XiangShan分支预测部分学习报告

Assign	
tag	homework
姓名	周鹏宇
学号	2019K8009929039

导言

关于香山CPU

香山是一款**开源**的**高性能 RISC-V 处理器**，**基于 Chisel 硬件设计语言**实现，**支持 RV64GC 指令集**。在香山处理器的开发过程中，使用了包括 Chisel、Verilator、GTKwave 等在内的大量开源工具，实现了差分验证、仿真快照、RISC-V 检查点等处理器开发的基础工具，建立起了一套包含设计、实现、验证等在内的基于全开源工具的处理器敏捷开发流程。目标是成为世界级的体系结构创新开源平台，服务于工业界、学术界、个人爱好者等的体系结构研究需求，**探索高性能处理器的敏捷开发流程**，建立一套基于开源工具的高性能处理器设计、实现、验证流程，大幅提高处理器开发效率、降低处理器开发门槛。

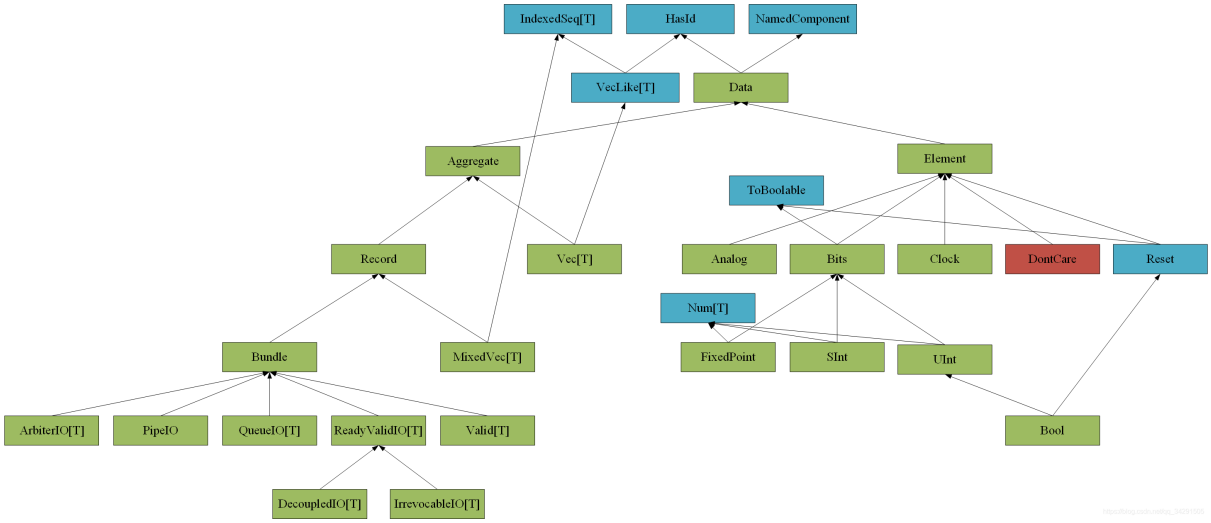
香山处理器坚持**开源策略**，坚定地开源所有的设计、验证、基础工具代码。在Chisel开源社区的现有总线工具、浮点运算单元、系统缓存等基础上，修改完善了它们的功能，同时优化了频率、吞吐等性能指标。香山**积极地拥抱开源社区**，并欢迎来自社区的贡献。

关于chisel

Chisel(Constructing Hardware In a Scala Embedded Language)是一门以Scala为宿主语言开发的硬件构建语言，它是由加州大学伯克利分校的研究团队发布的一种新型硬件语言。

相较于传统的硬件设计语言（比如Verilog），chisel的优势在于：

- 在硬件电路设计中引入了面向对象的特性



chisel的数据类型，其中箭头指向者为父类或者超类（绿色模块为class，红色模块为object，蓝色则为trait）

- 优化了部分Verilog的语法，减少了不必要的部分。这具体体现在一些Verilog可能会出现可以通过仿真（testbench）但无法生成比特流（bitstream）的语法结构，如果参与过国科大的体系结构研讨课或者组成原理研讨课应该会对此深有体会，以笔者自身经历为例，前几日在做异常支持的工作时，出现了一处混合使用上升沿和下降沿触发的触发器，最后可以通过仿真，但在综合时却出现了“组合逻辑环”这一致命错误。而chisel在转换成Verilog语言（这是必要的，因为目前没有直接支持chisel的EDA）时，会**避免出现不可综合的语法**。
- 利用Scala的模式匹配、特质混入、类继承等特性，能够迅速改变电路结构

基于以上特点，相较于传统开发，利用chisel开发要具有**更高的效率**和**更少的代码量**。

关于分支预测模块

关于分支预测

分支预测技术是**提高通用处理器性能**的重要方法，本质是克服指令控制相关，提高指令并行度，从而使得处理器的性能得到提高。其重要性可以由以下几点来说明：

- 现在的通用处理器大多采用深度流水线和宽发射机制，两者背后分支预测是关键技术支撑。
- 计算机和计算器最大的区别就在于计算机有了分支指令，使得计算机从简单的数字计算转变为完成各种任务和运算。
- 分支预测技术不仅在高性能通用处理器中采用，而且在嵌入式处理器也广泛采用。

关于香山中的分支预测模块

香山中的分支预测模块是整个前端的重要组成部分，其通过接受并分析当前PC以及历史信息（His），给出预测结果并最终传递给取指令模块。在这里给出目前香山前端的组成：

- frontend
  - Bim
  - BPU
  - Composer（Class）
  - Frontend
  - FrontendBundle
  - FTB
  - Ibuffer

- ICache
- IFU
- ITTAGE
- local
- NewFtq
- PreDecode
- RAS
- SC
- Tage
- uBTB

其中，本学习报告重点阅读的将是Bim、BPU、Composer、FTB、NewFtq、RAS、SC和uBTB部分。

当然，单纯如同报菜名一样列出文件组成是毫无意义的（除了水一水页数），为了进一步说明香山CPU分支预测部分的数据传递和面向对象思想，我将以BPU中的Predictor类为例，简单介绍一下分支预测过程中各个部件的配合。

```
//BPU.scala文件中Predictor类的部分信息
class Predictor(implicit p: Parameters) extends XSModule with HasBPUConst {
  val io = IO(new PredictorIO)

  val predictors = Module(if (useBPD) new Composer else new FakePredictor)
......
}
```

