Diplomacy

- 整个 Diplomacy 架构主要由三种对象，`LazyModule`，`LazyModuleImpLike`，`BaseNode`和它的相关子类型（最基本的`MixedNode`，`SinkNode`，`SourceNode`等等）
  - `LazyModule`中的`module`成员是`LazyModuleImpLike`类型，对应 chisel 中的`Module`子类，是实际的硬件模块。通常使用 Diplomacy 结构时，定义新的模块继承`LazyModule`，override `module`成员，来指定我们最终需要的硬件模块
  - 除了重载`module`成员，在我们定义的`LazyModule`子类中还会定义若干`BaseNode`，一个`BaseNode`实际上定义了`module`的接口形状。不同`BaseNode`的相互连接则定义了参数传递的方向和`module`间的连接（通常是不同`LazyModule`中的`BaseNode`间的连接）
  - 基本的`:=`连接运算符会产生一条有向边，有向边的源节点的`module`会产生一个`Output Port`，类型为`BO`，目的节点会产生一个`Iutput Port`，对应的数据类型为`BI`，然后源节点的`Output Port`和 目的节点的`Input Port`会自动连接
- 具体参数的传递和协商：
  - `BaseNode`的连接构成了一张有向图（要求连接不能构成环，否则会抛出异常）
  - 实际的源头参数只由三个地方生成：`SourceNode`向 Downward 方向传递参数`DO`，`SinkNode`向 Upward 方向传递参数`UI`，`BaseNode`连接时获取到的隐式参数`Parameter`。
  - `SourceNode`和`SinkNode`以外的节点有自己的`mapParamsD`和`mapParamsU`函数，节点根据自己收到的`DI`参数，调用`mapParamsD`产生需要输出的`DO`参数，而根据自己收到的`UO`参数，调用`mapParamsD`产生需要输出的`UI`参数
  - 因此总的说来，`upward parameter`从`SinkNode`流向`SourceNode`，`downward parameter`从`SourceNode`流向`SinkNode`。
  - 参数协商的结果由`edgeI`，`edgeO`函数决定。`edgeI`函数根据这条边上的`DI`，`UI`参数（`DI`由`SourceNode`主导，而`UI`由`SinkNode`主导，因此这里的协商综合考虑了两者）和产生这条边的`BaseNode`连接时隐式传入的`Parameter`参数决定最终的参数`EI`。`edgeO`函数则根据这条边上的`DO`，`UO`参数和`Parameter`参数决定最终的参数`EO`。`bundleI`函数根据`EI`的类型产生输入端口`BI`，`bundleO`函数根据`EO`的类型产生输入端口`BO`
- `a :*= b`和`a :=* b`都表示两个`BaseNode`进行多次连接（对应产生多个 port），具体的连接次数由`resolveStar`函数决定，两个运算符区别在于前者调用`a.resolveStar`，而后者调用`b.resolveStar`
- `a :*=* b`将根据`a, b`的实际连接情况，退化为`a :*= b`（当`a, b`仅出现在`:*=`连接符中时），或退化为`a :=* b`（当`a, b`仅出现在`:=*`中时），如果在`:*=*`的可达子图中的节点出现在了`:*=`和`:=*`中，报错。如果`a, b`的`flexibleArityDirection`均为`false`，`a :*=* b`退化为`a := b`
- 一个之前困扰我的地方在于，`LazyModuleImpLike`中生成的`AutoBundle` IO 没有指明 port 的方向，即没有调用`Input`，`Output`方法。然后我发现了这个 how to understand the flip in autobundle（） and in makeios 后才搞明白。其实就是 Chisel 里面 port 如果没有指明方向，会默认是 Output，如果叠加了 Flipped，则奇数次时是 Input，偶数次是 Output。由于`AutoBundle`类中对于 Input 有一个 flip 操作，因此如果 BI 类型中只需要 Input，而 BO 类型中只需要 Output，在生成 BI，BO 时就不需要显式指定方向（这就是为什么 adder example 能够正常运行）。

