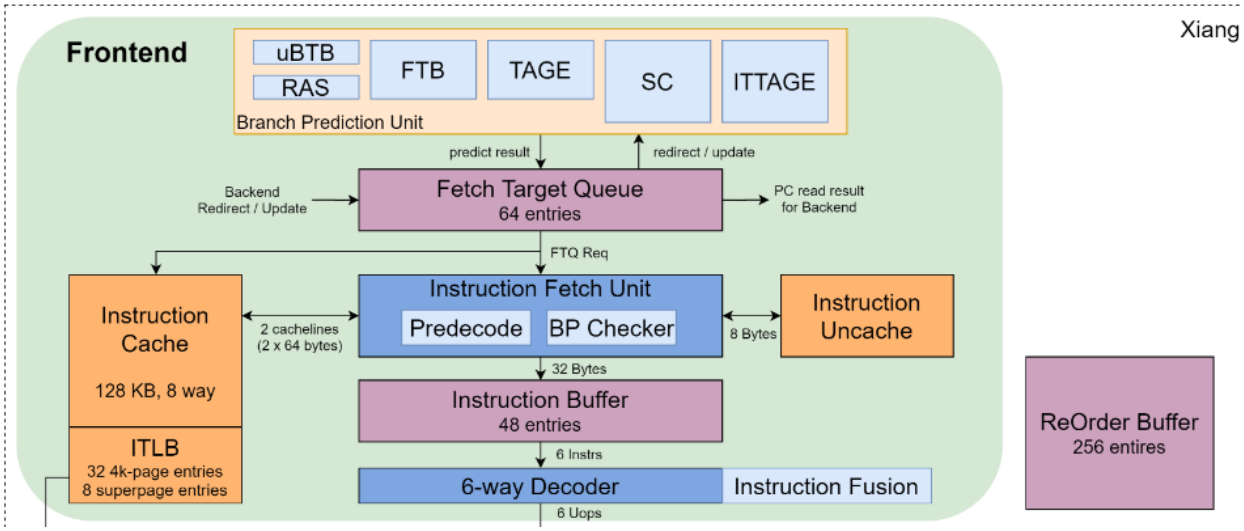
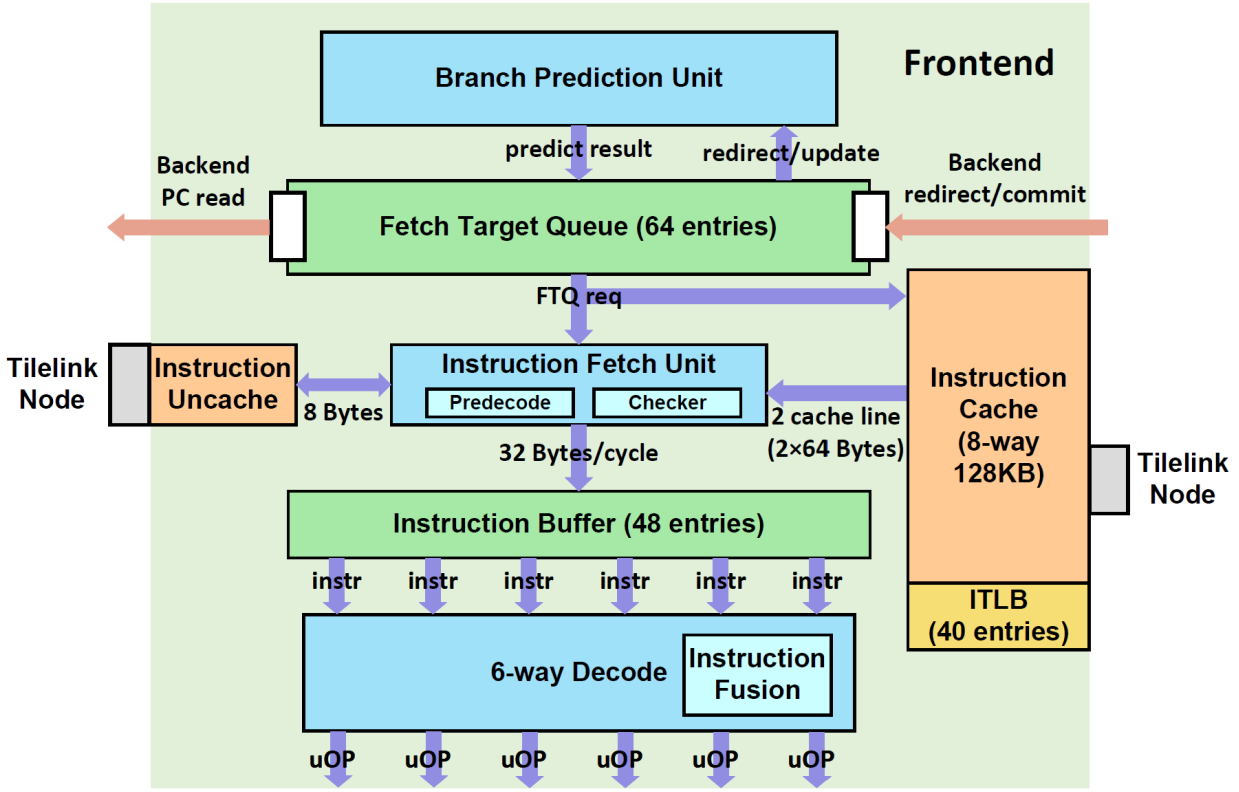
Todo List：

1. TAGE的更新逻辑
2. SC的工作逻辑
3. 细化BPU顶层模块的工作逻辑
4. 添加关于icache的章节
5. 用draoio画图帮助描述各模块的行为
6. 添加模块间互动的时序描述章节
7. 添加代码层次的梳理章节

一.整体概览

根据香山文档的划分，南湖架构的前端部分包含有分支预测单元（Branch Prediction Unit)，取指目标队列（Fetch Target Queue），取指单元（IFU），指令缓存（Instruction Buffer），译码器（Decoder）和ICache，ITLB组成，下图为香山文档中给出的前端部分结构示意图。





香山南湖架构的前端设计总体上参考了《A scalable front-end architecture for fast instruction delivery》这篇论文，将取指令单元和分支预测单元解耦，在中间使用FTQ进行连接，解决了传统设计上分支预测对于流水线性能影响过大的问题。

二、模块分析

（一）.分支预测单元（Branch Prediction Unit）

BPU的具体实现模块为xiangshan/frontend/BPU.scala中的Predictor类，此外还有一个FakePredictor模块，该模块是一个“假”的分支预测单元，当要预测的PC到来时只会空转三个周期，last_stage_meta信号恒为0。