可以注意到，Predictor类继承了抽象类`XSMoudle`（其实基本上等价于chisel给出的`Moudle`类，但多了一层封装；至于chisel给出的`Moudle`类则是用于帮助工程师定义硬件模块的，一切硬件模块都是继承自`Moudle`类的），同时混入了特质（有点类似单例对象，由于Scala没有多重继承而提出的一种解决方案）`HasBPUConst`：

```
//BPU.scala文件中HasBPUConst的部分信息
trait HasBPUConst extends HasXSParameter with HasIFUConst {
  val MaxMetaLength = 1024 // TODO: Reduce meta length
  val MaxBasicBlockSize = 32
  val LHistoryLength = 32
  val numBr = 2
  val useBPD = true
  val useLHist = true
  val shareTailSlot = true
  val numBrSlot = if (shareTailSlot) numBr-1 else numBr
  val totalSlot = numBrSlot + 1
  ......
}
```

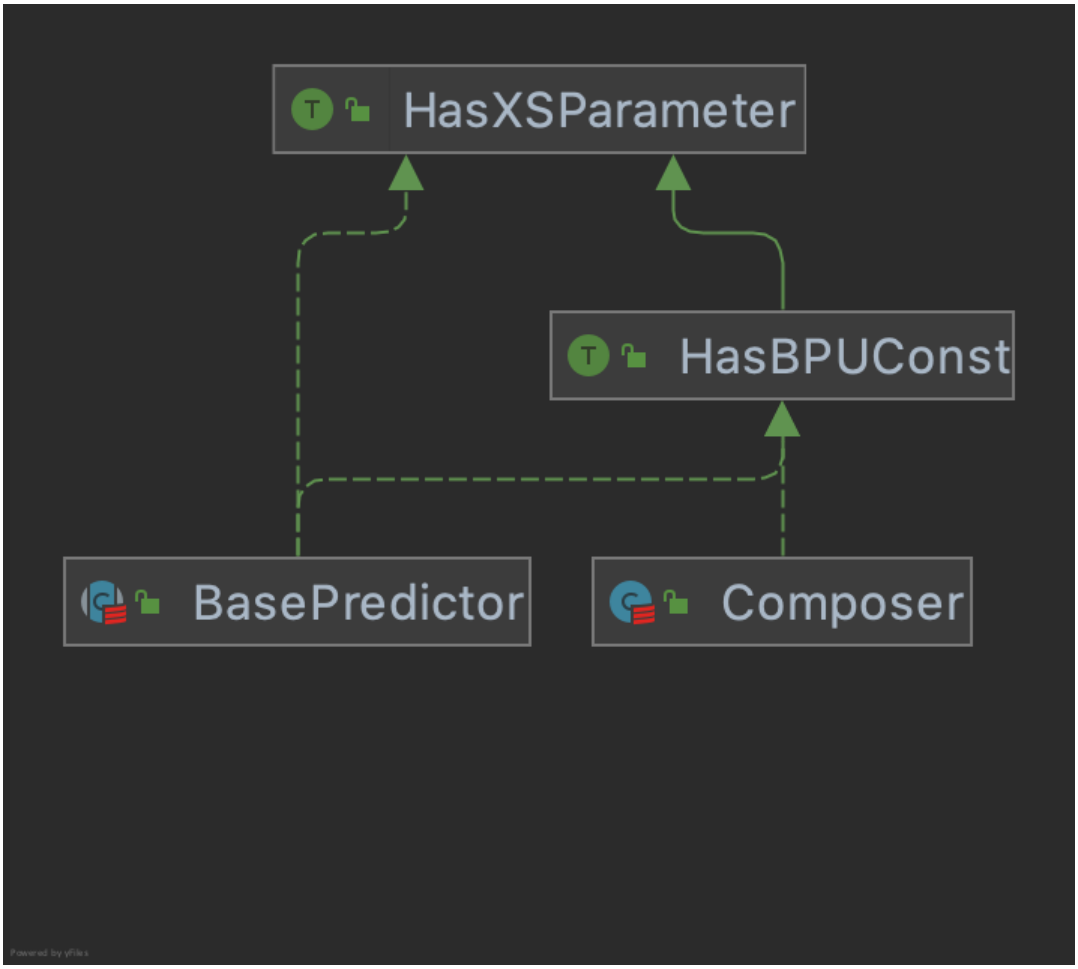
可以注意到，该特质的主要用处是定义一些与BPU相关的变量（可以注意到`useBPU`为`true`），那么回到`Predictor`类，其后的`val io = IO(new PredictorIO)`是定义这个模块的接口，具体可以参考`PredictorIO`并继续向上追索，本处不予以赘述（追索线条过长，而且由于不是以继承的形式传递，因此很大程度上只能面向代码追索）。

随后，其例化了一个模块（用面向对象的语言描述，就是实例化了一个类），该模块为`composer`（因为`useBPU`必然是`true`），因而转到`composer`类进行分析。

```
//BPU.scala文件中HasBPUConst的部分信息
class Composer(implicit p: Parameters) extends BasePredictor with HasBPUConst {
  val (components, resp) = getBPDComponents(io.in.bits.resp_in(0), p)
  io.out.resp := resp
```

```
  ......
}
```

在这里我们注意到，其调用了`getBPDComponents`方法，故而继续追索，此时用UML图简单观察：



composer类的UML图

而`getBPDComponents`方法，就在`HasXSParameter`特质中被定义的。

```
//getBPDComponents方法部分信息
def getBPDComponents(resp_in: BranchPredictionResp, p: Parameters) = {
    coreParams.branchPredictor(resp_in, p)
  }
```

可以注意到，其结果又调用了`XScoreParameters`样例类（在`HasXSParameter`中被实例化）中的方法`branchPredictor`，进而继续追索，到达本次旅途的终点站：

```
//branchPredictor方法
branchPredictor: Function2[BranchPredictionResp, Parameters, Tuple2[Seq[BasePredictor], BranchPredictionResp]] =
    ((resp_in: BranchPredictionResp, p: Parameters) => {
      // val loop = Module(new LoopPredictor)
      // val tage = (if(EnableBPD) { if (EnableSC) Module(new Tage_SC)
      //                             else          Module(new Tage) }
      //             else          { Module(new FakeTage) })
      val ftb = Module(new FTB()(p))
      val ubtb = Module(new MicroBTB()(p))
      val bim = Module(new BIM()(p))
      val tage = Module(new Tage_SC()(p))
      val ras = Module(new RAS()(p))
```

```
      val ittage = Module(new ITTage()(p))
      // val tage = Module(new Tage()(p))
      // val fake = Module(new FakePredictor()(p))

      // val preds = Seq(loop, tage, btb, ubtb, bim)
      val preds = Seq(bim, ubtb, tage, ftb, ittage, ras)
      preds.map(_.io := DontCare)

      // ubtb.io.resp_in(0)  := resp_in
      // bim.io.resp_in(0)   := ubtb.io.resp
      // btb.io.resp_in(0)   := bim.io.resp
      // tage.io.resp_in(0)  := btb.io.resp
      // loop.io.resp_in(0)  := tage.io.resp
      bim.io.in.bits.resp_in(0)  := resp_in
      ubtb.io.in.bits.resp_in(0) := bim.io.out.resp
      tage.io.in.bits.resp_in(0) := ubtb.io.out.resp
      ftb.io.in.bits.resp_in(0)  := tage.io.out.resp
      ittage.io.in.bits.resp_in(0)  := ftb.io.out.resp
      ras.io.in.bits.resp_in(0) := ittage.io.out.resp

      (preds, ras.io.out.resp)
    })
```

