第 27 章

香山 SoC 的 MemBlock

在计算机系统结构中，除 CPU 之外最重要的成分就是“内存”（相对于磁盘等外部存储设备而言，也称“主存”）即存储器了。“内存”，或者“存储器”，并不是专指起着存储作用的存储单元阵列，而是泛指整个存储子系统，包括存储阵列及其控制器，尤其还包括为使 CPU 能达到更高运行速度和更高效率而设的高速缓存，以及在多处理器环境下为保证存储一致性和协同性而采取的措施和电路。处于最底层、实际起着存储作用的存储单元阵列一般采用速度较低但价格便宜的动态 RAM 即 DRAM 技术和器件，而上层最靠近 CPU、直接与 CPU 相连的电路和器件则采用速度与 CPU 相称（比 DRAM 快数十倍）但价格较贵的静态 RAM 即 SRAM（作为高速缓存）和 CMOS 电路技术（用于控制）。SRAM 所采用的技术与 CPU 中的逻辑电路是一样的，如果缓存的规模较大就占用资源十分可观。一般而言，在 SoC 芯片上 SRAM 所占用面积的比例会远远超过 CPU 本身。

存储子系统的结构样式有点像金字塔，更准确地说是像个上窄下宽的梯形平台。处于最底层的是速度最慢但规模最大（可达数十 GB 甚至更大）的 DRAM，那是在专门的 DRAM 存储芯片上。最顶端的是速度最高但规模最小（一般是数十 KB）的 SRAM 加上控制电路，一般称为“一级缓存”，一级缓存是集成在 SoC 或 CPU 芯片上的。在这两层之间，则一般有二级缓存，甚至也许还有三级缓存，二级缓存是否与 CPU 集成在同一芯片上就取决于具体的设计了。一般而言，随着集成密度的提高，现在都倾向于集成在同一芯片上。

在多处理器或称“多核”的系统中，一级缓存是与 CPU 紧密结合在一起的，每个 CPU 或“Core”都有自己的一级缓存，二级缓存一般也是属于具体 CPU 的，三级缓存就是公共的了。不过也有的系统其实没有二级缓存但有三级缓存，所以三级缓存实际上就成了公共的二级缓存。总之，在多核系统中需要有一层公共的缓存。从专用的一级缓存到公用的三级缓存，这里明显有个转折，存储的“协同性”问题也就随之而生，这在本书第一卷中已经讲过。当然，像 ICache 那样的只读存储器是不存在（跨 CPU）协同性问题的，也不存在（同一 CPU 上不同进程间的）一致性问题，所以 ICache 的结构就相对要简单得多。然而我们后面会看到，可读可写的 DCache 就有协同性的问题，因而就复杂多了。

在采用流水线结构的处理器中，尤其是采用超标量结构的处理器中，内存读写指令的执行一直是个问题，因为对内存的读写速度存在着相当大的不确定性。一般而言，对一级缓存的读写速度与 CPU 本身的速度是匹配的，如果所读写对象能在一级缓存中命中就不会造成延迟，可是如果在一级缓存不命中而得要临时去存储子系统后方“批发”，那延迟就大了。这时候，最简单的办法是让指令流水线暂时停顿（stall）下来等待。可是，对于流水线，尤其是对于超标量流水线，这个损失可就不小了。比较好的办法是在电路中有个代理机制，遇有访内指令就把对内存的读写委托给代理，而流水线本身并不停下前进的脚步。对于写内存操作，代理只需要向 CPU 作出一个坚决完成任务的承诺，并按接受委托的先后写入内存就行；对于从内存读出，则代理得要保证按次序从内存读出，并将读出的数据直接写入所指定的目标寄存器。当然，如果后面有指令依赖于从内存读入的数据还得停下等待，这也是为什么要有脱序（Out-of-Order）执行的主要原因。这样还有个好处，就是如果代理发现对目标地址的读出之前恰好有对同一目标地址的写入，那就可以抄近路把需要写入的内容立即就交给流水线，而无需等待对内存的写入完成。沿着这个访内代理的思路，结合超标量流水线中多执行单元、多发执行的现实，很容易就会想到：访内指令应该有独立的执行单元，直接就起着访内代理的作用，并与存储子系统有紧密的结合。事实上也正是这样，香山流水线中有个“存储模块”即 MemBlock，里面专门有访内指令 ld（从内存加载，即读入）和 st（向内存存储，即写入）的执行单元。访内指令有个特殊性，就是对内存的读写很可能会有较大的延迟，但是另一方面对内存的读写一般而言又都不是持续的存在，而是