BPU的参数设置在BPU.scala的HasBPUConst中，内容如下：

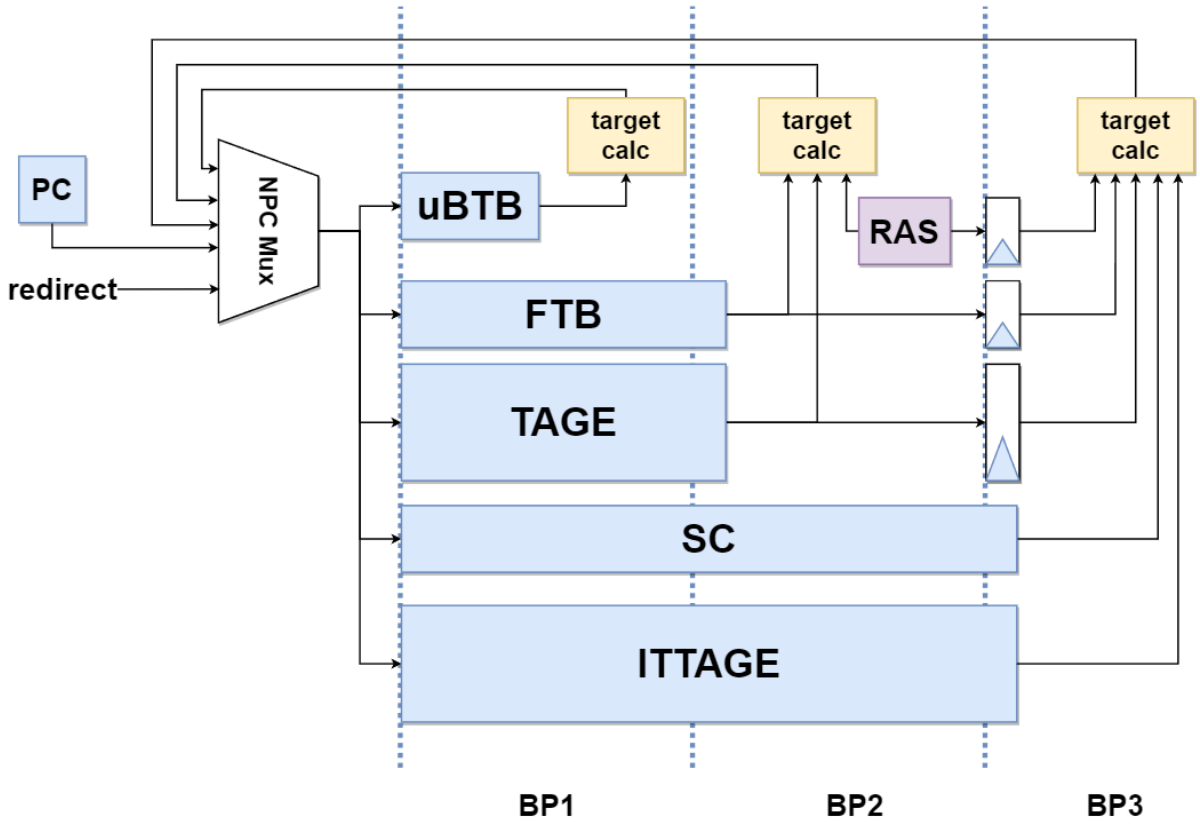
```
trait HasBPUConst extends HasXSParameter {
  val MaxMetaLength = if (!env.FPGAPlatform) 512 else 256 // TODO: Reduce meta length
    //最大历史长度
  val MaxBasicBlockSize = 32            //该参数只在BPU.scala中的BranchPredictionUpdate中被使用，但已被注释。现行香山使用的BranchPredictionUpdate模块位于FrontendBundle.scala中，其中没有使用该参数
  val LHistoryLength = 32               //在代码中未见使用
  // val numBr = 2                      //每次能够预测的分支指令的最大数量。被移至HasXSParameter中
  val useBPD = true                     //是否开启分支预测，为1时实例化Composer，为0时实例化FakePredictor
  val useLHist = true                   //在代码中未见使用
  val numBrSlot = numBr-1               //FTB中
  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            //分支预测单元的内部流水级总数

  val debug = true
  // TODO: Replace log2Up by log2Ceil
}
```

香山的BPU以预测块为单位进行，在默认参数下（FetchWidth=8）预测块的大小为64个字节。BPU由uBTB、FTB、TAGE-SC、RAS、ITTAGE五个模块组成，其中uBTB为只使用PC索引的目标地址的BTB（无tag对比），1周期内完成预测；FTB的结构类似于cache，用PC的高20位作为tag索引，低十二位用于索引对应的bank。FTB作为TAGE预测器中的Base Predictor，与TAGE预测器一样都是2周期完成预测。SC统计矫正器用于校正TAGE的预测结果，在TAGE不够准确时反转TAGE的预测方向。ITTAGE基本上是一个在表项上多记录了预测的跳转地址的TAGE预测器，用于预测间接跳转，和SC一样延迟都为3个周期。以下是香山文档中给出的分支预测单元流水线示意图：



BPU中每个模块都继承自抽象类BasePredictor，统一了各个预测器的接口。

```
class BasePredictorIO (implicit p: Parameters) extends XSBundle with
HasBPUConst {
  val reset_vector = Input(UInt(PAddrBits.W))
  val in  = Flipped(DecoupledIO(new BasePredictorInput)) // TODO: Remove
DecoupledIO
  // val out = DecoupledIO(new BasePredictorOutput)
  val out = Output(new BasePredictorOutput)
```

```
  // val flush_out = Valid(UInt(VAddrBits.W))

  val ctrl = Input(new BPUCtrl)

  val s0_fire = Input(Bool())       //理解为流水线的锁存信号，置为有效时数据将流向下一个流水级
  val s1_fire = Input(Bool())
  val s2_fire = Input(Bool())
  val s3_fire = Input(Bool())
                                    //越靠后完成预测的预测器越精准
  val s2_redirect = Input(Bool())
  val s3_redirect = Input(Bool())

  val s1_ready = Output(Bool())
  val s2_ready = Output(Bool())
  val s3_ready = Output(Bool())

  val update = Flipped(Valid(new BranchPredictionUpdate))       //从FTQ传入的更新信息
  val redirect = Flipped(Valid(new BranchPredictionRedirect))   //重定向信息
}
```

顶层模块BPU主要完成对各个分支预测子模块的实例化，流水线的管理以及全局分支历史的管理。全局分支历史管理通过一个优先多路选择器实现，由于对分支指令结果的预测是从s0流水级开始的，全局历史的选择也在这一流水级内进行。s0_fold_gh的来源按照优先级从低到高排序结果如下：