在这里我们可以了解到分支预测部件和分支预测整体模块关系的一隅：在这里，各种分支预测部件被实例化（比如Bim、ITTage等），他们的结果则是由**覆盖重定向**得出的。这里简单介绍一下不同部件的作用：

- bim：进行方向预测
- ubtb：存储跳转地址并进行跳转
- tage：更先进（准确）的方向预测
- ittage：处理间接跳转（比如jalr）
- ras：也是处理间接跳转，可以处理syscall/ertn等，正确率相当高

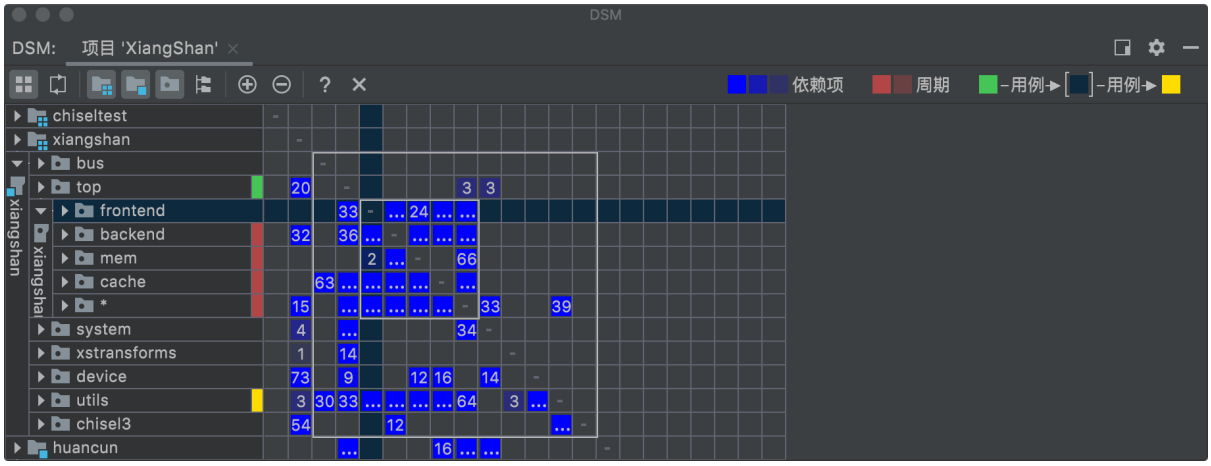
当多级预测产生不同的历史结果时，用后级覆盖并冲刷前级流水线，从而保证每次预测使用的历史准确，进而提高预测成功率。

至于分支预测部分在整个前端中的体现，则体现在`frounted`文件中对其的实例化：

```
//decouped-frontend modules
  val bpu     = Module(new Predictor)
  val ifu     = Module(new NewIFU)
  val ibuffer =  Module(new Ibuffer)
  val ftq = Module(new Ftq)
```

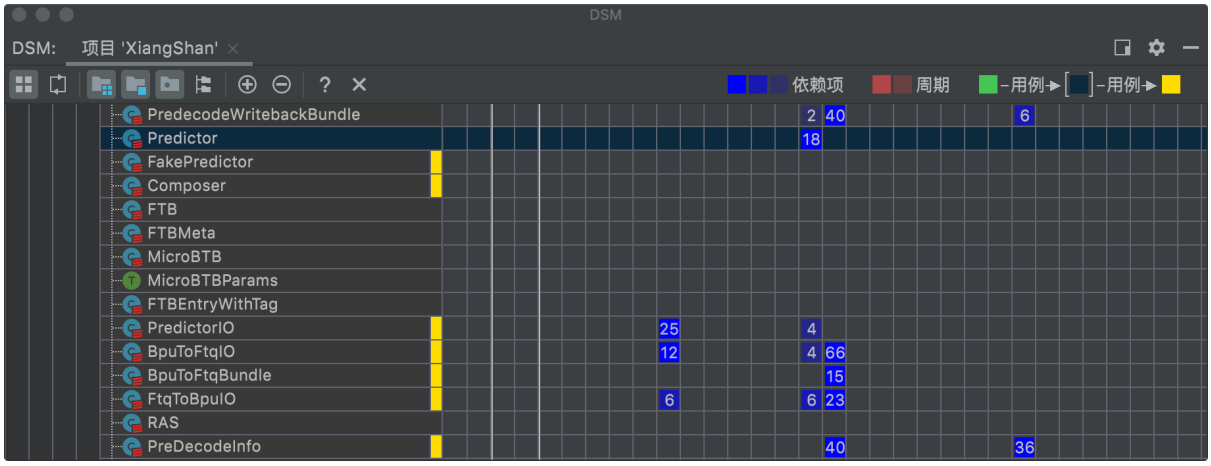
现在，BPU中的`Pridictor`类与其他各个分支预测部件的关系已经清晰不少：Predictor根据继承`HasBPUConst`的信息决定实例化`Composer`类，而`Composer`类继承了`HasXSParameter`，并通过调用其中的`getBPDComponents`方法来进一步获取分支跳转结果，而`getBPDComponents`则调用`branchPredictor`方法，其中例化了各个分支预测部件并通过**覆盖重定向**的方法给出最终的结果。

以上的分析也可以通过依赖分析矩阵予以侧面验证，一下展示部分依赖分析：



前端的依赖分析

前端部分与后端、存储器和Cache相互间都有交互（依赖），top模块则依赖前端，这符合FPGA的设计思路



Predictor的依赖分析

`Predictor`类依赖于`Composer`类提供的操作和数据结构，依赖于`PreDecode`类提供的预译码信息，同时也依赖于几个IO类的接口设置。

可以注意到同层次的类相互间各有关联，不同层次的类之间相互调用，利用接口达到解耦的效果。

导言小结

导言部分重点介绍了香山CPU开源项目，chisel语言以及分支预测部分，并以BPU中的Predictor为例管窥蠡测此项目的类、对象、实例化、继承和封装。

其实可以注意到，各个模块间的协同配合很少使用**继承**语法，被继承最多的往往是常用参数的声明类或特质（比如`HasXSParameter`），其余大部分时候都是会选择使用例化（实例化）等语法，这不仅满足开放封闭原则、高内聚低耦合等面向对象要求的原则，也符合硬件设计上的习惯，同时，硬件设计的要求又使之必须满足接口功能单一、接口隔离原则（毕竟如果BPU能随便把ICatch内容修改一番，那混乱的指令足以立刻使CPU宕机）。

