香山Note

一、总体架构

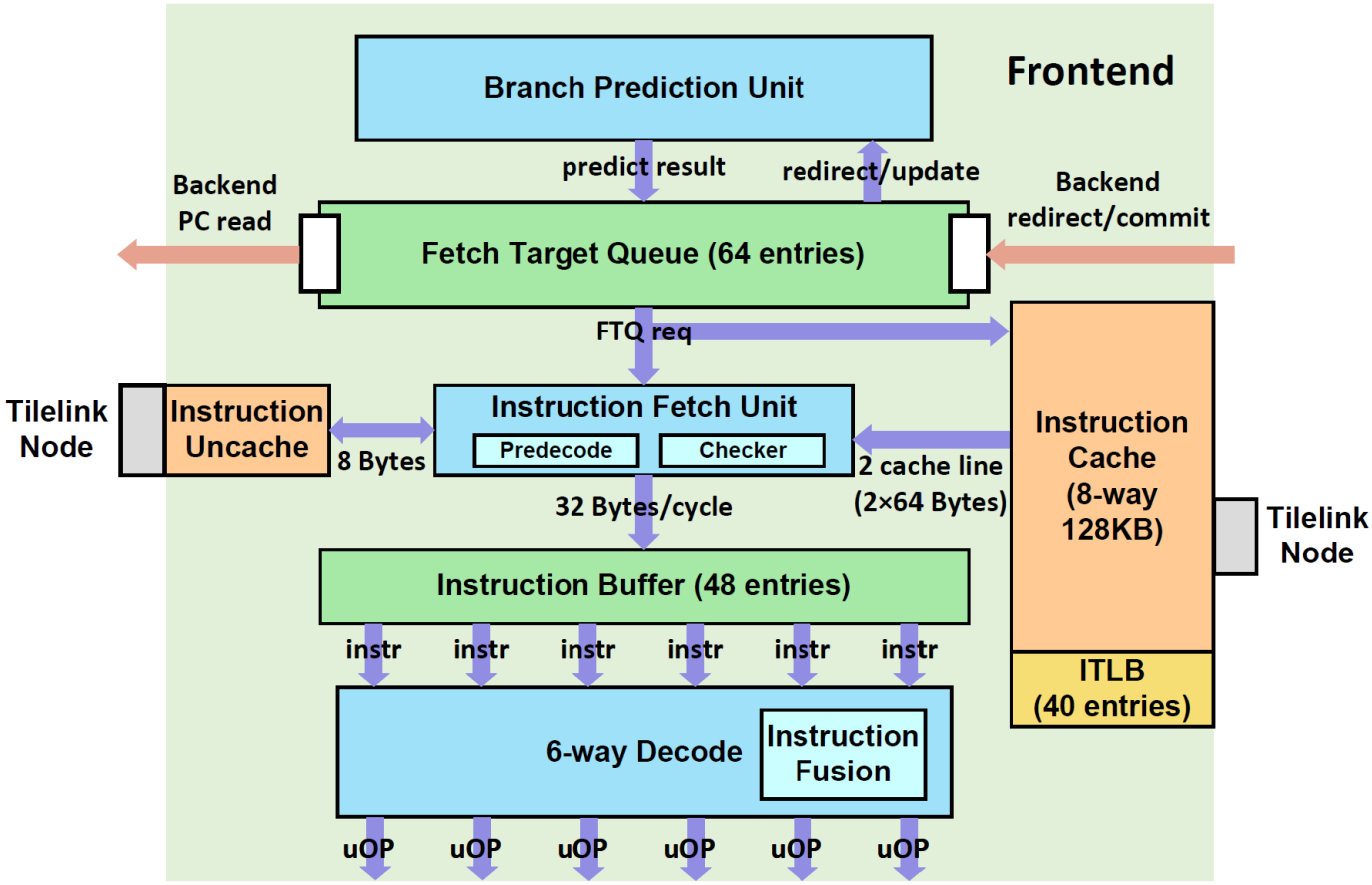
乱序六发射架构设计。

- 前端：分支预测单元、取指单元、指令缓冲单元
- 后端：译码、重命名、重定序缓冲、保留站、整型/浮点寄存器堆、整型/浮点运算单元
- 访存系统：Load&Store流水线

分支预测：三个流水级

访存架构：两条3级load流水线，两条2+2级store流水线

二、前端架构



分支预测单元提供取指请求，写入一个队列，该队列将其发往取指单元，送入指令缓存。 取出的指令码通过预译码初步检查分支预测的错误并及时冲刷预测流水线，检查后的指令送入指令缓冲并传给译码模块，最终形成后端的指令供给。（图片皆搬运自香山处理器官方文档）

1.分支预测（BPU）

为什么需要分支预测？流水线译码到当前指令时，识别到当前指令时跳转指令。但由于跳转与否尚不明晰，故而需要等待跳转结果的产生（在组成原理课程中，我们使用了**转发**来加快分支指令结果的返回，应该是没有做有关分支预测的工作）。停机等待分支结果是浪费性能的，故而人们尝试研究分支预测。

高性能CPU中，据我了解，每次取指应当不是仅取一条指令，而是从I-Cache中取一组指令。在这一组指令中，可能会包括各种call、ret等指令。根据学者的统计，分支指令在代码中的“含量”达到 20%且其在流水线中是否能够被处理得当对处理器的性能有着至关重要的影响。

在遇到分支指令的时候，不进行任何预测的做法是愚蠢的，这凭空增加了无谓的等待时间。即便是默认遇到分支指令还继续执行顺序指令的做法也能节省很多性能。下面介绍分支预测的指导方法。