选择信号	来源	优先级	备注
true	s0_folded_gh_reg	0	流水线阻塞
do_redirect.valid	update_ptr	2	发生新的重定向请求
s3_redirect	s3_predict_fh	3	s3级的重定向请求
s1_valid	s1_predicted_fh	4	s1_valid在代码中被初始化为0，未见其它赋值
s2_redirect	s2_predicted_fh	5	s2级的重定向请求

1.微目标地址缓存（Micro Branch Target Buffer）

香山的官方文档中提到在uBTB中摒弃了tag对比的做法，直接使用分支历史和PC的低位异或得到的结果来寻址存储表，但我在FauFTB.scala中看到这个模块仍然有tag对比的行为，本部分准确度存疑。Fa前缀可能是fake的意思。

uBTB以一整个预测块为单位，提供无气泡的简单预测，其实现位于xiangshan/frontend/FauFTB.scala中。uBTB的组成类似于cache，其参数位于一个trait FauFTBParams中：

```
trait FauFTBParams extends HasXSParameter with HasBPUConst {
  val numWays = 32
  val tagSize = 16
```

```
  //在uBTB中未使用
  val TAR_STAT_SZ = 2
  def TAR_FIT = 0.U(TAR_STAT_SZ.W)
  def TAR_OVF = 1.U(TAR_STAT_SZ.W)
  def TAR_UDF = 2.U(TAR_STAT_SZ.W)

  //两种分支指令的偏移量长度。BTB中不直接记录跳转地址，而是记录偏移量并将其与当前预测的PC值的高位拼接形成目标地址。
  def BR_OFFSET_LEN = 12
  def JMP_OFFSET_LEN = 20

  def getTag(pc: UInt) = pc(tagSize+instOffsetBits-1, instOffsetBits)
}
```

uBTB的行为如下：

s0流水级：

- 若u.valid有效，则将从外部传入的BTB更新信息的tag与表项中的tag对比
- 否则，将当前预测的PC值的tag部分与表项中的tag对比

s1流水级：

- 若u_s1_valid有效（延迟一拍的u.valid）则：
  - 若BTB命中，则将更新信息写入对应的BTB表项，并更新2-bit饱和计数器的值
  - 否则将更新信息写入PseudoLRU替换器指示的表项中
- 否则，根据2-bit饱和计数器的值预测分支方向

2.取指目标缓存（Fetch Target Buffer）

FTB是香山里分支预测单元中精确预测器的核心部分，文档中提到FTB除了提供预测块内分支指令的信息外，还提供预测块的结束地址。其实现位于xiangshan/frontend/FTB.scala中，参数由trait FTBParams提供：

```
trait FTBParams extends HasXSParameter with HasBPUConst {
  val numEntries = FtbSize
  val numWays    = FtbWays
  val numSets    = numEntries/numWays // 512
  val tagSize    = 20

  //三个用于计算跳转地址的状态位
  val TAR_STAT_SZ = 2
  def TAR_FIT = 0.U(TAR_STAT_SZ.W)      //TARGET_FITTING
  def TAR_OVF = 1.U(TAR_STAT_SZ.W)      //TARGET_OVERFLOW
  def TAR_UDF = 2.U(TAR_STAT_SZ.W)      //TARGET_UNDERFLOW

  //不同分支指令的偏移量位宽
  def BR_OFFSET_LEN = 12
  def JMP_OFFSET_LEN = 20
}
```

根据香山的官方文档，FTB的表项满足以下特点：

- FTB项由预测块的起始地址start索引，start通常为上一个预测块的end或来自BPU外部的重定向的目标地址（如异常返回）
- FTB项内最多记录两条分支指令，其中第一条一定是条件分支指令
- end一定满足以下条件之一：
  - end - start = 预测宽度
  - end为从start开始的预测宽度范围内第三条分支指令的PC
  - end是一条无条件跳转分支指令的吓一跳指令的PC，同时它在从start开始的预测宽度范围内

FTB所使用的存储表定义在class FTB内定义并实例化，名称为FTBBank。表中存放的数据结构为FTBEntryWithTAG，这个类只是在另一个名为FTBEntry的类的基础上添加了tag段。FTBEntry的结构定义位于FrontendBundle.scala中，内容如下：

```
class FTBEntry(implicit p: Parameters) extends XSBundle with FTBParams with BPUUtils {
  val valid       = Bool()      //valid：表项有效位
  val brSlots = Vec(numBrSlot, new FtbSlot(BR_OFFSET_LEN))  //brSlots：分支指令的预测跳转地址
  val tailSlot = new FtbSlot(JMP_OFFSET_LEN, Some(BR_OFFSET_LEN))   //tailSlot:预测块的结束地址

  // Partial Fall-Through Address
  val pftAddr     = UInt(log2Up(PredictWidth).W)    //下一个预测块的低位地址，当预测结果为not taken时，下一个预测块的起始地址为Cat(高位PC, pftAddr, 0.U(log2Ceil(指令占用字节数)).W)。若carry为1.B，则高位PC需要进位（即起始地址为Cat(高位PC+1, pftAddr, ...)）
  val carry       = Bool()          //用于计算下一个预测块
  val isCall      = Bool()          //跳转指令的类型
  val isRet       = Bool()
  val isJalr      = Bool()
  val last_may_be_rvi_call = Bool() //疑似是判断是否为压缩后的call指令

  val always_taken = Vec(numBr, Bool()) //预测跳转总是发生的计数器，当该位为1时，预测对应的跳转指令结果为总是跳转，此时不以其结果训练预测器；
    //当遇到不跳转情况时，将该位置0

     //这里省略了类中定义的其它函数
}
```

与一般BTB中直接存储跳转的目的地址不同，FTBEntry只存储地址的低位（pftAddr），并将其与高位拼接来得到下一个预测块的起始地址。当预测结果为not taken时，下一个取指块的地址就由ptfAddr和当前预测块的PC高位拼接得到。这段设计来自于论文《A scalable front-end architecture for fast instruction delivery》。

跳转地址的存储使用了另一个数据结构FtbSlot：

```
//offsetLen: 不同分支指令的跳转偏移长度，有JMP_OFFSET_LEN（20位）和BR_OFFSET_LEN（12位）两种值
```

```
//subOffsetLen: 目前只见于FTBEntry中的tailSlot里使用
class FtbSlot(val offsetLen: Int, val subOffsetLen: Option[Int] = None)
(implicit p: Parameters) extends XSBundle with FTBParams {
  if (subOffsetLen.isDefined) {
    require(subOffsetLen.get <= offsetLen)
  }
  val offset  = UInt(log2Ceil(PredictWidth).W)  //
  val lower   = UInt(offsetLen.W)        //跳转的偏移量
  val tarStat = UInt(TAR_STAT_SZW)       //跳转目标的类型
  val sharing = Bool()                   //若slot中存储的是预测块内能够预测的最后一条条
件分支指令（第numBr条分支指令），该位置为1
  val valid   = Bool()

  val sc      = Bool() // 用于预取
    //这里省略了类中定义的其它函数
}
```

