Project 7 实验报告

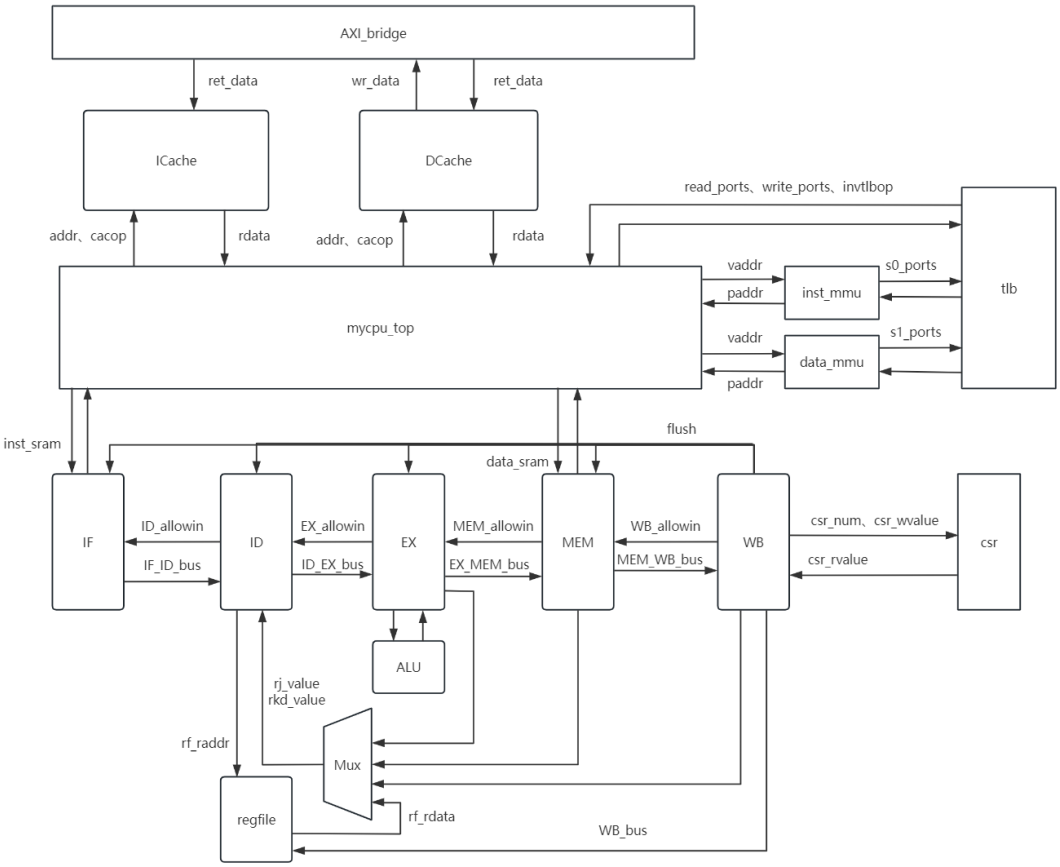
钱翰林 周远扬 石曜铭

2026 年 1 月 13 日

1 小组分工情况说明

- 周远扬: 设计 cache 模块，修改转接桥，报告撰写
- 石曜铭: 将 cache 集成到 cpu，调试
- 钱翰林: 添加 cacop 指令，报告撰写

2 处理器结构设计图



3 主要设计点

3.1 Cache 模块设计

Cache 需要处理读写操作，并且会有是否命中问题，因此依照着状态机的节奏，处理一次操作先需要 check，检查命中情况，若未命中再向内存交互，之后 replace 并 refill，脏块写回与新数据写入，并完成一整次的操作。对于写情况，另有一个状态机控制单独写，主状态机认为传给写状态机之后写操作即完成。

need_pause 包含两种情况：(1) 端口冲突：WriteBuffer 正在向一 Bank 写，而新来的读请求也要读该 Bank。(2) 数据相关：Lookup 阶段是 Store，而新来的读请求访问同一地址 (RAW)。

3.2 修改转接桥

为了集成 Cache，需要 bridge 支持 burst 传输，所以需要改动状态机，使得读写可以多拍延续，并且考虑 last 信号。

对于读写, 可能是一拍或四拍，所以要考虑 last 等信号与实际传输过程，与 Cache 交互依赖状态机即可，与内存交互，就要自行计数。

3.3 在 CPU 中集成 Cache

修改转接桥与 top 模块的相关接口，将 ICache 和 DCache 实例化。

在实际将 Cache 集成进 cpu 的过程中，由于随机延迟的存在，需要 bridge 和 Cache 之间多次握手，防止错过信号。比如 bridge 中控制着传入 Cache 的 rd_rdy 等握手信号，这些信号又控制着 Cache 主状态机，所以处理不当就会错过，需要一步步握手，控制信号保持，以不错漏信号处理