静态预测：采取不变即静态的策略进行预测即静态预测。比如，我们可以选择遇到分支指令就跳转的策略，也可以选择遇到for结构的分支指令就跳转的策略，这是一种简单且易于实现的预测方法。

动态预测：这部分包含了很多种不同的动态预测策略，且具备记忆能力是它们发挥作用的关键。当然为了实现静态的预测策略，我们可能需要更多的硬件来辅助实现。

动态预测的方法有许多，不过在了解预测方法之前，我们需要明确我们应该如何找出分支指令（因为取指阶段往往是在译码阶段之前的）。看到这里你也就应该可以理解香山处理器为什么要预译码，提前译码是一个比较容易想到的解决方案。我们从I-Cache中取出指令块（指令缓存，SDRAM的读取速度较慢，需要设置一个Instruction Cache来加速缓存），然后让其经过快速译码器来进行预译码的工作（I-Cache的读取也可能需要多个周期，这将会比较耗时）；我们也可以尝试在指令从L2 Cache写入I-Cache的时候就“夹带私货”，进行预译码，但这样也比较耗时。

我们再考察一个问题：对于一条指令来说，它的物理地址可能会发生变化，但是它的虚拟地址应当是稳定的；即便是发生了进程切换，我们也可以通过冲刷分支预测器或是结合AISD进行预测。这也就是说，依靠PC判断指令是分支指令与否是可行的。至此，我们拥有了一些分支指令的识别方法。

Base on 两位饱和计数器的分支预测方法：我们之前提到了动态预测，是“有记忆”的预测策略。在众多的“有记忆”的策略中，人们发现基于二位饱和计数器的预测方法最为有效。

什么是**二位饱和计数器**？我们可以使用一个拥有四个状态的有限状态机描述这个硬件：

状态码	状态	饱和	跳转
00	Strongly taken	是	跳转
01	Weakly taken	否	跳转
10	Weakly not taken	否	不跳转
11	Strongly not taken	是	不跳转

每一个二位饱和计数都对应了位于某一个PC的一条指令。我这么说的时候，心理也曾经出现了如下的对话：

Q：你这样子做很浪费空间啊！一台32位的机器就要使用几百MB的存储器来模拟二位饱和计数器，这显然是不合理的，为什么使用一个全局的计数器？

A：你说的没错，这样做的空间开销确实很大，但是一个全局的计数器受到的影响实在是太多了，可能有成千上万条分支指令影响着计数器的记忆，这样计数器根本起不到预测分支的效果。

Q：那也不行啊！每一条PC上的指令肯定不会全部都是分支指令，这样子不会浪费很多计数器吗？

A：你说的很对，所以有人提出了一个折中的策略，就是令**将2^k**长度的PC内的指令使用同一个计数器，这样既可以降低用作计数器的存储器的容量，又不至于对计数器的记忆产生过于混乱的影响（当然，影响肯定存在，这种情况称为别名）。

在得到以上的结论后，我们使用PHT（Pattern History Table）存放PC值的一部分对应计数器的值，也就是用做计数器的存储。那么2^k的k取多少合适呢？基于某些研究与测试，这个值取在13比较理想，可以领预测的正确率达到90%。我们可以从直观上感觉到，这种预测方法对于循环和递归的分支预测是非常有效的；对于结果较为随机的预测序列也拥有一定的抗干扰能力。不过，我们还有预测成功率更高的策略。

