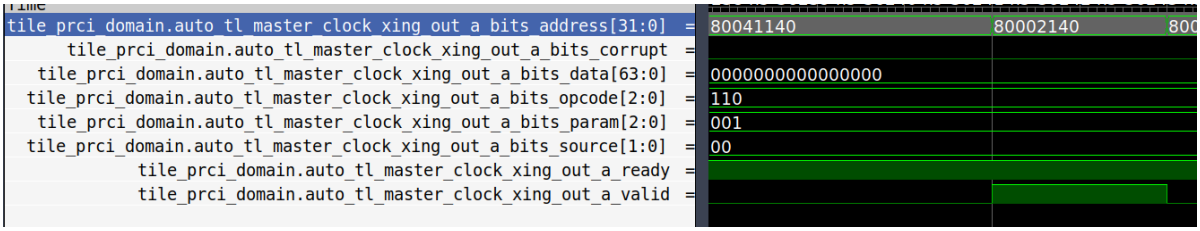
测试程序

以下程序基本能使得在后续的store中能够造成cache miss

```
for (int i = 0 ; i < 128 ; i++) {
    {   // 填8个字节
        for (int j = 0 ; j < 8 ; j ++) {
            Table[(i << 11) + j] = i + j;
        }
    }
  }
  // 再读一次,
  for (int i = 0 ; i < 128 ; i++) {
    Table[i << 11] = i;
  }
```