TileLink 1.8.1

- ~~4.2. Request-Response Message Ordering，这一节中要求 agents 能够在发送 request 的同一周期接收 response，这和 forward progress rules 有啥关系呢？~~ 我懂这啥意思了，就是说为了满足 forward progress rules，可以推理出 agents 不能够在 A Channel 的第一个beats 被接收后延迟一周期才拉高 D Channel 的 ready 信号。因为如果 master agents 这么设计后，D Channel 不能够满足 rule3.1：因为如果 slave agent present reponse message，但由于 request message 还没被接收（此时 A D Channel 的 valid 信号都被拉高，但 ready 信号都被拉低，这是合理的状态），整个系统处于死锁状态。而 assumptions 告诉我们，在证明 D Channel 满足 rule3.1 时，不能假定更低优先级的 A Channel 满足 rule3.1（即假定 prensented request in A Channel will be accepted by slave agents）。因此可知 D Channel 不满足 rule3.1，说明这个 master agent 的行为不符合 forward progress rules
- 协议设计的关注点：目前看到的复杂总线协议（AXI4，TileLink）都基于 request-response 模型，following features are necessary to pay attention
    - Can we allow multiple in-flight request? If yes, response priority or totally out of order response?
    - Supported atomic operation? Common atomic operations include(TileLink only supports the second type)
        - LL/SC (load link, store conditional)
        - read-arithmetic-write (read value, and do specified arithmetic on it, write result into original address, return original value)
        - compare-swap (master sends two values, compare value and swap value, if compare value is matched, swap value will be written)
- 不是很懂为啥要在 Channel B 上实现 get, pull 等一系列 operation，说是为了支持 update 类型的一致性协议，就算此时允许 slave 向 master 获取数据，那具体 update based coherence protocol 要咋实现呢？

Chipyard and Boom

- Chipyard 的入口程序在于 `Generator.scala`，在 `make verilator` 进行模拟时，默认的顶层模块是 `Rocket-chip/TestDriver.v`，它负责提供 clock 和 reset 信号。它封装了 chisel 这边的顶层模块 `TestHarness`，再下一层是 `ChipTop`，它又封装了 `DigitalTop`
- `DigitalTop` 继承了一个重要的 trait `InstantiatesHierarchicalElements`，这个 trait 的构造函数中使用传入的 `Parameters` 读取 `TilesLocated` 参数，得到一个 `Seq[CanAttachTile]`，随后调用它们的 `instantiate` 方法以及内部的 `tileParams` 的 `instantiate` 方法。实际上 `RocketConfig` 或者 `BoomConfig` 中都设置了 `TilesLocated` 对应的值，例如 `BoomConfig` 中对应为 `Seq[BoomTileAttachParams]`，`BoomTileAttachParams` 继承自 `CanAttachTile`，它内部的 `BoomTileParams` 的 `instantiate` 方法会实例化出 `BoomTile`。因此不同的 Config 可以配置出不同的 core
- `BoomTile` 中包含的一些子模块
    - BoomCore
    - LSU
    - PTW
    - BoomFrontend
    - BoomNonBlockingDCache
    - rocc
- `DecodeUnit` 是完全的组合逻辑，输入的 `MicroOp` 由前端传过来，有些位是无效位，解码后输出的 `MicroOp` 对这些位进行了填充。
- rename stage 的结构（TODO: 当理解了分支预测后增加分支预测相关的内容）