分支预测自然是越快越好，倘若能在当前取指的同时预测出结果，那么下一次取指就可以依靠本次预测的结果。

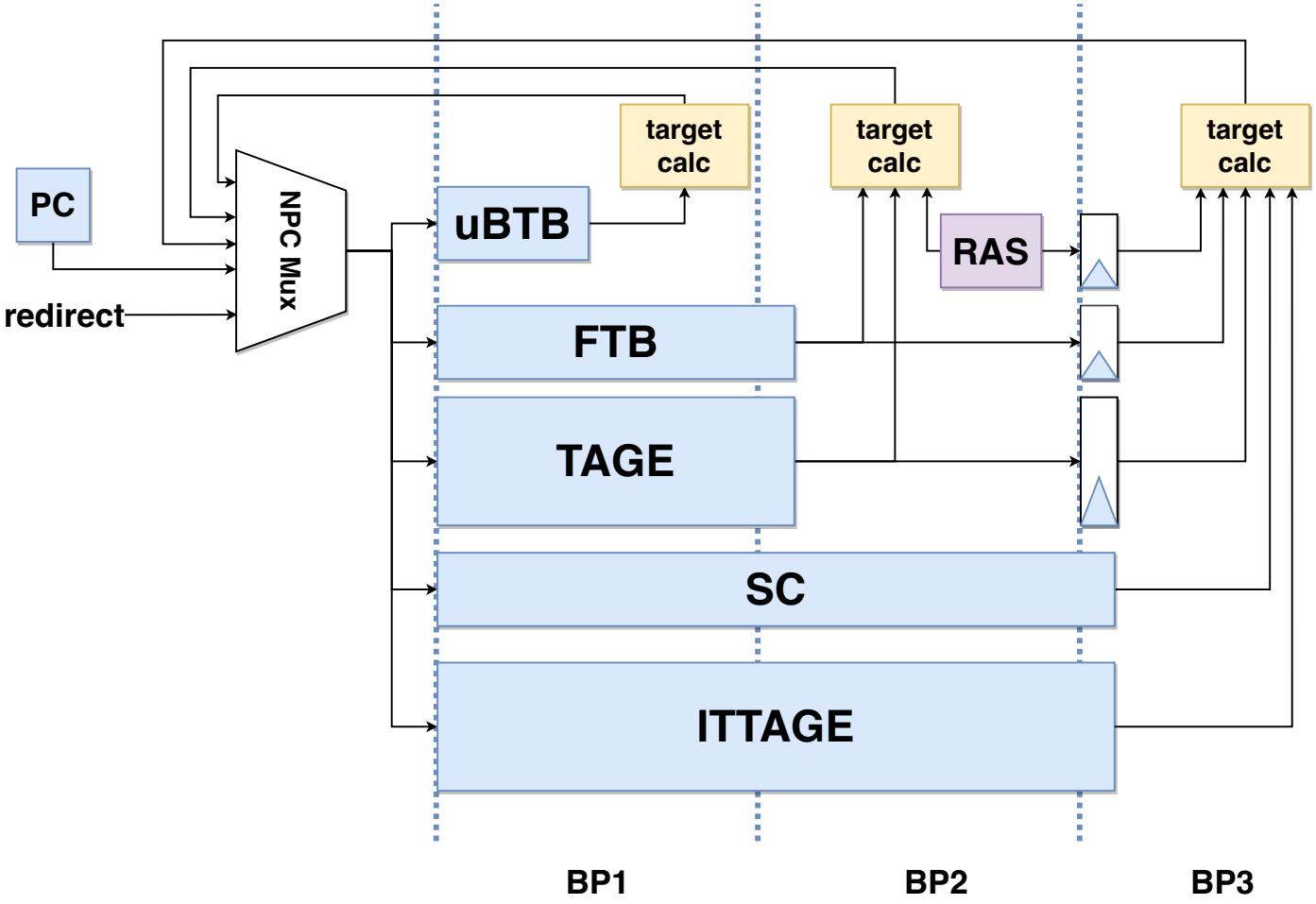
香山处理器使用了一款名为**TAGE**的分支预测部件，它在2006年的分支预测大赛中获得了冠军，具有很高的预测准确率，其同样为一种具有“记忆”能力的分支预测部件。在BPU中，它作为二级预测发挥作用。

香山处理器具有多级预测技术，如结合历史信息进行预测的BTB部件（这也是香山处理器的一级预测部件）。TAGE、RAS等是后继的预测层次，TAGE可以做更加精确的预测，RAS和递归跳转的预测关系更大。当然，如果预测在前面的预测级别就被判定有效，那是最理想的。

当前指令的PC来源有多种，包括后端、前段的重定向、顺序取指等，Next PC由此给出。

预测结束后，指令会被送至IBuffer并送至后端进行处理。

香山处理器的BPU部件参考了Berkeley Sonic BOOM的架构。



下一行预测器（NLP）： uBTB

精确预测器（APD）：FTB、TAGE-SC、ITTAGE、RAS

握手逻辑：BPU 的各个流水级都会连接FTQ（即指令队列），一旦第一个预测流水级存在有效预测结果，或者后续预测流水级产生不同的预测结果，和FTQ的握手信号有效位都会置高

uBTB：根据分支历史和PC的低位异或索引存储表提供精简的预测信息

取址目标缓冲（FTB）：提供预测块内分支指令的信息和预测块的结束地址

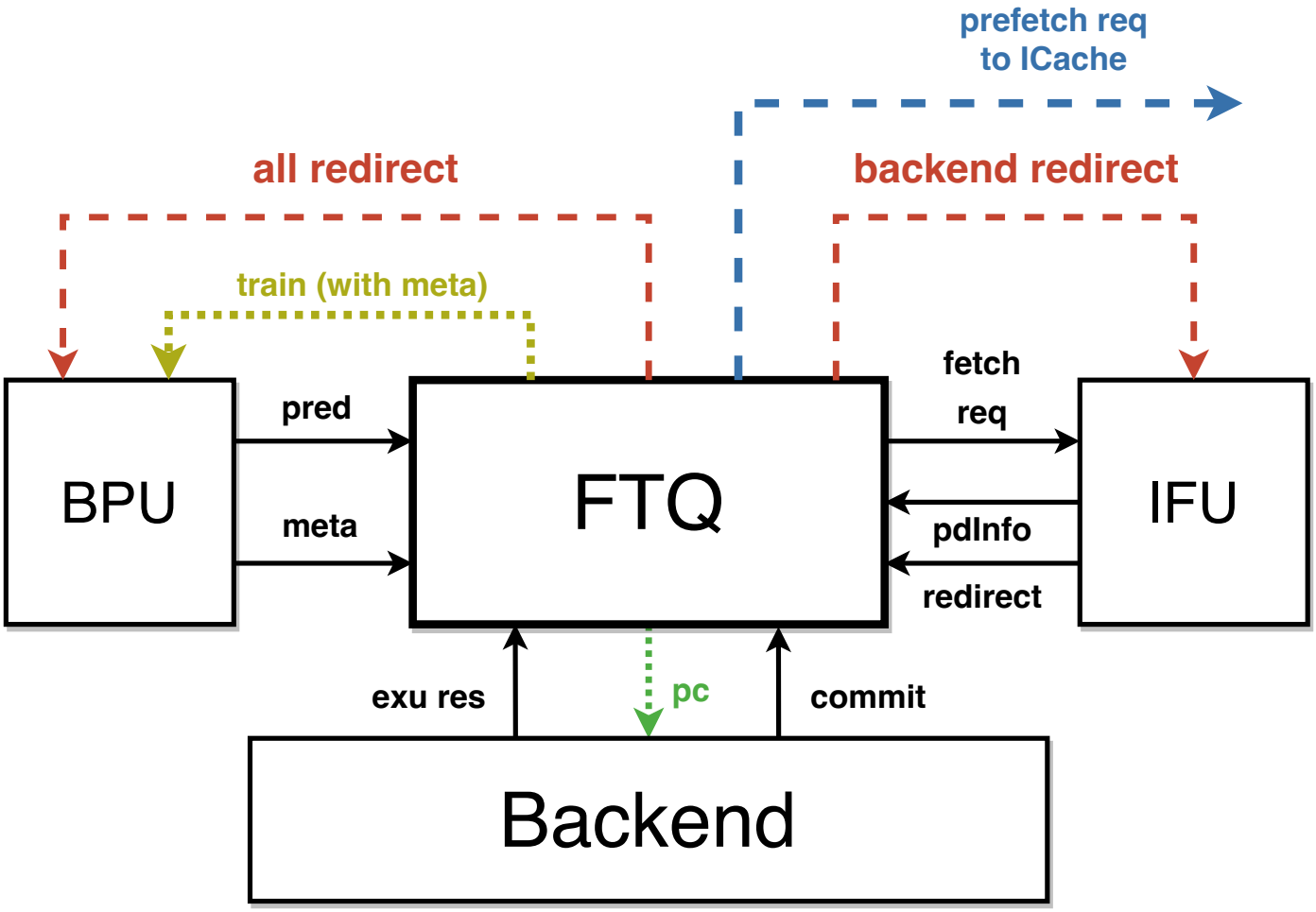
TAGE-SC：TAGE使用当前PC以及历史预测表提供的分支历史信息进行预测，同时他也拥有一个备选逻辑预测，用于在预测信心不足的时候使用。SC是统计矫正器，它可以协助TAGE进行预测，矫正TAGE的预测结果