而且同时集成 ICache，DCache, 就会导致读操作冲突，因为两个读端口缩为一个，这里就需要重新构建 bridge 中的 ar 状态机来完成控制，状态分别是 init,wait,req，初始为 init，wait 负责在 rdata 时等待阻塞结束，req 负责 rinst 和 wait 之后的 rdata，发出握手。

3.4 添加 CACOP 指令

Cacop 指令在 EX 级发出，Cache 收到 cacop 操作的相关信号后，Hit 判断可以复用 LookUp 访问的 Tag 读出和比较部分，Cache 行中 V 的修改可以复用 Refill 访问的数据通路，写回内存可以复用 Replace 访问的数据通路。

需要注意的是 hit_invalidate 进行虚实地址转换时会用到两个 mmu, 需要修改原先的数据通路，让 EX 级接收两个 mmu 转换的物理地址以及产生的异常信号。对于 inst_mmu，Cacop 指令的优先级高于 IF 级的取指，这样导致 IF 级可能出现 PC 错误，因此在执行完 Cacop 指令后会 flush 进行指令重取，也可以避免相关的资源冲突。

4 调试

4.1 cacop 指令时序问题

cacop_en 拉高时没有及时更新 tagv_addr 和 tagv_en，而用的是慢了一拍的 reg_cacop_en，导致 cacop 指令的实现时序与 cache 状态机的状态不符。

4.2 I,D 轮流读时，被 need_wait 覆盖

Cache 中的 `need_wait`，指写的同时需要读，那么读等待。

由于延迟随机，新的读请求被 `need_wait` 覆盖掉，其实是 ar 状态机的握手信号问题，wait 时不该继续拉高，所以做出上面的改动，把 wait 时单独处理，不要握手以防丢掉信号。

4.3 脏块回写时内容错误

写缺失时, 要写的数据为内存旧数据叠加输入新数据，不能直接靠 wstrb 控制使能解决，因为 wstrb 为 0 就不再写入。

所以应该先得到要写入的叠加数据，再全部写入。

4.4 wlast 返回错误

因为写操作，考虑到 burst, 有一拍或四拍，所以 wlast 应该在两种情况下都能正确拉高，依靠 bridge 内部的写计数完成。

4.5 上板错误

在通过 exp23 的仿真之后，发现上板出现问题：在烧写后第一遍跑的时候可以通过，但按 reset 之后就一直卡在 0，并且亮两个绿灯，意味着一个点都没有跑完，却认为通过了测试。为了复现这个问题，我们阅读了整体的代码框架，并对 `mycpu_top.v` 进行了修改，将只跑一次修改为循环测试，并关闭了 trace 比对。

为了验证修改的正确性，我们将这份代码复制到 exp22 中并仿真，发现能够完整实现循环测试的设想。于是我们在 exp23 中进行了仿真，发现确实复现了上板的现象：在第一遍跑完之后，跑第二遍时一个点都没过的情况下显示了 PASS。通过观察波形发现，在跑第二遍时 reset 后，对起始 PC `0x1c000000` 取指竟然取出了 `ffffffff`。进而导致触发了指令不存在异常，而此时跳转的目标恰好是直通测试结束的指令地址，因此一条指令都没有执行就结束了。在这个过程中，一次寄存器写都没有，因此也导致没有亮红灯。

为了定位问题，我们首先思考发现，既然 exp22 的表现正常，那么问题一定出现在某条 cacop 指令的执行上；另一方面，既然从不该更改的地址上读出了非预期的数据，说明一定有一刻往 `0x1c000000` 这个地址发了一个修改的请求。通过阅读框架代码，发现内存实际上是实例化了一个 `axi_ram` 模块，直接查看对这个模块的读写，并未发现对 `0x1c000000` 的写操作。这时我们想到，实际地址空间一定没有这么大，所以可能是对某个根本不应该写的地址进行了写的操作，导致恰好映射到了这里。向前查找 cacop 指令的写，发现有一条指令向 `0x00000000` 发了一条写操作，写的内容恰好是 `ffffffff`。而当我们特判了地址为 0 则不写之后，能够正常 reset，也确实证明是这里的问题。

进一步排查写 0 地址的原因，发现根源是查询 index 为 0 的 cache 行的 tagv，返回的结果是 tag 全 0，但 V 为 1。这里写的是 way0，所以从这里向前查找 D-Cache 中，上一次写 addr 为 0 的时候，发现这是一条 cacop 的 cache 初始化指令。最终定位到问题是：cache 初始化时应该把 tag 和 v 均设为 0，而非仅仅将 tag 设为 0。