Chipyard 代码导读

梁书豪

目录

1 Chipyard 简介

Chipyard 是一个面向全系统设计和验证，以敏捷性、可配置性和可重用性为目标的 SoC 设计平台。它包含了一套 SoC 常用的硬件组件库，提供了一个基本的 SoC 设计模板，并囊括了 SoC 从设计到出片的一系列工具。通过 Chipyard，用户可以方便地选配处理器核、外设、总线拓扑等，设计完成后一站式导出 RTL 模型，并用模拟器或 FPGA 验证设计的正确性，从而高效地设计自己的 SoC。

根据《SoC 设计方法与实现（第 3 版）》对 SoC 设计方法的分类，SoC 设计可分为基于代码的设计、基于 IP 核复用的设计和基于平台的设计，三者的抽象层次依次提高。最原始的基于代码的设计由于复杂度高、集成度低、迭代时间长，在工业界已经成为历史。基于 IP 核复用的设计是现在普遍使用的方法，它把 SoC 的一个关键部分——IP 核进行了抽象，很大程度降低了设计难度。但随着 SoC 规模的继续扩大，上市时间的进一步缩短，客户需求的多样性增加等因素，仅抽象 IP 核还是无法把设计复杂度降低到一个可以接受的水平，于是就有了比 IP 核复用效率更高的基于平台的设计。目前对于“平台”还没有一个统一的定义，一般认为平台应该包含一个可重用的组件库（“IP 核库”），一个可配置的体系结构框架（“SoC 模板”）和一套综合、仿真、验证工具（“配套软件”）。Chipyard 便是这样一个 SoC 设计平台。

Chipyard 的文档（Welcome to Chipyard's documentation）其实已经写得比较详细了，但它的定位是供开发人员查阅的“技术手册”，适用于已经对 Chipyard 的代码有一定了解的用户；对于新手，该文档的学习曲线有点过于陡峭了。因此，为了让新手也能顺利入门 Chipyard，我写了这份《Chipyard 代码导读》。我不会按 Chipyard 文档的顺序来写，而是按一种可以舒服地接受 Chipyard 的概念的顺序。当然，一些内容由于 Chipyard 的文档已有，我只会给出引用，需要读者自行阅读。

2 FIRRTL 基础

Chipyard 的绝大部分代码是用 Chisel 写成，而 Chisel 和 Firrtl 又是紧密相关的；Chipyard 不仅把 Firrtl 当做 IR，还用了 Firrtl 库中的很多方法，所以我需要首先介绍一下 Firrtl。

2.1 FIRRTL 简介

Firrtl（Flexible Internal Representation for RTL）是一个多层次的用于描述数字电路设计的 IR（Intermediate Representation）。Firrtl 的部分功能和其他的 RTL 设计语言（如 Verilog）类似，都是在描述电路的 RTL 模型；但 Firrtl 的一个显著特点是它允许在电路层面上进行编译转换。借助语法中多层次的表达能力，Firrtl 可以在从抽象到具体的各个层次中变换，在这个过程中实现简化、技术映射、验证等目的。

我在知乎看到一个问题：Firrtl 在基于 Chisel 的项目开发中的意义，题主的疑问是为什么 Chisel 要先 elaborate 成 Firrtl 再转换到 Verilog，Chisel 完全可以直接转换为 Verilog，再让 EDA 工具去优化 RTL 模型。这个问题其实关乎到 Firrtl 的野心，Firrtl 并不只是想做一个简单的 IR，它希望完全替代掉当前的 RTL 后端语言（如 Verilog、Netlist）。Firrtl 的愿景是，以后电路设计者只需写高层次语言（如 Chisel，SystemC），而 EDA 后端实现者只需优化 Firrtl，这样处在中间尴尬位置的 RTL 语言就可以退休了；硬件设计就会变成和软件设计一样，大部分人都在写高级语言，只有少部分实现者需要接触底层架构。

2.2 FIRRTL 的语法

Firrtl 其实非常易读，它和 Verilog 的很多概念是类似的，因此我不在这里讲解，感兴趣的读者请自行阅读 Specification for the FIRRTL Language。基本上只要了解以下 3 个概念即可：

1. 设计哲学（第 1 章）：Firrtl 的 High/Middle/Low 层次与 Lowering 过程
2. 模块（第 3 章）：`Circuit` 、`Module` 与 `Instance`
3. 连接（第 5 章）：连接（ `<=` ）与部分连接（ `<-` ）

2.3 FIRRTL 中的变换

作为一个 IR，变换（transform）是 Firrtl 的重中之重。这也是 Firrtl 的定义中没有说明的内容，因为如何变换是和实现相关，和定义无关，因此这里重点进行讲解。当然我也只是讲个大概，想深入了解变换或自己写变换的读者可在 Firrtl 官方仓库首页上找到更多资料。

首先，Firrtl 定义了一个变换的基类 `TransformLike`（firrtl.options.TransformLike），如下图所示。Firrtl 中的变换是“同类变换”，即变换前后都是同一类型 `A`，只是内容变了。`TransformLike` 本身是一个抽象类，它派生出两个类：`Transform` 和 `Phase`，旨在限定 `A` 的类型，分别用于 IR 的变换和编译的变换。这两个类比较容易混淆，因为它们都是做变换，只是变换的对象不一样；下面通过一些例子来辨析这两个类。

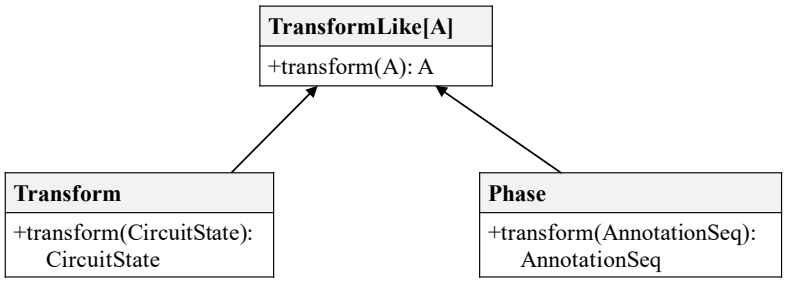

Figure 1: *TransformLike 和它的派生类*

先看 `Transform`（firrtl.Transform），它的几个典型的派生类如下图所示。`Transform` 作用的对象是 IR 描述的电路（`CircuitState`），也就是 Firrtl 的内存形式。如前文所述，所谓 IR 变换便是在各个层次间转换（但通常是从高到低的单向转换）和进行简化、验证等操作。可以看到，在高层形式（接近 Chisel）上，`Transform` 做的事情是诸如推断数据类型和宽度这样的高层次操作，经过这些操作，IR 就会消去不确定的内容；在中层形式上，`Transform` 仍然需要进一步消除 IR 中的抽象，例如展开 Bundle 连接等，便于映射到不支持此类抽象的 RTL 语言；而到了低层形式（接近 Verilog），`Transform` 就会做一些优化，这里和软件编译有些类似，如常量传播、消除死代码等，感兴趣的读者请参阅 VLSI 算法相关内容。

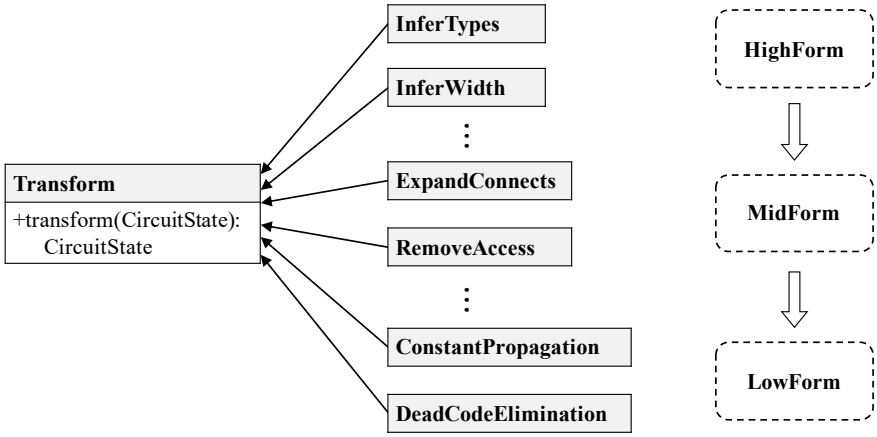

Figure 2: *Transform 和它的典型派生类*

再看 `Phase`（firrtl.options.Phase）。以 Chisel 生成 Firrtl 的过程为例，这其中涉及到的一些 `Phase` 如下图所示。`Phase` 的作用对象是 `AnnotationSeq`（注释序列），这是一种描述编译过程中间信息的类，相当于外部传递给编译器的“参数”；`AnnotationSeq` 也是一个复杂的类，先忽略不谈。我们知道，Chisel 生成 Firrtl 的过程无非是先检查数据类型和宽度的对应，然后运行 Chisel 代码以 elaborate，接下来就是输出 Firrtl（`.fir`）和额外的注释（`.anno.json`），以及关于技术映射的描述文件（`.rom.conf`），最后还可以生成一些自动化测试脚本（`.d`）。这些步骤都是编译过程中的大的阶段，因此用 `Phase` 来描述。可以看出，`Phase` 的范围比 `Transform`

大，一个 `Phase` 中常常包含了多个 `Transform` ；但这并不是说 `Phase` 的复杂度比 `Transform` 高，我的意思是 `Phase` 的逻辑层次更高，显然 `Transform` 的算法应该更复杂。

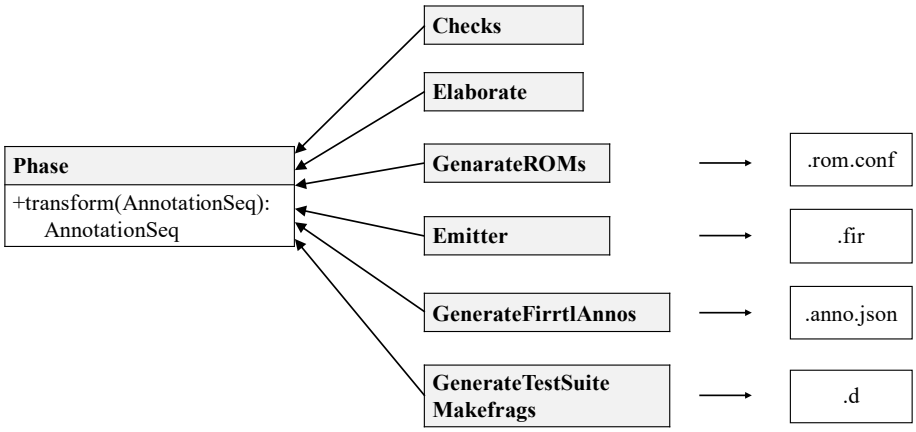

Figure 3: *Phase 和它的典型派生类*

另外还有一种特殊的 `Phase` ，叫做 `Stage` （firrtl.options.Stage），如下图所示。Stage 的作用对象也是 `AnnotationSeq` ，它无非比 `Phase` 多了命令行解析功能（ `shell` ），从而可以接受命令行参数。在以命令行的方式构建 Chipyard 时， `Stage` 会被作为 Scala 的主类来调用，主类解析命令行参数后再调用各个进行实际操作的 `Phase` 。显然，命令行调用也是编译步骤的一部分，这再次说明了 `Phase` 的定位。