    - freelist：核心存储是 `numPregs` 个寄存器，每个 bit 表示对应的物理寄存器是否空闲。另外还有额外的 corewidth 个 宽度为 `numPregs` 的寄存器，保存了下一次分配的物理寄存器的 oneHot 编码。分配的组合逻辑为，每个周期检查 corewidth 个待分配物理寄存器是否填充好了，如果没有则从 freelist 中选出空闲的填充进来。当外界需要分配物理寄存器时（ren2 阶段），直接将待分配物理寄存器的编号送出去就好了。我猜测分配物理寄存器的过程提前一个周期的目的是为了缓和 rename 的时序，毕竟这里送出去的物理寄存器还要在同一个周期用来修改 maptable
    - maptable 的核心存储是 长为 `log2Ceil(numPregs)`，宽为 `numLregs` 构成的寄存器阵列，对应一个逻辑寄存器到物理寄存器的映射。
    - busytable 的核心存储是 `numPregs` 个寄存器，每个 bit 表示对应的物理寄存器是否将要被写。**从本质上讲，重命名使得指令间的依赖变为了物理寄存器间的读写依赖（不论是 corewidth 间的指令还是 corewidth 内的指令），当指令需要读的物理寄存器不再 busy 后就可以发射了**。
- 一些时序上的情况，decode 和 rename 共两个周期，为了方便，我们简称为 decode 周期 和 rename 周期
    - decode 周期完成的内容：
        - 从前端传来的 `uops` 由 `DecodeUnit` 解码后生成 `dec_uops`，其中 `dec_uops.br_tag` 和 `dec_uops.br_mask` 由 `BranchMaskGenerationLogic` 产生，这些都是一个 cycle 内的组合逻辑完成。
        - rename 模块根据 dec_uops 要读写的寄存器做 bypass 处理（因为此时可能 rename 周期正在修改 maptable 的映射，因此这里有 bypass 逻辑来规避同时读写的冒险），读 maptable 来获取要读写的逻辑寄存器对应的物理寄存器（我想要获取写寄存器对应的物理寄存器在于该指令完成后就可以释放该物理寄存器到 freelist 中了），maptable 读取的结果或 bypass 的结果填充在微指令的 `prs1`, `prs2`, `prs3`, `stale_pdst` 中。**但请注意，此时还未考虑 corewidth 这一排指令内部的依赖**
        - 看上去这个周期做的事情很多，但 decode， bypass，读 maptable 这几件事其实都是并行的，所有时序应该不会太差（因为 riscv 的逻辑寄存器在固定的编码位置，因此 bypass 和 读 maptable 不依赖于 decode 的结果）
        - decode 周期结束后的微指令存放在 `AbstractRenameStage` 的 `r_uop` 寄存器中，然后还有对应的 `r_valid` 寄存器来指示其是否 valid。
    - rename 周期完成的内容：
        - 微指令 `ren2_uops` 从 `r_uops` 中读出，读取 busytable 设置好微指令的 `prsX_busy` 位。如果外部指示微指令可以发送到下一周期了（即输入 `io.dis_fire` 拉高），那么读取 freelist 中预分配好的物理寄存器，并将映射写入 maptable 中，同时将该物理寄存器在 busytable 中的位拉高。这些写入操作将在下一周期的时钟上升沿完成。**请注意，这些写入的操作仅依赖于指令要写的寄存器，与 corewidth 这一排指令内部的依赖无关。**
        - 接下来在 corewidth 这一排指令内部进行 bypass，从而为每条微指令设置正确的 `prsX` 和 `prsX_busy` 值
        - 由于实际上有 general / float register rename table 两个 table，因此为得到最终的微指令，还需要一级 mux（根据指令的源寄存器类型来选择正确的 `prsX` 等）
        - 每条微指令的 `iq_type` 表明了该微指令需要占用的 issue queue，`BasicDispatcher` 逻辑很简单，假设微指令会进入到每个 issue queue 中，因此当所以 issue queue 对应的 dispatch slot 都 ready 时才拉高 ready 信号。然后根据微指令的 `iq_type` 发射到对应 issue unit 中即可。另一个更复杂的 `CompactingDispatcher`，它保证了仅有指令需要的 issue queue 满了才会停止发射（就是说会先查看该条指令依赖的 issue queue，这些 issue queue ready 了，dispatcher 回传的 ready 信号就会被拉高），同

时允许 corewidth > dispatch width of issue queue。但从 `Compactor` 这个模块的实现上看，为保证 `CompactingDispatcher` 的正确性，需要有额外的两点保证：

- 如果 issue queue 的 dispatch slot i 的 ready 信号为 0，那么大于 i 的 dispatch slot 的 ready 信号也必为 0，保证总是编号最低的那一批 uop 被选中发射（不会有后面的指令比前面的指令先发射），**TODO：指令进入到 issue unit 中的顺序与在 ROB 中的顺序是什么关系**
- 如果一条指令可能被发射到多个 issue queue，那么这些 issue queue 的 dispatch slot 的 ready 数目需要相同，否则可能导致在一个周期内，一条指令被发射到了某个 issue queue，但没被发射到另一个 issue queue

另外，由于只要 issue queue 和 iq_type 对上了就会被发射，因此在配置时要求不能有功能重叠的 issue unit?