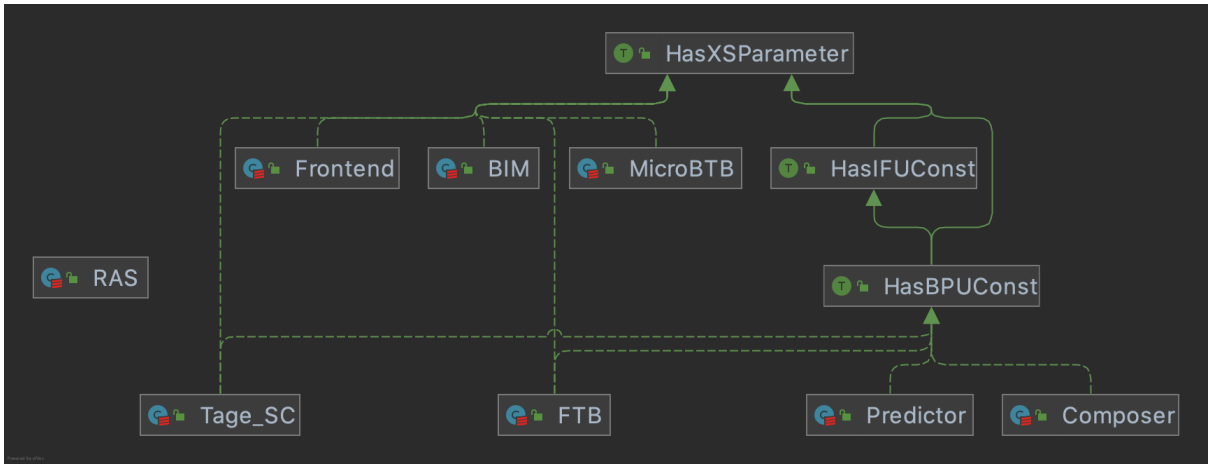
简言之，无论是chisel的语法设计还是香山项目对其的运用，都无时无刻不在硬件设计思路的影响下体现着面向对象的宗旨：**抽象和封装**。

```
//开个玩笑
import chisel3._

Class Open_XiangShan extends OOD with FPGA{}
```

香山CPU是个仍在不断进步的开源项目，事实上，笔者阅读的时候甚至还有幸多次目击了项目组git push的现场（IDEA突然弹出有新的提交内容，是否git pull），而且目前的架构也已经与在repo中的早期架构图有了不少出入，这一方面增大了我学习和阅读的难度，但另一方面也给了我目睹一个开源项目逐渐成长的机会。在之后的报告中，我将重点基于分支预测部分，学习其代码中体现的面向对象的思想和FPGA的设计思路等。

不过，笔者之前甚少接触面向对象语言，一直以来一直以C和Verilog作为主要的练习对象，两者一个是严格的面向过程语言，另一个虽然有大量的封装，但没有继承和多态，也称不上是面向对象语言，因此只能在阅读中学习，不仅仅是代码的语法和技巧，更重要的是面向对象的思想和设计思路。

最后用一个本次涉及到的所有的类、特质所构成的UML图结束此部分。



Part1:香山BPU设计的面向对象设计和核心流程

1. 类和对象

首先我们给出BPU.scala中的所有类：

```
class BPUCtrl(implicit p: Parameters) extends XSBundle{}

class BasePredictorInput (implicit p: Parameters) extends XSBundle with HasBPUConst{}

class BasePredictorOutput (implicit p: Parameters) extends XSBundle with HasBPUConst {}

class BasePredictorIO (implicit p: Parameters) extends XSBundle with HasBPUConst {}

abstract class BasePredictor(implicit p: Parameters) extends XSModule with HasBPUConst with BPUUtils {}

class FakePredictor(implicit p: Parameters) extends BasePredictor {}//此类可以忽略掉

class BpuToFtqIO(implicit p: Parameters) extends XSBundle {}

class PredictorIO(implicit p: Parameters) extends XSBundle {}

class FakeBPU(implicit p: Parameters) extends XSModule with HasBPUConst {}//此类可以忽略

class Predictor(implicit p: Parameters) extends XSModule with HasBPUConst {}
```

以BPUCtrl为例

```
class BPUCtrl(implicit p: Parameters) extends XSBundle {
  val ubtb_enable = Bool()
  val btb_enable  = Bool()
  val bim_enable  = Bool()
  val tage_enable = Bool()
  val sc_enable   = Bool()
  val ras_enable  = Bool()
```

```
  val loop_enable = Bool()
}
```

讨论类自然离不开抽象，具体到这个类，则很明显可以看出是关于分支预测部件**使能**的抽象，即若相应信号拉高（如`ubtb_enable`为true），则相应部件可用（如可以使用`ubtu`部件），这个类中只有对**字段**的定义。

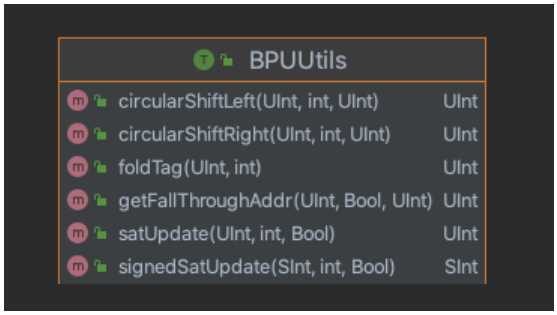
（这里IDEA似乎出了一些问题，其无法正确识别字段和方法，导致自动生成的UML图出现问题）