多少有些阵发性，所以需要有队列加以平滑，而且读入和写出需要各有一个队列。实际上读入队列就是读入代理，写出队列就是写出代理；执行单元只需把读写的操作要求挂入相应的队列，下面就是两个队列的事了。

进一步，如果处理器的指令系统支持对内存单元的“原子操作（Atomic Operation）”（一般都会支持），即在一条指令内对一内存单元实施不可被中断、不会被干扰的“读-改-写”操作，那么既然对内存的读写各有互相独立的执行单元，这种“AMO 指令”的执行就理应另有专门的执行单元，而且这个执行单元应该与存储子系统靠得越近越好，最好就是直接做在存储子系统内部。事实上也正是这样，香山的 AMO 指令执行单元就是在数据缓存 DCache 内部。当然，要不是把数据缓存与 CPU 集成在同一块芯片上，想要这样做也并不容易。

从功能和逻辑上说，CPU 的一级缓存现在大多都将指令和数据分开，指令有指令缓存 ICache，数据有数据缓存 DCache，这就是所谓“非冯”的“哈佛结构”，因为在冯诺伊曼的结构中是不分指令和数据的。事实上指令和数据在计算机的运行中确实各有不同的鲜明特色，至少 ICache 是只读的而 DCache 则有回写的问题，只读的 ICache 不存在（对于多 CPU 的）存储协同性问题。另外，指令和数据的局域性也不同，这又影响访问的命中率。实践下来的结果是以分开为好。但是在二级缓存就不一样了，二级缓存一般都是不分指令与数据的，那就又成了冯诺伊曼的结构。比之一级缓存，二级缓存不只是规模扩大了（速度比一级缓存低但比内存高，造价比一级缓存低），更重要的是要保证存储的（多处理器）协同性，因为二级缓存通常都是由多个 CPU 核公用的，这是个相当复杂的问题。事实上，二级缓存的复杂性主要就来自对存储协同性的处理。

在香山的设计中，一级缓存采用哈佛结构，分成指令缓存 ICache 和数据缓存 DCache 两个模块。其中 ICache 我们已经在前面结合指令流水线前端看过了，数据缓存 DCache 则又与 ld/st 指令的执行密切相关，所以香山把 DCache 和 ld/st 指令的执行单元结合在一起，再加 AMO 指令执行单元，成为一个 MemBlock 模块。

另外，就 RISC-V 的存储协同性解决方案 Tile Link 而言，DCache 处于其顶端，DCache 才是一次内存访问的 Trunk Tip（见本书第十一章关于 TileLink 的介绍），所以 DCache 必须得支持 TL-C 规程，而不像 ICache 那样只需支持 TL-U 规程就行。在整个 SoC 芯片的结构中，我们以 CPU 为上层，MemBlock 处于 CPU 之下，而 MemBlock 内部的顶层则是其一级缓存，下面是二级缓存，真正的内存处于最底层。

二级缓存则是个相对独立的模块，名为 HuanCun（缓存），事实上 HuanCun 的代码是个独立的开源软件包。因而 MemBlock 模块仅包含一级缓存中的 DCache 和 ld/st 指令执行单元，而 HuanCun 模块则是对二级缓存的实现。所谓香山的存储子系统则既包括一级缓存也包括二级缓存，既包括 MemBlock 也包括 HuanCun，一级缓存既包括 DCache 也包括 ICache。本章的内容着重于对 MemBlock 的介绍，但不包括 ICache，那已经放在流水线前端中加以介绍。

