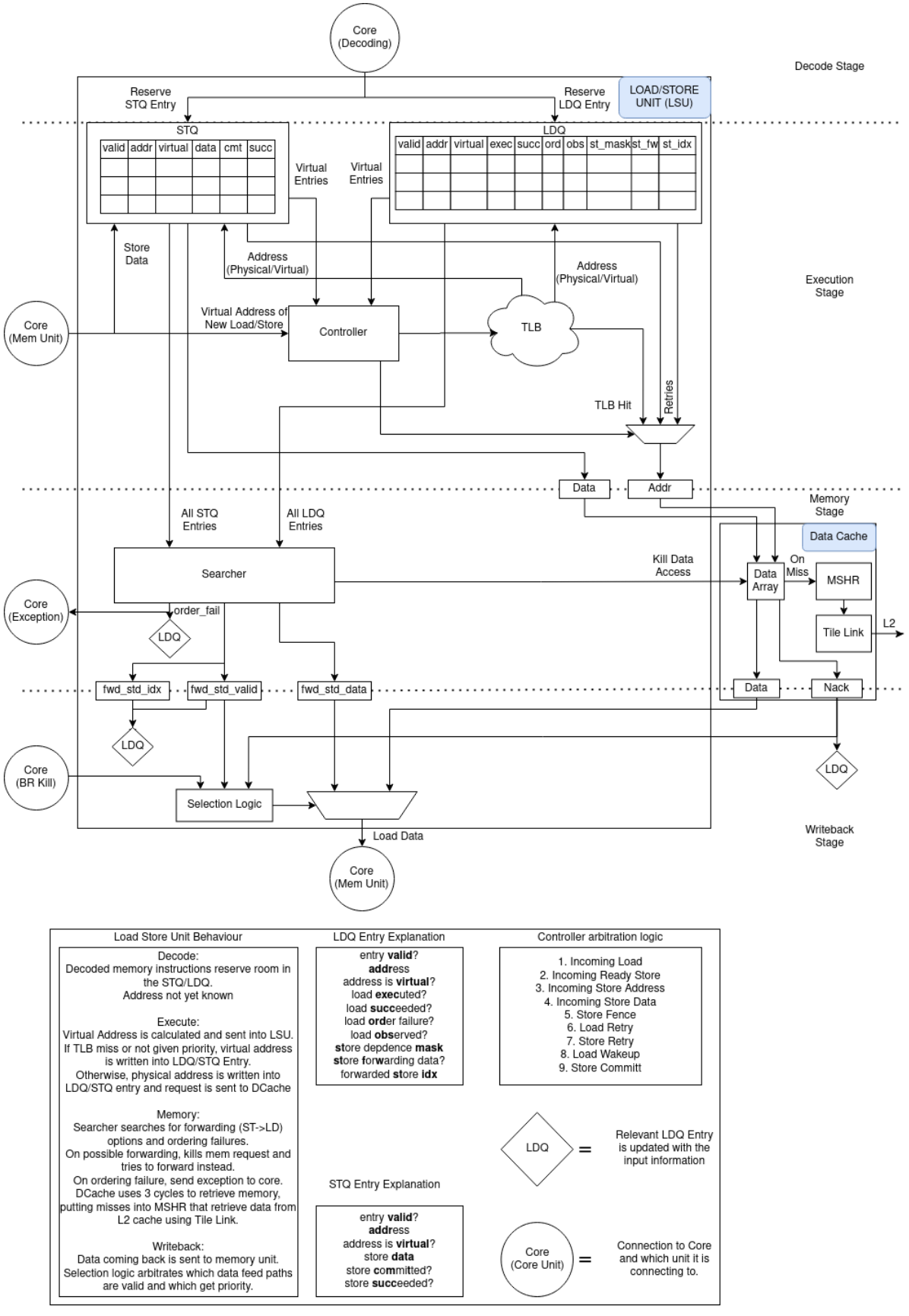
lsu




```
class LDQEntry(implicit p: Parameters) extends BoomBundle()(p)
    with HasBoomUOP
{
  val addr                = Valid(UInt(coreMaxAddrBits.W))
```

```
  val addr_is_virtual      = Bool() // Virtual address, we got a TLB miss
  val addr_is_uncacheable = Bool() // Uncacheable, wait until head of ROB to
execute

  val executed             = Bool() // load sent to memory, reset by NACKs , 发送给
cache
  val succeeded            = Bool() // 成功获得cache的数据 ，包括forward及cache 返回的
  val order_fail           = Bool() // load-load 或者 store-load 错误
  val observed             = Bool() // 被cache release 的addr 会 置位

  val st_dep_mask          = UInt(numStqEntries.W) // list of stores older than
us， 第i位为1,即stq_idx为i的store 是老于load的
  val youngest_stq_idx     = UInt(stqAddrSz.W) // index of the oldest store
younger than us，比load 早的最新的一条store

  val forward_std_val      = Bool()
  val forward_stq_idx      = UInt(stqAddrSz.W) // Which store did we get the
store-load forward from?

  val debug_wb_data        = UInt(xLen.W)
}
```

```
class STQEntry(implicit p: Parameters) extends BoomBundle()(p)
   with HasBoomUOP
{
  val addr                 = Valid(UInt(coreMaxAddrBits.W))
  val addr_is_virtual      = Bool() // Virtual address, we got a TLB miss
  val data                 = Valid(UInt(xLen.W))

  val committed            = Bool() // committed by ROB
  val succeeded            = Bool() // D$ has ack'd this, we don't need to
maintain this anymore

  val debug_wb_data        = UInt(xLen.W)
}
```