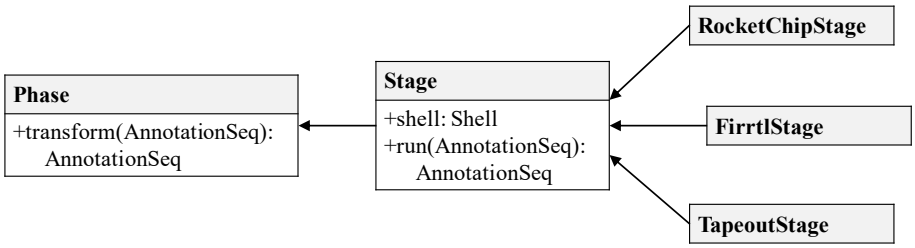

Figure 4: *Stage 和它的典型派生类*

3 TileLink 基础

TileLink 是 Chipyard 使用的总线协议，但它不仅是一个协议，还提供了整个互连的框架，深刻影响了 Chipyard 的代码格局，是读 Chipyard 不可不知的部分。Chipyard 的官方文档 9. TileLink and Diplomacy Reference 写得很好，强烈建议读者把这一章全部读完，不会很长。我在这里只是提一些重点，和文档没有提到但是也比较重要的内容。

3.1 Diplomacy 与 TileLink

Diplomacy 和 TileLink 是 Chipyard 的互连系统中紧密合作的两个组件，其中 Diplomacy 是一个协议无关的互连框架，TileLink 则是一个面向 SoC 的总线协议。但在当前的实现中，TileLink 包装了 Diplomacy，把自己变成了一个同时具有协议和互连框架的完整的库；所以 Chipyard（包括 Rocket 和 Boom）的代码只需要和上层的 TileLink 打交道，不需要关心下层的 Diplomacy，但我们仍然不应该忘记 Diplomacy 的存在。关于两者的设计理念和两者关系的更多讨论可以看 CARRV 2017 上的这篇论文：*Diplomatic Design Patterns: A TileLink Case Study*，相信看完之后会对整个互连系统都有更深的认识。

作为一个互连框架，Diplomacy 最大特点是可以自动传递参数，所有独立的参数只要定义一次，它就会自动传播到网络中的每个节点。例如，假设我们在 Cache 侧定义了总线宽度是 64B，那么就可以在 Core 侧“查询”

到 64B 这个参数，不用重复定义；Core 查询到总线宽度为 64B 之后就可以定义自己的接口宽度，因为 Chipyard 里的模块都是可配置的，Core 根据接口宽度就可以进一步配置传递消息需要的拍数等；这很符合 Chisel 这样的 HLS 语言的设计思想。Diplomacy 的自动传递参数就引出了 *Two-Phase Elaboration* 的概念。我们过去理解的 Elaboration 就是一步把软件代码修饰成硬件代码，但在 Two-Phase Elaboration 中多了一个参数传递的阶段：在第一阶段，先让各个节点定义的参数在网络中充分传递，同时检查参数之间是否存在冲突（例如地址冲突，读写权限冲突）；在第二阶段，每个节点都得到其所需的参数后，再正式 elaborate 各个节点上的模块。

为了延迟模块的 elaboration，Diplomacy 利用了 Scala 的 `lazy` 关键字，定义了一个 `LazyModule` 接口。所有和 TileLink 直接连接的模块都要实现这个接口，实现方式为分别继承 `LazyModule` 和 `LazyModuleImp` 这两个类，在 `LazyModule` 中只对属于自己的节点参数进行设置，在 `LazyModuleImp` 中再获取完整的节点参数和定义模块逻辑。例如，下面是从 TCAM.scala 中摘录的代码：

```
class TCAM(...)(implicit p: Parameters) extends LazyModule {
  val node = TLHelper.makeManagerNode(beatBytes, TLSlaveParameters.v1(...))
  ...
  lazy val module = new TCAMModule(this)
}
class TCAMModule(outer: TCAM) extends LazyModuleImp(outer) {
  val (tl, edge) = outer.node.in(0)
  ...
  tl.d.valid := acq.valid
  acq.ready := tl.d.ready
}
```

为了实现一个 TCAM 模块，我们需要先定义一个继承 `LazyModule` 的 `TCAM` 类，在这个类中设置 TileLink 节点（`makeManagerNode`）；再定义一个继承 `LazyModuleImp` 的 `TCAMModule` 类，在其中实现 TCAM 的真正逻辑。`TCAMModule` 作为一个 `lazy val module` 变量在 `TCAM` 中出现，显然，由于此时 `module` 没有被访问，`new TCAMModule(this)` 并不会被执行。等到第一阶段 elaboration 结束后，`LazyModule` 才会自动访问 `module`，导致 `TCAMModule` 被实例化（在 Chisel 中，实例化才会触发 elaborate）。这时在 `TCAMModule` 中，我们就可以查询节点的所有参数（`edge`），和对 TileLink 的通道（`tl.d`）进行访问了。

这种接口 + 实现的设计范式在 Java 里其实也非常常见，只不过在 Java 里是为了让接口与实现分离，在这里是为了延迟 elaboration。需要再次提醒读者的是，千万不要认为这些 LazyModule 是多此一举或是某种可做可不做的优化，所有和 TileLink 直接连接的模块都*必须这样写*，否则节点在初始化时可能还没有得到完整的参数。当然，不和 TileLink 直接连接的模块就不用这样写，因为它们的父模块已经是 `lazy` 的了，它们可以放心地使用父模块的参数。

TileLink 是一个片上共享内存互连协议，它针对片上（而不是片外）通信做了优化，支持多层次组合，并且确保在事务层面不会死锁。关于 TileLink 的设计思想、组成架构、消息格式等，SiFive TileLink Specification 这本手册已经做了详细描述，我不在此重复，建议读者读一下它的第 2 章（*Architecture*）。TileLink 协议本身没有什么好说的，因为这个协议和其他总线协议相比并没有太特殊之处，我们也不是协议设计人员，很难评价它的好坏。作为 Chipyard 的使用者，我们只要会用 Rocket 提供的 TileLink 库就行了，所以接下来两节我都会关注 TileLink 的实现。

3.2 TileLink 的节点

TileLink 节点类型是实现和原始定义差别比较大的地方；9.1. TileLink Node Types 这篇文档介绍了当前实现中的几种节点类型，但它没有说清楚节点之间的关系，所以我在本节再对节点类型做一下归纳，

首先在 TileLink 的原始定义中并没有区分节点类型，所有节点都叫做 Agent。它区分的只是节点和连接之间的接口（IF），根据是发起请求还是接收请求，接口分为 Master IF 和 Slave IF，如下图所示：

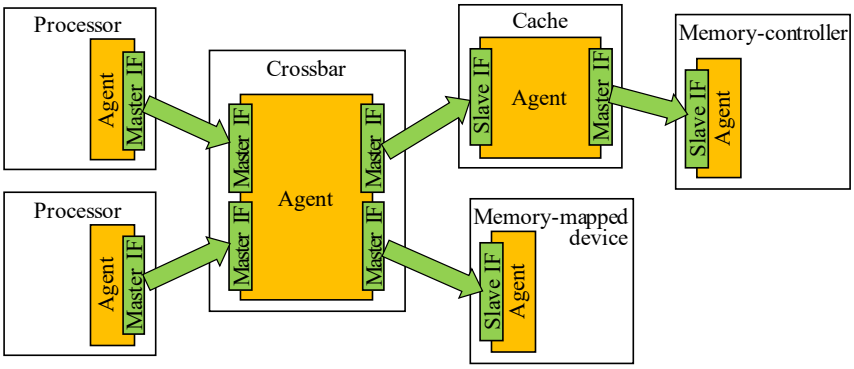

Figure 5: 一个示例的 *TileLink* 网络拓扑

但到了实现中，接口的概念就被融合到节点里了，接口不再显式定义，而是节点根据接口的不同被划分为不同的类型。Diplomacy 提供的基础节点定义在 `diplomacy/Nodes.scala` 中，每个节点都有详细的注释，故我不在此赘述；我把它们的继承关系画成了如下这张图，供读者参考：

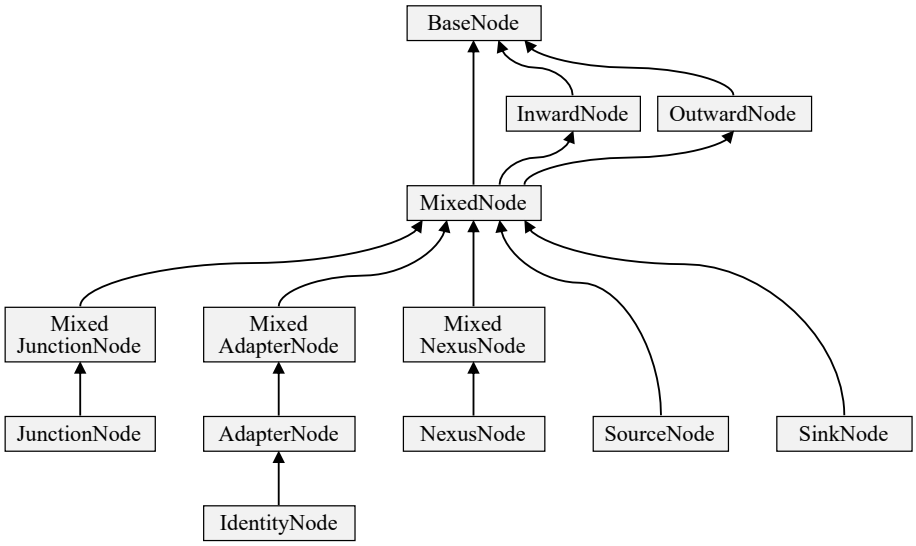

Figure 6: *Diplomacy* 节点的继承关系

注意到，这些节点由于是 Diplomacy 定义的，所以都是没有指定协议的“元节点”；各个协议需要在此基础上添加协议相关的实现，从而得到具有实际功能的节点。例如，TileLink 在 `tilelink/Nodes.scala` 中继承了这些类，得到诸如 `TLClientNode`（继承 `SourceNode`）、`TLManagerNode`（继承 `SinkNode`）这样的以 `TL` 开头的节点；AXI4 则是在 `amba/axi4/Nodes.scala` 中定义了属于自己的节点。9.5. Diplomatic Widgets 还介绍了一些经过多层封装的高级节点，供用户方便地调用。

关于节点的命名，一件神奇的事情是 TileLink 协议手册中的 Master IF 在实现中对应的是 `TLClientNode`，而 Slave IF 在实现中对应的是 `TLManagerNode`，也就是主仆关系反过来了。但是节点的 `Parameters` 的名字还是叫 `Master` 和 `Slave`，于是我们经常会在代码中看到这样的用法：

```
val node = TLManagerNode(Seq(TLSlavePortParameters.v1(...)))
```

这行代码把 `Slave` 的参数传给了 `Manager` 节点……不过想想也没错，`Manager` 节点通常是存放资源的地方（如 Cache），所以叫“管理员”，`Client` 节点通常是请求数据的一方（如 Core），所以叫“客户”；但是请求关系还是 `Client` 向 `Manager` 请求，所以 `Manager` 的接口才是 `Slave`，可能是翻译成中文的时候比较容易引起误解。当然，各大 Git 托管网站把主分支改名为 `main` 是 20 年的事，而 TileLink 的协议手册 17 年就写好了，不知道 Rocket 的开发者们有没有改名的计划。总之，上面举的那张拓扑图到了实现中就会变成这个样子：