ITTAGE：主要用于处理特殊的jalr指令（不作为函数返回的jalr等），基本逻辑和TAGE类似

RAS：实现了一个寄存器栈，用于处理call指令的相关预测

2.取指目标队列（FTQ）

主要职能为存放BPU预测的取指目标，并根据目标向IFU发送取指请；同时，它也可以暂存BPU各个预测器的预测信息，并在指令提交后将这些信息送回BPU用作预测器的训练。



FTQ的队列结构，其存储结构主要由几种不同的寄存器堆实现，此处可以参考文档。

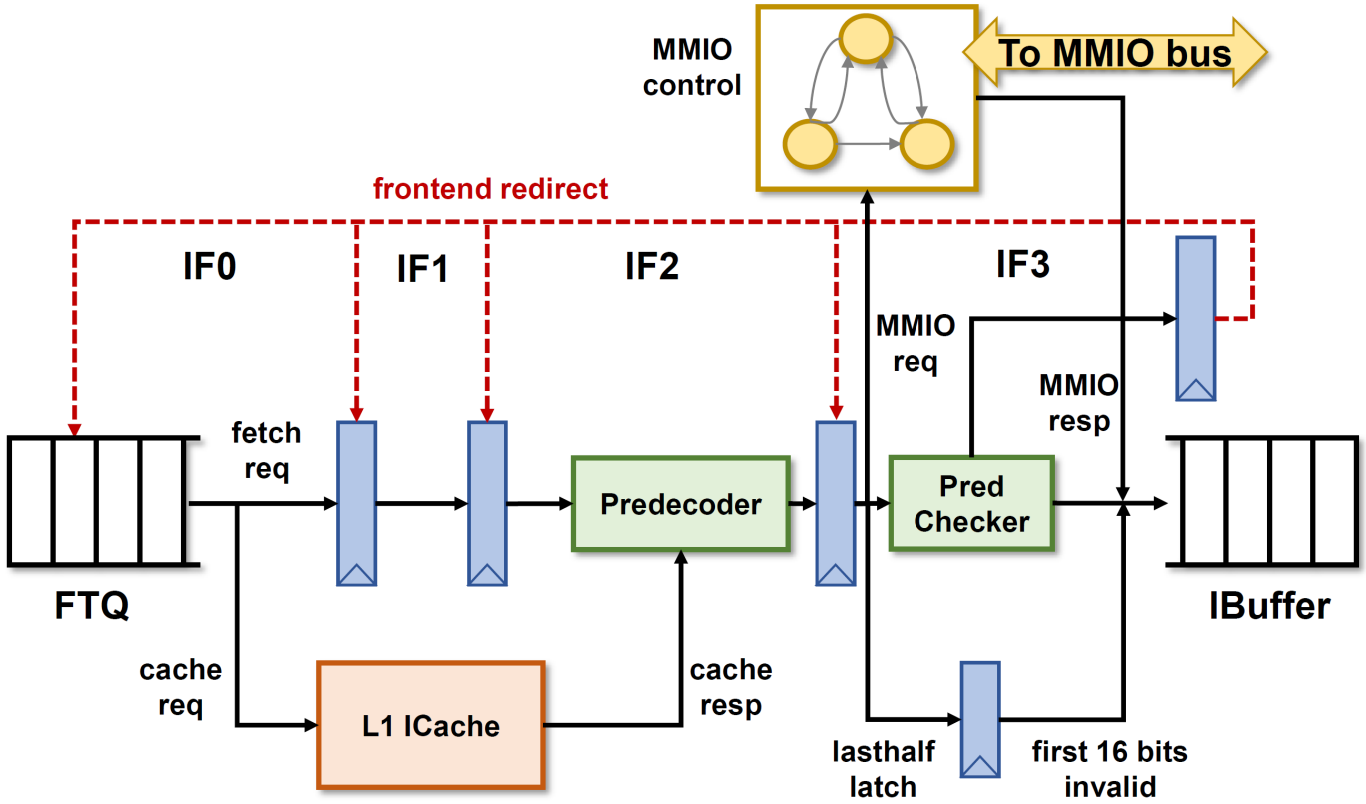
对于一个**预测块**（分支预测单元 (BPU) 每次给取指目标队列 (FTQ) 的请求基本单位，它描述了一个取指请求的范围，以及其中分支指令的情况）来说，其在FTQ当中的生命历程大致如下：