sfence

sfence 是一种store barrier 写屏障 ，sfence 之后发出的任何写入指令都不会启动，直到sfence 之前发出的写入指令完成为止

在sfence的时候，会把整个exe_req都赋值成sfence的指令

```
for (i <- 0 until memWidth) {
    when (io.core.exe(i).req.bits.sfence.valid) {
      exe_req := VecInit(Seq.fill(memWidth) { io.core.exe(i).req })
    }
  }
```

多个fire

```
 // Can we fire a incoming load
  val can_fire_load_incoming = widthMap(w => exe_req(w).valid &&
exe_req(w).bits.uop.ctrl.is_load)

  // Can we fire an incoming store addrgen + store datagen
  //
  val can_fire_stad_incoming = widthMap(w => exe_req(w).valid &&
exe_req(w).bits.uop.ctrl.is_sta
                                                              &&
exe_req(w).bits.uop.ctrl.is_std)

  // Can we fire an incoming store addrgen
  val can_fire_sta_incoming  = widthMap(w => exe_req(w).valid &&
exe_req(w).bits.uop.ctrl.is_sta
                                                              &&
!exe_req(w).bits.uop.ctrl.is_std)

  // Can we fire an incoming store datagen
  val can_fire_std_incoming  = widthMap(w => exe_req(w).valid &&
exe_req(w).bits.uop.ctrl.is_std
                                                              &&
!exe_req(w).bits.uop.ctrl.is_sta)

  // Can we fire an incoming sfence
  val can_fire_sfence        = widthMap(w => exe_req(w).valid &&
exe_req(w).bits.sfence.valid)

  // Can we fire a request from dcache to release a line
  // This needs to go through LDQ search to mark loads as dangerous
  val can_fire_release       = widthMap(w => (w == memWidth-1).B &&
io.dmem.release.valid)
  io.dmem.release.ready     := will_fire_release.reduce(_||_)

  // Can we retry a load that missed in the TLB
  val can_fire_load_retry    = widthMap(w =>
                               ( ldq_retry_e.valid                            &&
                                 ldq_retry_e.bits.addr.valid                  &&
                                 ldq_retry_e.bits.addr_is_virtual             &&
                                !p1_block_load_mask(ldq_retry_idx)            &&
                                !p2_block_load_mask(ldq_retry_idx)            &&
                                RegNext(dtlb.io.miss_rdy)                     &&
                                !store_needs_order                            &&
                                (w == memWidth-1).B                           &&
// TODO: Is this best scheduling?
                                !ldq_retry_e.bits.order_fail))

  // Can we retry a store addrgen that missed in the TLB
  // - Weird edge case when sta_retry and std_incoming for same entry in same
cycle. Delay this
  val can_fire_sta_retry     = widthMap(w =>
                               ( stq_retry_e.valid                            &&
                                 stq_retry_e.bits.addr.valid                  &&
                                 stq_retry_e.bits.addr_is_virtual             &&
```

```
                                (w == memWidth-1).B                          &&
