XiangShan Cache 源码阅读

蔡洛姗 2018K8009929051

一、XiangShan简介

1.1 什么是XiangShan

“香山”是一款由中国科学院计算所和鹏城实验室联合开发的开源高性能RISC-V处理器。这里的开源，一是指”香山“ 实现的是一套开放免费的RISC-V指令集，二是指处理器的微架构设计与实现完全开源，其设计文档和源码都公开在https://github.com/OpenXiangShan仓库中，三是指其设计流程以及设计、综合、验证过程中使用的EDA等工具也都是开源的。所谓高性能，主要体现在香山处理器的微结构设计采用了较为先进的乱序、多发射、分支预测、存储层次等方法，最大化单个芯片上多种类型多个层次的并行性，实现高并发、高吞吐量、低延迟，使基准测试程序的运行时间减少到一定程度，以满足特定应用领域的需求。香山处理器采用 Chisel 硬件设计语言开发，目前第一版”雁栖湖“是首个稳定的微架构，而第二版”南湖“还在不断开发，因此我在本次大作业中阅读的源码版本主要为第一版稳定的代码。

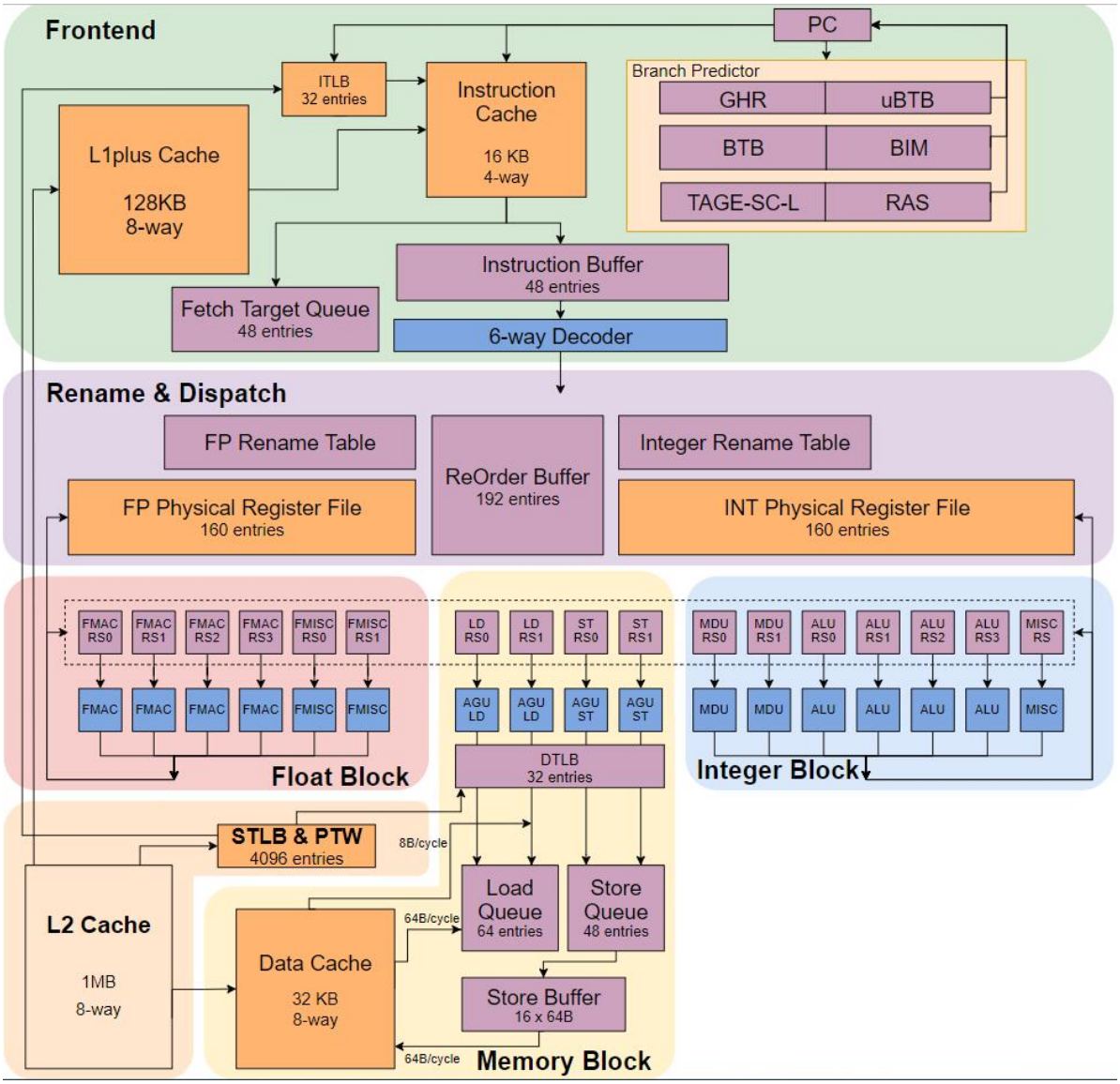
1.2 什么是Chisel

香山处理器项目拥有面向对象的特性，主要原因在于它采用了Chisel这样一种编程语言。Chisel是一种硬件构建语言(Hardware Construction Language)，它嵌入在Scala语言中，从而可以重用scala语言的面向对象和函数式等强大特性。本质上它的代码仍是Scala程序，通过增加一些表示硬件原语的类和函数（如Reg, Wire, Mux, FIFO），Chisel就能够表示同步数字电路的组成，在借助sbt和JVM编译后会生成verilog硬件描述语言的代码，然后进一步综合、验证。

由于融入了高级编程语言的语法，Chisel这类硬件构建语言与verilog等硬件描述语言相比，具有更高的代码开发效率：设计者能在更短时间内搭建项目，同时利用面向对象中的复用思想分摊验证的成本。另外，学生和个体贡献者也能够更简单地自行创新，有利于开放、开源、共享、共治的处理器芯片生态构建。这些优势将会在后文中结合代码进行进一步的分析。

