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。

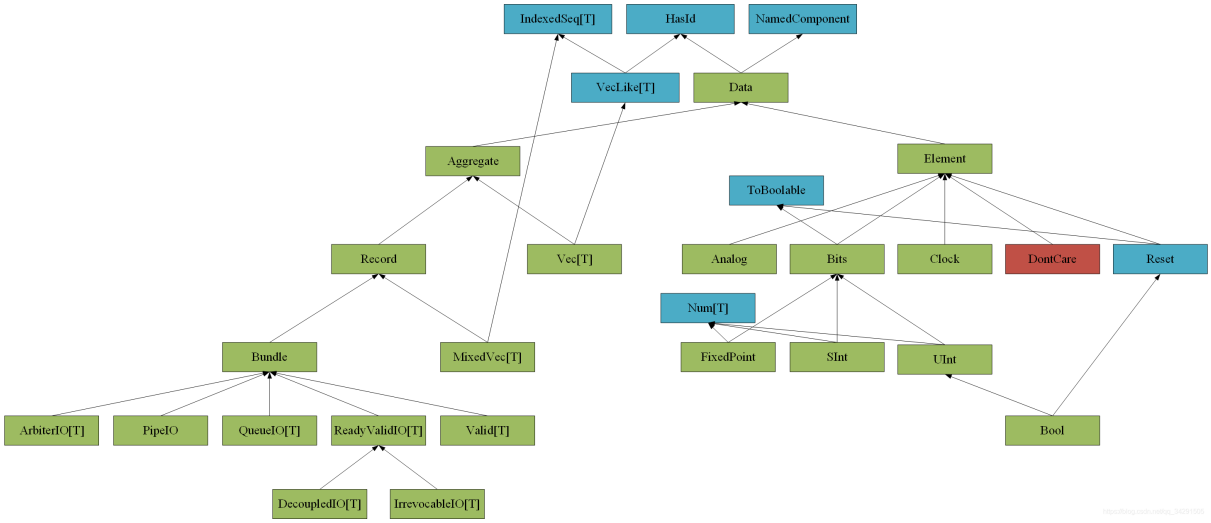
香山处理器坚持**开源策略**，坚定地开源所有的设计、验证、基础工具代码。在Chisel开源社区的现有总线工具、浮点运算单元、系统缓存等基础上，修改完善了它们的功能，同时优化了频率、吞吐等性能指标。香山**积极地拥抱开源社区**，并欢迎来自社区的贡献。