- 总的来说，当使用 `BasicDispatcher` 时会有一点低效，编号为 k 的指令（编号从 0 开始）要想成功发射，需要对应的 issue unit 有至少 k + 1 个空槽位才行。另外，只有当 dis_fire 拉高时，dispatcher 收到的对应的 valid 信号才会拉高，因此实际上 issue queue 收到 valid 信号是组合依赖于自己发出的 ready 信号的
- TODO：将该指令写入 rob 中

- 最后提一下 corewidth 内部指令的发射顺序，内部编号越小的指令 pc 更小，因此在数据依赖和控制依赖上只能是编号大的指令依赖于编号小的指令。如果指令 I 能从 decode 周期发射到 rename 周期，或者从 rename 周期发送到 dispatcher，那么编号比 I 小的指令也一定能发射（因此保证是发射是顺序的）。当 corewidth 中的指令全部都发射完成后，才会拉高对应的 ready 信号（例如，当这一排指令都从 decode 进入到 rename 周期时，才会拉高 dec_ready 信号。都从 rename 进入到 dispatcher 后时才拉高 dis_ready 信号）。这可能损失了一些性能，例如先有两条指令进入了 rename 周期，不能从前端补上这两条指令占的 decode slot，不然就没法保证只有编号大的指令依赖于编号小的指令了。但这也简化了控制逻辑。另外一个稍微违反直觉的结果是，先有两条指令进入了 rename 周期，那么需要等这两条指令发射到下一周期时剩下的指令才能进入到 rename 阶段如果指令（因为一旦有 stall，dis_ready 信号就会被拉低）。但大部分情况下都是所有 decode 指令一起进入到 rename 阶段，因为唯一可能发生部分指令先进入 rename 的情况是 branch mask 用完了，但这种概率我觉得很小
- TODO：解释 dec_ready 和 dis_ready 信号，~~解释 corewidth 内部指令的发射顺序~~
- issue 模块
  - 在每个 issue queue 中，每个 issue slot 检查该微指令是否可以发射了（即所有 busy 位都被拉低了，每个周期 wakeup port 会告知哪些物理寄存器可用了，如果匹配上了，issue slot 拉低相应的 busy 位）。
  - 在 `issue-unit-unordered` 中根据 issue slot 的空闲位回传给 dispatcher 的 ready 信号的产生逻辑是：在循环里，每找到一个空闲的 issue slot，就把它分配给编号最低的还没有找到对应 issue slot 的 dispatch slot。注意这里没有考虑 dispatcher 的 valid 信号。
    - 对于 `BasicDispatcher`，就可能出现的情况是，corewidth = 4 中前两条指令发送给 issue unit 后，该 issue unit 可能后来又多了两个空位，但这两个空位会默认送到 corewidth 前两个槽位中，一直要等到 issue unit 有 4 个空槽位时才能将剩下的两条指令发过来？
    - 但对于 `CompactingDispatcher` 还好，因为 `Compactor` 模块的逻辑是，对于这一排 corewidth 指令，每有一个 valid 的，就发送给还未使用的下一个 dispatch width slot（即它考虑了 valid 信号，但没有考虑 ready，因此不会出现 `BasicDispatcher` 中说的问题），另外由于 `issue-unit-unordered` 保证了高位 ready 了必然低位 ready，因此不会出现说后面的指令比前面的指令先进入 issue unit