再以一个`trait`为例：

> `trait`简单介绍：特质是Scala代码复用的基本单元，将方法和字段定义封装起来，然后将他们通过混入类的方式来实现复用。就组成结构而言，他们和类还是很相似的，但其主要的用处是为了解决多重继承问题。个人理解：`trait`也可以当作一种**抽象**，在很大程度上**可以与类进行类比**。

```
trait BPUUtils extends HasXSParameter {
  // circular shifting
  def circularShiftLeft(source: UInt, len: Int, shamt: UInt): UInt = {
    val res = Wire(UInt(len.W))
    val higher = source << shamt
    val lower = source >> (len.U - shamt)
    res := higher | lower
    res
  }

  def circularShiftRight(source: UInt, len: Int, shamt: UInt): UInt = {
    val res = Wire(UInt(len.W))
    val higher = source << (len.U - shamt)
    val lower = source >> shamt
    res := higher | lower
    res
  }

  // To be verified
  def satUpdate(old: UInt, len: Int, taken: Bool): UInt = {
    val oldSatTaken = old === ((1 << len)-1).U
    val oldSatNotTaken = old === 0.U
    Mux(oldSatTaken && taken, ((1 << len)-1).U,
      Mux(oldSatNotTaken && !taken, 0.U,
        Mux(taken, old + 1.U, old - 1.U)))
  }

  def signedSatUpdate(old: SInt, len: Int, taken: Bool): SInt = {
    val oldSatTaken = old === ((1 << (len-1))-1).S
    val oldSatNotTaken = old === (-(1 << (len-1))).S
    Mux(oldSatTaken && taken, ((1 << (len-1))-1).S,
      Mux(oldSatNotTaken && !taken, (-(1 << (len-1))).S,
        Mux(taken, old + 1.S, old - 1.S)))
  }

  def getFallThroughAddr(start: UInt, carry: Bool, pft: UInt) = {
    val higher = start.head(VAddrBits-log2Ceil(PredictWidth)-instOffsetBits-1)
    Cat(Mux(carry, higher+1.U, higher), pft, 0.U(instOffsetBits.W))
  }

  def foldTag(tag: UInt, l: Int): UInt = {
    val nChunks = (tag.getWidth + l - 1) / l
    val chunks = (0 until nChunks).map { i =>
      tag(min((i+1)*l, tag.getWidth)-1, i*l)
    }
    ParallelXOR(chunks)
  }
}
```

简单来说，这是关于BPU模块中可能用到的方法（函数）的抽象，但由于现阶段阅读的重点并不在函数的功能，因此无法给出这些方法的具体作用。但作为对类的说明，其可以补充`BPUCtrl`未涉及的部分：类中对**方法**的定义。



BPUUtils的UML图

至于对象，则是对抽象的类的实例化，直接以导论中的

```
    val ftb = Module(new FTB()(p))
    val ubtb = Module(new MicroBTB()(p))
    val bim = Module(new BIM()(p))
    val tage = Module(new Tage_SC()(p))
    val ras = Module(new RAS()(p))
    val ittage = Module(new ITTage()(p))
```

为例，可以在BIM.scala文件中找到：

```
class BIM(implicit p: Parameters) extends BasePredictor with BimParams with BPUUtils {}
```

在`branchPredictor`方法中，`BIM`被例化（实例化）为一个对象，其结果也在之后的预测结果中被使用（或者被覆盖）。

2.关于继承

在香山项目中，最重要的父类自然是`Module`类相关的部分，毕竟对于硬件设计而言，模块化是断然不可缺少的，而chisel语言也已经提供了一个`Module`类供程序员使用。

对于香山而言，在BPU乃至于整个设计中，通用的`Module`类被命名为`XSModule`，为一个抽象类，继承自chisel的`Module`类并混入了许多其他特质，如`HasXSParameter`，`HasExceptionNO`以及`HasFPUParameters`。而其本身又声明了一个抽象方法：`io`。具体关于抽象方法和抽象类的讨论，我将放在下一个小节。

```
abstract class XSModule(implicit val p: Parameters) extends MultiIOModule
  with HasXSParameter
  with HasExceptionNO
  with HasFPUParameters {
  def io: Record
}
```

继承的意义在于可以使子类使用绝大多数父类的字段和方法，具体到此处，任何在CPU设计中的模块都继承自`XSModule`，这些模块都可以使用其中定义的方法，比如`io`以及其他设计所必须的参数等。

对于硬件设计而言，每个模块都必须有对应的端口，而chisel语言解决端口的定义问题则依赖于Bundle类，而对于香山项目，又为此定义了一个抽象类`XSBundle`

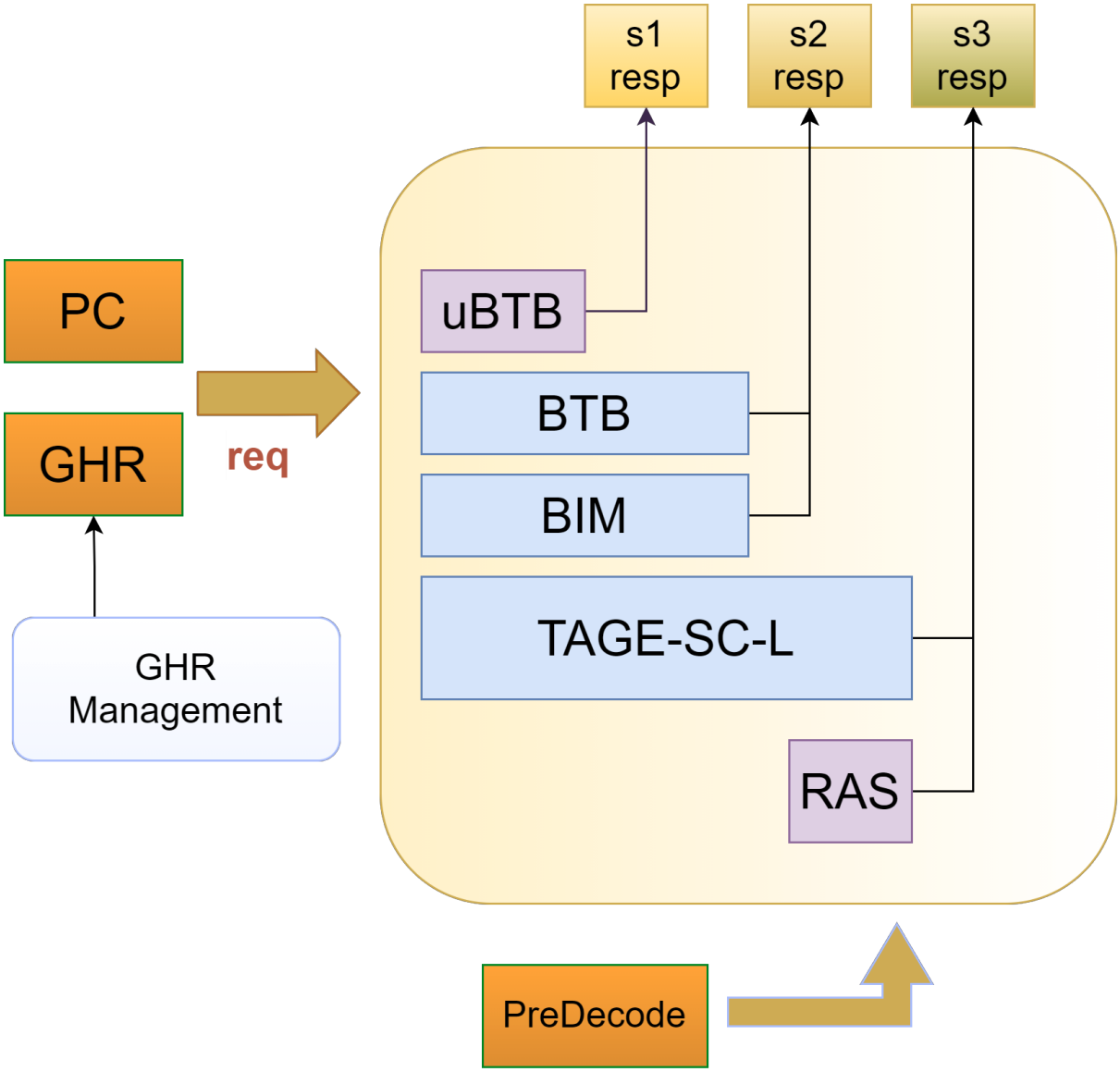
```
abstract class XSBundle(implicit val p: Parameters) extends Bundle
  with HasXSParameter
```