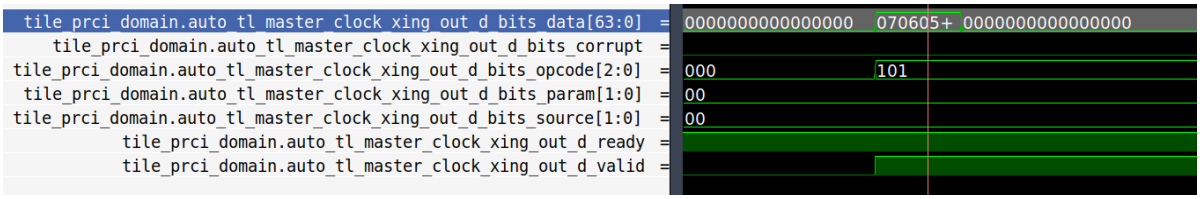
波形分析

以地址0x80002140为例



发送了 acquireblock请求 (opcode=6) ,source = 0,param = 1 ,NtoT,权限升了

通过d通道来回应相应的block,64Byte　opcode = 5 : GrantData param = 0 ,to T



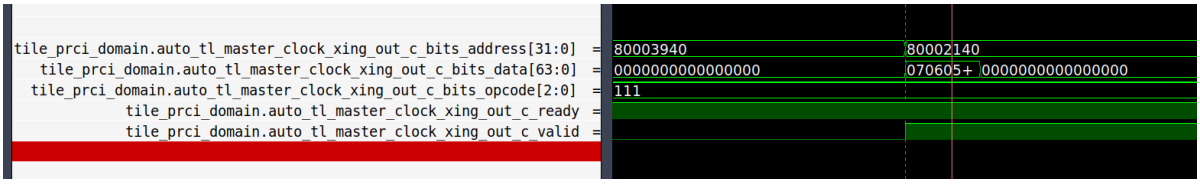
在访问0x80005140的时候,会把0x80002140踢出去

1000 0000 0000 0000 0101 0001 0100 0000

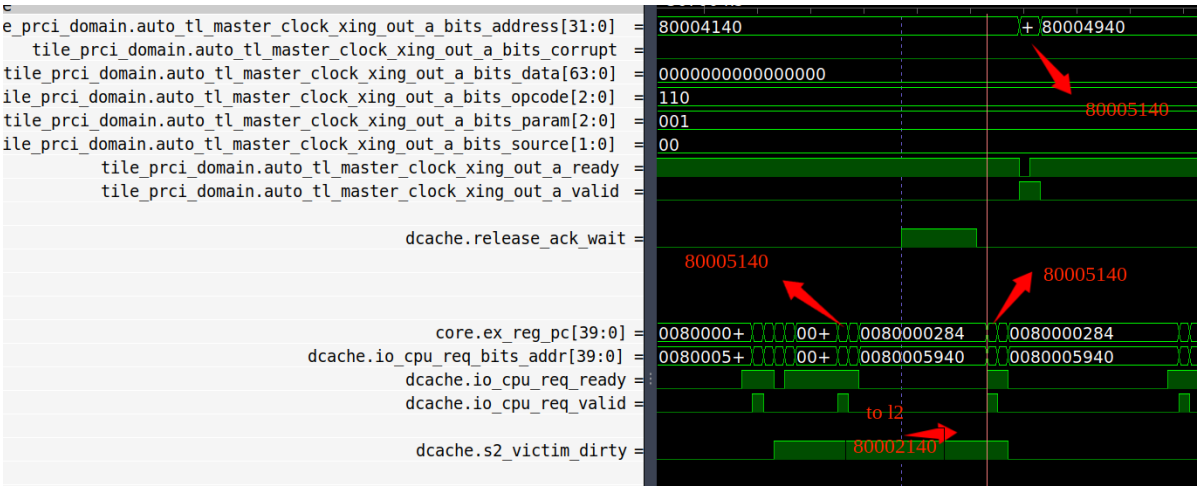
1000 0000 0000 0000 0010 0001 0100 0000

通过c通道向l2发送相应的写请求 ,opcode 为7　releasedata

param 1 TtoN ,权限降了,这样其它核可以有权限去访问这块内存了

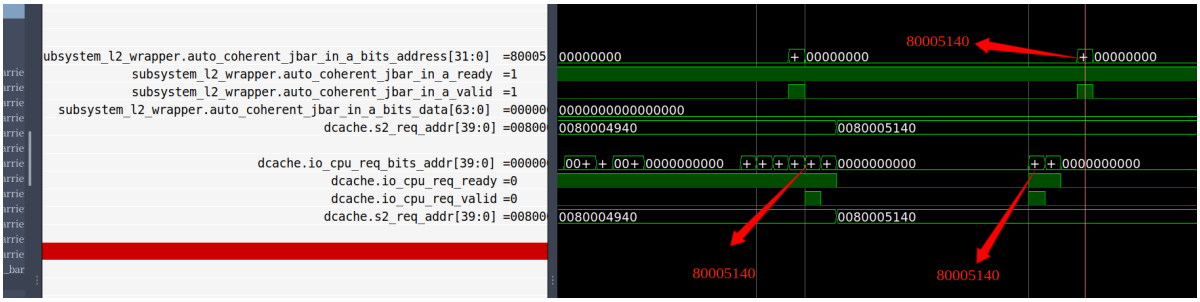


cpu重发消息了吗?? 会不会是第一次req是l1写入l2,第二次req是l1读取l2，应该不是



虽然说80005140发送了两次，但是从代码上来看a_bits_address依赖于s2_req_addr,这个 它在第一次store的时候就进行了赋值，所以按理来说即使第二次不发送，也是能够对l2发起read的，