MemBlock 是在抽象类 XSCoreBase 中（从而也是在 XSCore 中）创建的，下面是 class MemBlock 的代码：

```
[XSTop > XSTile > XSCore > XSCoreBase > MemBlock]
class MemBlock()(implicit p: Parameters) extends LazyModule with HasXSParameter with HasWritebackSource {
  val dcache = LazyModule(new DCacheWrapper())                    ///DCache 的外层包装
  val uncache = LazyModule(new Uncache())                         ///访问外设的接口
  val pf_sender_opt = coreParams.prefetcher.map(_ => BundleBridgeSource(() => new PrefetchRecv))   ///数据预取（可选）

  lazy val module = new MemBlockImp(this)                         ///MemBlock 的硬件实现

  override val writebackSourceParams: Seq[WritebackSourceParams] = {          ///MemBlock 也是指令执行结果回写的源头之一
    val params = new WritebackSourceParams
    params.exuConfigs = (loadExuConfigs ++ storeExuConfigs).map(cfg => Seq(cfg))
    Seq(params)
  }
  override lazy val writebackSourceImp: HasWritebackSourceImp = module
}  ///end class MemBlock
```

在抽象类 XSCoreBase 中只创建了一个 MemBlock 模块，而扩充了抽象类 XSCoreBase 的具体类只有一个，那就是 XSCore，创建 XSCore 的则是 XSTile，然而一个 SoC 芯片上却可以有多个 XSTile。所以，SoC 芯片上有几个 XSTile，即几个 XSCore，就有几个 MemBlock 模块，但是一个 XSCore 只有一个 MemBlock。

MemBlock 对 DCacheWrapper 的创建是不言而喻的，因为 MemBlock 的主体就是 DCache。此外，由于在 RISC-V 的系统结构中把外设看成存储子系统的一部分，这里还创建了一个 Uncache 模块，这与我们在前端中看到的 InstrUncache 是一样的道理。至于 PrefetchRecv，那是对数据的“预取”，其实就是让 DCache 提前从内存将某一块数据装载到缓存中。这里还提供了一个名叫 writebackSourceParams 的 val，这是对 trait HasWritebackSource 中同名 val 的覆盖。对于 val 的“赋值”来自其“赋值函数”，但这是在创建 MemBlock 对象的初始化过程中就要执行的，普通的函数则仅在被调用时才执行。凡是继承了 trait HasWritebackSource 的某类对象，都有可能被回写，而 writebackSourceParams 则说明了回写的来源，这里说是 loadExuConfigs ++ storeExuConfigs，那就是 ld/st 指令的执行单元。

我们先看 DCacheWrapper：

```
[XSCoreBase > MemBlock > DCacheWrapper]
class DCacheWrapper()(implicit p: Parameters) extends LazyModule with HasXSParameter {
  val useDcache = coreParams.dcacheParametersOpt.nonEmpty                    ///DCache也可以不用，一些嵌入式应用中无需数据缓存。
  val clientNode = if (useDcache) TLIdentityNode() else null
  val dcache = if (useDcache) LazyModule(new DCache()) else null             ///如果真要创建Dcache
  if (useDcache) {
    clientNode := dcache.clientNode
  }

  lazy val module = new LazyModuleImp(this) with HasPerfEvents {             /// DCacheWrapper的硬件实现
    val io = IO(new DCacheIO)
    val perfEvents = if (!useDcache) {
      // a fake dcache which uses dpi-c to access memory, only for debug usage!
      val fake_dcache = Module(new FakeDCache())                             ///可以只是个摆摆样子的FakeDCache
      io <> fake_dcache.io
      Seq()
    }
    else {
      io <> dcache.module.io                                                 ///要不然就将io连接到上面创建的Dcache上
      dcache.module.getPerfEvents
    }
    generatePerfEvent()
  }
}
```

注意 Lazy 模块 DCacheWrapper 是在 MemBlock 的 Diplomacy 部分创建的，而且这里确实有了参与参数一致化的 node，这是个 TLIdentityNode，事实上 DCache 与二级缓存 HuanCun 之间就有参数一致化的问题，而且二者之间的连接就是通过 TileLink。我们在第一卷中讲过，TileLink 提供了保证存储协同性所需的手段。

另外，MemBlock 之是否采用 DCache 是可选的。在一些简单的嵌入式应用中也可以不用 DCache，但 MemBlock 仍旧是 MemBlock，表面上也仍有 DCache，然而那却是个 FakeDCache，假的、摆摆样子的 DCache。当然，一般而言是真要有 DCache 的，那也是在这里创建。