                                RegNext(dtlb.io.miss_rdy)                    &&
                                !(widthMap(i => (i != w).B               &&
                                                can_fire_std_incoming(i) &&
                                                stq_incoming_idx(i) ===
stq_retry_idx).reduce(_||_))
                              ))
  // Can we commit a store
  val can_fire_store_commit  = widthMap(w =>
                               ( stq_commit_e.valid                          &&
                                !stq_commit_e.bits.uop.is_fence              &&
                                !mem_xcpt_valid                              &&
                                !stq_commit_e.bits.uop.exception             &&
                                (w == 0).B                                   &&
                                (stq_commit_e.bits.committed || (
stq_commit_e.bits.uop.is_amo      &&

 stq_commit_e.bits.addr.valid      &&

!stq_commit_e.bits.addr_is_virtual &&

 stq_commit_e.bits.data.valid))))

  // Can we wakeup a load that was nack'd
  val block_load_wakeup = WireInit(false.B)
  val can_fire_load_wakeup = widthMap(w =>
                             ( ldq_wakeup_e.valid
     &&
                               ldq_wakeup_e.bits.addr.valid
     &&
                              !ldq_wakeup_e.bits.succeeded
      &&
                              !ldq_wakeup_e.bits.addr_is_virtual
      &&
                              !ldq_wakeup_e.bits.executed
     &&
                              !ldq_wakeup_e.bits.order_fail
     &&
                              !p1_block_load_mask(ldq_wakeup_idx)
     &&
                              !p2_block_load_mask(ldq_wakeup_idx)
     &&
                              !store_needs_order
      &&
                              !block_load_wakeup
      &&
                              (w == memWidth-1).B
     &&
                              (!ldq_wakeup_e.bits.addr_is_uncacheable ||
(io.core.commit_load_at_rob_head &&

 ldq_head === ldq_wakeup_idx &&

 ldq_wakeup_e.bits.st_dep_mask.asUInt === 0.U))))

  // Can we fire an incoming hellacache request
```

```
  val can_fire_hella_incoming  = WireInit(widthMap(w => false.B)) // This is
assigned to in the hellashim ocntroller

  // Can we fire a hellacache request that the dcache nack'd
  val can_fire_hella_wakeup    = WireInit(widthMap(w => false.B)) // This is
assigned to in the hellashim controller
```

is_sta is_std是在exe阶段进行赋值的， is_sta && is_std 说明 此时 addr 和 data 的值都是准备好的， is_sta说明此时addr是准备好的

虽然store 指令被分成多个uop , 但是 可以根据 stq_idx来跟踪具体的load , store 指令

3个数据来源， exe ,ldq ,stq

load retry w = memWidth - 1

sta_retry w = memWidth - 1

store_commit w = 0

load_wakeup w = memWidth - 1

release w = memWidth - 1

选择的时候按照age , 越老的应该会越早进行 retry 等操作

在 load incoming 和 load_retry 的时候 会进行 load queue 的赋值， 包括addr 的valid , 修改 vaddr 为 paddr , 写入 pdest , addr_is_virtual , uncacheable