1.3 XiangShan架构与主要功能模块

香山“雁栖湖”架构是一个具有11级流水、6发射、4个访存部件的乱序处理器核，微架构图如下所示：



它可以分为前端、后端、访存三个主要功能模块。前端包括取指、分支预测、指令缓冲等单元，顺序取指。后端包括译码、重命名、重排序缓冲、保留站、整型/浮点寄存器堆、整型/浮点运算单元。访存子系统按照`load`和`store`分割开，包括两条`load`流水线和两条`store`流水线，以及独立的`Load Queue`和`Store Queue`，`Store Buffer`等，缓存包括`L1Cache(ICache、DCache)`、`L2Cache`、`TLB`和预取器等模块，在访存部件内。我在本次大作业中选择了`L1Cache`进行阅读，L1缓存包括指令缓存`ICache`和数据缓存`DCache`，二者结构相似只是`DCache`的访问模式比`ICache`更加丰富，指令的访问是只读不可写且位宽固定，而数据的访问可读可写且有字节访问、字访问和行访问等多种形式，因此在下文中我们主要以`DCache`为例进行分析。

二、XiangShan Cache功能流程分析

2.1 什么是Cache

缓存(Cache)技术是在处理器运算速度和内存访问速度差异日渐扩大导致的"存储墙"问题背景下被提出的，主要利用了程序访问内存的时间局部性和空间局部性，使用速度较快、容量较小的`Cache`来临时存放处理器常用的数据，从而提高数据访问效率。

Cache的主要功能是有序存放处理器常用的数据，响应来自上层的读/写请求，从自身或下层存储器中查找对应的数据块并及时替换。现代处理器普遍在片内集成多级Cache，香山也在每个处理器核中配备了一级指令Cache和数据Cache，二级Cache，以及多核共享的三级Cache。共享存储系统将会引入缓存一致性和总线协议等更加复杂的问题，而本文的重点是面向对象思想，所以我们选择相对简单的一级