再看 Uncache，由于可读可写，这就比 InstrUncache 复杂一些，我们先把这个分支解决掉，然后再回到 DCache 这个正题。

27.1 外设接口 Uncache

Uncache 就是 MemBlock 的 MMIO 接口，这也是个 Lazy 模块：

```
[XSCoreBase > MemBlock > Uncache]
// convert DCacheIO to TileLink. for Now, we only deal with TL-UL          ///既然是Uncache即无缓存，当然只需支持TL-U规程。
class Uncache()(implicit p: Parameters) extends LazyModule with HasXSParameter {
  def idRange: Int = UncacheBufferSize
  val clientParameters = TLMasterPortParameters.v1(
    clients = Seq(TLMasterParameters.v1( "uncache", sourceId = IdRange(0, idRange) ))
  )
  val clientNode = TLClientNode(Seq(clientParameters))                ///Client相当于Master

  lazy val module = new UncacheImp(this)
}
```

Uncache 模块也要参与参数一致化的协调，协调的对方就是具体的外设模块。而 Uncache 模块自身的硬件描述则在 UncacheImp 中：

```
[XSCoreBase > MemBlock > Uncache > UncacheImp]
class UncacheImp(outer: Uncache)extends LazyModuleImp(outer) with HasTLDump with HasXSParameter with HasPerfEvents{
  val io = IO(new UncacheIO)          ///见后

  val (bus, edge) = outer.clientNode.out.head          ///这是通过与下层绑定获得的连接

  val req = io.lsq.req
  val resp = io.lsq.resp
  val mem_acquire = bus.a             ///TLBundle中的A通道，注意在A通道上既可发出读命令也可发出写命令。
  val mem_grant = bus.d               ///TLBundle中的D通道，读出的数据在D通道上。

  val req_ready = WireInit(false.B)
  val need_fence = WireInit(false.B)

  // assign default values to output signals
  bus.b.ready := false.B              ///对B通道的设置，Uncache无需使用B通道，因为外设数据不可缓存。
  bus.c.valid := false.B              ///Uncache也无需使用C通道。
  bus.c.bits  := DontCare
  bus.d.ready := false.B              ///D通道不会从上层接收，故将其ready信号设成false。
  bus.e.valid := false.B              ///Uncache无需使用E通道
  bus.e.bits  := DontCare

  val enqPtr = RegInit(0.U.asTypeOf(new UncachePtr))          ///将下面创建的多个MMIOEntry模块看成一个队列，轮流使用。
  val issPtr = RegInit(0.U.asTypeOf(new UncachePtr))
  val deqPtr = RegInit(0.U.asTypeOf(new UncachePtr))
```

```
  val fence = RegInit(Bool(), false.B)

  io.lsq.resp.valid := false.B
  io.lsq.resp.bits := DontCare

  val entries = Seq.fill(UncacheBufferSize) { Module(new MMIOEntry(edge)) }  ///创建一串4个MMIOEntry模块
///class UncacheImp，  待续。
```

首先，既然是与外设的接口，是 Uncache 即无缓存，当然只需支持 TL-U 规程。所以只需要支持 A 和 D 两个通道就够了，因而这里把 B/C/E 三个通道都封了。

Uncache 的外部接口是 UncacheIO：

```
class UncacheIO(implicit p: Parameters) extends DCacheBundle {
  val hartId = Input(UInt())                          ///Uncache也是与CPU绑定在一起的，所以有个hartId。
  val enableOutstanding = Input(Bool())
  val flush = Flipped(new UncacheFlushBundle)         ///如果指令流转向。
  val lsq = Flipped(new UncacheWordIO)                ///来自LoadQueue和StoreQueue，是Uncache的上层。
}
```

UncacheIO 主要是对 lsq，即 LoadQueue 和 StoreQueue 的接口，通过 UncacheWordIO 从这两个队列接受请求和作出回应。这里有个看似奇怪的问题，Uncache 不应该是个终端节点，有输入必有输出，可为什么在 UncacheIO 中只有来自 lsq 的输入，却没有看到有输出。这是因为 Uncache 在参数协调阶段被绑定到了底层的外设接口，这就蕴含着 Uncache 与底层外设接口既有参数协调又有 TLBundle 的连接。我们看一下 class XSTile 中的相关代码：

```
class XSTile()(implicit p: Parameters) extends LazyModule with HasXSParameter with HasSoCParameter {
  …
  misc.d_mmio_port := core.memBlock.uncache.clientNode
  lazy val module = new LazyModuleImp(this){…}
}
```