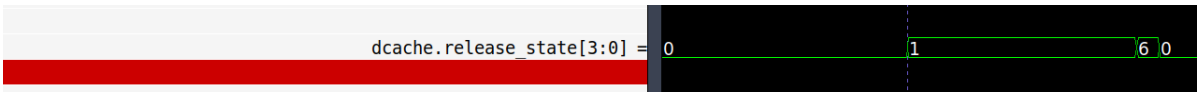
不过为什么会发送两次



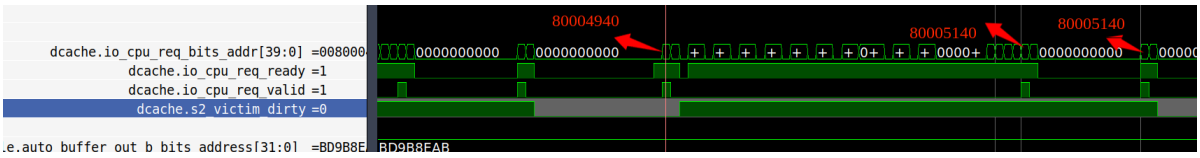
代码

rocket用的是普通的block-cache

s_voluntary_writeback



到底什么时候会release ,就是被踢出去的时候

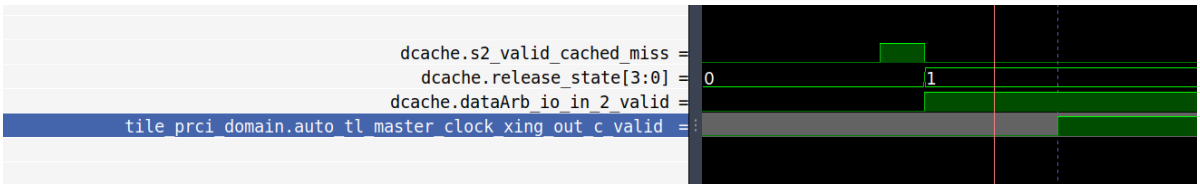


tl_out_c.valid

看下来就是在cached_miss且dirty的时候c_valid会拉高

```
// s2_victim_dirty 在出现没有命中的时候会拉高,在出现命中的时候会拉低
val s2_victim_state = Mux(s2_hit_valid, s2_hit_state, Mux1H(s2_victim_way,
s2_meta_corrected).coh)

val (s2_victim_dirty, s2_shrink_param, voluntaryNewCoh) =
s2_victim_state.onCacheControl(M_FLUSH)

// cache_miss
val s2_want_victimize = (!usingDataScratchpad).B && (s2_valid_cached_miss ||
s2_valid_flush_line || s2_valid_data_error || s2_flush_valid)
```

```
val s2_victimize = s2_want_victimize && !s2_cannot_victimize

when (s2_victimize) {
// 状态变换
release_state := Mux(s2_victim_dirty && !discard_line, s_voluntary_writeback,
                        Mux(!cacheParams.silentDrop.B && !release_ack_wait &&
release_queue_empty && s2_victim_state.isValid() && (s2_valid_flush_line ||
s2_flush_valid || s2_readwrite && !s2_hit_valid), s_voluntary_release,
                        s_voluntary_write_meta))
    }
val inWriteback = release_state.isOneOf(s_voluntary_writeback, s_probe_rep_dirty)



dataArb.io.in(2).valid := inWriteback && releaseDataBeat < refillCycles.U
  dataArb.io.in(2).bits := dataArb.io.in(1).bits

  val s1_release_data_valid = RegNext(dataArb.io.in(2).fire())
  val s2_release_data_valid = RegNext(s1_release_data_valid && !releaseRejected)

  tl_out_c.valid := (s2_release_data_valid || (!cacheParams.silentDrop.B &&
release_state === s_voluntary_release)) && !(c_first && release_ack_wait)
```



t1_out_c.addr and t1_out_c.data

```
//address
when (s2_victimize) {
probe_bits := addressToProbe(s2_vaddr, Cat(s2_victim_tag, s2_req.addr(tagLSB-1,
idxLSB)) << idxLSB)
tl_out_c.bits.address := probe_bits.address
}
```

```
//data
val s2_data_decoded = decodeData(s2_data)
val s2_data_corrected = (s2_data_decoded.map(_.corrected): Seq[UInt]).asUInt
tl_out_c.bits.data := s2_data_corrected
```

t1_out_a.valid

```
// a.valid
tl_out_a.valid := !io.cpu.s2_kill &&
    (s2_valid_uncached_pending ||
      (s2_valid_cached_miss &&
       !(release_ack_wait && (s2_req.addr ^ release_ack_addr)(((pgIdxBits +
pgLevelBits) min paddrBits) - 1, idxLSB) === 0.U) &&
       (cacheParams.acquireBeforeRelease.B && !release_ack_wait &&
release_queue_empty || !s2_victim_dirty)))
  tl_out_a.bits := Mux(!s2_uncached, acquire(s2_vaddr, s2_req.addr,
s2_grow_param),
    Mux(!s2_write, get,
    Mux(s2_req.cmd === M_PWR, putpartial,
    Mux(!s2_read, put, atomics))))
```

(s2_valid_cached_miss &&
! (release_ack_wait && (s2_req.addr ^ release_ack_addr)(((pgIdxBits + pgLevelBits) min paddrBits) - 1, idxLSB) === 0.U) ## 应该是说不在release

cacheParams.acquireBeforeRelease.B 一个配置,在release之前发起请求,看样子是false

!release_ack_wait && release_queue_empty || !s2_victim_dirty ,脏了是要写回的

附录

参考文献

版权信息