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
  ......
}
```

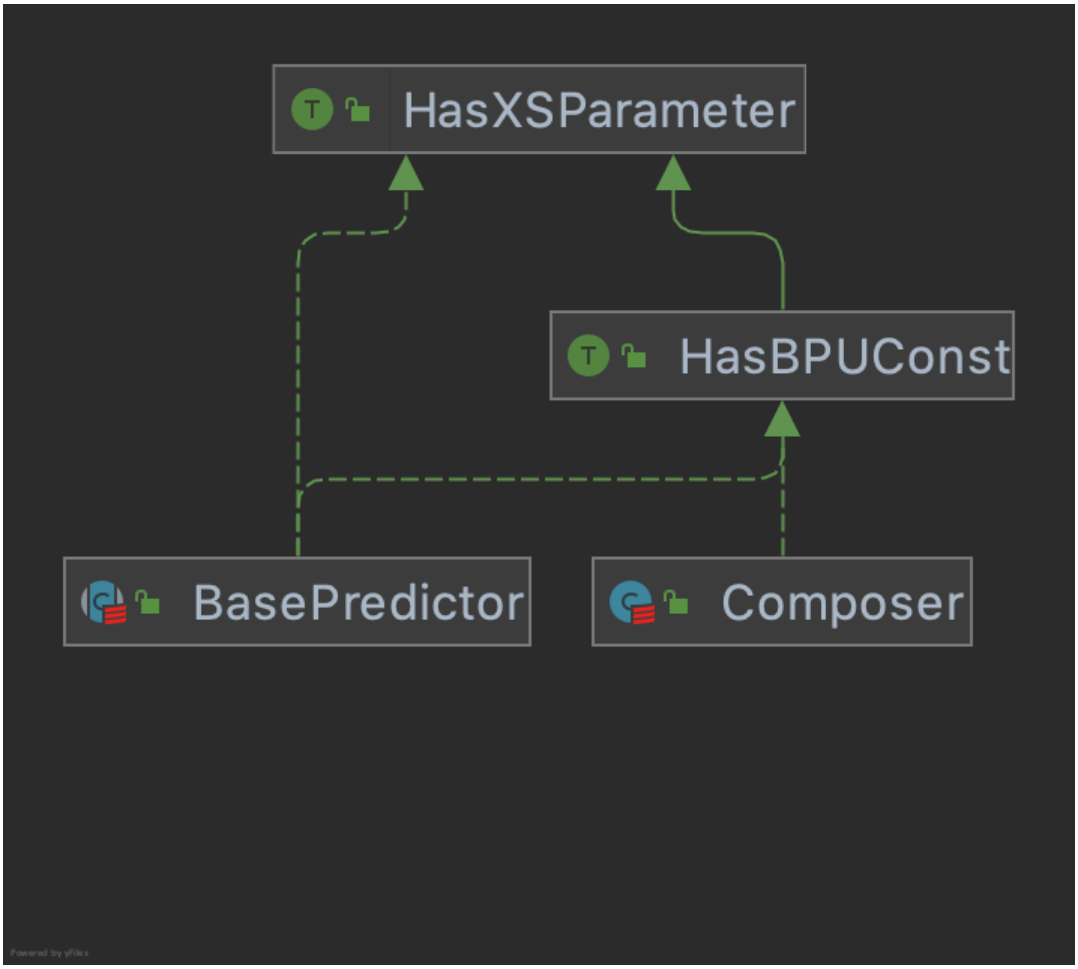
可以注意到，该特质的主要用处是定义一些与BPU相关的变量（可以注意到`useBPU`为`true`），那么回到`Predictor`类，其后的`val io = IO(new PredictorIO)`是定义这个模块的接口，具体可以参考`PredictorIO`并继续向上追索，本处不予以赘述（追索线条过长，而且由于不是以继承的形式传递，因此很大程度上只能面向代码追索）。

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

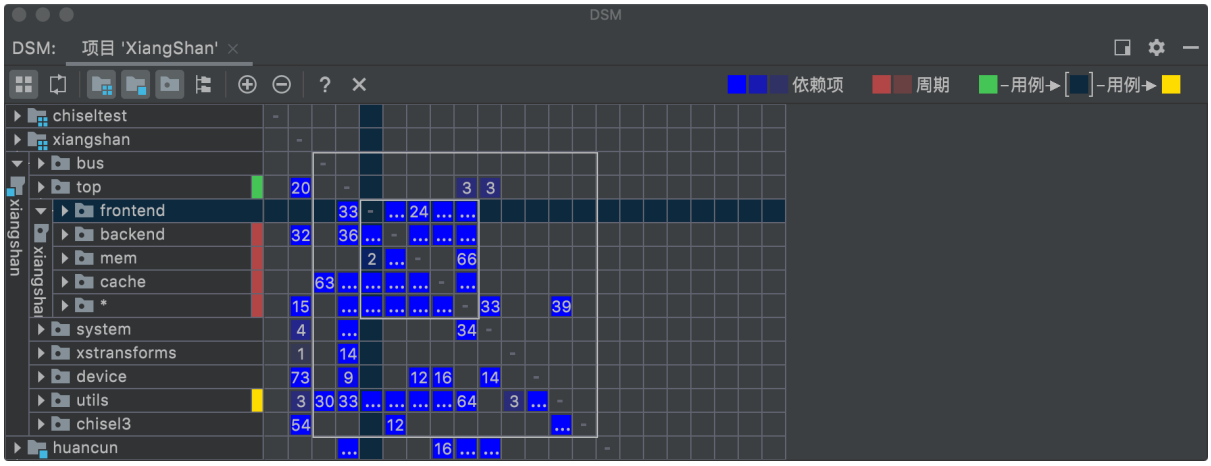
当多级预测产生不同的历史结果时，用后级覆盖并冲刷前级流水线，从而保证每次预测使用的历史准确，进而提高预测成功率。

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)
```