在 sta_incoming stad _incoming sta_retry 的时候更新 store queue , 包括addr 的valid， addr还是转化为物理地址， addr_is_virtual 赋值， 为什么这里会更新 pdst

stq data 需要判断是浮点数还是 int , 然后选择相应的数据写入相应的store queue ,此时data.valid 会拉高， 浮点貌似只有在w = 0 这一路才能够输入进来

发现br_mask需要随时通过 新的core.brupdata来更新

load_coming , stad_coming , sta_coming , sfence , hella_incoming , load_retry , sta_retry 不会同时拉高 且按照优先级

如果在s0周期data.valid 了， 第s1周期由于是把s0周期的表项进行打拍，正常来说此时data是没有valid的 ，就是说在上个周期赋值的data 是没办法clr_busy_valid的 ，错了， 它是在 sta_incoming 的时候判断data有没有valid,，意思就是判断data 和 addr都valid , sta_incoming 代表上一个周期已经把data写入了

s0 在exe resp 的时候 会判断 will_incoming 等一系列的，发起tlb访问， 发起dcache 访问， 同一拍可以返回tlb 的值， 更新 store queue 和 load queue

首先发起tlb的访问 ，在同一个周期返回是否tlb miss , 如果没有miss 且 uncache 的话， 发起dcache 访问

dcache 有多个数据来源 ，

1. load incoming , exe 发起的 load 请求， 需要查找tlb
2. loadd retry , load queue 发起的 retry 请求， 需要 查找tlb
3. store commit ,

4. load_wakeup
5. hella_incomig
6. hella_wakeup

dcache 有一个总体的ready 信号

位向量 s0_executing_loads 记录送到cache 执行的 load_index

stq有一个stq_execute_head 队列指针， 指向需要commit 的 store

如果exe 发起的时候 dcache 没有ready 好， 这个周期就无法访问dcache , 此时会怎么样 ？

对于store , 本身就没有在incoming执行的选项，是根据execute指针进行执行的

对于load ,在后续通过ldq_wakeup来向dcache发送req

s1 具体需要做以下的事情

1. 向rob 发起 clr_bsy的信号， 针对store 需要store 的data 和 addr都就绪，此时就可以把rob_bsy给清空了，当然valid 还需要在当前周期，前一个周期，前两个周期都没有出现异常 （浮点数也需要另做处理，貌似浮点数只在最后一个口输出给rob）
2. 计算可能可以forward的 load queue index , valid , addr， lcam_addr包含store 和 load 的addr , lcam_ldq_idx只有load 的index , 还有一个store_index ，
3. 设置observe位，在遍历load queue时，如果load_queue 的 addr 和 do_release 的addr block_match , 设置observe
4. 遍历load queue , 1.如果load没有forward_std_val (没有做forward), 2.如果load queue的数据是forward得到的，但是当周期的store指令有和这个load地址一致并且这个store要比forward的store年轻一点，说明之前forward的store是错误的. 需要 置位 ldq_order_fail , 并标记fail_load(会产生异常) 。 （store - load 违例）

   如果当前流水的load 比较老，load queue 对应的load 已经执行了， 并且load queue的load observe为1， 则发生了 order failure, 如果此时load queue 没有执行的话， 更新 s1_set_execute, 暂停cache 访问(就是直接停止访问cache)， 设置can_forward （不太懂）
5. 根据execute_mask表设置load queue 的 execute位 （发送给cache ）
6. 遍历store queue , 如果有地址和store queue地址一致的load , 需要向cache 发起kill 信号，并且可以设置相应的forward（因为此时从cache中取数据是错误的，可以直接forward），addr_match 标志位 , 获取到mem_forward_valid 和 mem_forward_stq_idx(即和load 地址匹配并且最年轻的store(貌似有可能在load后面))， store-load 冲突 , 还有更新s1_set_execute
7. 更新clr_unsafe, (不过貌似这个clr_unsafe是保持为0的)
8. 推测唤醒， 只要 fire_load_incoming (向cache 发送了请求)，可以直接把目的寄存器唤醒

s2 需要做

1. 在 nack_valid 了 之后， lsu 需要 re-execute this ,当 nack uses_ldq时， 把ldq 的executed 重新置为0， nacking_loading 相应位置1， 如果 used_store_ldq的时候，如果stq_idx 比目前的 stq_execute_head 和 stq_head 都要老的话， 更新 stq_execute_head , 相当于要重新commit store 了 、