- 预测块从BPU出发，进入FTQ
- FTQ向IFU发出取指请求，等待预译码信息写回

- IFU写回预译码信息，如果检测出预测错误，发送信息给BPU
- 指令进入后端执行，错误预测，则向BPU发送重定向请求
- 指令成功提交，返回信息给BPU进行训练

其中维护两个指针bpuPtr和commPtr，当前者指向n+1时，预测块生命开始；当后者指向n+1时，预测块生命结束。

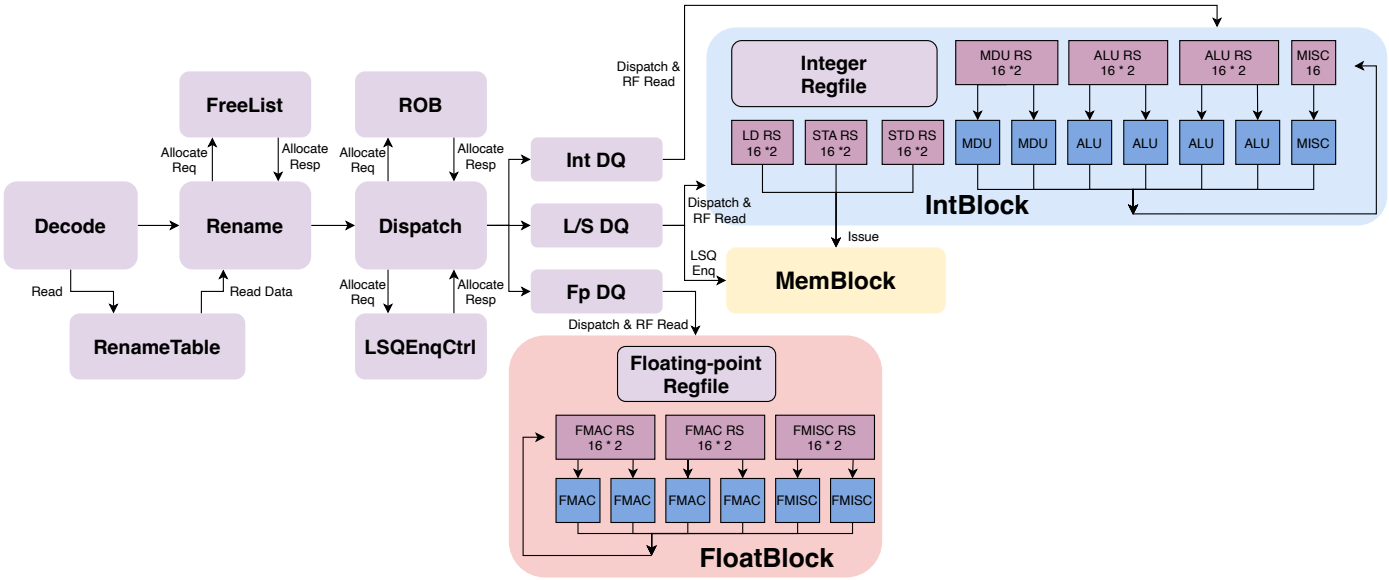
3.取指单元（IFU）



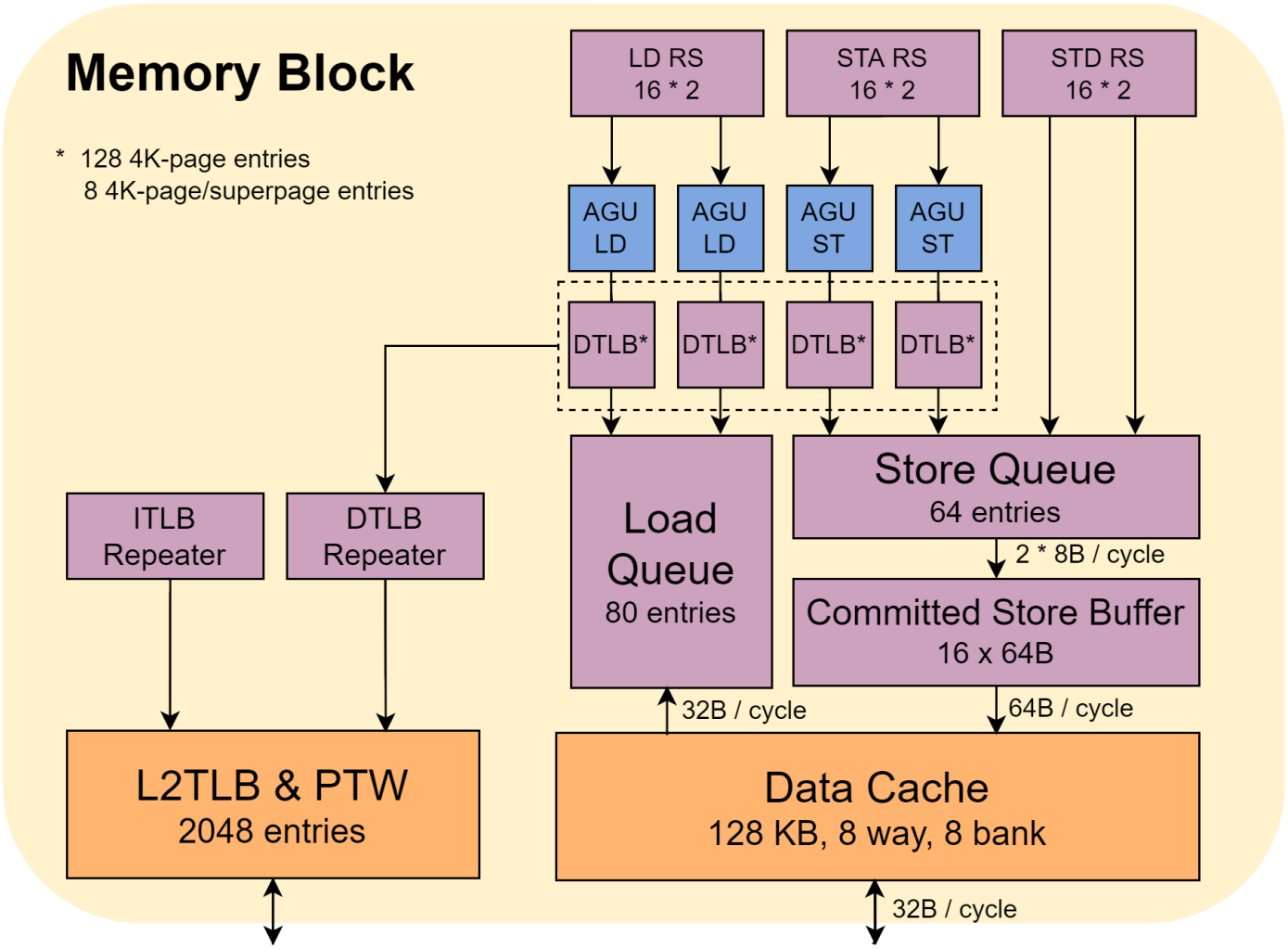
南湖IFU采用四级流水线结构（分支预测-指令缓存解耦）。取指请求在IFU中会经历下面几个阶段：

- FTQ发送指令请求，包括一个32bytes的指令块的起始地址和下一个跳转目标的地址
- IF0阶段同时发送请求给IFU流水线和ICache模块
- IF1会进行一些简单的计算
- IF2阶段，指令缓存ICache会返回数据。首先进行指令切分，得到有效范围的指令码。随后送入预译码器进行预译码，同时将 16 bits 的压缩指令扩展为 32 bits 的指令
- IF3阶段，首先将预译码结果送到分支预测检查器，出错则刷新IFU流水线并且从FTQ重新取指，未发现错误指令则在IBuffer中等待译码

三、后端架构


FreeList
ROB
Allocate Req
Allocate Resp
Allocate Req
Allocate Resp
Int DQ
Decode
Rename
Dispatch
L/S DQ
Dispatch & RF Read
Integer Regfile
MDU RS 16 *2
ALU RS 16 * 2
ALU RS 16 * 2
MISC 16
LD RS 16 *2
STA RS 16 *2
STD RS 16 *2
MDU
MDU
ALU
ALU
ALU
ALU
MISC
IntBlock
Issue
LSQ Enq
MemBlock
Read
Read Data
Allocate Req
Allocate Resp
Fp DQ