如前所述，MemBlock 模块是在 XSCore 中创建的，而 XSCore 又是在 XSTile 中创建的，所以 XSTile 中的 core.memBlock.uncache.clientNode 就是前面 class Uncache 中的那个 clientNode，XSTile 把它绑定到了 misc.d_mmio_port，这就是外部设备的接口。这样，一方面是 Uncache 与外设接口之间需要有参数一致化，另一方面也蕴含着二者之间有个 TileLink 连接，因为 Uncache 的 clientNode 是个 TLClientNode。这样，在 Uncache 的外部接口 UncacheIO 上就只要有个对上的连接就可以了。

不过因为 Uncache 的底层是外设接口，而外设的数据是不缓存的，所以 B/C/E 通道对其没有意义，只要使用 A/D 两个通道就行。

回到 UncacheImp 的代码，下面就是创建若干 MMIOEntry 模块，具体个数取决于参数 UncacheBufferSize；这个参数定义为 4，所以就是一排 4 个 MMIOEntry 模块，一个 MMIOEntry 模块就是一个通往作为 MMIO（Memory Mapped IO）的外设接口：

```
[XSCoreBase > MemBlock > Uncache > UncacheImp > MMIOEntry]
// One miss entry deals with one mmio request
class MMIOEntry(edge: TLEdgeOut)(implicit p: Parameters) extends DCacheModule {
  val io = IO(new Bundle {
```

```
    //   MSHR ID     ///MSHR就是Miss Status Holding Register
    val hartId = Input(UInt())
    //   Control IO
    val enableOutstanding = Input(Bool())

    //   Client requests
    val req = Flipped(DecoupledIO(new UncacheWordReq))                    ///来自上层(即Uncache)的读/写请求
    val resp = DecoupledIO(new DCacheWordRespWithError)                   ///向上层的回应

    //   TileLink
    val mem_acquire = DecoupledIO(new TLBundleA(edge.bundle))             /// mem_acquire在A通道上，这是向下的请求。
    val mem_grant = Flipped(DecoupledIO(new TLBundleD(edge.bundle)))      /// mem_grant在D通道上，这是来自下层的回应。
                                                                          ///注意所说的mem实际上是MMIO即外设
    //   This entry is selected.
    val select = Input(Bool())
    val atomic = Output(Bool())                                           ///表示这是原子操作
  })  ///end val io

  //   ========================================================
  //   FSM state description:
  //   s_invalid     : Entry is invalid.
  //   s_refill_req  : Send Acquire request.
  //   s_refill_resp : Wait for Grant response.
  //   s_send_resp   : Send Uncache response.
  val s_invalid :: s_refill_req :: s_refill_resp :: s_send_resp :: Nil = Enum(4)
  val state = RegInit(s_invalid)                                          ///状态机的初始状态是s_invalid

  val req = Reg(new UncacheWordReq)                                       ///寄存器，保存来自上层的读/写请求
  val resp_data = Reg(UInt(DataBits.W))                                   ///寄存器，保存要向上层发送的单项数据
  def storeReq = req.cmd === MemoryOpConstants.M_XWR                      ///存储请求的操作码是M_XWR即写入

  //   Assign default values to output signals.
  io.req.ready := false.B                                                 ///向上层即Uncache表示准备接受读/写请求
  io.resp.valid := false.B                                                ///暂无数据向上发送
  io.resp.bits := DontCare

  io.mem_acquire.valid := false.B                                         ///暂无向下层即外设的读/写请求
  io.mem_acquire.bits := DontCare
  io.mem_grant.ready := false.B                                           ///没有准备从下层接收

  io.atomic := req.atomic
  //   Receive request