与`XSModule`类似，`XSBundle`也是基于chisel原有的`Bundle`块，并混入了与香山设计相关的一些因素（`HasXSParameter`）

```
trait HasXSParameter {

  implicit val p: Parameters

  val coreParams = p(XSCoreParamsKey)
  val env = p(DebugOptionsKey)

  val XLEN = coreParams.XLEN
  val hardId = coreParams.HartId
  val minFLen = 32
  val fLen = 64
  def xLen = XLEN

  val HasMExtension = coreParams.HasMExtension
  val HasCExtension = coreParams.HasCExtension
  val HasDiv = coreParams.HasDiv
  val HasIcache = coreParams.HasICache
  val HasDcache = coreParams.HasDCache
  val AddrBits = coreParams.AddrBits // AddrBits is used in some cases
  val VAddrBits = coreParams.VAddrBits // VAddrBits is Virtual Memory addr bits
  val PAddrBits = coreParams.PAddrBits // PAddrBits is Phyical Memory addr bits
  val AsidLength = coreParams.AsidLength
  val AddrBytes = AddrBits / 8 // unused
  val DataBits = XLEN
  val DataBytes = DataBits / 8
  val HasFPU = coreParams.HasFPU
  val HasCustomCSRCacheOp = coreParams.HasCustomCSRCacheOp
  val FetchWidth = coreParams.FetchWidth
  val PredictWidth = FetchWidth * (if (HasCExtension) 2 else 1)
  val EnableBPU = coreParams.EnableBPU
  val EnableBPD = coreParams.EnableBPD // enable backing predictor(like Tage) in BPUStage3
  val EnableRAS = coreParams.EnableRAS
  val EnableLB = coreParams.EnableLB
  val EnableLoop = coreParams.EnableLoop
  val EnableSC = coreParams.EnableSC
  val EnbaleTlbDebug = coreParams.EnbaleTlbDebug
  val HistoryLength = coreParams.HistoryLength
  val PathHistoryLength = coreParams.PathHistoryLength
  val BtbSize = coreParams.BtbSize
  ......
}
```

可以注意到，在此段代码中，又声明了诸如`PredictWidth`、`EnableBPU`、`EnableSC`等在分支预测相关的部分有控制作用的变量。至于对这些变量的赋值等操作，则在更顶层的部分。

起初我对继承的理解存在很大的误解，以为继承应当是最常用的类关系，因为任何一个CPU模块都要**依赖**于其下的子模块，如ALU、Cache等，而继承的字面定义则为“如果类A具有类B的**全部属性和全部操作**，而且具有自己特有的某些属性或操作，则A叫做B的特殊类，B叫做A的一般类”，而CPU在功能上确实具有子模块的“**全部操作**”，具体在结构设计图上，引用设计文档中的设计图，其关系大致为：



香山的设计文稿中给出的分支预测部分的设计图，但需要注意的是，目前的结构已经和设计文档中的结构有了很大不同，比如此图涉及到三级流水，但实际上目前的香山CPU分支预测部件的设计已经是四级流水了

因此起初我猜测，BPU类应当继承自BIM、类uBTB等，但实际上，上述的类都是以实例化的形式在BPU、Pridictor等类中使用的，被继承，或者说最常作为父类（超类）的，基本上就是前文列举出的`XSModule`和`XSBundle`类。而之所以如此，是因为继承所带来的**耦合度是很高**的，会严重违反高内聚、低耦合的设计原则。试想，若香山分支预测部分的设计很不幸的按照我设想的，各模块之间以继承关系为主的方式进行设计，那么在项目组关于分支预测流水级的改动中，就很可能会出现仅仅因为流水级增加了一级，所有的部件都需要进行复杂的改动，哪怕其实他们的主要工作就是在一个流水级内完成的。当然，可以通过覆写等方法在一定程度上缓解此问题，但依然是治标不治本，盲目的使用继承大概率会带来维护上的严重问题。

3.关于抽象类和抽象方法

首先给出一个BPU中的抽象类

```
abstract class BasePredictor(implicit p: Parameters) extends XSModule with HasBPUConst with BPUUtils {
  val meta_size = 0
  val spec_meta_size = 0

  val io = IO(new BasePredictorIO())

  io.out.resp := io.in.bits.resp_in(0)
```

```
  io.out.s3_meta := 0.U

  io.in.ready := !io.redirect.valid

  io.s1_ready := true.B
  io.s2_ready := true.B
  io.s3_ready := true.B

  val s0_pc       = WireInit(io.in.bits.s0_pc) // fetchIdx(io.f0_pc)
  val s1_pc       = RegEnable(s0_pc, resetVector.U, io.s0_fire)
  val s2_pc       = RegEnable(s1_pc, io.s1_fire)
  val s3_pc       = RegEnable(s2_pc, io.s2_fire)
}
```

遗憾的是，在整个分支预测乃至于前端部分的代码中，我也只能找到这么一个并不正统的抽象类，因为其中并没有抽象方法。不过，我们依然可以通过分析其的使用，来进一步学习抽象类的用法和作用。