Dispatch & RF Read
RenameTable
LSQEnqCtrl
Floating-point Regfile
FMAC RS 16 * 2
FMAC RS 16 * 2
FMISC RS 16 * 2
FMAC
FMAC
FMAC
FMAC
FMISC
FMISC
FloatBlock

?.访存机制


Memory Block
* 128 4K-page entries
8 4K-page/superpage entries
LD RS 16 * 2
STA RS 16 * 2
STD RS 16 * 2
AGU LD
AGU LD
AGU ST
AGU ST
DTLB*
DTLB*
DTLB*
DTLB*
ITLB Repeater
DTLB Repeater
Load Queue 80 entries
Store Queue 64 entries
2 * 8B / cycle
Committed Store Buffer 16 x 64B
32B / cycle
64B / cycle
L2TLB & PTW 2048 entries
Data Cache 128 KB, 8 way, 8 bank
32B / cycle

总体架构

香山处理器的Memblock用于实现访存功能。其主要使用3级Load流水线和2+2级Store流水线实现，具体包括：两条Load流水线、两条Store Addr流水线、两条Store Data流水线。

- Load Queue用于辅助Load指令的高速缓存机制，Store Queue负责暂存指令提交前的Store数据，并且可以提供数据转发。
- 在Store指令提交之后, Store Queue 会将其中的数据搬运到Committed Store Buffer，这个部件会合并存请求，并且在将满的时候统一写入（节约时间）。
- Data Cache为数据缓存，对外提供读写端口（目前不太明白端口作用，以后可能会补充）
- 39sV MMU内存管理单元（3*9的三级页表+12位的页内偏移），负责将虚拟地址翻译成物理地址，同时检查权限，其包含了L2TLB、Repeater等组件。