Cache，特别是同时具备读写响应的L1 DCache来避开项目自身的实现复杂性，只探究核心的需求建模和流程分析。

2.2 DCache的基本结构

我们首先来看香山处理器的一个DCache模块的实现类形如：

```
class DCacheImp(outer: DCache) extends LazyModuleImp(outer) with
HasDCacheParameters {
  val io = IO(new DCacheIO)
  val (bus, edge) = outer.clientNode.out.head

  //----------------------------------------
  // core data structures
  val dataArray = Module(new DuplicatedDataArray)
  val metaArray = Module(new DuplicatedMetaArray)

  //----------------------------------------
  // core modules
  val ldu = Seq.fill(LoadPipelineWidth) { Module(new LoadPipe) }
  val storeReplayUnit = Module(new StoreReplayQueue)
  val atomicsReplayUnit = Module(new AtomicsReplayEntry)
  val mainPipe   = Module(new MainPipe)
  val missQueue  = Module(new MissQueue(edge))
  val probeQueue = Module(new ProbeQueue(edge))
  val wb         = Module(new WritebackQueue(edge))

  //----------------------------------------
  // meta array read/write
  // data array read/write
  // load pipe connect
  // store pipe and store miss queue connect
  // atomics excecute
  // refill to load queue connect
  // mainPipe connect
  // wb connect
  /* ......
     ......
     ...... */
}
```

基于此，我们可以提取出香山DCache所需要的主要类和方法：

```
[用例名称]
    XiangShan L1DCache

[用例描述]
    1. 由XSCore的后端访存模块Memblock实现DCache，向上连接Load/Store Queue并输入访存请求，向下连接L2Cache输出请求；
    2. pipe流水线接收请求，查找对应的metadata和data，根据metaArray中的内容判断是否命中：
        若命中则到dataArray中进行读写操作；
        若不命中则将请求发送到missQueue，从下一级存储中读写；
    3. 将需要替换的数据块从WritebackQueue写回下一级存储，将新数据块写入dataArray和metaArray
    4. 向Load/Store Queue返回请求结果

[抽象提取类]
```

```
[类]：DCacheIO
[属性]：与其它模块交互的接口信息

[类]：DuplicatedDataArray、DuplicatedMetaArray
[方法]：读操作、写操作、ECC校验
[属性]：配置规格、存储的数据、IO请求地址与数据

[类]：MainPipe、LoadPipe
[方法]：分析请求，查找和读写目标数据块、发出替换和写回脏块的请求
[属性]：配置信息、处理请求的状态机(miss/hit/req/resp)

[类]：MissQueue、WritebackQueue
[方法]：接收请求并适当合并、发送请求到其它模块
[属性]：队列元素、入队出队命令、处理请求的状态机
```

2.3 DCache的请求处理流程

接下来我们进一步阅读代码并分析DCache具体如何基于上述类，响应一个来自CPU的读写请求。

2.3.1 Load

从`DCacheImp`类的子类模块连接代码中，可以提取下面这些有关Load请求的操作：

```
  ldu(w).io.lsu <> io.lsu.load(w)
  val dataReadArb = Module(new Arbiter(new L1DataReadReq, DataReadPortCount))
  dataReadArb.io.in(LoadPipeDataReadPort)  <> ldu(LoadPipelineWidth -
1).io.data_read
  dataArray.io.read(LoadPipelineWidth - 1) <> dataReadArb.io.out
  dataArray.io.resp(LoadPipelineWidth - 1) <> ldu(LoadPipelineWidth -
1).io.data_resp
```