    - 再说一下 `issue-unit-unordered` 模块中如何选择将 issue slot 中微指令发送到 `RegisterRead` 模块中
        - 每个 issue queue 的一个 issue port 对接一个 exe unit。同时每个 issue slot 中的微指令如果可以发射了，就拉高 request 信号，如果是 high priority 指令，同时拉高 request_hp 信号。然后从编号最小的 issue port 开始循环，对于每个 issue port，从最小 issue slot 开始循环检查是否可以发射指令且是否与该 port 对应的 exe unit 匹配（high priority 指令优先），如果匹配，那么将该 slot 的微指令发送到该 port 中，同时拉高该 issue slot 的 grant 信号。**注意，不同于 issue queue，这套组合逻辑允许多个 exe unit 有重复的功能单元**
        - 容易发现，这种发射方式的优先级与 issue slot 编号有关，而与指令进入 issue slot 的顺序无关，这当然是不好的，可能导致一条指令进入了高编号的 slot，半天都没办法发射。通常我们当然希望这个 issue queue 具有 FIFO 属性，尽可能地发射早进入队列的。
    - 然后我们再来看 `issue-unit-age-ordered` 模块对 issue queue 的实现。核心逻辑上来讲，它每次将 issue slot 向前滑动，来保证新进入的指令的发射优先级一定更低。
        - 发射指令到 exe unit 的逻辑和 `issue-unit-unordered` 完全一样，但会更简单一点，因为不再考虑 request_hp 信号了。依然是考虑每个 issue port，然后 issue slot 编号最小的满足发射要求的指令优先发射
        - 核心在于滑动逻辑，issue slot 的 `will_be_valid` 信号给出下个周期该 slot 是否依然 valid （在不考虑有新微指令被写入该 slot 的情况下）。最重要的 `shamts_oh` 信号用 onehot 编码给出了每个 slot 需要滑动的距离（它的产生逻辑也很简单，从最小编码的 slot 开始扫描，每碰到一个下周期将要空闲的 slot 就把 `shamts_oh` 左移动一位）。最后根据 `shamts_oh` 给出的移动距离来搬运 issue slot 即可。如果一个 slot 对应的 `shamts_oh` 不为 0，拉高它的 clear 信号（因为这个 slot 中的微指令要被搬到更小的 slot 里了，但同时这个 slot 又可能搬进来新的微指令，因此 valid 信号的优先级会高于 clear 信号）。这也说明 issue slot 中的 IO 信号某些只用于 `issue-unit-unordered`，例如 request_hp，一些又只用于 `issue-unit-age-ordered`，例如 clear
        - **滑动逻辑中相当精妙的一点在于，设置了滑动距离上限为 dispatchWidth，这就是相当体现硬件思维的地方了。因为我们完全没必要向下滑动时把所有空位占满，让所有 valid 的 issue slot 紧贴在一块。只需要在滑动距离达到 dispatchWidth 时，保证所有的新指令都能够放进 issue slot 中就行了。通过设置滑动上限，简化了硬件电路的实现。**
- register Read 阶段，不太理解为什么这个模块要分两个周期完成
    - 第一个周期进行第二阶段的译码来填充微指令的 `ctrl` 位，这也是精妙的设计，没有必要在 decode 阶段译码出需要的所有的控制逻辑（这样避免了 decode 阶段成为关键路径），因为 decode 和 rename 做得越快，我们就可以越早地进入到乱序的阶段。
    - 第二个周期向寄存器进行读操作，然后接收额外的 bypass 操作数（这里其实有两层的 bypass，因为读寄存器堆时，寄存器堆那里发生了一次 bypass，然后这里又发生了一次 bypass）。最终将读出的数据写入到寄存器中，在第三个周期发送到
- exe unit 阶段
    - 每个 issue port 对应一个执行单元，register Read 阶段完成后，该微指令被送往对应的执行单元。暂时不考虑浮点流水线时，issue queue 有两个，一个是 `mem_iss_unit`，`memWidth` 表示它的 issue 宽度，对应的有 `memWidth` 个仅有 mem 的 `ALUExeUnit`。另一个是 `int_iss_unit`，`intWidth` 表示它的 issue 宽度，对应的有 `intWidth` 个有 alu 的 `ALUExeUnit`。
    - 每个 issue port 对应一个执行单元，每个执行单元中可能包含多个功能部件