  when (state === s_invalid) {                                            ///当状态机处于s_invalid状态时：
    io.req.ready := true.B                                                    ///准备从上层接受请求
    when (io.req.fire) {                                                      ///如有请求到来，就：
      req := io.req.bits                                                          ///将请求的内容打入寄存器req
```

```
      req.addr := io.req.bits.addr
      state := s_refill_req                                    ///转入s_refill_req状态
    }
  }

  //   Refill          ///Refill就是从下层(外设)读入或向下层写出
  //   TODO: determine 'lgSize' in memend
  val size = PopCount(req.mask)                                ///req.mask中的1位代表一个字节
  val (lgSize, legal) = PriorityMuxWithFlag(Seq(1.U -> 0.U, 2.U -> 1.U, 4.U -> 2.U, 8.U -> 3.U).map(m => (size === m._1) -> m._2))
  assert(!(io.mem_acquire.valid && !legal))

  val load = edge.Get(fromSource = io.hartId, toAddress = req.addr, lgSize = lgSize)._2
  val store = edge.Put(fromSource = io.hartId, toAddress = req.addr, lgSize = lgSize, data = req.data, mask = req.mask)._2

  XSDebug("entry: %d state: %d\n", io.hartId, state)

  when (state === s_refill_req) {                              ///当状态机处于s_invalid状态时：
    io.mem_acquire.valid := true.B && io.select                    ///在A通道上向下层发送
    io.mem_acquire.bits := Mux(storeReq, store, load)              ///根据请求中的操作码决定是发送store还是load

    when (io.mem_acquire.fire) {                               ///发送出去（取决于下层是否ready）以后就：
      state := s_refill_resp                                       ///转入s_refill_resp状态
    }
  }

  val (_, _, refill_done, _) = edge.addr_inc(io.mem_grant)
  when (state === s_refill_resp) {                             ///当状态机处于s_refill_resp状态时：
    io.mem_grant.ready := true.B                                   ///准备在D通道上接收

    when (io.mem_grant.fire) {                                     ///当下层的回应到来时：
      resp_data := io.mem_grant.bits.data                              ///在D通道上接收回应数据
      assert(refill_done, "Uncache response should be one beat only!")
      state := Mux(storeReq && io.enableOutstanding, s_invalid, s_send_resp)
                ///如果操作请求是M_XWR即写入，而且又允许Outstanding，就转入s_invalid，否则转入s_send_resp。
    }
  }

  //   Response
  when (state === s_send_resp) {                   ///当状态机处于s_send_resp状态时：
    io.resp.valid := true.B
    io.resp.bits.data    := resp_data              ///向上的回应数据就是从D通道上接收到的下层数据
    // meta data should go with the response
    io.resp.bits.id      := req.id                 ///回应的编号与请求的编号相同
    io.resp.bits.miss    := false.B
    io.resp.bits.replay  := false.B
    io.resp.bits.tag_error := false.B
```

```
    io.resp.bits.error   := false.B

    when (io.resp.fire()) {                          ///向上的回应发出以后（取决于上层是否ready），就：
      state := s_invalid                                  ///转入s_invalid状态
    }
  }
  //  End
}  ///end class MMIOEntry
```