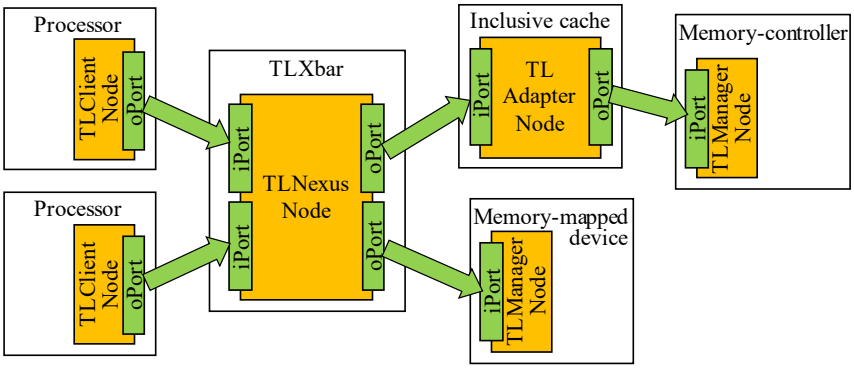

Figure 7: 一个示例的 *TileLink* 网络拓扑的实现

3.3 TileLink 的连接

如前面所述，在 `LazyModuleImp` 中，用户可以从已经完成参数传递的 `node` 上获取 `tl: TLBundle` 和 `edge: TLEdge` 两个对象，其中 `tl` 就是 TileLink 在当前节点上的 I/O 接口，对其进行读写操作就可以收发消息，这个比较好理解；`edge` 根据字面意思看似指的是连接当前节点的边，某种程度上确实是这个意思，但它其实并不具备读写功能，因为节点的 I/O 已经通过 `tl` 进行了，我们不需要再多一个 `edge` 用来做 I/O。事实上，`edge` 的真正作用是记录这个节点乃至整个地址空间的配置，和提供一些收发消息所需的辅助函数。下面举一些例子就容易明白了。

首先，`edge` 最常见的用法就是从中获取 `bundle: TLBundleParameters` 这一成员，从而确定 I/O 接口的宽度，这正是 Diplomacy 的 Two-Phase Elaboration 最基本的特征。例如，在 Boom 的 DCache 中，我们就可以见到大量的用 `edge.bundle` 配置 I/O 的例子（`dcache.scala: 24-35`）：

```
class BoomWritebackUnit(implicit edge: TLEdgeOut, p: Parameters) extends ... {
  val io = new Bundle {
    val req = Flipped(Decoupled(new WritebackReq(edge.bundle)))
    ...
    val release = Decoupled(new TLBundleC(edge.bundle))
  }
```

其中 `req` 和 `release` 都需要 `edge.bundle` 来确定宽度。

第二，`edge` 提供了一些静态函数用以确认一个地址的性质，从而在请求端就处理例外的请求。例如，在 Boom 的 TLB 中，我们就用 `edge.manager.fastProperty` 这个函数确定了当前地址是否可 Cache、是否有 RWX 权限、是否支持逻辑和算数的原子操作等（`tlb.scala:158-166`）：

```
  def fastCheck(member: TLManagerParameters => Boolean, w: Int) =
    legal_address(w) && edge.manager.fastProperty(mpu_physaddr(w), member, ...)
  val cacheable = widthMap(w => fastCheck(_.supportsAcquireT, w) && ...)
  val prot_r    = widthMap(w => fastCheck(_.supportsGet, w) && pmp(w).io.r)
  val prot_w    = widthMap(w => fastCheck(_.supportsPutFull, w) && pmp(w).io.w)
  val prot_al   = widthMap(w => fastCheck(_.supportsLogical, w))
  val prot_aa   = widthMap(w => fastCheck(_.supportsArithmetic, w))
  val prot_x    = widthMap(w => fastCheck(_.executable, w) && pmp(w).io.x)
```

一旦 TLB 发现当前请求的地址不支持当前请求的操作，它就可以立即引发异常，不用等到末端设备返回错误。这里我必须强调，`edge` 的函数都是*静态函数*，也就是不需要在总线上发起请求，而是直接用本地的逻辑进行判断。由于节点的参数已经在第一阶段的 elaboration 中充分传递，所以每个节点都知道整个地址空间中的每个段都有什么样的性质。当然，对于 TLB，这里能够确定的只是在硬件层面上固化的性质，软件运行时配置的性质还需要读页表项来确定。

第三，`edge` 提供了很多辅助收发消息的函数（显然也是静态函数），它们在 9.3. TileLink Edge Object Methods 中都有详细介绍，我就不在这里复述了。由于我对 Boom 的访存系统比较熟，这里再举一个 MSHR 的状态机中 Refill 操作的例子（`mshrs.scala:211-231`）：

```
} .elsewhen (state === s_refill_req) {
  io.mem_acquire.valid := true.B
  io.mem_acquire.bits  := edge.AcquireBlock(...)._2
  when (io.mem_acquire.fire()) {
    state := s_refill_resp
  }
} .elsewhen (state === s_refill_resp) {
  when (edge.hasData(io.mem_grant.bits)) {
    io.mem_grant.ready := io.lb_write.ready
    io.lb_write.valid  := io.mem_grant.valid
    ...
}
```

Refill 在请求和回复阶段分别使用了 `edge.AcquireBlock` 和 `edge.hasData` 这两个函数。当发生 Cache miss 时，MSHR 就要通过 `AcquireBlock` 构造一个请求，向下级 Cache 请求所需的数据。至于为什么要检查这个请求是否包含数据，了解 MESI 协议的读者就知道，Boom 所使用的 MESI 协议中 Cache miss 未必是数据不存在，也可能只是缺少权限；所以这个请求可能并不需要返回数据，只需要申请一个权限，通过 `hasData` 就可以判断我们是否需要填充 Cache line。

4 Chipyard 的配置系统

Chipyard 的配置系统是一定是初学者的噩梦，我一开始就无比好奇 `Config` 到底是如何通过一个 `case =>` 就被定义了，又是如何通过一个 `p: Parameters` 就传递到了模块里。虽然 Chipyard 的文档提到了这件事（6.8. Keys, Traits, and Configs），但这篇文档讲得很浅，没有说出这套配置系统的本质。

事实上，Chipyard 的配置系统可以清晰地分为两个部分：(1) 基于 Trait 的类修饰系统，和 (2) 基于 Config 的参数系统。把这个本质看清楚了，这套配置系统就很好理解了。下面分别介绍这两个部分。

4.1 基于 Trait 的类修饰系统

我们知道，Scala 支持非常丰富的继承语义，Chipyard 就利用了这一点，通过继承对模块进行修饰。下面便是从 `ChipTop.scala` 中选出的部分代码：

```
class DigitalTop(implicit p: Parameters) extends ChipyardSystem
  with testchipip.CanHaveTraceIO
  with sifive.blocks.devices.i2c.HasPeripheryI2C
  with sifive.blocks.devices.uart.HasPeripheryUART
  ...
class DigitalTopModule[+L <: DigitalTop](l: L) extends ChipyardSystemModule(l)
  with testchipip.CanHaveTraceIOModuleImp
  with sifive.blocks.devices.i2c.HasPeripheryI2CModuleImp
  with sifive.blocks.devices.uart.HasPeripheryUARTModuleImp
  ...
```

其中 `with` 的这些类就是 Scala 的 Trait。可以看到，无论是 `DigitalTop` 还是 `DigitalTopModule` 都被许多 Trait 修饰着，通过这些修饰给它们定义了 `TraceIO`、`PeripheryI2C` 和 `UART` 等组件。如前文所述，直接和 TileLink 连接的模块都需要用写成 `LazyModule` + `LazyModuleImp` 的形式，故这些组件也需要在两个模块中分

别定义。但并非所有模块都需要定义两次，这里我也没搞清楚，似乎是只有在第二阶段还需搭线的才要定义 `LazyModuleImp` ，否则定义第一个即可。

我们发现 Trait 的名称有两种，即 `CanHave` 和 `Has` 。从字面上看， `CanHave` 的意思是这个模块可以有，但不一定有； `Has` 的意思是这个模块一定有。但其实*根本不是*！这两种名称没有任何差别，都是不一定有。事实上，不管点进 CanHave 还是 Has 的 Trait，它们都长成这个形式：

```
trait HasXxx { this: BaseSubsystem =>
  val xxx = p(XxxKey).map { param =>
    LazyModule(...)
  }
}
```

也就是需要在 Config 参数系统（见下一节）里定义过才会有，否则只是一个空接口。这是 Chipyard 的配置系统比较混乱的地方，可能是历史原因所致。

4.2 基于 Config 的参数系统

基于 Trait 的修饰系统虽然很直观，但我们每次修改都要直接对类所在文件做修改，导致需要重新编译文件，这不符合参数化的思想。于是 Chipyard 提供了基于 Config 的参数系统，用这种方法定义一个参数时，用户虽然需要在多处地方写定义，但可以把参数都集中到一个单独的文件里（故名"参数系统"），修改时不用去改类的代码。这套参数系统涉及 3 个重要概念： `Parameters` 、 `Key` 和 `Config` （事实上 `Config` 是 `Parameters` 的子类，不过概念上可以把它们当成两个东西），这 3 个概念共同组成了基于 Config 的参数系统。

首先说 `Parameters` ，Chipyard 中几乎每个模块都会有一个 `(implicit p: Parameters)` 的参数域，这借用了 Scala 的 `implicit` 功能，让用户不用每个类都写 `p` ，它会自动一层一层地传递。模块里面的 `Parameters` 其实就相当于一个字典，而 `Key` 就是查询这个字典所需的键。以 `HasPeripheryUART` 为例：

```
trait HasPeripheryUART { this: BaseSubsystem =>
  val uartNodes = p(PeripheryUARTKey).map { ps =>
    UARTAttachParams(ps).attachTo(this).ioNode.makeSink()
  }
}
```

其中 `this: BaseSubsystem =>` 是 Trait 的 Self-Typing 语法，限定这个 Trait 只能修饰 `BaseSubsystem` 及其子类，于是这里的 `p` 指向的就是 `BaseSubsystem` 中的 `Parameters` 实例。查询 `UARTParams` 的方法很简单，就是用 `PeripheryUARTKey` 作为键查询 `p` ，若 `UARTParams` 在 `p` 中被定义过，就会返回一个 `Some(UARTParams)` 对象，否则返回 `None` ；这里借助了 Scala 的 `Option` 特性，无论返回 `Some(UARTParams)` 还是 `None` ， `map` 都不会报错，只是为 `None` 时 `map` 得到一个空 `Seq` ，使得此处的 `uartNodes` 只是一个空接口，不会出现在硬件中。

那么如何往 `p` 中添加 `UARTParams` 呢？这时就需要 `Config` 登场了。还是以 UART 为例，我们首先需要为 UART 定义一个新的 `Config` ，这些继承 `Config` 的类通常都叫 `WithXxx` ，例如这里叫 `WithUART` ：

```
class WithUART(baudrate: BigInt = 115200) extends Config((site, here, up) => {
  case PeripheryUARTKey => Seq(
    UARTParams(address = 0x54000000L, nTxEntries = 256, nRxEntries = 256, ...))
})
```

`WithUART` 这个类本身没有内容，它只是在继承 `Config` 时往 `Config` 传入了一个函数，这个函数的函数体又是一个函数（Scala 的 *Partial Function*），它在接收到对应的 `Key` 时（这里是 `PeripheryUARTKey` ）返回 `UARTParams` 。于是，这个定义和 `HasPeripheryUART` 中的查询就对应上了：如果这里定义过这个映射关系，`HasPeripheryUART` 中就能用 `PeripheryUARTKey` 查到对应的 `UARTParams` ，否则就查到 `None` 。需要注意

的是，单纯定义 `WithUART` 并不会让映射生效，我们还需要将它和更高层的 Config 汇合在一起，并在 elaborate 时指定那个最高层的 Config。例如，`WithUART` 先是被汇合到了 `AbstractConfig` 中：

```
class AbstractConfig extends Config(
  ...
  new chipyard.config.WithUART ++
  ...)
```

然后 `AbstractConfig` 再被各个处理器配置的 `Config` 引用，例如 `MediumBoomConfig` ：

```
class MediumBoomConfig extends Config(
  new boom.common.WithNMediumBooms(1) ++
  new chipyard.config.AbstractConfig)
```