跳转的目标地址依旧是由PC的高位和偏移量拼接完成，不过与fall-through addr不同的是跳转的方向可以是向上也可以是向下的，因此需要一个tarStat状态位指示PC高位是否需要+1或-1。如一个jal指令的预测跳转地址计算方式应该是**Cat(** PC高位, lower(offLen-1, 0), 0.U(1.W) **)**。

其他诸如查询和更新的行为就与FauBTB比较类似了，FTB的行为如下：

当update_valid为0时：

s0流水级：

- 将s0_pc（最新的预测块的起始地址）的tag段和Index段分别锁存至req_tag和req_idx内
- 以s0_pc的Index段向FTB使用的SRAM发送读请求。

s1流水级：

- 将req_tag内的值与表项中的tag对比，记录命中情况s1_hit
- 若s1_fire有效，锁存住s1_hit的值到s2_hit中
- 若s1_fire有效，将从FTB读出的信息锁存至ftb_entry中

s2流水级：

- 当s2_hit有效且ftb_entry中的always_taken域有效，或者从TAGE预测器传入的预测信息中s2的预测信息表示预测发生跳转时，设置s2的br_taken_mask为1，表示预测发生跳转。
- 若s2_fire有效，锁存s2_hit的值到s3_hit中
- 若s2_fire有效，锁存ftb_entry的值到s3_ftb_entry中

s3流水级：

- 当s3_hit有效且s3_ftb_entry中的always_taken域有效，或从TAGE预测器传入的预测信息中s3的预测信息表示预测发生跳转时，设置s3的br_taken_mask为1，表示预测发生跳转。

当update_valid为1时：

- u_meta.hit为1时，update_now为1，此时writeWay由u_meta.writeWay给出，将数据直接写入FTB中：
- u_meta.hit为0时，update_now为0，此时writeWay依赖于tag对比。注意由于FTB使用的是单端口SRAM，故在此需要通过将s1_ready置0以阻塞流水线来实现u_req_tag和表项中tag的比较。SRAM的写行为需要在tag对比完成之后，因此update.pc，update.entry和SRAM的写有效信号均需要延迟两周期（在代码中为delay2_pc，delay2_entry）。
    - s0流水级：
        - 将update_pc的tag段锁存至u_req_tag内。
        - 以uodate_pc的index段为地址向SRAM发送读请求。
        - 由于流水线阻塞，req_tag和req_idx保留原来的值。
    - s1流水级：
        - 将u_req_tag与表项中的tag段对比，记命中情况为u_hit。
        - 由于流水线阻塞，req_tag和req_idx保留原来的值。
    - s2流水级：
        - 若未命中，则分配一个新的way写入更新信息。
        - 若命中，则将更新信息写入命中的way中。
        - 流水线阻塞解除，req_tag和reg_idx按照s0_pc更新。

FTB使用的SRAM从接受读请求到返回数据需要两个周期，因此当update_valid刚置为1时，s0级已发送的请求在s1所返回的预测数据就会流水线阻塞的原因而丢失。为了避免这种情况发生，香山会在发生update的时候将读出来的预测信息暂时锁存住，直到流水线阻塞解除。这在代码层面上通过object HoldUnless完成：

```
val pred_rdata   = HoldUnless(ftb.io.r.resp.data, RegNext(io.req_pc.valid && !io.update_access))
//当req_pc.valid && !update_access为0时，保存原值；为1时pred_rdata = ftb.io.r.resp.data

//原型
object HoldUnless {
  def apply[T <: Data](x: T, en: Bool): T = Mux(en, x, RegEnable(x, 0.U.asTypeOf(x), en))
}
```

3.TAGE预测器

TAGE（Tagged GEometrical Length Predictor）分支预测是综合O-GEHL分支预测和PPM-like分支预测所设计的分支预测算法。由base predictor T0，和一序列（partially）tagged predictor components Ti组成，tagged predictor components Ti所采用索引历史长度是不同的，成几何长度关系。从它的名称上就可以看出来设计上的特点：

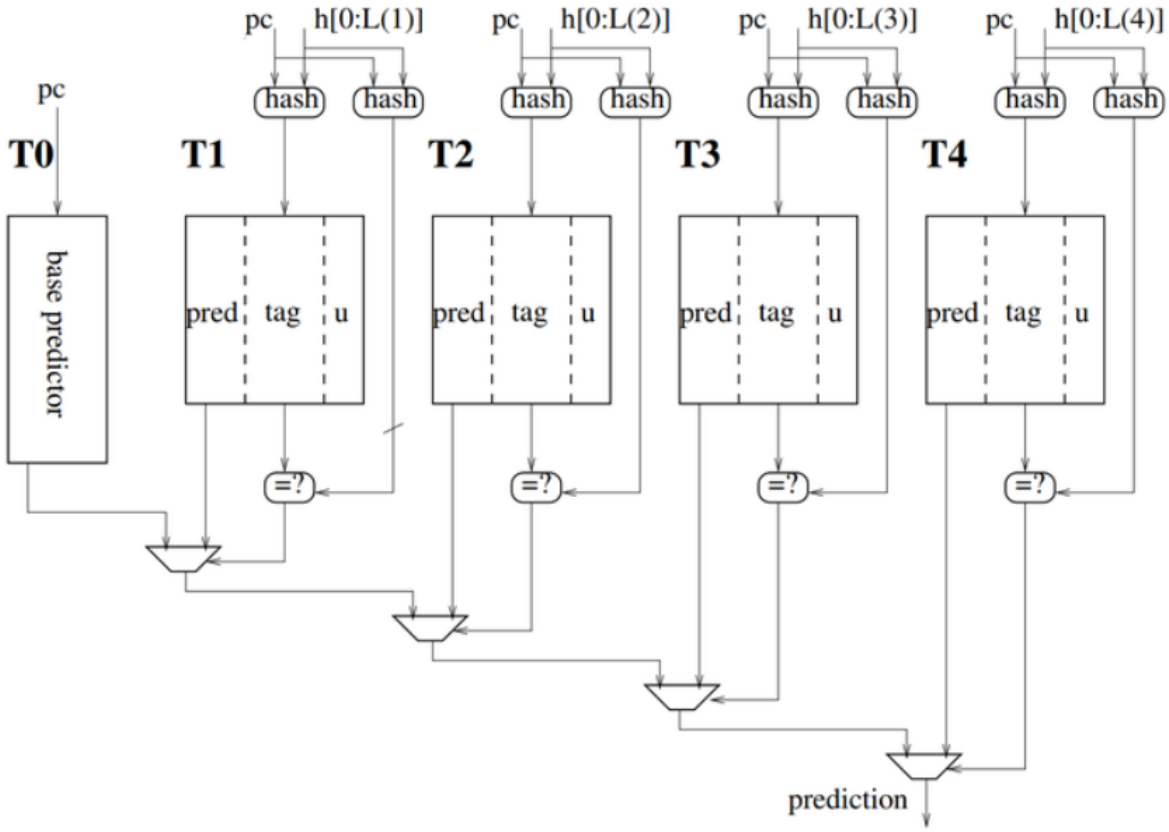
1.TA（Taged）：

由于分支预测的索引通常为PC或者PC和全局历史的hash值，一般为了提高存储效率PC只有部分段用于索引，因此会有造成别名（aliasing），TAGE中带有tag的设计能够更好地处理这一问题。