- `ALUExeUnit` 是执行单元之一（区别于功能部件），其中存在的功能部件包括：ALU，RoccShim，Mul，IntToFP，DIV，MemAddrCalc。目前有两种可能的外部实例化。一种是仅有 mem，它的 `writesLlIrf` 为真，但 `writesIrf` 为假。一种是有 alu，没 mem，其它的单元可能有，它的 `writesIrf` 为真。
- 基类 `ExecutionUnit` 有四个可能的结果输出端口，`iresp`，`fresp`，`ll_iresp`，`ll_fresp`。`ALUExeUnit` 不会使用 `fresp`。
  - `ALU`，`Mul`，`DIV` 共用 `iresp`，当 `Mul` 存在时，`ALU` 会额外流水两个周期来对齐 `Mul`，保证不会与 `Mul` 争抢输出端口。而 `DIV` 使用输出端口的优先级最低，当 `ALU` 和 `Mul` 都不输出时它才能输出。
  - 只有 `ALU` 的结果会进行 bypass
  - MemAddrCalc 单元计算 Store 指令的地址，并将地址和数据传输给 lsu，使用 `ll_iresp` 和 `ll_fresp` 来输出从 lsu 接收到的 load 数据。
  - IntToFP 使用 `ll_fresp` 进行输出，RoccShim 使用 `ll_iresp` 进行输出。TODO：搞清楚这些单元做了什么
- TODO：Alu 的中间结果和输出会用于 wakeup 和 bypass，fast wakeup 和 slow wakeup，详细描述这一行为
  - wakeup port 由两部分组成，一部分是一个周期就可以完成的 alu 指令（bypassable），它们在从 issue slot 中发送给 register read 的同时发送给 wakeup port，这些构成了 fast wakeup。另一部分是整个 exe units 中 unbypassable 的数据（包括 mul，div，mem 这些部分），当整个 exe unit 走完之后才能传给 wakup port，这些属于 slow wakeup。
  - 有两个模块包含 wakeup port，一个是 rename stage 中的 busytable，它接收 wakeup 信息，清除对应物理寄存器的 busy bit，注意这里不包含对读的 bypass 逻辑，即 wakeup 对 busytable 的写不会前递给同一周期的读操作
  - 另一个模块是 issue unit，每个 issue queue 的 issue slot 都需要监听所有的 wakeup port，来判断自己的需要的操作数是否准备就绪了（因此这里逻辑复杂度是 issue entry x wakeup port）。上面讲到在 busytable 中没有对 wakeup port 的数据进行 bypass，是因为 issue slot 中已经存在监听的逻辑了，issue slot 中不是帮助 register 中的微指令进行监听，而是帮助即将写入 register 的微指令监听，换句话说，指令最开始的监听发生在 rename stage。因此微指令不可能错过 wakeup
  - 仅有 alu 指令会进行 bypass，当 alu 和 mul 同在一个 exe unit 中时，alu 类指令花费三个周期完成（三个 pipeline register），当 exe unit 中有 alu 但没有 mul 时，alu 类指令只花一个周期完成（一个 pipeline register）。bypass 是输出当周期 alu 的结果，以及除最后一级流水线寄存器外的寄存器中的结果（最后一级流水线中的数据直接对接寄存器堆写回了）。**因为 alu 实际的执行只需要一个周期，bypass 和 fast wakeup 保证了有数据依赖的 alu 指令能够在相邻的周期完成（在 mips r10000 中的 instruction queue 章节中也提到了这个设计）。对于 fpu 指令来讲，bypass 和 fast wakeup 依然有一些提升，只是从提升的 cycle 百分比上来看不如 alu，不太清楚是不是因为这个原因 boom 没有做浮点的 bypass 和 fast wakeup**
  - 现在我们来考虑 wakeup 和 bypass 的功能是否正确。即要保证每个指令都不会错过需要的 wakeup 信号，这个在第三点已经说过了。因此我们重点关注第二条，每个发射的指令能否保证读到正确的操作数，不管是 bypass 过来的还是从 register 中读到的。在最早情况下，该指令落后依赖的 alu 指令一个周期发射（fast wakeup），此时该指令在读寄存器时，依赖的 alu 指令正在 exe unit 的第一个周期，由于 bypass 会输出当周期的 alu 结果，因此在 register read 处的 bypass 逻辑使得我们读到了正确的操作数。最晚则是该指令读寄存器时，alu 指令的结果已经输出到了最后一级寄存器中，这个时候寄存器堆内部的 bypass 逻辑就派上用场了（还记得我们前面提到的在 register read 阶段