从而在顶层配置中生效。

默认情况下，Config 使用的是覆盖策略，也就是对于同一个 `Key` ，上面的配置会覆盖下面的配置。例如，假设我们在 Boom 中再次定义了 UART：

```
class MediumBoomFastUARTConfig extends Config(
  new boom.common.WithNMediumBooms(1) ++
  new chipyard.config.WithUART(BigInt(3686400L)) ++
  new chipyard.config.AbstractConfig)
```

那么最终我们只会得到一个波特率为 3686400Hz 的 UART，`AbstractConfig` 中的 115200Hz 的定义会被覆盖。它的原理是同一个 `Key` 的 `Params` 其实会排成一个链表，`Parameters` 在查询时会从上到下查询，默认情况下查到第一个为 `true` 的为止。但某些情况下我们可能希望多个 Config 进行组合，这时候就需要直接操作 Config 的 `(site, here, up)` 这几个变量。操作过程比较复杂，感兴趣的读者可以看 8.6. Context-Dependent-Environments，我这里只举一个WithBringupPeripherals的例子：

```
class WithBringupPeripherals extends Config((site, here, up) => {
  case PeripheryUARTKey => up(PeripheryUARTKey, site) ++ List(UARTParams(...))
  ...
}
```

这个类就不会覆盖原有的 `PeripheryUARTKey` ，而是在链表后面接上自己的 `UARTParams` 。

另外，`AbstractConfig` 中还有两种特殊的 `Config` ，即 `HarnessBinder` 和 `IOBinder` ，分别用于定义 Harness 组件和 IO 单元（见下一章）。它们的写法和其他的 `WithXxx` 不一样，并且虽然是 `Config` 的派生类（`OverrideIOBinder` 最终继承自 `Config` ），但作用方式更像是 `Has` / `CanHave` 。以 `WithUARTIOCells` 为例：

```
class WithUARTIOCells extends OverrideIOBinder({
  (system: HasPeripheryUARTModuleImp) => {
    val (ports: Seq[UARTPortIO], cells2d) = system.uart.zipWithIndex.map({ case (u, i) =>
      val (port, ios) = IOCell.generateIOFromSignal(u, s"uart_${i}", system.p(IOCellKey), ...)
      (port, ios)
    }).unzip
    (ports, cells2d.flatten)
  }
})
```

它并不通过 `case =>` 定义一个参数，而是依赖于别的参数定义好的 `system.uart` 。根据 Scala `map` 的语法可知，别的参数定义了几个 UART，它就生成几个 `IOCell` ；若 `system.uart` 为空，它不会生成任何 `IOCell` 。这恰恰是 `Has` / `CanHave` 的语义，读者一定要注意区分。

5 Chipyard 的 SoC

Chipyard 作为一个 SoC 设计平台，提供了一个 SoC“设计模板”。它的特点是将 SoC 划分为几个层次，每个层次实现不同的功能；这既让设计更加有条理，也便于同时测试和出片。想要读懂这个 SoC，重点是读懂它的层次。文档 8.1. Tops, Test-Harnesses, and the Test-Driver 对此做了一些介绍，但是这个文档没有画图，也没有结合代码（Chipyard 的代码还挺绕的）。所以这一章我会图文并茂地给大家介绍这个 SoC。

5.1 SoC 概览

由于 Chipyard 的 SoC 可配置性很强，不同的配置会有很大的差别，所以本章我以一个有代表性的 `MediumBoomConfig` 配置为准。我先不加解释地给出这个 SoC 的结构图：

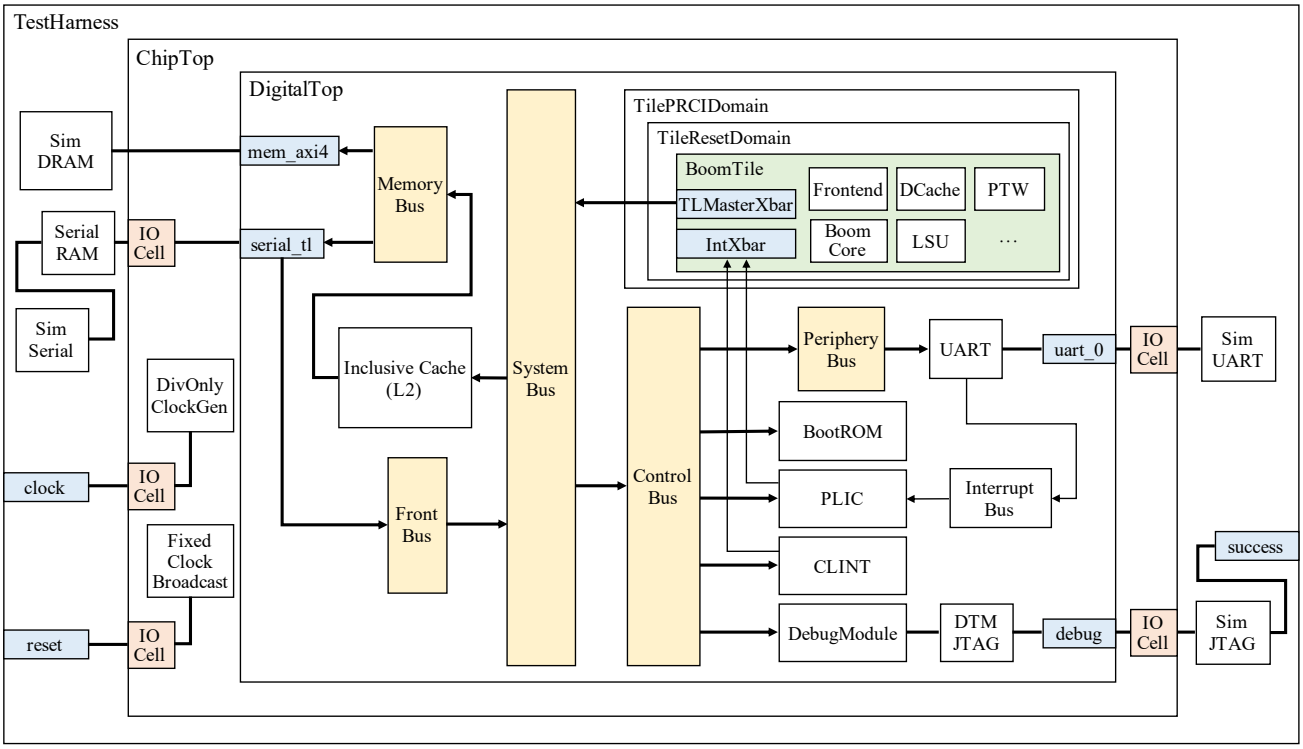

Figure 8: *Chipyard SoC 结构图*

从这张图可以看到，该 SoC 在 Tile 的外部分为 3 个大的层次——TestHarness、ChipTop 和 DigitalTop，SoC 的时钟、总线和外设等都分布在这 3 个层次上。我接下来就会分别介绍这 3 个层次。至于 Tile 的内部结构，我假设读者都有 Core 的设计经验，Tile 对于读者不是一件很难理解的事情，所以不在此讲解。

5.2 TestHarness

TestHarness 是在模拟器环境下用于验证 *DUT*（*Design Under Test*）的外围环境，所谓 DUT 就是图中的 ChipTop。在实现到 FPGA 上时，ChipTop 会连接到真实的硬件模块，但现在为了方便调试，这些模块大多是用 C++ 实现的，故带有 `Sim` 前缀。TestHarness 的 4 个主要模块为：

1. `SimDRAM` ：使用 DRAMSim2 库实现的内存模型，可以比较真实地模拟内存的行为；
2. `SimSerial` ：实现了 RISC-V 定义的 *TSI*（*Tethered Serial Interface*）协议接口，用于加载程序、代理系统调用等，是最主要的调试接口；
3. `SimUART` ；使用 UART 协议进行简单的 I/O；
4. `SimJTAG` ：实现了 JTAG 协议调试接口，主机可接 GDB 对 DUT 上的程序进行运行时调试。

可以看到，除了 `SimDRAM` ，其他 3 个模块都和主机有交互，因为交互是调试必不可少的功能。关于 DUT 如何和主机交互的更多信息请见 8.2. Communicating with the DUT。

至于 TestHarness 是如何定义上述模块的，Chipyard 的代码非常绕，它首先是在 `AbstractConfig.scala` 中定义了一系列的 HarnessBinder 参数，如下所示：

```
class AbstractConfig extends Config(
  new chipyard.harness.WithUARTAdapter ++
  new chipyard.harness.WithBlackBoxSimMem ++
  new chipyard.harness.WithSimSerial ++
  ...
```

根据前面所述，这些 `With` 不一定会生效，还需要定义 `Config` 才能真正被实例化，如何定义的这里就不深究了。总之，假设已经定义了 `Config` ，TestHarness 会用 `ApplyHarnessBinders` 这个函数实例化这些模块，并将它们和 ChipTop 连接在一起：

```
  lazyDut match { case d: HasIOBinders =>
    ApplyHarnessBinders(this, d.lazySystem, d.portMap)
  }
```

其中 `portMap` 就是 ChipTop 对外暴露的 I/O 端口，需要被 `HarnessBinders` 连接；`lazySystem` 指的是 DigitalTop，但是 Chisel 不允许跨层连接 I/O，所以它在这里不是用于连接的，只是提供 DUT 的一些参数。

5.3 ChipTop

ChipTop 是芯片出片（*Tape-out*）时的顶层模块。我们在把 Chipyard 生成的 RTL 模型拿到 FPGA 去综合的时候拿的就是 ChipTop，因为 FPGA 不需要 TestHarness。ChipTop 上其实没有太多内容，它和 DigitalTop 区分开是因为即使我们做的是数字电路，SoC 上仍然有一些处理模拟信号的模块；把这些模块都放在 ChipTop 中可以让纯数字电路的部分（也就是 DigitalTop）更加干净。ChipTop 包含的内容通常有：

1. *IO 单元*（*I/O Cell*）：有一定电路基础的读者就会知道，芯片的管脚是不会直接和微电路连在一起的，因为微电路无法驱动那么大的输出，外部信号的波动也容易烧坏微电路，所以我们需要在管脚处设置 IO 单元，让 IO 单元来放大输出信号和保护输入信号；
2. *时钟信号接收器*（*Receiver*）*与多路复用器*（*Multiplexer*）：用于分发时钟和调整频率；
3. *复位信号同步器*（*Synchronizer*）：由于复位信号可以在任意时刻出现，我们需要一个同步器将其和时钟同步；
4. *其他处理模拟信号的 IP 核*：如果该 SoC 需要收发模拟信号（如温度计、麦克风的信号），就可以在这里进行处理。

当前的配置除了 4 都有。

从结构图上看，我们会发现内存接口是没有 IO 单元的，这可能是因为 Chipyard 默认它还会接一个片上的内存控制器？时钟在这里连接了一个 `DividerOnlyClockGenerator` （由 `AbstractConfig.scala:42` 定义），这是一个可以把外部传入的时钟按倍数变慢的模块，但是在这里似乎没什么用，因为倍数就是 1。另外 Chipyard 是允许每根总线有自己的时钟的，但当前配置下所有的设备和总线的时钟都是 100MHz，相当于用同一个时钟。复位信号接了一个 `FixedClockBroadcast` ，这个应该起到的就是同步的作用，但我甚至没找它的定义。说实话我基本没看懂时钟和复位的代码，所以这里写得比较潦草，感兴趣的读者可以自己研究一下。

5.4 DigitalTop

DigitalTop 就是数字电路的顶层了，所有的总线和外设都放在这一层。DigitalTop 在 Chipyard 的代码中又被称为 System，因为它继承自一系列叫做 `XxxSystem` 和 `XxxSubsystem` 的类。DigitalTop 需要关注的重点一是它的总线拓扑，二是它包含的外设（或模块）。先说总线拓扑，Chipyard 默认定义了 5 根总线，介绍如下：