分析控制逻辑和数据通路我们可以得出，读请求首先从IO接口进入`LoadPipe`，`LoadPipe`类的代码较为复杂，我们只关注其三级流水来理解它在Dcache中所承担的功能，而不细究其实现，这也是面向对象里“封装”和“信息隐藏”思想的体现，对于自上而下进行代码阅读的读者来说，能够大大提高效率。

```
class LoadPipe(implicit p: Parameters) extends DCacheModule {
  ......
  // Pipeline
  // ------------------------------------------------------------------------
-----
  // stage 0
  val s0_valid = io.lsu.req.fire()
  val s0_req = io.lsu.req.bits
  val s0_fire = s0_valid && s1_ready

  // ------------------------------------------------------------------------
-----
  // stage 1
  val s1_valid = RegInit(false.B)
  val s1_req = RegEnable(s0_req, s0_fire)
  val s1_addr = io.lsu.s1_paddr
  val s1_fire = s1_valid && s2_ready
  s1_ready := !s1_valid || s1_fire

  // tag check
```

```
  val meta_resp = VecInit(io.meta_resp.map(r => getMeta(r).asTypeOf(new
L1Metadata)))
  def wayMap[T <: Data](f: Int => T) = VecInit((0 until nWays).map(f))
  val s1_tag_eq_way = wayMap((w: Int) => meta_resp(w).tag ===
(get_tag(s1_addr))).asUInt
  val s1_tag_match_way = wayMap((w: Int) => s1_tag_eq_way(w) &&
meta_resp(w).coh.isValid()).asUInt
  val s1_tag_match = s1_tag_match_way.orR
  val s1_hit_meta = Mux(s1_tag_match, Mux1H(s1_tag_match_way, wayMap((w: Int) =>
meta_resp(w))), s1_fake_meta)

  // data read
  val data_read = io.data_read.bits
  data_read.addr := s1_addr
  data_read.way_en := s1_tag_match_way
  data_read.rmask := UIntToOH(get_row(s1_addr))
  io.data_read.valid := s1_fire && !s1_nack

  // ----------------------------------------------------------------------------
-----
  // stage 2
  val s2_valid = RegInit(false.B)
  val s2_req = RegEnable(s1_req, s1_fire)
  val s2_addr = RegEnable(s1_addr, s1_fire)

  val s2_tag_match_way = RegEnable(s1_tag_match_way, s1_fire)
  val s2_tag_match = RegEnable(s1_tag_match, s1_fire)
  val s2_hit_meta = RegEnable(s1_hit_meta, s1_fire)
  val s2_hit = s2_tag_match && s2_has_permission && s2_hit_coh ===
s2_new_hit_coh

  // load data gen
  val s2_data_words = Wire(Vec(rowWords, UInt(encWordBits.W)))
  for (w <- 0 until rowWords) {
    s2_data_words(w) := s2_data(encWordBits * (w + 1) - 1, encWordBits * w)
  }
  val s2_word = s2_data_words(s2_word_idx)
  val s2_word_decoded = s2_word(wordBits - 1, 0)

  // send load miss to miss queue
  io.miss_req.valid := s2_valid && !s2_nack_hit && !s2_nack_data && !s2_hit
  io.miss_req.bits.addr := get_refill_addr(s2_addr)

  // send back response
  val resp = Wire(ValidIO(new DCacheWordResp))
  resp.valid := s2_valid
  resp.bits.data := s2_word_decoded
  io.lsu.resp.valid := resp.valid
  io.lsu.resp.bits := resp.bits
}
```

从上述代码中可以看出，`LoadPipe`在`stage0`时根据地址`addr`向`metaArray`和`dataArray`发出读请求，在`stage1`时读取到`cache_line`的内容进行`tag`比较，判断是否命中，在`stage2`如果`hit`，则将读到的`data`返回；如果`miss`，则发送`miss`请求到`MissQueue`，等待请求进入`L2cache`读取整个`block`的数据，将对应数据返回给`Load Queue`。同时`MissQueue`还会向`MainPipe`发送`refill`请

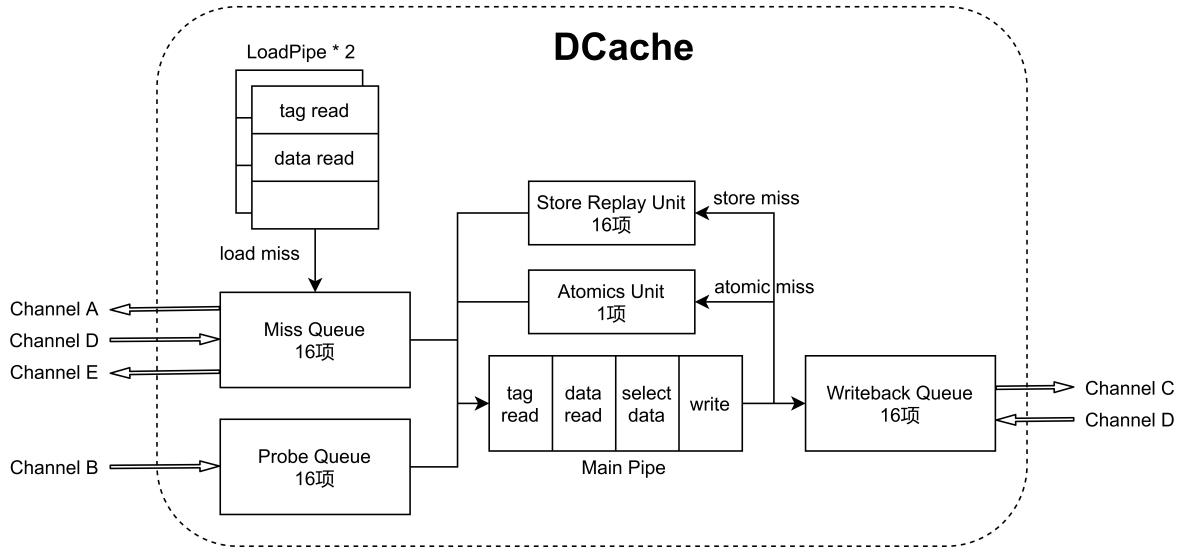
求，读出被替换的路，写入新的`tag`和`data`。如果被替换的路是`dirty`的，则将其从`WritebackQueue`写回`L2cache`。