2.GE（GEoimetrical）

主流观点认为一般global history match的长度越大，预测就越准确。所以TAGE选取了几何增长的history长度，如32，64，128，256。原则上，每次优先选取最长的match做预测。当然也有一些简单地branch可能不需要那么长，短的就好。TAGE的userful counter 就发挥了作用。如果短的history predict的效果就很好，就会通过useful counter 反映出来。

更详细的原理介绍参见这篇专栏或这篇论文。



TAGE预测器

图中的base predictor就是一个简单的由饱和计数器组成的预测器，当所有bank都没有命中时就会采用base predictor的预测结果。香山中的base predictor实现位于xiangshan/frontend/Tage.scala中的TageBTable类。其他的Tagged Predictor使用的是TageTable类，被称为Provider。

在没有发生update的情况下，TAGE的时序逻辑比较简单，只需要以s0_pc的idx段和tag段索引SRAM，在s1读出预测数据并选择历史长度最长的预测器的预测结果，该预测器被称为provider。如果没有命中任何预测器，则会使用base predictor的预测结果。此外，香山还参考了这篇论文实现了一个**备选预测(Alternative Provider)**的逻辑。备选预测（altpred）是命中的预测器中具有第二长度的预测器（若没有命中则为base predictor）。论文中提到在一些情况下（比如最长历史项是刚刚分配出来的新项）altpred的结果比provider要更准确，因此还设置了一个USE_ALT_ON_NA寄存器，用以指示是否使用备选预测器的预测结果。香山中处于时序的考虑，始终使用base predictor作为备选预测的结果。

注意香山中的TAGE预测器与FTB一样，能够同时预测numBr条（目前numBr = 2）分支指令的结果，因此TAGE的预测表中一个表项也最多能够存储两条分支指令的结果。同时，香山的TAGE内没有存储分支指令的跳转地址（这个功能已经由FTB完成了），TAGE只是简单地预测分支是否发生（taken or not）。

s0流水级：

- Base Predictor：

    - 直接以s0_pc的idx段（s0_idx）为地址访问base predictor使用的预测表。
    - 当s0_fire有效时，锁存s0_idx到s1_idx中。
- Tagged Predictor：
    - 将s0_pc与分支表的历史进行异或hash得到s0_idx，用于索引预测表。
    - 将s0_pc与分支表的历史进行异或hash得到s0_tag。
    - 当s0_fire有效时，分别锁存s0_idx和s0_tag到s1_idx和s1_tag中。
- 在Tage的顶层模块中还涉及到如下信号：
    - provided：是否命中至少一个Tagged Predictor。
    - providerInfo: 命中的Tagged Predictor的index以及预测数据，若无命中

s1流水级：

- Base Predictor：
    - 直接输出预测表中的预测数据。
- Tagged Predictor:
    - 将s1_tag与预测表中的tag对比，记录每条分支指令（至多numBr条）的命中情况为per_br_hit。
    - 根据per_br_hit和预测表中的值输出预测数据

```
for (i <- 0 until numBr) {
  io.resps(i).valid := per_br_hit(i)
  io.resps(i).bits.ctr := per_br_resp(i).ctr  //饱和计数器
  io.resps(i).bits.u := per_br_u(i)   //useful域
  io.resps(i).bits.unconf := per_br_unconf(i)
}
```

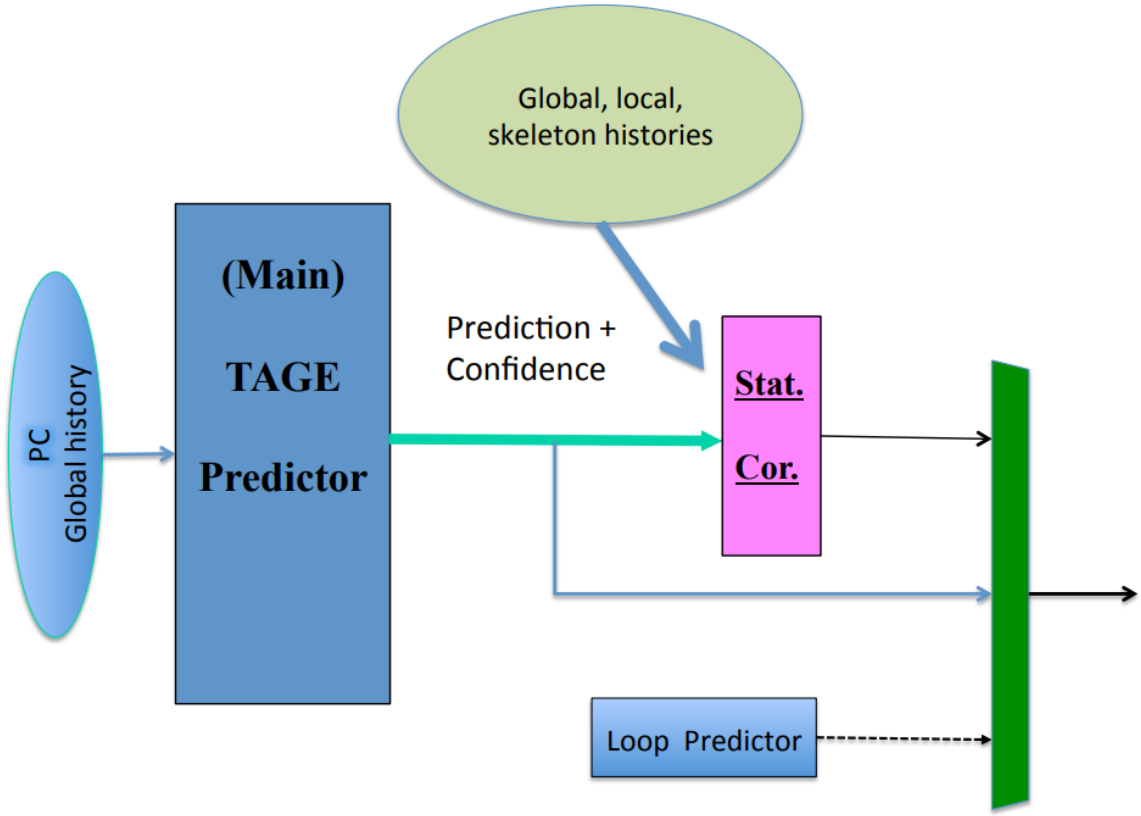
- 在Tage的顶层模块中还涉及到如下信号：
    - s1_provideds：锁存自provided
    - s1_providers：命中的历史最长的预测器的编号（index），锁存自providerInfo.tableIndex。
    - s1_altUsed: 是否使用备选逻辑。当provided为0或USE_ALT_ON_NA指示使用备选预测时，在下一周期被置为1。
    - s1_baseCnt: 锁存自从base predictor中读出的饱和计数器的值。
    - s1_finalAltPreds: 备选逻辑的预测结果。锁存自从base predictor中读出的饱和计数器的最高位。
    - s1_tageTakens: 预测结果，为1表示taken。当s1_altUsed为1时，取base predictor的预测结果。否则取providerInfo中的预测结果。