1. *System Bus*：直接和 Core 相连，是 System（DigitalTop）上最重要的总线，也是唯一一根必须实现的总线；当其他总线没有实现时，原本和那条总线连接的设备就会连到 System Bus 上来。
2. *Periphery Bus*：外设总线；由于当前配置只有 UART，所以它只孤零零地连了一个 UART，但其实很多的 MMIO 外设都可以连到上面；文档 6.6. MMIO Peripherals 提供了一个添加自定义 MMIO 外设的例子，该外设连接的就是 Periphery Bus。
3. *Front Bus*：是唯一一根和 System Bus"逆向而行"的总线，因为其他总线对于 System Bus 而言都是 Slave，而 Front Bus 对于 System Bus 是 Master；Front Bus 的作用是将外部访问总线的请求传递到 ChipTop 的总线系统中，否则外部将没有其他方法*主动*发起请求，只能被动接受请求；例如在当前设计中，`SimSerial` 就通过 Front Bus 来访问内存，达到调试的目的。
4. *Memory Bus*：连接内存控制器的总线；Memory Bus 上的请求必须是已经没有一致性约束的请求，也就是任何一致性约束都必须在 Memory Bus 之前解决，因此我们在 System Bus 和 Memory Bus 之间放置了 Inclusive Cache（L2），用它解决一致性的问题。
5. *Control Bus*：连接比较重要的外设的总线；Control Bus 连接了 BootROM、PLIC、CLINT、DM 这几个外设，它们都是和 Core 关系比较密切的外设，所以放在离 Core 更近的 Control Bus 而不是 Periphery Bus 上；这样设计还有能耗的考量，虽然现在所有的外设都使用一样的时钟频率，但在实际产品中我们常常希望降低不频繁使用的外设的时钟频率以降低能耗，这就可以通过给 Periphery Bus 和连接到其上的外设配置更低的时钟频率来实现。

注意到，DigitalTop 上还有一根叫 Interrupt Bus 的总线，但它不是 DigitalTop 总线家族的一员，因为它不能访问内存空间，只能传递一些简单的中断信号。读者可以在 `BaseSubsystem` 类中找到这 5 根总线的定义，在 `CoherentMulticlockBusTopologyParams` 和 `HierarchicalMulticlockBusTopologyParams` 这两个参数模板中找到各个总线以及 L2 之间互连关系的定义。

DigitalTop 上的主要外设（或模块）介绍如下：

1. *BootROM*：一个只读存储器，用于存放系统上电后所需的启动代码；系统上电时 Core 会将 PC 指向 BootROM 中的一个地址（称为 *Reset Vector*），然后执行 BootROM 中的启动程序；的关于启动流程的更详细说明请见 6.12. Chipyard Boot Process。
2. *Platform-Level Interrupt Controller*（*PLIC*）：RISC-V 自己定义的一套外部中断处理机制，用于处理外设等引发的中断，例如 UART 收到消息的中断；PLIC 的规范目前还在开发中，可见 RISC-V Platform-Level Interrupt Controller Specification。
3. *Core Local Interruptor*（*CLINT*）：也是 RISC-V 自己定义的一套中断处理机制，但是是用于处理 Core 自己配置的中断，例如时钟中断；CLINT 的规范可见 RISC-V Advanced Core Local Interruptor Specification。
4. *DebugModule*（*DM*）：调试模块，可以根据外部请求向 Core 发出中断，让其跳至调试程序处；DM 自带一个很小的 RAM，用于存放从外部传进来的临时调试代码；易知，由于 DM 无法直接向内存拷贝代码，它必须用这个 RAM 来存放临时代码，然后才能借这些代码做更复杂的操作。
5. *Debug Transport Module*（*DTM*）：调试请求转换模块，用于将某种协议的调试请求转换为对 DM 的命令；事实上，Chipyard 支持用多种协议来访问 DM，所以 DTM 的目的就是对内统一这些协议，让不同的协议均可共用一个 DM；更多关于 DM 和 DTM 的信息可见 RISC-V External Debug Support 手册。
6. *Inclusive Cache*（*L2*）：SiFive 提供的一致性 LLC（*Last-Level Cache*），使用目录和 MESI 协议实现了内存一致性；Chipyard 目前只能配置 L2 Cache，关于 L3 可以看香山的 Non-inclusive L2/L3 Cache。

这些外设（或模块）由 `AbstractConfig.scala:44-60` 处的一系列 `With` 语句包含进系统中。

6 Chipyard 的构建流程

Chipyard 提供了现成的 Makefile 供用户构建多种形式的 DUT，当然，以模拟器为主；文档 2. Simulation 详细介绍了用命令行构建各种模拟器的方法。Chipyard 提供的 Makefile 虽然方便，但是过度的包装也会让用户看不到构建过程中的细节，忽略一些可能出现的问题。因此，本章我将以构建 `MediumBoomConfig` 的 Verilator 模

拟器的流程为例说明 Chipyard 的构建流程。

这里向读者推荐一个看 Makefile 的好方法，就是真的运行一遍构建流程，然后把 Makefile 的输出拷贝到文本编辑器里，把其中的命令整理成易读的格式，再一条一条地分析；这比直接看 Makefile（尤其是 Chipyard 层层嵌套的 Makefile）要清晰得多，我在写本章的时候就是这么做的。

6.1 准备文件

由于 Chipyard 的代码不仅来自 Chipyard 仓库，还来自很多别的仓库，所以构建流程的第一步是先将所需的文件从各个仓库拷贝到工作目录下。这一部分的命令由 `mkdir` 、 `cp` 和 `echo` 组成。这一步完成后，我们会在工作目录中看到如下文件：

```
$ ls $CHIPYARD/sims/verilator/generated-src/chipyard.TestHarness.RocketConfig
EICG_wrapper.v  SimSerial.cc      mm.cc           remote_bitbang.cc  testchip_tsi.h
SimDRAM.cc      bootrom.rv32.img  mm.h            remote_bitbang.h   verilator.h
SimDTM.cc       bootrom.rv64.img  mm_dramsim2.cc  sim_files.f
SimJTAG.cc      emulator.cc       mm_dramsim2.h   testchip_tsi.cc
```

这些文件可以分为：

1. 来自 *chipyard* 的 `emulator.cc` ，是 Verilator 要求的包装文件，可参考 Example C++ Execution。
2. 来自 *rocket-chip* 的 `SimDTM.cc` 、 `SimJTAG.cc` 和 `remote_bitbang.cc` ，用于模拟调试接口。
3. 来自 *testchipip* 的 (1) `SimSerial.cc` 和 `testchip_tsi.cc` ，同样用于模拟调试接口，但是协议不同；(2) `SimDRAM.cc` 、 `mm_dramsim2.cc` 和 `mm.cc` ，用于模拟内存；(3) `bootrom.rv32.img` 和 `bootrom.rv64.img` ，作为 BootROM 的内容。

6.2 Elaboration

第二步就是运行 Chisel 代码以 elaborate 我们的设计了。Chipyard 使用 sbt 来自动编译和运行代码，所以首次运行的时候可能需要从线上仓库获取一些库。调用的命令如下所示：

```
$ java ... -jar $CHIPYARD/generators/rocket-chip/sbt-launch.jar ... \
    ";project chipyard; runMain chipyard.Generator ... \
      --name chipyard.TestHarness.MediumBoomConfig \
      --top-module chipyard.TestHarness ..."
```

可以看到，它先用 `java` 启动了 sbt 的包（ `sbt-launch.jar` ），再用 sbt 自动编译和运行 Chipyard 的主类（ `chipyard.Generator` ），同时在命令行参数中指定我们的配置（ `MediumBoomConfig` ）。运行过程中，控制台会输出一些对于我们的设计的描述，主要是设备树文件（*DTS*）、地址空间映射表和时钟频率分配表。运行完毕后，我们会在工作目录看到新增的文件：

```
chipyard.TestHarness.RocketConfig.0x0.0.regmap.json
chipyard.TestHarness.RocketConfig.0x10028000.0.regmap.json
...
chipyard.TestHarness.RocketConfig.anno.json
chipyard.TestHarness.RocketConfig.d
chipyard.TestHarness.RocketConfig.dts
chipyard.TestHarness.RocketConfig.fir
chipyard.TestHarness.RocketConfig.rom.conf
...
```

其中最主要的当然是 `.fir` 文件。至于其他的文件，有的是作为 Firrtl 的附加描述（Firrtl 的语法无法描述电路的一些特殊性质，需要额外的描述文件），有的是用于测试，有的只是单纯的 log 输出而已，这里不一一介绍，感

兴趣的读者可以自己研究。

相比于其他步骤，elaboration 并不是一个很耗时的操作，因为 Chisel 生成 Firrtl 时并不会做优化，只是把整个设计遍历一遍。对于 `MediumBoomConfig` 这样的小型配置，elaboration 只需要约 1 分钟。

6.3 出片

出片（*Tape-out*）在电子设计领域指的是芯片交给制造商之前最终的设计。在 Chipyard 中出片的意思有点偏差，它指的其实是将 Firrtl 转换为 Verilog 这个步骤。得到 Verilog 后，前端设计流程就算完成了，接下来就可以把 Verilog 代码交给后端。这一步中 Chipyard 还会做一件事，就是把 ChipTop 和 TestHarness 拆开。我们知道，软件模拟器需要 TestHarness 才能运行，而 EDA 工具只需要 ChipTop；为了让模拟器和 EDA 工具共用一份 ChipTop，避免测试和出片的内容不一致，Chipyard 把 ChipTop 作为一个单独的模块从 TestHarness 中拿了出来；这样 ChipTop 既可以作为 TestHarness 的外部模块来运行，也可以直接交给 EDA 工具。

Chipyard 使用 Barstools 仓库提供的 `GenerateTopAndHarness` 工具同时完成拆分和转换的任务，调用该工具的命令如下所示：

```
$ java ... -jar /root/chipyard/generators/rocket-chip/sbt-launch.jar ... \
    ";project tapeout; runMain barstools.tapeout.transforms.GenerateTopAndHarness \
      -i .../chipyard.TestHarness.MediumBoomConfig.fir \
      -o .../chipyard.TestHarness.MediumBoomConfig.top.v \
      -tho .../chipyard.TestHarness.MediumBoomConfig.harness.v \
      --syn-top ChipTop \
      --harness-top TestHarness ..."
```

该命令的输入为上一步中 elaborate 得到的 `.fir` 文件，输出为 `.top.v` 和 `.harness.v` 两个文件，分别对应 ChipTop 和 TestHarness。

出片这个步骤非常耗时，实测对于 `MediumBoomConfig` 就需要 8 分钟左右，对于更大的 `MegaBoomConfig` 甚至需要 30 分钟。其实它并没有做什么优化，光是 Firrtl 从高层形式转换为低层形式就可以花很多时间；而且目前 `GenerateTopAndHarness` 的实现是对 ChipTop 和 TestHarness 都要跑一遍 Firrtl 到 Verilog 的转换，所以用时几乎翻倍。当然，优化这个流程并非是一件容易的事，需要开发者对 Firrtl 的算法和实现都有很深的理解。

6.4 技术映射

技术映射（*Technology Mapping*）是电路设计从抽象模型向具体模型过渡的重要步骤，它将实现无关的布尔描述转换为实现相关的器件描述。我们需要进行技术映射的原因是，芯片制造厂商的工艺库不是万能的，它未必能包含所有的门器件（如与非门，或非门），这时就需要把原有设计中不可实现的门技术映射为可实现的门。技术映射的结果通常是网表（Netlist），此时模型已经几乎不具备可读性。