如果你想看视频讲解，在此附上一个链接：https://www.bilibili.com/video/BV1Y64y1X7Pw?spm_id_from=333.999.0.0&vd_source=d973f2b13853d90dbdaeda352964226c

香山处理器采取乱序访存机制，接下来进行介绍。

乱序访存机制Overview

这一部分作为概览简单说明香山处理器的乱序访存机制，具体细节之后会详细介绍。

Load Hit：在一条load指令命中时, 这条指令会经历3个Stage:

- Stage 0：计算地址，读 TLB，读 D-Cache
- Stage 1：读 dcache data
- Stage 2：获得读结果，选择并写回

在 Stage 2 之后, 还会有一个额外的 stage 处理 stage 2 来不及完成的状态的更新。

Store：对于store指令，分为两个部分：一个是地址计算部分Sta，另一个是数据流水线Std。Sta会经历四个状态：

- Stage 0：计算地址，读TLB，和Load指令类似
- Stage 1：将addr和其他控制信息写入store queue并开始违例检查
- Stage 2：违例检查
- Stage 3：违例检查，允许 store 指令提交

store指令的data计算部分在从保留站中发射后，会直接从保留站中将数据搬运到Store Queue。

Tips：保留站是为了解决相继进入流水线的指令间数据或资源的相关性，在功能部件的输入端设置的暂存寄存器。

Load Miss：对未命中的load指令进行了处理。

Replay From RS：这部分为Load指令和Sta操作从保留站中重发的机制。我们需要在什么时候重发？以load指令为例，在以下的情况中我们可能需要重发：

- TLB缺失
- L1 D-Cache的MSHR（一个用于记录信息的寄存器组）满
- D-Cache冲突
- 转发时数据没有到位

它们有一些共性的特点：

- 发生频率不高(相比于正常的访存指令)

- 这些事件发生时访存指令无法正常执行
- 在一段时间后再执行相同的访存指令, 这些事件不会发生

重发机制就是可能产生问题的指令在保留栈中等待反馈信号，如果反馈信号表示发生了问题，那么这些指令就会从保留站中重新发出。Load流水线安排了两个发送反馈信号的端口（分别在Stage1和Stage2后的额外Stage），而Store流水线则只安排了一个反馈端口。这些反馈端口不仅会返回重发的信号，还会返回一些其他的信息（重发原因等）。

Store To Load Forward：

即转发（北航喜欢叫转发，好像大部分人都称这个机制为前递，大家都知道，不多做解释，我还是习惯叫转发）。

Store到Load的转发被分配到三级流水线执行，使得转发尽可能充分。在转发期间，转发逻辑会检Committed Store Buffer和Store Queue中是否存在load指令需要的数据，倘若存在，则将数据合并到本次load的结果中。

违例操作检查与恢复：将在下文进行介绍。

乱序访存机制大概包括上述内容。这一部分的内容比较丰富，同时也呈现出了和我们在组成原理课程中设计的CPU的不同特点：多条流水线访存。关于其中的一些机制的细节，将会在下文进行介绍。

Load Pipeline

load流水线是访存流水线的重要组成部分。前面提到过，load流水线分为三级，另有一级用于处理一些时序延后的数据。简要介绍其每个流水线的工作内容：

- Stage 0：取出指令和操作数，并计算虚拟地址。将虚拟地址送入TLB进行查询，并送入虚拟缓存进行Tag索引
- Stage 1：拿到Stage 0阶段向TLB请求返回的物理地址，同时完成快速异常检查。将虚拟地址Data索引，并将物理地址送入数据缓存与Tag比较。在这一级，我们还可以将物理地址和虚拟地址送入Store流水线的部件，请求转发。除此之外，还需要根据一级数据缓存返回的命中向量以及初步异常判断的结果，产生提前唤醒信号送给保留站，如果在这一级就出现了会导致指令从保留站重发的事件，通知保留站这条指令需要重发（这是刚刚提到的Load流水线第一个重发反馈点）。可以看到，这一部分需要做的工作还是不少的。如果你没办法全部看明白，也可以**不用纠结，继续往下看**
- Stage 2：本阶段首先需要完成异常检查。而后，可以根据一级数据缓存（即Cache）以及转发的结果选择数据，并对返回值做裁剪操作。同时更新Load Queue的状态，并根据结果的类型向不同模块发送信息
- Stage 3（一个额外的Stage）：根据 dcache 的反馈结果, 更新 load queue 中对应项的状态；根据 dcache 的反馈结果, 反馈到保留站, 通知保留站这条指令是否需要重发（第二个重发反馈点）。本流水级是专门为处理时序延迟而设计的

接下来我们来讨论一下Load Miss，这是Load流水线中一种比较特殊的阻碍。为了移除这一阻碍，处理器会采取以下的操作：

禁用档期那指令的提前唤醒：Stage 1阶段物理地址进入D-Cache与Tag进行比较，流水线可以得知当前指令是否Miss。倘若Miss，该指令就不会被提前唤醒，提前唤醒位被置低。

更新Load Queue的状态：load指令Miss后，结果不会协会寄存器堆，也不会占用写回端口，同时会更新Load Queue的状态。

分配D-Cache MSHR：Miss的load指令会在Load Queue中等待D-Cache重填，D-Cache也会为该load指令分配MSHR。

这里简单介绍一下MSHR，即缺失状态处理寄存器。早起的缓存设计中，一旦发生Cache Miss，处理器的后续请求就会被阻塞直至Miss的数据被重填。这显然损失了性能，于是引入MSHR来记录缺失状态，同时保持Cahce对处理器的响应（即不阻塞）。拥有了记忆能力，Cache就可以持续相应其他的请求，并可以随时"顺手解决"记录在MSHR中的缺失记录。当然MSHR要是填充满了，再发生缺失时，就不得不通过阻塞来等待Refill。除此之外，MSHR还有诸多种实现方法。