2.3.2 Store

同样从`DCacheImp`类中，我们可以找到`Store`请求相关的操作：

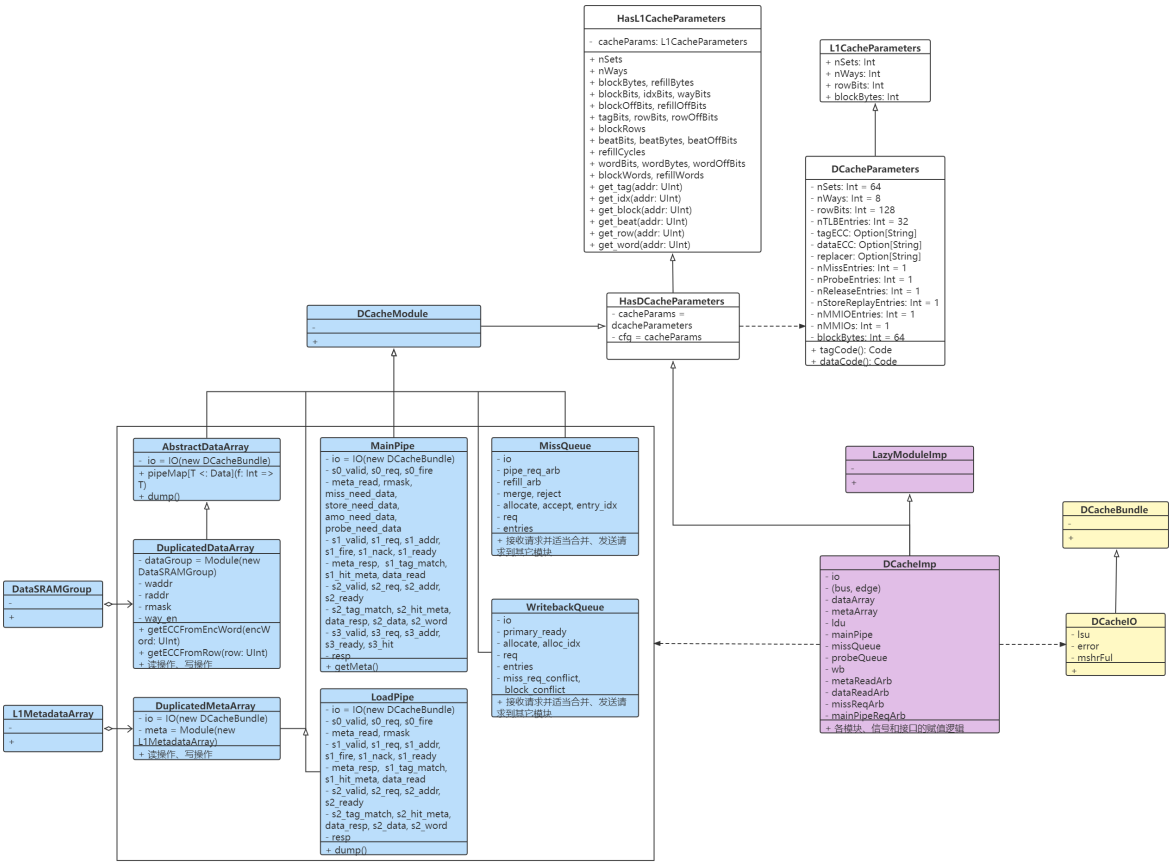
```
  storeReplayUnit.io.lsu <> io.lsu.store
  val mainPipeReqArb = Module(new RRArbiter(new MainPipeReq,
MainPipeReqPortCount))
  mainPipeReqArb.io.in(MissMainPipeReqPort)     <> missQueue.io.pipe_req
  mainPipeReqArb.io.in(StoreMainPipeReqPort)    <> storeReplayUnit.io.pipe_req
  dataArray.io.write <> mainPipe.io.data_write
  dataArray.io.resp(LoadPipelineWidth - 1) <> mainPipe.io.data_resp
  missQueue.io.pipe_resp          <> mainPipe.io.miss_resp
  storeReplayUnit.io.pipe_resp    <> mainPipe.io.store_resp
```