根据前面提到的 Firrtl 的野心，Firrtl 应该是想支持技术映射的，不过目前在 Chipyard 中没有这么用；而且 Chipyard 并不需要做器件的技术映射，因为 Chipyard 只是面向前端设计，技术映射等后端流程可以留给 EDA 工具完成。Chipyard 唯一需要做的就是内存的技术映射，因为 Chisel 对于内存的描述非常抽象（见 Chisel/FIRRTL Memories 对 Chisel 内存模型的介绍）；如果不做技术映射，Firrtl 就会把所有内存都映射为 flip-flop 数组，即使有些内存本来应该映射为 SRAM；这会大大损害芯片的性能。

Chipyard 同样使用 Barstools 库进行技术映射，技术映射在 Barstools 中被叫做 *MacroCompiler*，因为这确实有点像 C 语言里面展开宏的过程；当然这只是在对内存做技术映射的情况下，在 VLSI 中做器件的技术映射是及其复杂的。4.6. Barstools 介绍了 Barstools 库做技术映射的一些细节，可作为延伸阅读。下面展示的是用 MacroCompiler 对 ChipTop 做技术映射的命令：

```
$ java ... -jar /root/chipyard/generators/rocket-chip/sbt-launch.jar ... \
    ";project barstoolsMacros; runMain barstools.macros.MacroCompiler \
      -n .../chipyard.TestHarness.MediumBoomConfig.top.mems.conf \
      -v .../chipyard.TestHarness.MediumBoomConfig.top.mems.v ... \
      --mode synflops"
```

这句命令的输入为 `.mems.conf` 文件，输出为 `.mems.v` 文件；`--mode` 参数指定为 `synflops` 的意思是把所有的内存都映射为 flip-flop，这是因为我们只生成模拟器，可以不使用 SRAM。

注意，由于上一步我们已经把电路拆成了 `top` 和 `harness` 两部分，所以这一步需要对 ChipTop 和 TestHarness 分别做技术映射；当然，对 TestHarness 做技术映射的命令和 ChipTop 基本相同，故不在这里展示。技术映射不需要对整个模型进行，它只针对其中的内存，因此非常快，只需要数秒。

6.5 构建 Verilator

Verilator 是一个开源的信号级模拟器，它使用编译后执行的方法，先将输入的 Verilog 文件转换为 C++，再将 C++ 编译为模拟器。编译后执行让 Verilator 模拟效率非常高，因为它在将 Verilog 转换为 C++ 的时候就可以做逻辑优化和多线程划分，不需要运行时再做。Verilator 的构建流程分为两步，第一步是用 `verilator` 分析工具将 Verilog 文件转换为 C++，同时生成 Makefile；第二步是运行 Makefile 得到模拟器的可执行文件。

调用 `verilator` 的命令如下：

```
$ verilator --cc --exe \
    -CFLAGS "..." -LDFLAGS "..." \
    --threads 8 --threads-dpi all -O3 --x-assign fast --x-initial fast \
    --output-split 10000 --output-split-cfuncs 100 --assert -Wno-fatal --timescale 1ns/1ps \
    --max-num-width 1048576 ... \
    --top-module TestHarness ... \
    .../chipyard.TestHarness.MediumBoomConfig.top.v \
    .../chipyard.TestHarness.MediumBoomConfig.harness.v \
    .../chipyard.TestHarness.MediumBoomConfig.top.mems.v \
    .../chipyard.TestHarness.MediumBoomConfig.harness.mems.v  \
    -o /root/chipyard/sims/verilator/simulator-chipyard-MediumBoomConfig ...
```

可以看到，我们需要给 `verilator` 指定一些模拟器的参数，和在前几步中生成的 `.v` 代码，包括 ChipTop、TestHarness 和内存的技术映射。`verilator` 的用时取决于设计的复杂程度，对于 `MediumBoomConfig` 用时为 3 分钟左右。

运行 Makefile 的命令如下：

```
$ make VM_PARALLEL_BUILDS=1 \
    -C .../chipyard.TestHarness.MediumBoomConfig \
    -f VTestHarness.mk
```

这个命令没有什么特殊的地方，就是一个普通的 `make` 。C++ 编译可以充分利用主机的多核性能，在 32 核服务器上这一步只需要不到 1 分钟。

最终我们就得到了编译好的模拟器：`simulator-chipyard-MediumBoomConfig` 。

7 案例学习：在 L2 Cache 上实现预取

预取（*Prefetch*）是隐藏 cache 延迟的重要方法，可以在不增加 cache 大小的情况下提升 cache 性能。Chipyard 使用的 Inclusive Cache（以下简称 L2）是 SiFive 公司提供的一个 cache 实现，它可以正确支持多核一致性，但不包含预取功能，这给我带来了做优化的机会。

事实上，L2 的性能对系统性能有不小的影响。下图是我在 FPGA 上做的增加内存延迟的实验。由于 FPGA 的内存频率（200Mz）比 Core 的频率（50MHz）还高，所以在不给内存增加延迟的情况下 L2 未命中的代价就会很低，某种程度上相当于 L2 无限大；我们有必要在内存和 L2 之间添加延迟，使得 L2 的未命中代价更接近真实情况。可以看到，增加延迟后多个程序的 IPC（即性能）都有显著下降，有的程序甚至下降了 50% 以上；反过来说，在真实系统中，如果我们能通过预取隐藏这些延迟，就有可能实现 100% 的性能提升。

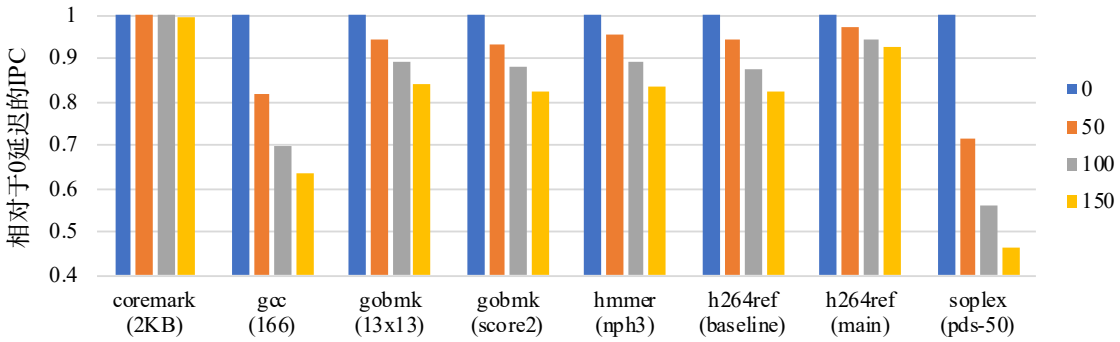

Figure 9: *评测程序的 IPC 与增加内存延迟（cycle）的关系*

本章我将展示我在 L2 上实现预取接口与一个简单的预取器的过程，作为读者研究 Chipyard 的案例学习。虽然我没有实现 SOTA 的预取器，但这个接口是通用的，读者可以在我的基础上实现更复杂的预取器。我所有的代码都开源在 GitHub - lshpku/sifive-inclusivecache-prefetch 上，欢迎读者取用。

7.1 SiFive Inclusive Cache 概述

L2 的代码写得还算不错，至少它的架构还是挺优美的，以至于香山的 Non-Inclusive Cache 也照着它写。这个 cache 本来是计划同时支持 MLC（L2）和 LLC（L3）的，代码中已经可以看到为了同时支持两者做的努力，但最终它并没有实现 MLC，只实现了 LLC；正好由于 Chipyard 只支持配置到 L2，所以它在 Chipyard 中既是 LLC 也是 L2。此处的 L2 并非现在主流处理器中作为 MLC 的 L2，请读者注意区分。

在 Chiyard 中，L2 的本体并非直接和总线连接，而是先经过一层 `CoherenceManagerWrapper` 的包装，如下图所示。该 wrapper 对输入输出进行了互连、转换、加缓冲等操作，提高了 Inclusive Cache 的通用性，也方便 Chipyard 配置不同的一致性方法（例如 `BufferlessBroadcast` ）。

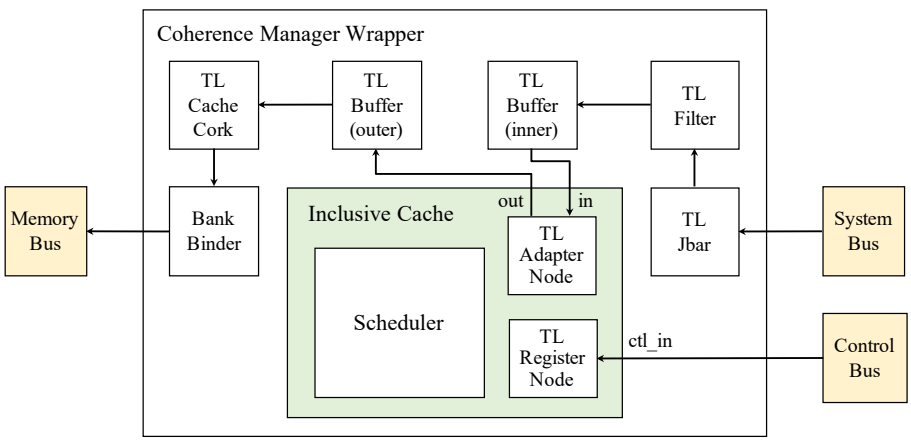

Figure 10: *L2 与总线的连接方式*

L2 的代码是有一定复杂度的，因为硬件实现一致性不是一件简单的事情；读者如果想读源码，我建议先好好读一下教科书上对于多核 cache 的介绍（比如什么是目录，什么是 MESI 协议、什么是 inclusive），还有把 TileLink 的 5 个通道分别做什么搞清楚。关于 L2 的源码讲解我发现一篇很好的博客：rocket-inclusive-cache | Francis's blog（真的写得很好 !），建议读者先去看一下这篇博客，再回来看我的拙笔。下面我也只是抽象地讲一下 L2 的原理。

L2 的顶层包括 Scheduler 和两个 TLNode，其中两个 TLNode 只是 Chisel 层面的接口，不反映在物理硬件上，因此 L2 的实际内容都在 Scheduler 中。Scheduler 的内部结构如下图所示：