在这个抽象类中，声明了一些与PC相关的变量，确定了端口以及一些和流水线相关的握手信号，有趣的是，一般情况下，我们可以把接口理解为特殊的抽象类，而这里的`val io = IO(new BasePredictorIO())`正是在对接口进行定义。之后我们来分析一下该抽象类的使用：

```
class BIM(implicit p: Parameters) extends BasePredictor with BimParams with BPUUtils
//进行方向预测的模块
class Composer(implicit p: Parameters) extends BasePredictor with HasBPUConst
//正如其名，是综合预测结果的一个模块
class FTB(implicit p: Parameters) extends BasePredictor with FTBParams with BPUUtils with HasCircularQueuePtrHelper
//从目标缓存中取指的一个模块
abstract class BaseITTage(implicit p: Parameters) extends BasePredictor with ITTageParams with BPUUtils
//一个比较先进的，用于处理jalr间接跳转类的预测模块，至于这里为何也是个抽象类
//是因为似乎目前该部分并没有实现完成，其内部被完全注释掉了
class RAS(implicit p: Parameters) extends BasePredictor
//也是用于处理间接跳转的，但更先进
class MicroBTB(implicit p: Parameters) extends BasePredictor
  with MicroBTBParams
//分支预测目标寄存器模块
```

以下截取BIM中的一部分进行分析

```
  bim.io.r.req.valid := io.s0_fire
  bim.io.r.req.bits.setIdx := s0_idx

  io.in.ready := bim.io.r.req.ready
  io.s1_ready := bim.io.r.req.ready

  val s1_read = bim.io.r.resp.data

  io.out.resp := io.in.bits.resp_in(0)
```

需要注意的是，正如导论中所说的，香山CPU设计的分支预测部分是在多个流水级中进行的，每个流水级中有不同的分支预测部分和分支预测结果，在之后的流水级中不断的重覆盖。因此在此处仅仅是对s1流水级的握手信号进行修改，并由`resp_in(0)`赋值给`resp`。

之后截取uBTB中的一部分进行分析

```
  io.s1_ready := bank.read_pc.ready

  io.out.resp := io.in.bits.resp_in(0)
  io.out.resp.s1.pc := s1_pc
  io.out.resp.s1.preds.hit := bank.read_hit
  io.out.resp.s1.ftb_entry := read_entry.entry
  io.out.resp.s1.preds.fromFtbEntry(read_entry.entry, s1_pc)
```

uBTB模块起到记录跳转地址并进行跳转的工作，而其和BIM一样，也主要在第一级流水完成预测工作，因此此处其也对第一级流水级的握手信号进行修改，同时也对第一级流水级预测出的pc进行修改，在之后的流水级中被采纳或者覆盖。此外还对历史记录进行更新，方便之后ITTage的预测。

根据以上两个类中对抽象类BasePredictor的使用，可以分析出：

- BasePredictor类为其子类，也就是更加具体的分支预测部件提供了一个通用，无论是简单或者复杂的分支预测部件都可以基于其进行操作和扩展
- 同时其也在一定程度上限制了子类的框架，比如BIM和uBTB的握手信号设计，输入输出设计和pc的赋值（也即resp的内容等）都有了一定的标砖，不至于过于随意，导致之后的覆盖出现问题。

此外也可以根据此进一步了解抽象类的几个特点：

- 抽象类中可以包含普通方法(有具体实现的方法)，**当然也可以不包含抽象方法**
- 抽象类不能被实例化，即不能用new来实例化抽象类
- 抽象类可以包含属性、方法、构造方法。但是构造方法不能用new来实例，只能用来被子类调用
- 抽象类只能用来被继承

此外，在前端中，上述分支预测部件的`Module`和`Bundle`也都被以抽象类的形式声明。

遗憾的是，由于在此部分没有抽象方法，因此无法对抽象方法的特性进行更进一步的分析。

4.关于多重继承

首先，在chisel，或者说Scala语法中并**不存在多重继承**这个概念，而是以特质和混入的方法实现。首先简单回顾一下概念：

- 特质是Scala中代码复用的基础单元，将方法和字段定义封装起来，然后以混入的方式实现复用
- 类继承要求继承自一个明确的父类，但混入却允许混入多个特质以实现代码复用

以下给出BPU中的一个trait

```
trait HasBPUConst extends HasXSParameter with HasIFUConst {
  val MaxMetaLength = 1024 // TODO: Reduce meta length
  val MaxBasicBlockSize = 32
  val LHistoryLength = 32
  val numBr = 2
  val useBPD = true
  val useLHist = true
  val shareTailSlot = true
  val numBrSlot = if (shareTailSlot) numBr-1 else numBr
  val totalSlot = numBrSlot + 1

  def BP_STAGES = (0 until 3).map(_.U(2.W))
  def BP_S1 = BP_STAGES(0)
  def BP_S2 = BP_STAGES(1)
  def BP_S3 = BP_STAGES(2)
  val numBpStages = BP_STAGES.length

  val debug = true
  val resetVector = 0x10000000L//TODO: set reset vec
  // TODO: Replace log2Up by log2Ceil
}
```

正如其名，该特质中定义了在BPU中会使用到的绝大多数参数，其继承自超类`HasXSParameter`，因此此特质只能被混入继承自`HasXSParameter`的类，关于此可以在对其的用法中体现，例如之后要提及的`HasBPUParameter`类。

```
trait HasBPUParameter extends HasXSParameter with HasBPUConst {
  val BPUDebug = true && !env.FPGAPlatform && env.EnablePerfDebug
  val EnableCFICommitLog = true
  val EnbaleCFIPredLog = true
  val EnableBPUTimeRecord = (EnableCFICommitLog || EnbaleCFIPredLog) && !env.FPGAPlatform
```

```
  val EnableCommit = false
}
```

对于此类，可以注意到其直接调用了在特质`HasXSParameter`中声明的`env`变量，换言之，在对变量的使用和代码的复用上，特质的混入和直接的对类的继承并没有显著区别，而这个类中则负责声明一些BPU相关的杂项。

理论上，特质可以直接调用其所继承的抽象类中的抽象方法，但遗憾的是，在香山的分支预测部件中，没有合适的抽象方法予以分析，因此这个能体现出Scala特色的方法无法被展示。

回到项目本身，可以注意到另一个特质的特点，即**线性化**。

以`HasBPUParameter`为例，其会先调用最右侧，也即HasBPUConst类中所声明的变量，原因也很简单，对于具体的BPU模块，我们肯定希望其优先调用专为其特化处理的`BPUConst`而非针对整个项目所设计的`XSParameter`，而对于一般的多继承，对于调用的顺序并没有很好的解决方案。

此时会产生一个新的问题，我们需要在什么时候用特质，而在什么时候使用抽象类呢？对于Scala语法而言，其并没有严格的规定，那么在香山项目中，我们可以试图去摸索出一点规律。

在本章的3节中，`BasePredictor`是以抽象类的形式被声明和构建的，其主要特点是在数个具有一定关联的模块中被复用，比如BIM、ITTage等，最终都需要统一的为前端提供分支预测结果；而在4节中所提到的`HasXSParameter`和`HasBPUConst`，都被用在整个BPU的多数模块中，这些模块可能彼此之间几乎没有关联，但的确都需要用到这些常数。那么，结论已经呼之欲出：