可见，MMIOEntry 就好比是 Uncache 读写 MMIO 即外设寄存器的中介，或者说读写“引擎”。Uncache 通过 MMIOEntry 访问外设。

前面说过，Uncache 中有多个 MMIOEntry 模块，具体数量取决于参数 UncacheBufferSize（定义为 4），这 4 个 MMIOEntry 模块就构成一个队列，轮流使用。之所以如此，是因为下层对外设的操作比较慢，占用 MMIOEntry 的时间比较长，如果只用一个 MMIOEntry 就排不过来，如果被下层拖住就不能再从上层接收请求，从而造成阻塞。另一方面，这样还能把 Uncache 通往下层的 A 和 D 两个通道的使用，以及 Uncache 从上层的请求接收，在时间上错开，加以流水线化。

我们回到 Uncache，看看 Uncache 是怎样使用这些 MMIOEntry 的：

```
[XSCoreBase > MemBlock > Uncache > UncacheImp]     续：
  for ((entry, i) <- entries.zipWithIndex) {                  ///对于entries中的每个MMIOEntry：
    entry.io.hartId := io.hartId
    entry.io.enableOutstanding := io.enableOutstanding

   //  Enqueue
    entry.io.req.valid := (i.U === enqPtr.value) && req.valid          ///所谓enqPtr，所指向的就是当前轮到使用的MMIOEntry。
    entry.io.req.bits := req.bits
    when (i.U === enqPtr.value) { req_ready := entry.io.req.ready }
                                                    ///这个MMIOEntry的ready信号就作为Uncache向其上层(lsq)发送的ready

    //  Acquire        ///所谓issPtr，所指向的就是当前可以向下层发送请求的那个MMIOEntry（相当于流水线的第2步）
    entry.io.select := (i.U === issPtr.value) && Mux(entry.io.atomic, issPtr.value === deqPtr.value, !fence)
    when (i.U === issPtr.value) { need_fence := entry.io.atomic }

    //  Grant          ///所谓deqPtr，所指向的就是当前可以接受来自下层回应的那个MMIOEntry（相当于流水线的第3步）
    entry.io.mem_grant.valid := false.B
    entry.io.mem_grant.bits := DontCare
    when (i.U === deqPtr.value) { entry.io.mem_grant <> mem_grant }

    entry.io.resp.ready := false.B
    when (i.U === deqPtr.value) { io.lsq.resp <> entry.io.resp }
  }  ///这样，就把对于A和D两个通道的使用，以及从上层的接受，一起流水线化了。

  io.lsq.req.ready := req_ready              ///向上层即lsq发出的ready信号来自req_ready
  when (io.enableOutstanding) {              ///代码中原有的注释说明了对MMIOEntry队列和A/D通道使用的流水线化：
    //  Uncache Buffer is a circular queue, which contains UncacheBufferSize entries.
    //  Description:
```

```
//     enqPtr: Point to an invalid (means that the entry is free) entry.
//     issPtr: Point to a ready entry, the entry is ready to issue.
//     deqPtr: Point to the oldest entry, which was issued but has not accepted response (used to keep order with the program order).
//
//   When outstanding disabled, only one read/write request can be accepted at a time.
//   Example (Enable outstanding):
//     1. enqPtr:
//        1) Before enqueue
//           enqPtr --
//                   |
//                   V
//           +--+--+--+--+
//           |  |  |  |  |
//           +--+--+--+--+
//
//        2) After
//           enqPtr+1 ---
//                      |
//                      V
//           +--+--+--+--+
//           |  |  |  |  |
//           +--+--+--+--+
//
//     2. issPtr:
//        1) Before issue
//           issPtr --
//                   |
//                   V
//           +--+--+--+--+
//           |  |  |  |  |
//           +--+--+--+--+
//
//        2) After issue
//           issPtr+1 --
//                      |
//                      V
//           +--+--+--+--+
//           |  |  |  |  |
//           +--+--+--+--+
//
//     3. deqPtr:
//        1) Before dequeue
//           deqPtr --
//                   |
//                   V
//           +--+--+--+--+
```

```
  //          |  |  |  |  |
  //          +--+--+--+--+
  //
  //      2) After dequeue
  //          deqPtr --                       deqPtr+1 --
  //                   |                                |
  //                   V                                V
  //          +--+--+--+--+       or       +--+--+--+--+--+
  //          |  |  |  |  |                |  |  |  |  |  |
  //          +--+--+--+--+                +--+--+--+--+--+
  //              (load)                       (store)
  //
  //      3) After response
  //          deqPtr+1 ---                    deqPtr--
  //                     |                            |
  //                     V                            V
  //          +--+--+--+--+       or       +--+--+--+--+--+
  //          |  |  |  |  |                |  |  |  |  |  |
  //          +--+--+--+--+                +--+--+--+--+--+
  //              (load)                       (store)
  //
```

```
  when (req.fire) { enqPtr := enqPtr + 1.U }                // Enqueue

  when (mem_acquire.fire) { issPtr := issPtr + 1.U }        // Issue
  when (mem_acquire.fire) { fence := need_fence }

  // Dequeue
  when (mem_grant.fire) { deqPtr := Mux(edge.hasData(mem_grant.bits), deqPtr /* Load */, deqPtr + 1.U /* Store */) }
  .elsewhen (io.lsq.resp.fire /* Load */) { deqPtr := deqPtr + 1.U }

  when (mem_grant.fire && fence) { fence := false.B }
} ///end when (io.enableOutstanding)
.otherwise {  /// ! io.enableOutstanding
  when (io.lsq.resp.fire) { enqPtr := enqPtr + 1.U, issPtr := issPtr + 1.U, deqPtr := deqPtr + 1.U }
} ///end when (io.enableOutstanding){…}.otherwise …

TLArbiter.lowestFromSeq(edge, mem_acquire, entries.map(_.io.mem_acquire))        ///A通道
io.flush.empty := deqPtr === enqPtr

///Debug与性能统计从略
} ///end class Uncache
```