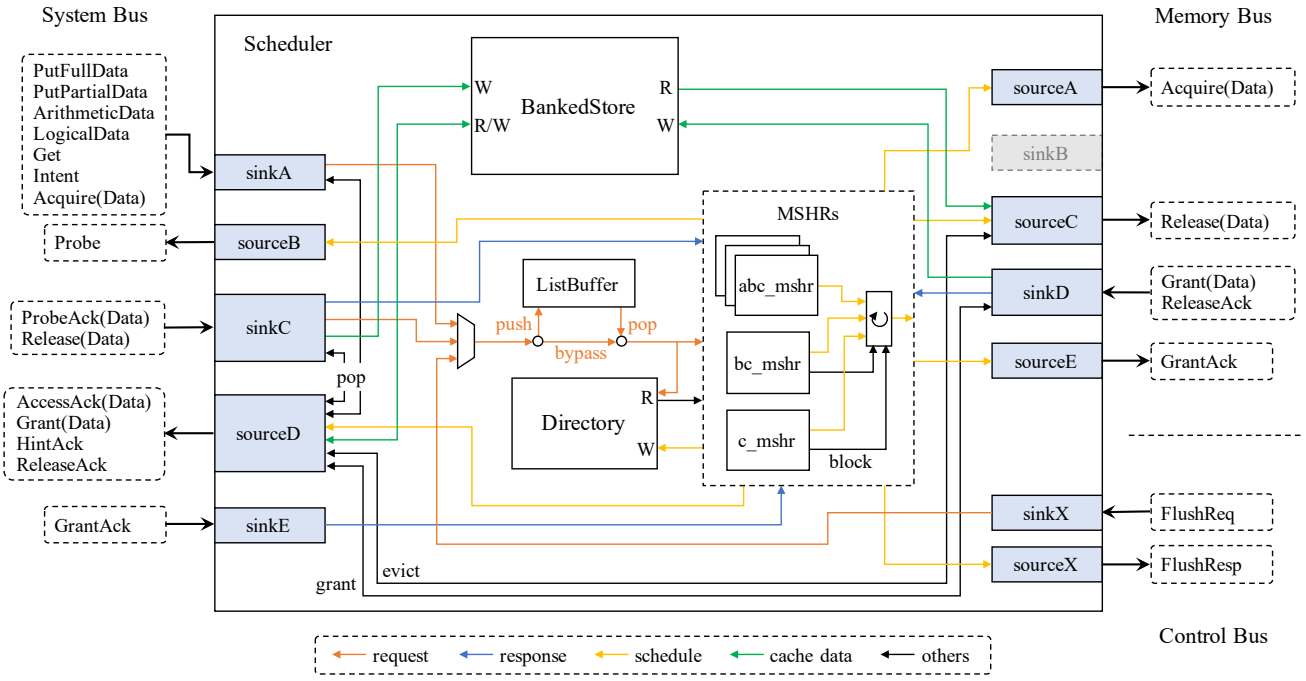

Figure 11: *Scheduler 的内部结构*

从这张图中可以看到，Scheduler 的主要模块为：

1. *IO 接口（sink*/source**）*：将总线上的 TileLink 请求转换为对 L2 内部的请求；IO 接口并非简单的转换器，一些接口也分担了重要的功能，甚至有多级流水线（如 sourceD）。
2. *Directory*：存储目录和 meta。
3. *BankedStore*：存储数据。
4. *ListBuffer*：用硬件实现的链表，提供了 N 个长度为 M 的虚拟队列，比直接开 N×M 的队列省空间。
5. *MSHRs*：处理请求的主要逻辑；注意，它虽然叫 MSHR，但它不只处理 miss 请求，实际上所有请求都要经过 MSHR；当然，Scheduler 自己和 IO 接口也承担了不少逻辑。

L2 内部的各个模块使用松散耦合的方式协作，因此 L2 不像 Core 一样有明显的流水线，但它仍然是按照明确的顺序来工作的。简单来说，L2 的工作流程无非是是以下几个步骤：

1. *接收请求*：从 sourceA、sourceC 和 sourceX 接收外部模块传入的请求，即图中的 request；L2 每周期只允许接收一个请求，若多个请求同时到达就要按照 C>X>A 的优先级接收。
2. *分配请求*：将每个请求分配（allocate）给一个 MSHR 处理；根据 MSHRs 的可用性，请求可能被分配给空闲的 MSHR，暂存到同 set 的 MSHR 的 ListBuffer（L2 规定同一时刻 MSHRs 的 set 不能相同），抢占高优先级专用的 MSHR，或阻塞（即不接收新的请求）。
3. *发送次级请求*：如果某个请求不能仅靠 L2 处理（例如 L2 未命中），L2 就需要向其他模块发送次级请求（例如向下级 cache 请求数据）；次级请求先由 MSHR 以 schedule 的形式派发给 IO 接口，再由 IO 接口发送出去；L2 每周期只允许发起一个次级请求，故 MSHRs 使用 Round-Robin 方式进行轮转。

4. *接收次级回复*：外部模块处理完次级请求，向 L2 发送次级回复，即图中的 response；次级回复可以直接发送给对应的 MSHR，不存在优先级或阻塞问题，因为它们一定是给某个已分配的 MSHR 的回复。
5. *发送回复*：向最初发起请求的核发送回复；回复同样用 schedule 的方式发送。

7.2 MSHR 是状态机吗

MSHR 是 L2 最复杂也是最重要的模块，读懂了 MSHR 就基本等于读懂了 L2。Francis 的博客虽然写得很全面，但唯独没有讲清楚 MSHR，尤其是作者说 MSHR 是一个状态机，这是我强烈反对的一点。L2 的 MSHR 并没有包含状态机的典型特征，例如状态编码和迁移函数；虽然 Rocket 和 Boom 的 L1 DCache 的 MSHR 确实是状态机，但显然 L2 的 MSHR 和 L1 是两码事，不能简单类比。那么 L2 的 MSHR 到底是什么呢？我的答案是：*使用微程序的处理器*。下面让我给出证据。

首先，MSHR 定义了 19 个布尔“状态”，每个状态对应着一个操作的完成情况；一个 `false` 的状态表明对应的操作未完成，而当一个操作完成后相应的状态就会被置为 `true` 。这些状态和操作如下所示：

状态	相关通道	操作
s_rprobe	sourceB	（release-probe）向由于自己release导致的需要back-invalidate的核发送probe
w_rprobeackfirst *w_rprobeacklast*	sinkC	等待*s_rprobe*的回复，由于回复可能是多拍，所以用first和last来一并概括
s_release	sourceC	向下级cache写回数据
w_releaseack	sinkD	等待*s_release*的回复
s_pprobe	sourceB	（probe-probe）向由于其他核请求权限导致需要降级的核发送probe
s_acquire	sourceA	向下级cache请求数据
s_flush	sourceX	向Control Bus发送Flush回复
w_grantfirst *w_grantlast* *w_grant*	sinkD	等待*s_acquire*的回复，first和last含义同上；另外还有一个*w_grant*，它是一个优化，相当于last，但在支持虫洞传输的数据上它可以立即被设置，于是L2不用等到所有数据返回才开始下一步操作，实现了critical word first
w_pprobeackfirst *w_pprobeacklast* *w_pprobeack*	sinkC	等待*s_pprobe*的回复，各个变体的含义同上
s_probeack	-	为支持MLC预留，目前无用
s_grantack	sourceE	收到下级cache的数据后再次向下级cache发送同步信息
s_execute	sourceD	向上级cache发送回复
w_grantack	sinkE	等待上级cache收到数据后的同步信息
s_writeback	-	将新的meta写回Directory

Figure 12: *MSHR 的 19 个布尔状态与它们对应的操作*

注意到，状态名称的前缀已经暗示了它的操作类型与通道，其中 `s_` 意为“schedule”，是向外发送某个请求或回复（除了 *s_writeback*），所以都和 source 通道有关；`w_` 意为“wait for response”，是接收某个回复，所以都和 sink 通道有关。

MSHR 收到新请求时，会根据请求本身和 L2 当前的状态进行“译码”，也就是标记哪些操作需要被执行（设置相应状态为 `False` ）。译码逻辑既会考虑请求的 `opcode` 和 `param` ，也会考虑 Directory 是否命中、meta 的状态等；故对于同一个请求，由于 L2 状态的不同，译码的结果也可能不相同。当 MSHR 需要执行多个操作时，这些操作会按上表中的顺序依次执行（注：事实上，一些操作可以并行执行，如 *s_pprobe* 和 *s_acquire*；另外当 set 不变时，*s_writeback* 可以和新的请求重叠）。容易知道，MSHR 的 19 个“状态”既不是状态机常见的整数状态，也不是 one-hot 编码的状态（因为多个状态可能同时被设置）；如果说 MSHR 是状态机，那么它就是一个有 2^{19} 种状态的状态机了，这显然很牵强。而如果把 MSHR 理解为使用微程序的处理器，那么这套“译码-执行-写回”的流程就都说得通了。

我不严谨地画了一张 MSHR 译码的逻辑图，如下所示。简单来说，在收到请求后，MSHR 首先判断它来自哪个通道，因为每个通道的请求类型是固定的，根据通道可以快速识别请求类型。识别出请求类型后，MSHR 就根据请求本身和 L2 的状态设置上述的 19 个布尔状态。至于 MSHR 为什么不用译码表而是用分支逻辑来译码，这是因为分支逻辑已经足够完成译码，用译码表反而更复杂。

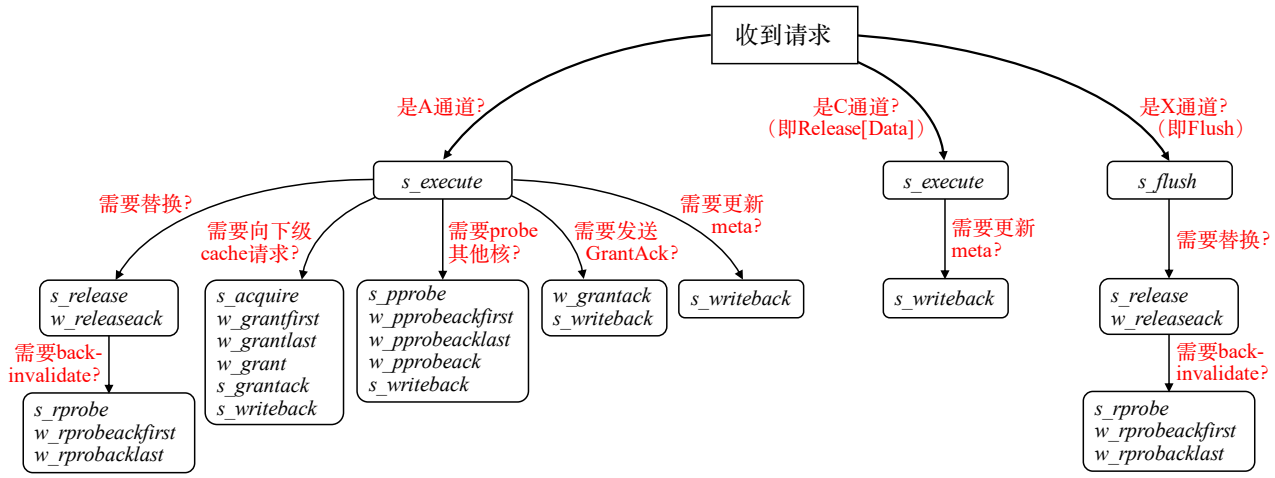

Figure 13: *MSHR 对请求进行“译码”的逻辑*

7.3 添加预取接口

我在 L2 中实现了一个通用的预取接口。该接口的设计目标是提供一套完备的预取器与 L2 的沟通机制，使得任何预取器都可以用这个接口与 L2 连接（虽然现在并没有实现）。该接口的结构图如下：

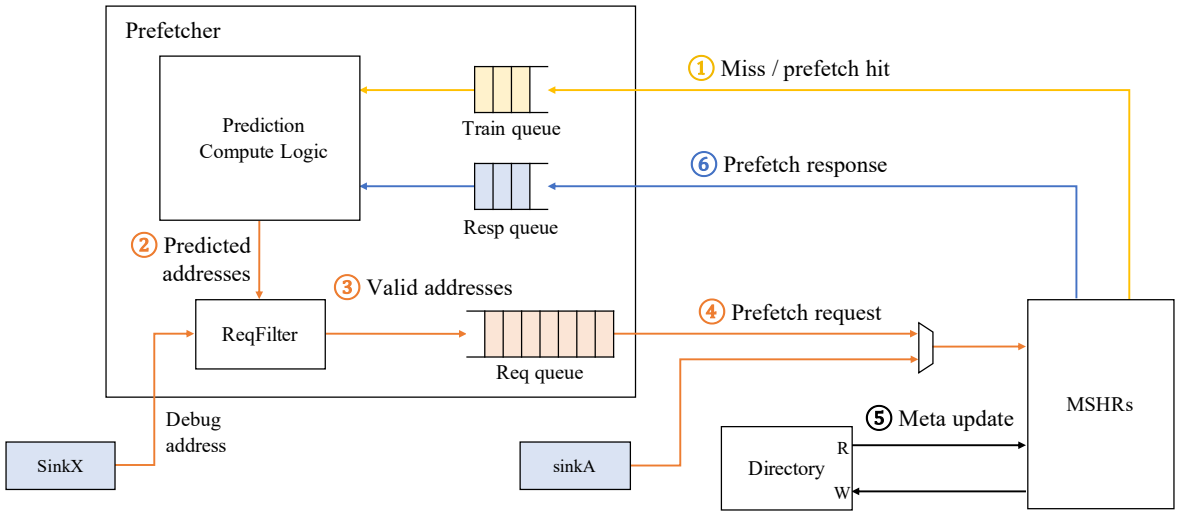

Figure 14: *L2 预取接口结构图*

其中 *Prefetcher* 和 Prefetcher 与 MSHRs 之间的连接就是整个接口的内容；*Prediction Compute Logic* 是放置真正的预测逻辑的地方，用户可以在这里实现自己的预取算法。我的预取接口其实很简洁，它只需要 3 个端口与 MSHRs 连接，即 *Train*、*Req* 和 *Resp*，另外还有一个调试用的 MMIO 端口。下面我用一次完整的预取过程说明这个接口的工作方式：