可以看出，写请求首先从IO接口进入一个`storeReplayQueue`，出队列后进入`mainPipe`。`mainPipe`拥有四级流水线，我们在此就不再展示其代码，其主要功能为处理`Refill`，`Probe`，`Store/AMO`请求。对于`Store`请求，`mainPipe`在`stage0`发出请求读取`Meta`，`stage1`根据`tag`判断是否命中并发出请求读取`Data`，`stage2`根据请求的`mask`选中`data`的写入部分并准备好命中时`Meta`和`Data`的新内容，`stage3`如果`hit`则将新的内容写入`dataArray`和`metaArray`，如果`miss`，则发送`miss`请求到`MissQueue`。接下来就和读请求`miss`一样，从下一级读取`block`，写入新块，替换和写回旧块。这也就是cache中常用的写回+写分配的策略，总结一下，DCache的主要部件以及处理流程图可以表示如下：



三、XiangShan Cache设计分析

在第二节中我们对 XiangShan DCache 的功能进行了需求建模和流程分析，找到了一部分承担主要任务的类，并探索了一个Load/Store请求被响应的具体流程，本节我们来继续分析 XiangShan Cache 还涉及哪些重要的类以及类间关系。



上图体现了我们在第二节中分析过的一些类以及相关类之间的关系。类间关系主要有继承、实现、关联和组合。

紫色的`DCacheImp`类继承自`LazyModuleImp`类，这是一个从外部的开源代码中引入的抽象类，定义了模块实现类的通用属性和方法。这样就使得硬件设计也可以像软件调用库函数那样，通过import和继承的方法调用一个外部的硬件组件实现，充分体现了复用与解耦的思想，可以减少冗余代码，提高开发效率。蓝色的`DuplicatedMetaArray, MainPipe, LoadPipe, MissQueue, WritebackQueue`等都继承了一个顶层定义的`DCacheModule`抽象类，它定义了Dcache所涉及的子模块通用的方法，但没有实现，而是由继承的子类来分别实现。这实际上表示的是这些模块共同构成了DCache，体现了封装和模块化的思想。

为了进一步解耦和便于扩展，我们在课上学过Java等语言提供了接口类，但Scala语言没有接口类，在硬件设计中模块与模块之间却必须有输入输出接口，我不确定用java中的接口类与硬件中的接口相比较是否贴切，不过Chisel在此基础上提供了`Bundle`类，用于捆绑不同类型的信号，可以整体引用和分别访问，非常适合在各个模块中实现IO接口。例如上图黄色的`DCacheIO`和其父类`DCacheBundle`就是`Bundle`类，拥有`req_valid, req_ready, req_addr, resp_valid, resp_ready, resp_data`等信号，在`DCacheImp`中被实现后可以通过`io.req_addr`来访问；此外`DecoupledIO`方法可以反转输入输出，比如与DCache模块相连接的`Load_queue`模块在实现IO接口时就需要用`DecoupledIO(new DCacheIO)`来表示其接线方向与DCache相反，这与`verilog`中需要对每一个输入输出信号都单独定义`input`和`output`的方式相比显然便捷很多。

此外，在阅读代码的过程中我发现还有一种特殊的类`Parameters`，通常伴随上述主要的类一起出现，如：`class DCacheImp(outer: DCache) extends LazyModuleImp(outer) with HasDCacheParameters`。这里的with表示复合类型，Chisel支持多继承，但这种多继承并不会产生冲突问题，因为`LazyModuleImp`才是DCacheImp主要继承的父类，而`HasDCacheParameters`只负责表示DCache的参数设置，如Cache的`nSets, nWays, blockBytes, tagBits, wordBits`等基本参数。这和verilog中的宏定义类似，但将所有关于DCache的参数都封装在一起，可配置又便于引用，更加直观。

