LSU功能描述和实现原理

在本次处理器设计中，Load-Store Unit（LSU）由4个子模块，Store Queue（STQ）、LoadPipeline、StorePipeline以及一个面向memory的请求仲裁器（Arbiter）构成。通过子模块的相互协作，LSU完成数据写入的暂存、加载过程中的数据选择与拼接、以及最终的 memory 请求发起，构建出一个支持乱序指令执行的访存后端。

STQ是LSU的核心缓冲结构，由三个指针(head,write\_valid,tail)维护的环形结构，以STQEntry的形式保存尚未提交的store指令。每个entry记录目标地址、写入数据，有效字节掩码以及指令类型（func3），支持SB，SH，SW存储操作的按字节精度控制。Store指令向mem的提交通过head和write\_valid指针进行控制，write\_valid的移动依赖于ROB的commit信号。只要当head和write\_valid不重合时，STQ会持续向仲裁器发起访存申请，当仲裁器接收后，head指向的store请求会存入mem，head后移一位

STQ不仅作为写入缓冲，还承担着Load执行过程中的数据前向源这一角色。系统会在STQ中根据LoadPipeline传入的范围（head-ld\_tail）、目标地址和Load指令的func3，将一个 load（最多4byte）拆成每byte单独处理，对于每一个byte，计算其实际地址，在相应的范围内查找是否存在地址匹配且尚未提交的Store条目覆盖了该byte。搜索完所有的byte后，记录每个byte是否成功找到(bytevalid)，并完成按byte拼接。在拼接过程中优先使用传入ld\_tail指针最近的匹配entry中匹配该byte的数据。最终将bitvalid（由bytevalid生成）和拼接的最终数据传回给LoadPipeline。

LoadPipeline是一个四级流水线，依次完成地址计算、STQForwarding和memory请求发起、数据合并以及最后的写回操作。在完成地址计算后，Load流水线向STQ传入Forwarding需要的范围（ld\_tail）、目标地址和Load指令的func3，进行Forwarding查询。此处为了防止LSV的发生，LSU和Dispatch之间有STQ尾指针状态的传递，每条load被dispatch时会记录当前STQ的tail的状态stq\_tail，stq\_tail会跟随这条load指令的uop一起动，最后作为我们Forwarding查找的范围依据，在load执行的时候，STQ 中更年轻的 store 被屏蔽，不能 forward 给这个 load。如果查询全部命中，访问memory 请求将被屏蔽；若仅部分命中，则会同时发起访问memory 请求。在Stage3中通过掩码（bitvalid）按位合并STQ和memory，将合并的数据根据load指令的func3进行符号扩展，适配LB/LBU等指令格式。在Stage4完成最终的写回操作

StorePipeline则是两级流水线，依次进行地址计算、最终写回STQ和ROB。功能简单，只完成地址的初步计算和指令信息的传递。

由于 LSU 内部所有访存操作共享单一 memory 接口，为避免访问冲突，系统采用轻量级的仲裁器对 LoadPipeline 和 STQ 中待提交 Store 的请求进行调度。在本设计中，memory 接口为理想 ready 信号模型，仲裁策略为轮询式公平仲裁，每周期仅允许一个请求通过，保障访存路径在高并发场景下的有序执行。当LoadPipeline发起申请但是轮询优先级没有轮到时，会stall一个周期