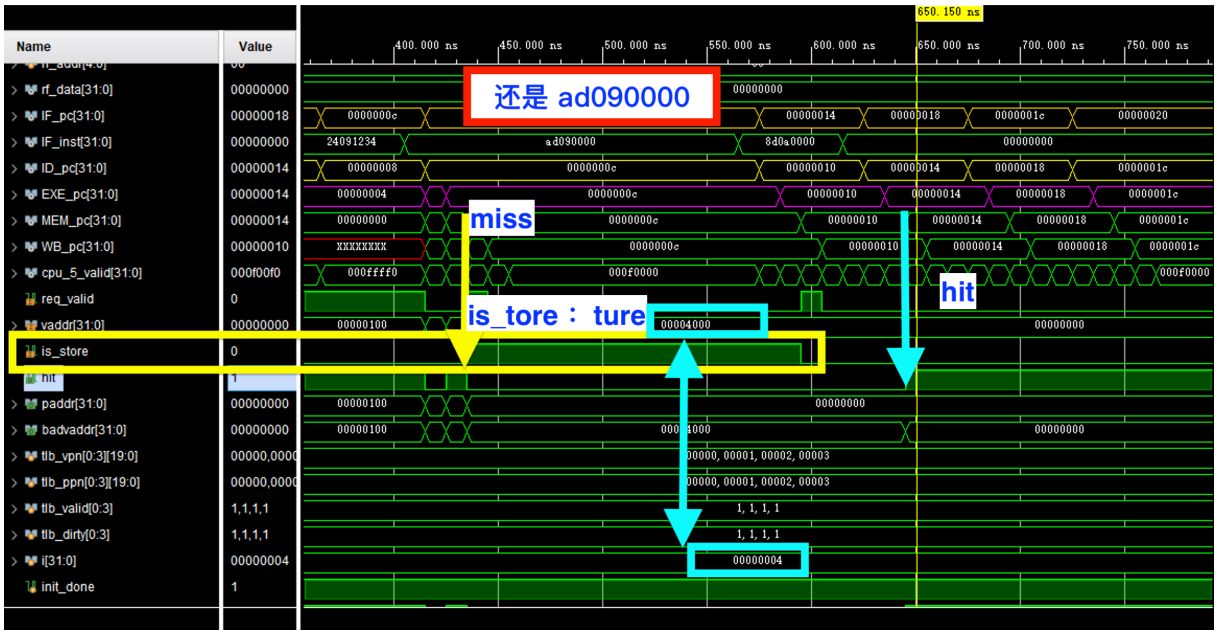


图 3.4: TLB 命中和非命中

异常：可以看到 badaddr、vaddr、i 都是同样的 0x00000004。（缺页异常（这里是往里面 store，但效果一样））。

3.4 流水暂停示例

除了 3.2 D-Cache 图中展示的那样（图中画出了级联流水线延迟），还有就是下面的一个

Cache miss 或 AXI busy 时，MEM_allow_in 拉低导致流水暂停；回填或写回完成后恢复。以下以一次 store 写直达为例：

```
D$ HIT-STORE-W: new=00000011
D$ WRITE-START: addr=0000001c wdata=00000011
PIPE_AXI_DATA: start rw=0 addr=0000001c len=1
... (MEM_allow_in=0 持续，流水暂停)
D$ WRITE-DONE
```

该过程中 MEM 阶段等待 AXI 写完成，EXE/ID 级无法继续前推，直到出现 `WRITE-DONE` 后流水恢复。

综合以上场景，仿真输出与预期一致，说明 TLB、I/D-Cache 与 AXI 接口集成正确。

4 实验总结

本次实验在原有五级流水 CPU 上完成了 TLB 与 I/D Cache 的最简集成，并通过 AXI 双通道实现指令与数据的并行访问。通过在 MEM 阶段统一产生访存异常、在 WB 阶段统一仲裁上报 CP0，保证了 TLB 异常与对齐异常的正确优先级与处理路径。

I-Cache 采用只读直映结构，D-Cache 采用写直达 + 写分配。仿真日志和仿真图都可以说明取指/访存命中路径、miss 回填路径、写直达时的流水暂停以及 TLB 异常上报均按设计运行。