s2流水级：

- 将s1级的各信号锁存一拍，对外输出预测结果

TAGE的更新：待补充

实际上TAGE预测器在应用上除了与SC（statistical corrector，统计校正器）搭配以外还经常与循环预测器搭配使用，组成TAGE-SC-L预测器，如下图所示。香山在南湖架构中舍弃了循环预测器，因为南湖架构中的分支预测单元是以预测块为单位进行预测，一条分支指令可能会存在于多个FTB项内，统计一条分支指令的循环次数比较困难。而在雁栖湖架构中，对每一条指令都会做出预测，一条分支指令在 BTB 中只会出现一次，故没有上述的问题。



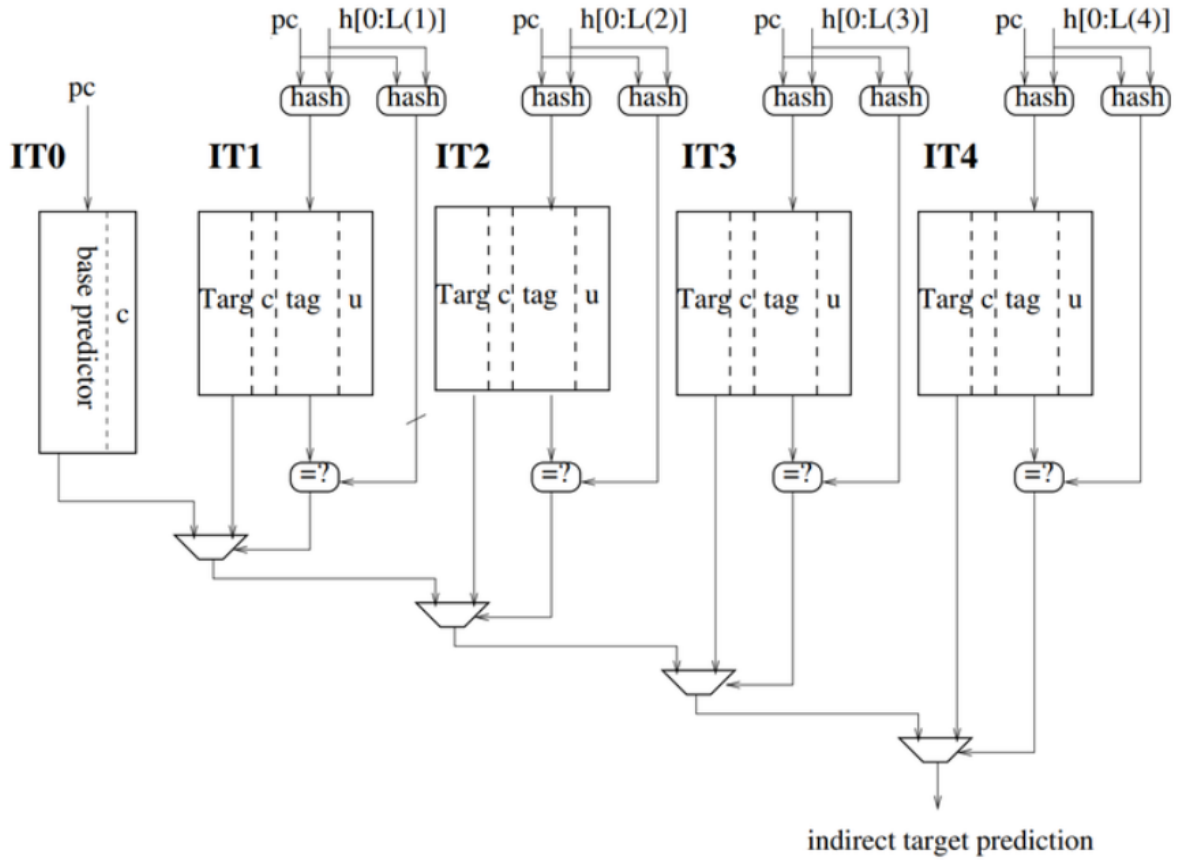
3.统计校正器（Statistical Corrector）

《待完成》

TAGE在预测与历史较相关的分支时非常有效，但对一些有统计偏向的分支效果不佳，如一条只对一个方向有微小的偏差，但与历史路径没有强相关性的分支。统计校正器SC负责在预测这种具有统计偏向的条件分支指令并在这种情况下反转TAGE预测器的结果。