在resp_valid 了 之后，如果是load, 通过core.exe.iresp 或者 fresp 返回对应的数据,后面就是写回相应的寄存器了，如果没有异常的话指令周期就结束了， ldq.success 会拉高， 如果是store , stq.successed 会拉高， 如果是amo指令， 需要返回data , uop 等值

3. 如果!dmem_resp_fire 并且 forward_fire , 可以做store data 的forward , 根据上一周期选择的forward_stq_idx等值，此时也会把load_queue的success拉高， 同时更新ldq 的 forward_std_val 和 forward_stq_idx
4. 判断ld_miss， 上周期做了推测唤醒， 如果这周期发现spec_ld_succeed 为 0 的话， ld_miss 就为 true 了， 具体判断条件为 1. 上周期做了推测唤醒，2. 这周期iresp.valid 且 ldq_idx 为 上周期incoming的idx ,有一路失败了，ld_miss就为1 了

forward的store 只有 sta_incoming stda_incoming sta_retry 这3个选项（刚获得物理地址）, load 有 load_incoming , load_wakeup , load_retry 这3个选项

dcache 的访问和 lsu 的流水线是并行的两条流水线

load_wake 和 store commit 的特殊处理， 本身优先级是 load_wakeup高于 store_commit , 但是如果can_fire_store_commit了 但是一直无法will_fire_store_commit的话， 当累计15个周期，会 强行 block load_wakeup一个周期，让store_commit

dcache 如果命中的话， s0 发起请求 tag 和 data，s1 返回tag 并比较， s2 周期会返回data(data返回需要两周期) 包含data 和 nack , data就是返回的一些数据， nack 是 s2_req， 或者说就是把请求重新发回给 lsu , s2_send_resp 和 s2_send_nack是互斥的， 不能够同时为1

每周期都需要遍历一次store表，更新表中的br_mask信息， 一旦发现br_mask 中所指的某一条分支出现分支错误，把当前条目的valid , addr.valid , data.valid 都 置为false， 更新live_store_mask的位

同理，每个周期都会遍历也会遍历load 表， 更新 br_mask ,如果发现分支预测错误，复位valid , addr.valid 信号

出现mispredict 的时候，也会直接将 stq_tail 和 ldq_tail 更新为 mispredict的那一个uop 对应的值， 另外说一下， 即使不是load , store 指令， 也会分配对应的load_idx 和store_idx， 确实方便

然后就是commit , 这是和rob的交互， 如果 commit_valid , 并且是 uses_stq 或者 uses_ldq , 如果 commit_store , 把stq.commit 拉高， commit_load， 把load的所有valid , addr.valid , executed , succeeded , order_fail , forwar_std_val 清空，然后更新 stq_commit_head 和 ldq_head , 逻辑很简单， 这里说一下就是 stq 是有3个指针的，包括 stq_head 和 stq_commit_head，stq_execute_head , 但是load只有一个ldq_head, stq_commit_head会快于stq_execute_head 快于 store_head , 因为在commit了之后还需要找机会写入cache , 才能清空表项

stq_commit_head 在 rob发起commit信号的时候 store.commit 之后后增加

stq_execute_head 在发起dcache 访问的时候会增加

stq_head 在commit 且 success 的时候会增加

commit_head > execute_head > head , 因为 commit 了之后才能发起execute , execute 成功了之后拉高success ,才能够增加stq_head