> 对于一些可复用的变量和行为，如果其可能被用于**多个互不相关的类**，用特质。

5.封装与模块——面向对象与FPGA设计的结合

- 本部分内容以注释的形式写在代码块里

```
class ITTageTable
(
  val nRows: Int, val histLen: Int, val tagLen: Int, val uBitPeriod: Int, val tableIdx: Int
)(implicit p: Parameters)
  extends ITTageModule with HasFoldedHistory {
//声明接口
  val io = IO(new Bundle() {
    val req = Input(Valid(new ITTageReq))
    val resp = Output(Valid(new ITTageResp))
    val update = Input(new ITTageUpdate)
  })

......

}

//Input(Valid(new ITTageReq))对应的类
class ITTageReq(implicit p: Parameters) extends ITTageBundle {
  val pc = UInt(VAddrBits.W)
  val hist = UInt(HistoryLength.W)
  val phist = UInt(PathHistoryLength.W)
}

//Output(Valid(new ITTageResp))对应的类
class ITTageResp(implicit p: Parameters) extends ITTageBundle {
  val ctr = UInt(ITTageCtrBits.W)
  val u = UInt(2.W)
  val target = UInt(VAddrBits.W)
}

//Input(new ITTageUpdate)对应的类
class ITTageUpdate(implicit p: Parameters) extends ITTageBundle {
  val pc = UInt(VAddrBits.W)
  val hist = UInt(HistoryLength.W)
  val phist = UInt(PathHistoryLength.W)
  // update tag and ctr
  val valid = Bool()
  val correct = Bool()
  val alloc = Bool()
```

```
  val oldCtr = UInt(ITTageCtrBits.W)
  // update u
  val uValid = Bool()
  val u = UInt(2.W)
  // target
  val target = UInt(VAddrBits.W)
  val old_target = UInt(VAddrBits.W)
}

//例化ITTageTable
val tables = TableInfo.zipWithIndex.map {
    case ((nRows, histLen, tagLen), i) =>
      // val t = if(EnableBPD) Module(new TageTable(nRows, histLen, tagLen, UBitPeriod)) else Module(new FakeTageTable)
      val t = Module(new ITTageTable(nRows, histLen, tagLen, UBitPeriod, i))
      // t.io.req.valid := io.pc.valid
      // t.io.req.bits.pc := io.pc.bits
      // t.io.req.bits.hist := io.hist
      // t.io.req.bits.mask := io.inMask
//由于之前的声明方式是（Validxxx），因此在这里需要有一个握手信号的确定，
//io.s0_fire的含义为s0级的握手信号对接成功
//而后通过pc，hist和phist三个在ITTageReq中定义的字段作为接口，向ITTageTable模块传递信息
//在这里就可以体现出和Verilog，或者说FPGA编程一样的封装思想：除了这几个接口，内部的信息是不会被
//直接访问的，也不会对外暴露的，也就是“信息隐藏”和“高内聚、低耦合”的思想的体现
      t.io.req.valid := io.s0_fire
      t.io.req.bits.pc := s0_pc
      t.io.req.bits.hist := io.in.bits.ghist
      t.io.req.bits.phist := io.in.bits.phist
  }
```

> ITTageTable的作用：之前已经简单提及ITTage部件的功能：根据64位历史等构成的历史表来预测分支跳转，尤其是间接跳转的目标地址，这里的ITTageTable正是在ITTage类中被例化，并作为记录预测目标地址的表被更新的。

6. 核心流程分析

对香山的各种细节的分析进行到这里，已经可以回到总体的设计思路了。分支预测模块作为前端中提供下一个pc的部件，其宏观上的工作流程为：从ICache中获取指令（并记录一些历史信息），在内部维护一个nextPC以及FTQ指针，经内部几级流水中不同的分支预测部件预测，并结果重覆盖后，将预测信息送入FTQ。

```
class Ftq(implicit p: Parameters) extends XSModule with HasCircularQueuePtrHelper
  with HasBackendRedirectInfo with BPUUtils with HasBPUConst {
  val io = IO(new Bundle {
    val fromBpu = Flipped(new BpuToFtqIO)
    val fromIfu = Flipped(new IfuToFtqIO)
    val fromBackend = Flipped(new CtrlToFtqIO)

    val toBpu = new FtqToBpuIO
    val toIfu = new FtqToIfuIO
    val toBackend = new FtqToCtrlIO
  ......
}
```

FTQ类的任务是接收来自BPU的入队信息，包括各种PC、预测信息等，理论上需要维护三个方向的信息，即BPU（`fromBpu`、`toBpu`）、IFU（`fromIfu`、`toIfu`）和Backend（`fromBackend`、`toBackend`）。

```
  val bpu     = Module(new Predictor)
  val ifu     = Module(new NewIFU)
  val ibuffer =  Module(new Ibuffer)
  val ftq = Module(new Ftq)

  //IFU-Ftq
  ifu.io.ftqInter.fromFtq <> ftq.io.toIfu
  ftq.io.fromIfu          <> ifu.io.ftqInter.toFtq
  bpu.io.ftq_to_bpu       <> ftq.io.toBpu
  ftq.io.fromBpu          <> bpu.io.bpu_to_ftq
  //IFU-ICache
```

```
  ifu.io.icacheInter.toIMeta    <>      icache.io.metaRead.req
  ifu.io.icacheInter.fromIMeta  <>      icache.io.metaRead.resp
  ifu.io.icacheInter.toIData    <>      icache.io.dataRead.req
  ifu.io.icacheInter.fromIData  <>      icache.io.dataRead.resp
```

上述的流程可以在前端的总体设计模块中得到体现，BPU的预测传递给FTQ，而FTQ通过与IFU的交互以及IFU和ICache的交互实现取址。在整个流程下，可以注意到香山项目利用chisel将封装、模块化、继承和混入等方法运用到了极致，各个模块各司其职，通过实例化来完成彼此之间的交互，而混入和继承又有效实现了一些参数和方法的复用（虽然混入的优势并没有被完全展现出来）。

至于对具体细节的分析，涉及到对各个模块的硬件设计的分析，一方面，我目前并没能搞清楚一些具体细节，因为在预测过程中有很多信息似乎并不是来自分支预测部件甚至于是整个前端，因此给我的分析带来了不少麻烦；另一方面，电路的具体设计其实本质上依然是硬件描述，对这一部分刨根问底也和本课程，也即面向对象的思想关系不大；最后，其实在前文中，我已经对如BIM、ITTage相关的部分做了部分介绍，并进行了一定的依赖分析，在一定程度上也算是对细节的展示。故不在此进行更深入的展开了。

Part2:香山CPU项目中体现出的面向对象设计思想