4.ITTAGE预测器

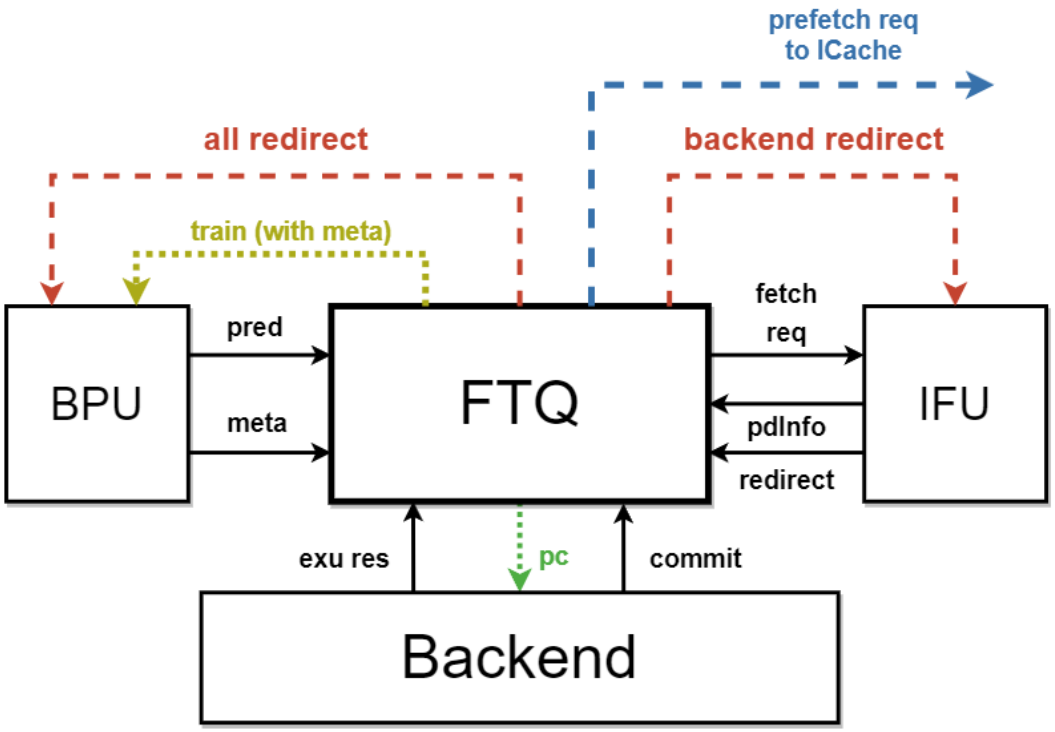
ITTAGE预测器是专门用于预测间接跳转指令的预测器，其预测算法与TAGE一致，只是在预测表中加上了跳转到的目标地址，并多延迟一拍以暂存读取出来的目标地址Target。原理在此不作重复。



ITTAGE预测器

（二）取指目标队列（Fetch Target Queue）

由于BPU在设计上与指令缓存进行了解耦，IFU和BPU所使用的PC被独立开来。而BPU中很少出现阻塞的情况，流水线的效率要大于需要访问Cache的IFU，为了最大化流水线的效率，香山设计了一个取指目标队列（FTQ）用于暂存BPU中已经完成预测的取指块，并向IFU发送取指令请求。除此之外，FTQ中还存储了BPU中各个预测器的预测信息（meta），在指令确认退休后，FTQ会将这些信息送回BPU用作预测器的训练，故它需要维护指令从预测到提交的完整的生命周期。除此之外，当后端需要指令PC时也会到FTQ中获得。



FTQ 与其它模块的交互

FTQ是一个循环队列结构，但队列中的内容是根据其自身特点存储在不同的结构中的，队列的指针并不与某个特定的存储结构绑定。FTQ主要使用的存储结构有ftq_pc_mem，ftq_pd_mem，ftq_redirect_sram，ftq_meta_1r_sram和ftb_entry_mem，其中除ftq_redirect_sram和ftq_meta_1r_sram用sram实现外都由寄存器堆实现。

FTQ维护了bpuPtr、ifuPtr、ifuWbPtr和commPtr四个队列指针：

- **bpuPtr**：当BPU发出新的预测块到FTQ时指针加一，指向最新进入FTQ的项。此时需要初始化FTQ各项的状态，并将预测信息写入存放预测信息的存储结构。若出现BPU的s2或s3流水级的预测结果覆盖当前预测结果时，需要恢复bpuPtr和ifuPtr并根据IFU中正在处理的预测块的索引号项决定是否清空IFU的各流水级。
- **ifuPtr**：当FTQ向IFU发出取指请求时指针加一，指向最新发送请求给IFU的FTQ项。
- **ifuWbPtr**：当IFU写回有效的预译码信息时指针加一，指向已在IFU中完成预译码的FTQ项。FTQ根据IFU写回的预译码信息有无预测错误（预测不跳转的jal、ret等）决定是否向BPU发送重定向请求并恢复bpuPtr和ifuPtr。
- **commPtr**：每条指令于后端中提交时都会通知FTQ，当一个FTQ项内的全部指令都已顺利退休时指针加一，指向最新的已确认指令执行结果的FTQ项。BPU可以利用这些已提交的指令的执行历史来训练预测器。bpuPtr和commPtr中间的即为生命周期仍未结束的预测块。

1. ftq_pc_mem

存储指令地址相关的信息，其基本构成单元为Ftq_RF_Components：

```
class Ftq_RF_Components(implicit p: Parameters) extends XSBundle with BPUUtils
with HasBPUConst {
  val startAddr = UInt(VAddrBits.W)              //预测块的起始地址
  val nextLineAddr = UInt(VAddrBits.W)           //下一行地址
  val isNextMask = Vec(PredictWidth, Bool())     //预测块
  val fallThruError = Bool()             //推测的下一行地址是否存在错误
  // val carry = Bool()

  //略去类内部定义的函数
}
```

ftq_pc_mem每个周期都会存储从bpu中发送来的预测块信息。每周期默认存储uBTB（一拍出结果）的预测结果，当其他预测器的预测结果与uBTB不一致时则需要矫正。

2. ftq_pd_mem

存储从IFU返回的预译码信息，其基本构成单元为Ftq_pd_Entry

```
class Ftq_pd_Entry(implicit p: Parameters) extends XSBundle {
  val brMask = Vec(PredictWidth, Bool())         //每条指令是否为分支指令
  val jmpInfo = ValidUndirectioned(Vec(3, Bool()))  //位于预测块末尾的无条件跳转指令的信息，包括是否存在，类型（jal or jalr，call or ret）等
  val jmpOffset = UInt(log2Ceil(PredictWidth).W)     //预测块末尾无条件跳转指令的位置
  val jalTarget = UInt(VAddrBits.W)                  //预测块末尾无条件跳转指令的目标地址
  val rvcMask = Vec(PredictWidth, Bool())            //是否为压缩指令的掩码
  def hasJal  = jmpInfo.valid && !jmpInfo.bits(0)
  def hasJalr = jmpInfo.valid && jmpInfo.bits(0)
  def hasCall = jmpInfo.valid && jmpInfo.bits(1)
  def hasRet  = jmpInfo.valid && jmpInfo.bits(2)

   //略去类中定义的其他函数
}
```

写入ftq_pd_mem的内容来自于IFU的第4个流水级，存储经过预译码的指令信息。除此之外，还会以commPtr的值为地址输出预测信息，用于训练BPU中的预测器。