在这当中，面向对象思想的核心要素，包括抽象、继承、关联和多态，得到了充分体现。

四、高级设计意图分析

4.1 设计模式分析

虽然《Design Patterns》这本书中所介绍的23种设计模式都是基于软件设计提出的，但是我认为使用拥有面向对象特性的语言进行的硬件设计，同样可以遵循这些设计模式来更好地复用，以达到敏捷开发的目的。首先我们来看看"香山"项目中使用到了哪些设计模式：

- **Builder建造者模式**

  定义：将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。

  应用场景：将各种产品集中起来进行管理，当创建复杂对象的算法应该独立于该对象的组成部分以及它们的装配方式时。

  例：DCache是一个复杂对象，由MetaArray, DataArray, MainPipe, LoadPipe等多个模块构成，但Array的大小、流水线的控制逻辑、miss后的替换算法是可以灵活选择的。如果想要将DCache的表示与实际对象的构建分里开来，就产生了一个抽象建造者DCache()，负责将这些固定的部件连接起来，还有一个具体建造者DCacheImp()，负责完成各模块的具体创建。

- **Adapter适配器模式**

  定义：将一个类的接口转换成客户希望的另外一个接口，使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。

  应用场景：想要使用一个已经存在的类，而它的接口不符合你的需求；想创建一个可以复用的类，该类可以与其他不可预见的类协同工作；想使用一些已经存在的子类，但是不可能对每一个都进行的子类化以匹配它们的接口。

  例：在 XiangShan 中，需要把很多模块进行对接，传递请求信号。但不同模块所暴露的接口，对请求的定义不一定是相同的。通过使用`ValidIO()`、`DecoupledIO()`、`Flipped()`等方法，比如

  ```
  val miss_req  = Flipped(ValidIO(new MissReq))
  val mem_req = ValidIO(DecoupledIO(new TLBundleA(edge.bundle)))
  ```

  就可以使不同的请求类转换成特定的形式（只暴露valid,ready信号，其它信号捆绑在bits结构中进行统一），从而实现模块接口的兼容。

4.2 Chisel vs. Verilog

当硬件设计开始使用Chisel这样一种新的语言时，大家最先提出的疑问自然是Chisel与Verilog这些经典的硬件描述语言相比有哪些优势。最重要的一点自然是它引入了软件设计和高级语言中面向对象的特性，这也是Chisel产生的出发点。下面是一个Chisel与Verilog代码的对比示例：

```
module dut(rst, clk, q);
  input rst;
  input clk;
  output q;
  reg [7:0] c;
  always @ (posedge clk) begin
    if (rst == 1b'1) begin
      c <= 8'b00000000;
    end
  else begin
    c <= c + 1;
  end
  assign q = c;
endmodule
```

```
class dut
extends Module {
  val q = IO( Output(UInt(8.w)) )
  val c = RegInit(0.U(8.w))
  c := c + 1.U
}
```

左侧的verilog代码和右侧的Chisel代码实现了同样的一个8位计数器，显然Chisel使用了更少的代码。此外实验和理论也已经证明，verilog和chisel同为寄存器级别的设计语言，可以实现的处理器性能是一致的。因此，面向对象为Chisel带来的高抽象性和可重用性，归根结底是提高了代码开发的效率。XiangShan的开发过程也验证了高性能处理器能够以半年左右的速度迭代，更加敏捷地开发。

但是，使用Chisel进行硬件设计也有缺点。因为scala程序经过编译输出的还是verilog代码，编译后的代码并不具有可读性，但是如果按照传统的仿真验证方法通过看波形定位bug，则仍然需要对应verilog代码中的信号名，从下图可以看出，XiangShan编译后输出的verilog变量名又长又臭，由波形定位到源代码中的bug变得更加困难。因此，XiangShan在设计过程中也使用和开发了Difftest，Checkpoint，仿真快照等一系列验证工具来帮助debug。

五、感想与总结

最后，用图灵奖得主David Patterson的一句话作结："Implementing specialized hardware along with effective methods for mapping the software onto the specialized hardware, is the key to achieving efficiency." 通过在这门课上学习面向对象思想，同时阅读了香山处理器的源码，我更加深刻地体会到来自软件设计的面向对象方法可以让硬件开发变得更加敏捷高效。不过在阅读的过程中，我还是明显感觉到能够体现面向对象特性的代码在XiangShan中的占比很少。如何充分利用面向对象的特性，依然是硬件开发者在编程过程中需要探索的问题。

Reference

[1] 香山官方文档仓库 https://github.com/OpenXiangShan/XiangShan-doc