把 TileLink 的 A/D 通道和 Uncache 与其上层（实际上是 LoadQueue 和 StoreQueue）的接口加以流水线化，这个设计还是挺巧妙的。

至此我们已经明白，MemBlock 中的 Uncache，其上方是 LoadQueue 和 StoreQueue，其下方则通过一组 MMIOEntry 通向外设。由于 Uncache 与其下方即外设端口有参数协调的绑定，这就蕴含着有个 TLBundle 通向其下方，所以它的外部接口上只要有通向上方的 lsq 就行。这样，我们就把 MemBlock 对外设的访问途径搞清了，下面就可以把注意力集中在通过 DCache 对存储器的访问。

27.2　MemBlock 的总体结构

Uncache 所处的环境和位置给了我们启示，因为对外设即 MMIO 的访问与对内存的访问是平行的。对于存储地址空间的访问，只要目标存在，就要么是通过 Uncache 进行，要么就是通过始于 DCache、实际上是 DCacheWrapper、一直向下的各层模块进行，只不过后者比前者要复杂得多。但是访问的来源却是一样的，对 Uncache 的访问来自 lsq 即 LsqWrappper，那么对 DCacheWrapper 的访问也来自 LsqWrappper。同样的情况我们在 Frontend 中也看到过，在那里是 ICache 与 InstrUncache 并立。

那么对 LsqWrappper 的访问又来自哪里呢？那就是 Load 指令和 Store 指令的执行，来自这两种指令的执行单元 LoadUnit 和 StoreUnit，其实还有 AMO 指令的执行单元 AtomicsUnit，总之就是访内指令的执行单元。我们在本书第二卷中看到访内指令的执行单元不在 ExuBlock 中，是另有安排的，实际上就是安排在 MemBlock 中。而访内指令执行单元的输入，则同样来自指令调度，来自 ReservationStation。需要指出的是，对访内指令的执行同样也可以是脱序（Out-of-Order）的，这种执行次序上的改变不仅可以发生在访内指令与非访内指令之间，也可以发生在访内指令与访内指令之间。显然，如果两条改变了执行次序的访内指令所访问的是不同的内存单元，那就实际上不会有冲突，但如果访问的是同一内存单元，那就可能有冲突了，所以在 MemBlock 内部得要有消除此种冲突的机制。

```
MemBlock
    dcache = LazyModule(new DCacheWrapper())
    uncache = LazyModule(new Uncache())
    MemBlockImp
        loadUnits = Seq.fill(exuParameters.LduCnt)(Module(new LoadUnit))
        storeUnits = Seq.fill(exuParameters.StuCnt)(Module(new StoreUnit))
        stdExeUnits = Seq.fill(exuParameters.StuCnt)(Module(new StdExeUnit))
        atomicsUnit = Module(new AtomicsUnit)
        lsq = Module(new LsqWrappper)
            loadQueue = Module(new LoadQueue)
            storeQueue = Module(new StoreQueue)
        sbuffer = Module(new Sbuffer)
```

MemBlock 的作用，就是把这些执行单元和结构成分创建出来，并做好这些成分之间的互相连接，也做好这些结构成分与作为一个整体的 MemBlock 的外部接口键的连接。在这些连接中，有些是输入端对输出端之间的对接，这都发生在内部结构成分互连；有些则是输入对输入，输出对输出的连接，这些都是内部成分与 MemBlock 外部接口之间的连接。比方说，MemBlock 有个外部接口 issue，这是个输入向量：

```
val issue = Vec(exuParameters.LsExuCnt + exuParameters.StuCnt, Flipped(DecoupledIO(new ExuInput)))
```

这个向量中的元素是 Flipped(DecoupledIO(new ExuInput))，都是来自 ExuBlock 的输入，更确切地说是来自 ExuBlock 中的 Scheduler，是 Scheduler 调度了访内指令投入执行，这才把相应的 ExuInput 发到了 MemBlock，因为访内指令的执行单元都在 MemBlock 中。向量的宽度是 LsExuCnt + StuCnt，其中 LsExuCnt = LduCnt + StuCnt。也就是说 MemBlock 中访内指令的执行单元总数是 LduCnt 加上 2 倍的 StuCnt。这是因为，对于存储