3. ftq_redirect_sram

存储重定向时需要恢复的预测信息，如RAS和分支历史等。其基本单元构成如下：

```
class SpeculativeInfo(implicit p: Parameters) extends XSBundle
  with HasBPUConst with BPUUtils {
  val folded_hist = new AllFoldedHistories(foldedGHistInfos)              //折叠后的全局历史
  val afhob = new AllAheadFoldedHistoryOldestBits(foldedGHistInfos)
  val lastBrNumOH = UInt((numBr+1).W)
  val histPtr = new CGHPtr                                                //全局历史指针（CGH，Circular Globlal History）
  val rasSp = UInt(log2Ceil(RasSize).W)                                   //RAS指针
  val rasTop = new RASEntry                                               //要恢复的RAS栈顶的项
}
```

由于在设计上RAS要在BPU的s3流水级才给出预测结果，ftq_redirect_sram存储的是BPU的最后一个流水级内的信息。同样，ftq_redirect_sram也会以commPtr的值为地址输出全局历史信息用于训练BPU中的预测期。

4. ftq_meta_1r_sram

存储其余的预测信息，即各个预测器的meta。

```
class Ftq_1R_SRAMEntry(implicit p: Parameters) extends XSBundle with HasBPUConst {
  val meta = UInt(MaxMetaLength.W)
}
```

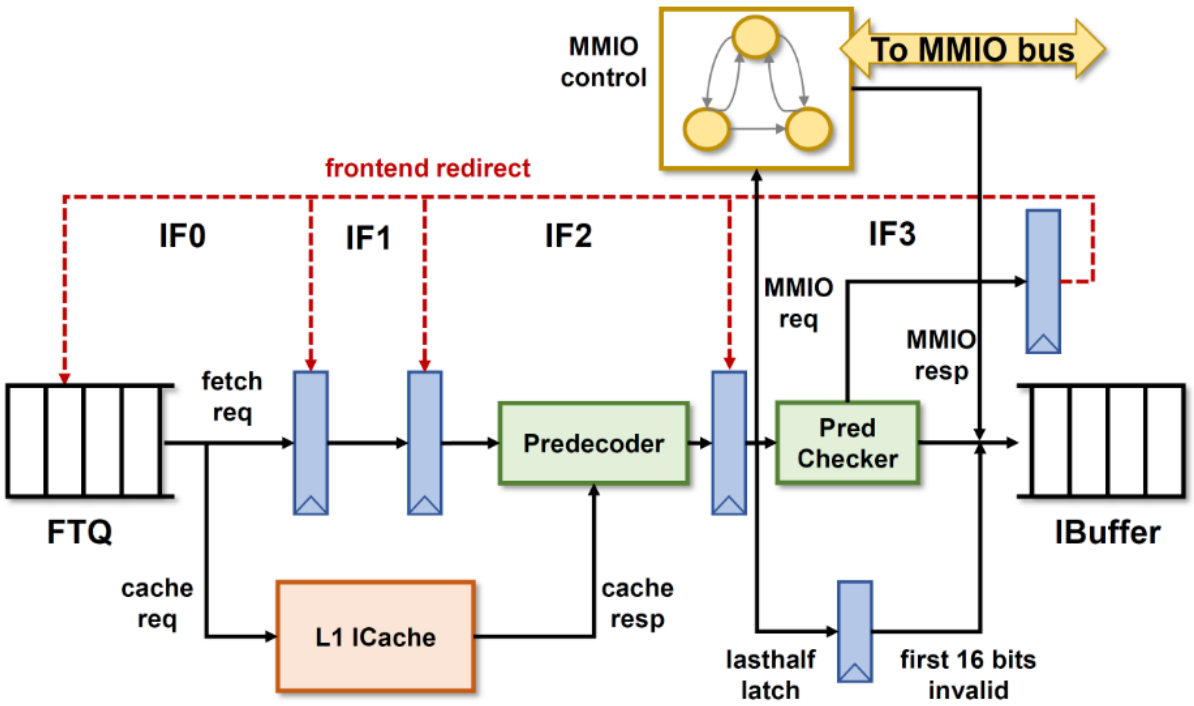
5. ftb_entry_mem

存储预测时FTB所必须的信息，在指令于IFU提交后用于训练信的FTB项。基本构成单元为前文介绍过的FTBEntry。

（三）取指令单元（Instuction Fetch Unit）

待完成：压缩指令的处理过程

取指令单元的实现位于xiangshan/frontend/IFU.scala中。南湖架构的IFU采用了四级流水线的结构，由于分支预测已经与取指部分解耦，取指单元的逻辑也大幅简化，只需要根据FTQ发来的请求从ICache中读取指令，再对指令进行预译码和分支预测正确性检查、MMIO检查即可。代码上该模块相对容易理解，各流水级的信号都以f[<num>]_<xxx>的形式命名，其中num为流水级的编号。详细行为如下：



从模块外输入的信息：

- fromFtq，从FTQ传入的信息，主要是取指令请求
- icacheInter，ICache的回应
- icachePerfInfo，ICache相关的性能计数器
- frontendTrigger，与CSR控制相关
- rob_commits，ROB中将要退休的指令信息

模块输出的信息：

- toFtq，传递给FTQ的信息，主要是分支预测的检查结果
- icacheStop，ICache的阻塞信号（MMIO访问）
- toIbuffer，传递给指令缓存（IBuffer）的信息，主要是完成预译码的指令/扩展后的压缩指令

IF0级：

- 若fromFtq.valid有效，则根据预测块的起始地址和有效范围向icache发送读请求。
- 根据取指的范围判断是否跨cache行使能f0_doubleLine信号。

IF1级：

- 计算指令块中每条指令对应的PC值

IF2级：

- 接收ICache返回的数据
- 生成例外（page fault、access fault、mmio等）
- 将每条指令的PC通过异或折叠得到fold_pc，用于译码阶段
- 将去除的指令送进预译码器进行译码，同时识别16bit的压缩指令并将其扩展为32bits的指令。

IF3级：

- 将指令的预译码信息送入分支预测检查器。在这一阶段能够检查出来的分支预测错误有：
  - jump类型指令不跳转：如预测块的有效范围内有jal、ret这类直接跳转指令，且分支预测给出的结果为not taken，则需要标记为预测错误。
  - 非跳转指令预测跳转：不含跳转指令或是跳转指令不在有效范围内时分支预测给出的结果为taken
  - 目标地址错误：部分跳转指令可从指令码出得出跳转地址（如jal），此时需要检测分支预测给出的跳转地址是否正确。
- 若分支预测检查器检查到错误，则需要将最先发生错误的分支指令的错误信息返回给FTQ，同时清空IFU所有流水级，等待FTQ发送新的取指请求。
- 若根据预译码结果需要从MMIO空间中取指令，则需要向MMIO模块发起取指令请求，并在MMIO模块返回指令前阻塞流水级。

三、模块间的交互

（待补充）