MSHR分配逻辑比较复杂，耗时较久，需要等待下一拍才能将结果反馈给Load Queue。

监听Refill与更新Load Queue：倘若Miss的地址已经回填，那么可以解除这一次Load Miss的标记，并将结果返回给Load Queue。同时在Stage 3，MSHR也会返回Miss分配的结果给Load Queue——倘若MSHR分配失败，则需要请求保留站重发这条指令。

这几是Load Miss对应的几个处理阶段，下面我们来介绍Load Pipeline的重发机制。

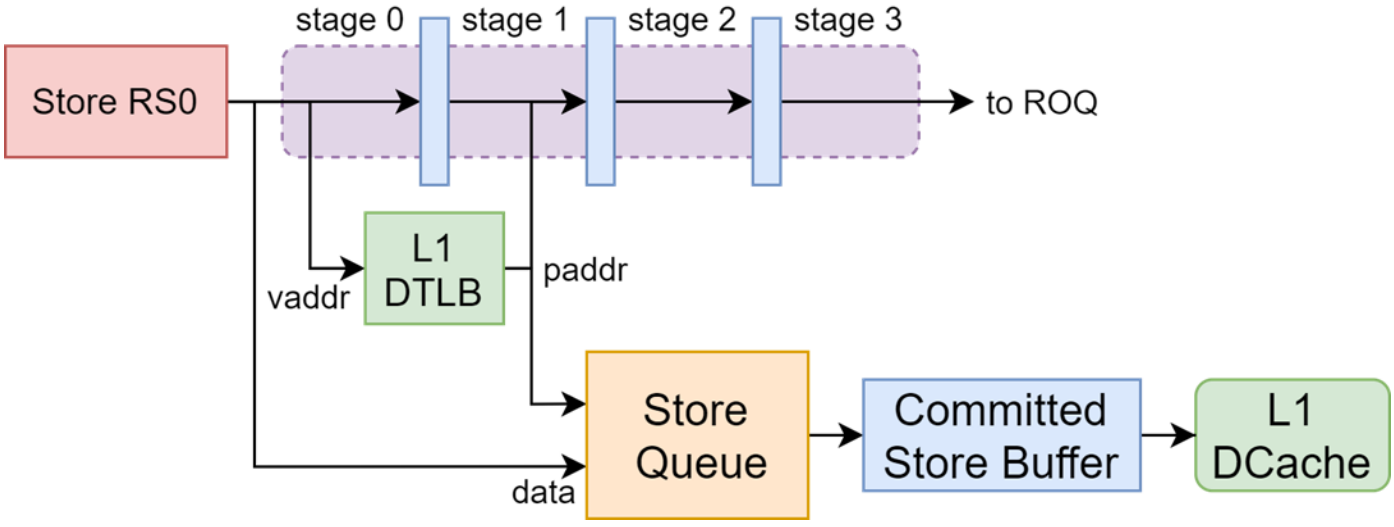
香山处理器的Load流水线是非阻塞的，即其一直保持有指令在流水中。除了Load Miss之外，其他Load指令的大部分异常情况都会通过保留站重发来实现该指令的重新执行。前面提到过，Load Pipeline有两个位置设置了Feedback端口（feedbackFast和feedbackSlow），可以向保留站发送重发请求。事件如**TLB Miss**会使用feedbackSlow端口，在Stage 3发送重发请求（即反馈信号），且保留站接受后等待延迟结束才重发指令，这是因为重填TLB需要时间，过早重发没有意义；而Bank Conflict（存储体冲突，即当存储体没有恢复的时候就又被访问而产生的冲突），可以在feedbackFast端口直接发出，并且保留站无延迟重发。

除此之外，Load Pipeline还有一些别的内容，不过碍于篇幅我们在此不多做介绍，读者可以自己到香山处理器的文档中Explore。

Store Pipeline

接下来介绍香山处理器的Store流水线，官方文档中标记此仍为南湖架构的流水线。

你可以看到，南湖架构的Store Pipeline拥有Sta和Std两种功能不同的流水线，每种流水线各有两条。这是因为其采取了**地址和数据**分离的执行方式（我们在组成原理课程中也是这样做的，还记得吗）。



Store Addr有四个流水级，前两个流水级负责将Store的控制信息和地址传递给Store Queue，后两个流水级负责等待访存依赖检查完成。具体来讲：

- Stage 0：计算虚拟地址，并将虚拟地址送入TLB
- Stage 1：TLB产生物理地址，并完成快速异常检查，开始访存依赖检查。将物理地址送入Store Queue等待数据
- Stage 2：访存依赖检查，更新Store Queue

- Stage 3：完成访存依赖检查，通知ROB可以提交指令

Store Data在保留站提供数据后，该流水线会立刻将数据写入Store Queue的对应项中，官方文档仅描述了其Stage 0，就是将保留站给出的数据写入Store Queue。

遇到了TLB Miss等问题时，处理的方法和Load Pipeline类似。

TLB为全相联，32项4KB Page，4项2MB &1GB Page，ITLB（指令快表）采用PLRU替换策略，DTLB（数据快表）采用随机替换策略