在load-load 冲突的时候，如果loadB 经历过 cache.release且已经被执行了 , 执行loadA的时候会出现 order-fature ,

在执行loadB的时候，发现loadA 没有执行会打断loadB的执行

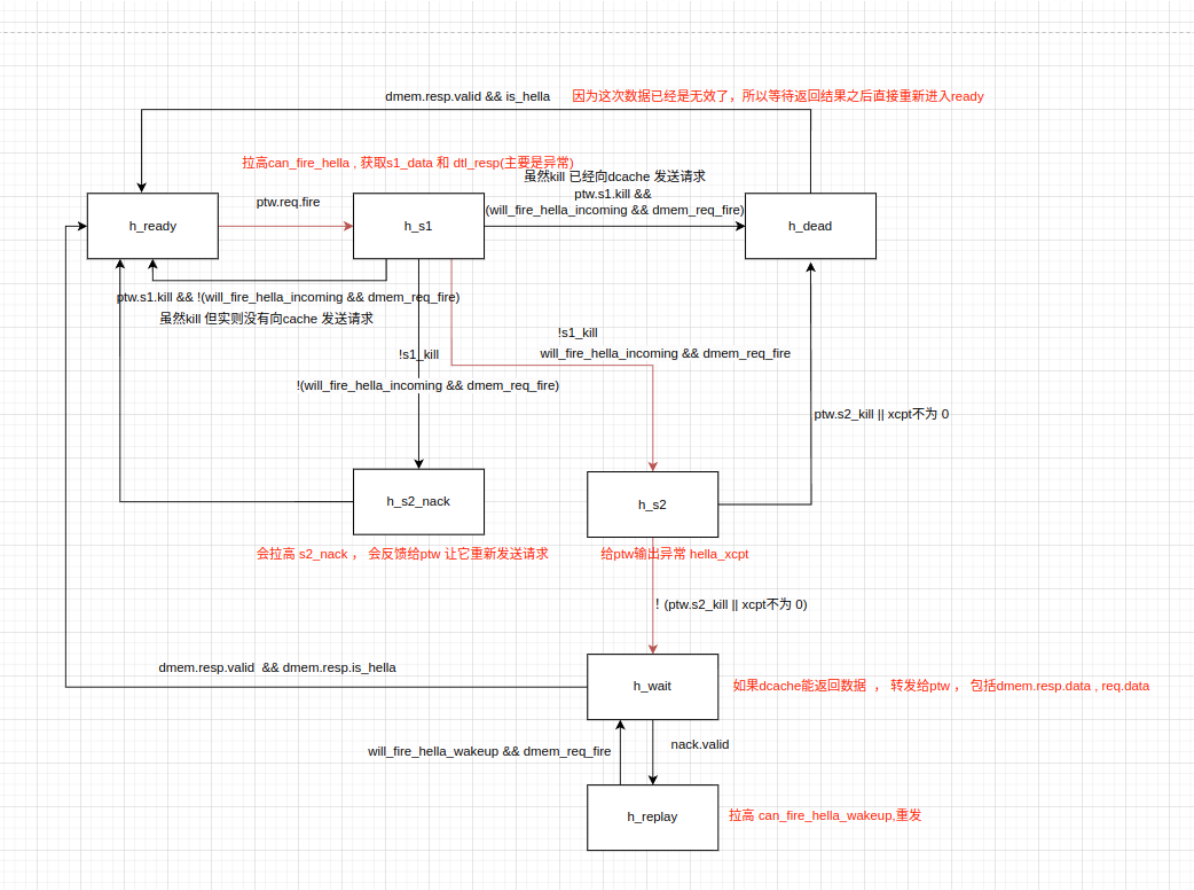
store-load 冲突就是 order-fail

在stq_head的指令又commit 又 successed (写入cache)之后， clear_store会拉高， 此时才清空 stq_head 对应的store queue， 另外 clear_store 还会清空 live_store_mask， 和 load, queue 对应的 st_dep_mask 位， 这样load 就不会通过这条store 来forward 了

发生异常的时候，清空所有的load queue , 把store queue 回退到 commit_head , 清空没有commit 且 successed 的 store queue , 更新live_store_mask 的相应位

最后就是hellacache ，一个状态机

主要实现的是和ptw的交互



附录

参考文献