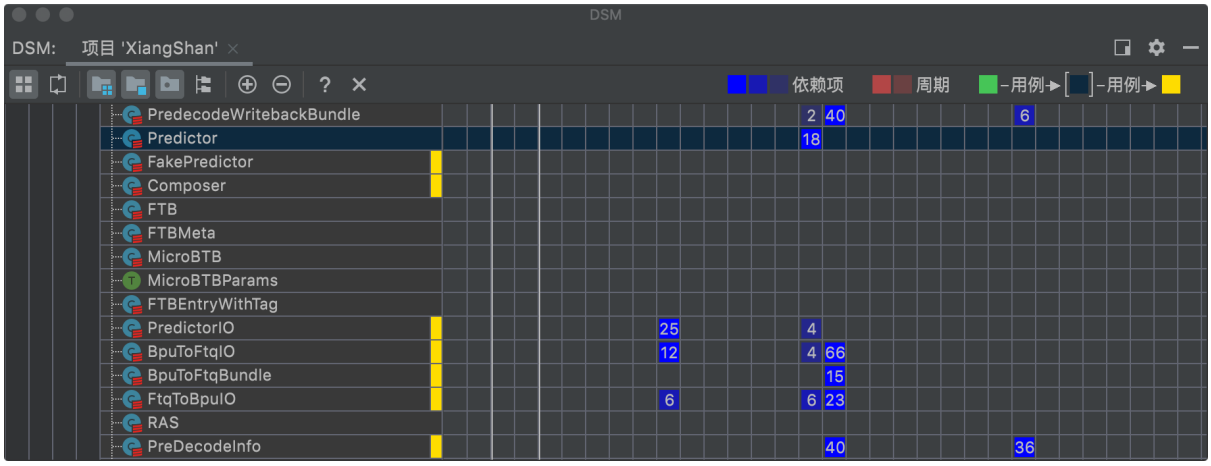
现在，BPU中的`Pridictor`类与其他各个分支预测部件的关系已经清晰不少：Predictor根据继承`HasBPUConst`的信息决定实例化`Composer`类，而`Composer`类继承了`HasXSParameter`，并通过调用其中的`getBPDComponents`方法来进一步获取分支跳转结果，而`getBPDComponents`则调用`branchPredictor`方法，其中例化了各个分支预测部件并通过**覆盖重定向**的方法给出最终的结果。

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**。

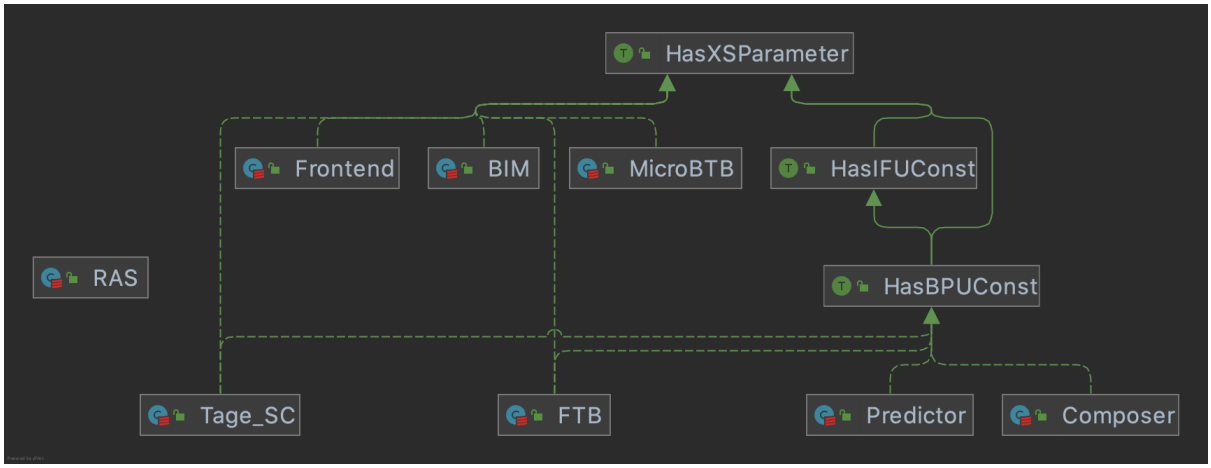
```
//开个玩笑
import chisel3._

Class Open_XiangShan extends OOP 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设计中的类，对象与封装

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。

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。

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

//在第一次报告中并非重点，暂时略过

3.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
      t
  }
```

> 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。