1. *发送训练信息*：当 MSHRs 发现 miss 或 prefetch hit 时，它会向预取器发送训练信息；这里的 miss 特指数据缺失，如果只是权限缺失则不算 miss，因为获取权限比获取数据要快，而且用预取器获取权限可能给 Core 带来不必要的 Probe；prefetch hit 指的是命中了一个被预取上来的块，如果没有预取，这个块本来应该是 miss 的，所以应当作为 miss 看待。

2. 生成预取地址：预测逻辑根据历史信息和当前训练信息生成一个（或一组）预取地址；除了预测逻辑，我还提供了一个调试用的 MMIO 接口，用户可以通过这个接口直接往 L2 发送预取地址。
3. 过滤预取地址：由于预测逻辑并不关心地址的范围与权限，所以可能会生成一些非法地址，需要在此过滤掉；另外由于 L2 使用的是物理地址，L2 上的页通常是不连续的，所以跨页的预测很可能指向了错误的页，需要被过滤掉。
4. 发送预取请求：预取接口使用预取地址构造预取请求，该请求和 sinkA 共用一个数据通路发送给 MSHRs；我的预取请求完全复用了 TileLink 的 *Intent*（或者叫 *Hint*）请求的格式，只是加上了一个 `prefetch` 位表明它是一个内部的预取请求（故 MSHRs 不需要发送 *HinkAck* 回复）；MSHRs 中的处理逻辑也是基本复用 Intent 的，所以其实我只修改了很少的代码就实现了预取接口。
5. 更新 *meta*：MSHRs 将预取地址对应的数据取回来后，需要更新目录中的 meta；我在 meta 中也加上了一个 `prefetch` 位，表明它是一个被预取上来的块，下次发生命中时就知道是 prefetch hit 而不是普通的命中；注意，发生 prefetch hit 后应当立即清除掉 `prefetch` 位，因为 prefetch hit 不应该重复发生。
6. 发送预取回复：MSHR 取回预取数据后还需要通知预取器，告知完成情况与所花时间等；发送预取回复不是必需的，许多预取器其实并不关心预取什么时候完成，但是一些高级的预取器需要用这个回复改善及时性（timeliness），所以我也提供了回复功能。

关于预取接口还有两个说明，读者若要修改我的代码务必注意：

1. 预取接口的 Train、Resp 和 Req 接口上都有队列，这些队列是可选的，是否需要队列取决于用户的预测算法的响应时间、生成地址的速率等；队列的形式也很有讲究，可以是简单的 FIFO 队列，也可以包含优先级排序、去重、超时丢弃等功能；队列配置得不好可能会对及时性造成影响；我发现 gem5 的 `prefetch::Queued` 写得还不错，读者可以参考模拟器写硬件代码。
2. L2 的 MSHR 对于请求有一个隐式要求，就是每个请求都必须至少有一个回复，否则这个请求占用的 MSHR 可能不会被正确释放（当 ListBuffer 不为空时）；所以即使预取回复没有用，我们也不能去掉这个回复，可以让它发送一个空回复。

我实现了一个 *Next-Line* 预取器，它总是在一个块 miss 的时候预取下一个块，是一种最简单的序列预取器。由于我的预取接口会对 prefetch hit 做出响应，所以它可以实现连续预取（也就是在一串连续地址的访问中只有第一个是 miss）。Next-Line 预取器的结构非常简单，读者可以直接看代码 `Prefetcher.scala` 。

7.4 实验结果

首先让我们设定一个符合真实处理器的内存延迟。我写了一个刷 cache 的程序，该程序用 64B 的跨步（64B 是 cache line 的大小，用 64B 的跨步可以避免两次访问间有空间局部性）访问一个大小为 N 的数组。我在多个内存延迟下观察 N 与平均访问延迟的关系，得到如下结果：

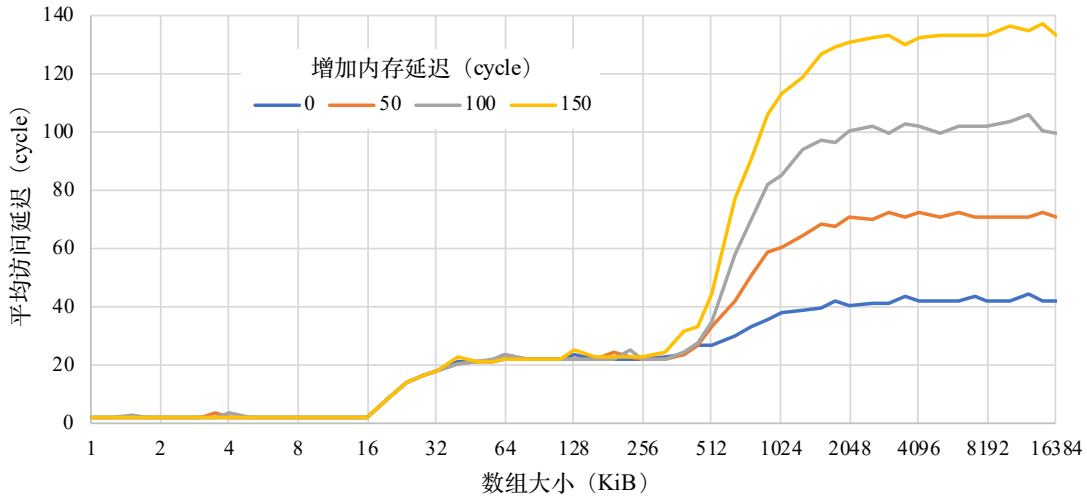

Figure 15: 平均访问延迟与数组大小的关系

容易发现，无论增加多少内存延迟，平均访问延迟随数组大小的变化都存在一个“三段式”的规律：

1. 当 N 不超过 16KiB 时，访问延迟约为 2.4 周期，因为 16KiB 是 L1 DCache 的大小，此时的延迟就是访问 L1 的延迟。
2. 当 N 超过 16KiB 后，延迟逐渐增加至 22 周期，因为跨步式的访问会让 L1 的每次访问都 miss，此时的延迟就是 L1 miss 的延迟；注意到，延迟直到 N 为 4 倍 L1 大小时才达到 22 周期，这是因为 Boom 的 L1 使用随机替换策略，而 L1 恰好有 4 路，所以 N 为 4 倍 L1 大小时才能比较充分地刷掉 cache。
3. 当 N 超过 512KiB 后，延迟再次增加至某个周期（取决于内存延迟），因为 512KiB 是 L2 的大小，此时的延迟就是 L2 miss 的延迟；由于 L2 也使用随机替换策略，所以 512KiB 附近的延迟增长曲线和 L1 类似；但是其实 N 还没到 512KiB 的时候延迟就开始增加了，这是因为我们的数组只是在虚拟空间上连续，但 Linux 不保证它在物理空间上连续，所以在 N 达到 512KiB 之前就已经出现了冲突。

我的实验结果很符合 The Ice Lake Benchmark Preview: Inside Intel's 10nm 这篇博客对 Intel 处理器做的结果，虽然 Boom 的 L1 miss 的延迟有点高（Intel 只用 13 周期，Boom 要用 22 周期）；这可能是因为 L1 MSHR 的状态机设计得比较繁琐；而且我测的是连续 miss，而 MediumBoom 的 MSHR 配置得很拮据（只有 2 个），连续 miss 会让 MSHR 不够用，从而使隐藏延迟的能力下降。最终，我选择给内存增加 100 个周期的延迟，因为此时 L1 miss 和 L2 miss 的延迟之比是 1:5 的关系，比较像 x86。

对预取器的一个完整评测除了加速比外，还应当包含*准确率*（*accuracy*）、*覆盖率*（*coverage*）和*及时性*（*timeliness*）这三点，但是这三点需要比较复杂的计数器才能统计出来；出于简便，这里我只用 IPC 计算了加速比。我在自己手写的两个程序、CoreMark 和一些 SPEC2006 程序上的结果如下：

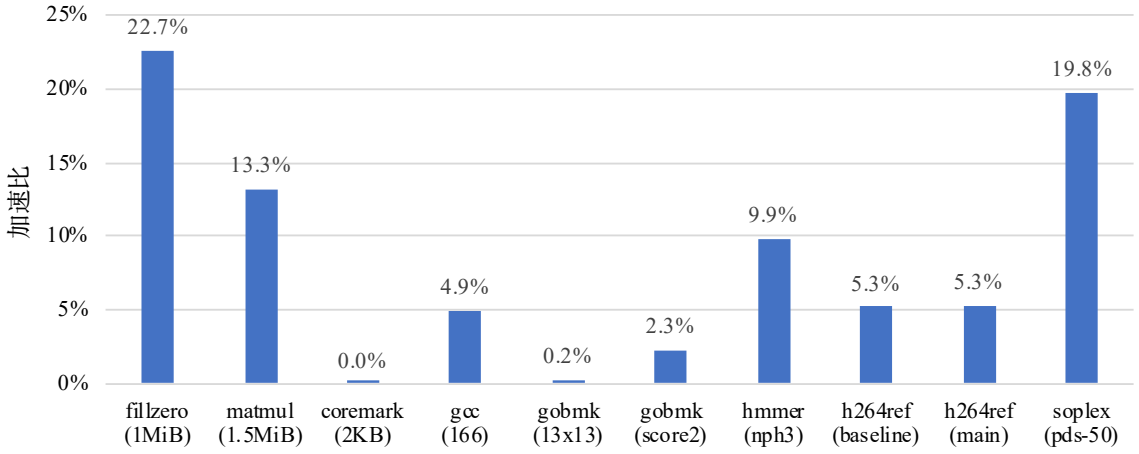

Figure 16: *L2 使用 Next-Line 预取器的加速比*

可以看到，大部分程序的性能都有一定的提升，一些访存规律性比较强的程序性能提升超过 10%，但是一些访存无规律或对 L2 压力小的程序的性能提升则很微弱；另外，加上预取之后的性能和前面提到的理想性能还有很大的差距。实验结果说明，我的预取接口和 Next-Line 预取器的实现是正确的，但是由于 Next-Line 的适用范围很窄（只用于无间隔顺序访存的程序），所以带来的性能提升有限。

读者若要在我的预取接口上继续研究，我有几个建议（也是对我自己的建议）：

1. gem5 里提供了很多预取器（`Prefetch.py`），用起来非常方便；读者应当先用 gem5 等模拟器比较各个预取器的优劣，再决定将哪个实现到硬件上。
2. 预取的研究不只局限于预取器，还和替换策略有关；我一直认为预取和替换应当是一体的，退一步说，也应该做到 *Eviction-Aware Prefetching* 和 *Prefetching-Aware Eviction*，所以 L2 现在用的随机替换策略是不合格的；预取和替换的协作是比较新的话题，读者可以多读论文。
3. 就 Boom 而言，我觉得 Boom 的整个内存系统从上到下都有很多可以改进的地方，比如 LSU、L1I、L1D、L1DTLB，不一定局限于 L2；如果要优化产品性能而不是单纯地做研究，我认为最好是优化它的短板，因为这样的性价比最高；至于哪个是短板就需要读者自己探究了。