发生的两次 bypass 吗），因为 exe unit 的最后一级流水线寄存器对接寄存器堆进行写回
    - TODO：pipeline mul 中使用了 retiming 技术，了解？
    - TODO：搞清楚 exe unit 和其它模块对接时的 valid ready 逻辑，对于 mul 和 alu 这种完全流水的部件，总是 ready 的，其它模块呢？
    - TODO：exe_unit 输出的 fu_type 间接地表明了自己是否 ready，但 fu_type 信号对接的是 issue_unit 而不是 register read，这会不会有什么问题？
    - **TODO：`PipelinedFunctionalUnit` 中有一组 `r_valids` 信号来记录流水线某个阶段的指令是否有效，但其只用于指示 `io.resp.valid`。但在其子类 `ALUUnit` 中，又新定义了一组 `r_val` 信号用作同样目的，只是它不会被 `brupdate` 和 `kill` 信号抹掉。然后这组信号仅用于 `bypass.valid`，为啥要这样分开整？`r_val` 信号不被抹掉会导致错误吗？**
    - TODO：MemAddrCalc 单元中的 BreakPoint 模块是干嘛的
    - TODO：查看浮点 pipeline 相关的代码
- write back 阶段与 ROB
    - 结果写回到 iregfile，清除该指令对应的 rob_bsy 位，表示该指令可以提交。
    - rob.io.commit 的一些关键传递对象 --> rename，ifu，
        - 传递给 rename stage 中的 freelist，释放 stale_pdst（与 mips R10000 的释放机制一致）
    - TODO：`rob_head_lsb` 的计算是不是有问题，根据当前的 `rob_head_vals` 来确定 `rob_head_lsb` 会导致它的值相比与真实值后延了一个周期？例如输出的 `rob_head_idx` 用于判断指令间的先后顺序，这是否会出问题呢
    - TODO：为什么 rob 的存储使用 reg 而不是 mem 呢
    - **直观来看，ROB 与 issue queue 的存储有许多重叠，那能不能直接让 ROB 充当 issue queue 呢？是可以这样做，但存在的弊端是没有区分整数队列和浮点队列，意味着 wakeup 逻辑会更复杂。另外 ROB 的 entry 通常会比 issue queue 多很多（因为 issue queue 只需要保存待发射的指令，而 ROB 需要保存所有的 in-flight 指令），这也会使得 wakeup 需要更多的逻辑（对 bypass 倒没有影响，因为 bypass 仅发生在读寄存器的阶段）**
    - **data in ROB(implicit renaming) vs data in physical register file(explicit renaming)：后者的好处的逻辑会更简单，数据不需要搬来搬去，而且只要在发射时读寄存器就好了，不需要把寄存器的值先自己单独存起来。因为对于前者来说，指令完成时结果是写入到 ROB 中，等指令提交时再将结果写入到逻辑寄存器堆中。由于这里会被搬一下，导致在发射队列中的指令需要提前把数据读过来，不然有可能这个数据被写到逻辑寄存器堆中了，然后该 ROB 的表项被新的指令占用。但 explicit renaming 处理异常会很慢，因为需要回滚 ROB，通常每个周期回滚一条指令，将重命名表还原到异常发生指令处的样子（本质原因在于说，重命名表影响了逻辑寄存器的值）。但 data in ROB 处理异常就很简单，只需要把重命名表的每个表项都指向对应的逻辑寄存器就行了**

Branch Design

`brupdate` 将会被送往 ifu，exe unit，fp_pipeline，branch mask generator，rename stage，issue units，register read，lsu，rob

- boom 的配置参数 `maxBrCount` 表示 `br_tag` 的数目。每条分支指令（**除了 jal 指令，TODO：为什么？我猜是前端直接会计算该指令的地址，不需要预测**）会有一个 `br_tag` 域，每条指令会有 `br_mask` 域。`br_mask` 是 `maxBrCount` 位宽的信号。 `br_mask` 中某一位为真，表示该指令依赖于对应 `br_tag` 的分支指令，如果那条分支指令预测错误了，那么 `br_mask` 对应位为真的指令

需要被 flush 掉

- Branch Generation：branch mask 由 `BranchMaskGenerationLogic` 生成，该模块核心的结构是 `maxBrCount` 个 bits 的存储 `branch_mask`。如果它的某一位为 1，表示该 `br_tag` 已经被分配出去了。在 decode 阶段，该模块为每条指令产生相应的 `br_tag` 和 `br_mask`。每条指令的 `br_mask` 组成有两部分，一部分来自于 `branch_mask`，每条指令当然依赖于已经分配出去的 `br_tag`，但这 coreWidth 条指令内部可能有分支指令，因此位于这些分支指令后面的指令的 `br_mask` 中会加上它们刚分配的 `br_tag`（注意分支指令自己的 `br_mask` 不会包含自己的 `br_tag`，这是当然的，分支指令的 mispredict 不应该 kill 掉它自己，分支指令本身还是合法的）
- `brupdate` 广播了已经计算出结果的分支指令。在分支指令进入 exe unit 的同一周期，exe unit 就会输出 brinfos，然后会进入一级寄存器缓存，下一周期各个模块就会得到 `brupdate.BrUpdateMasks`，同时一些组合逻辑用来确认第一条 mispredict 的 分支指令（如果有的话），这里会稍微麻烦一点，因为每个 alu 都可能输出一条分支指令的结果，我们需要用指令的 `rob_idx` 来确认它们的先后顺序。第一条 mispredict 的分支指令的信息经过流水线寄存后输出到 `branch.BrResolutionInfo` 中（因此比 `BrUpdateMasks` 晚一个周期）
- 我们要保证不会每条指令不会错过对应的 `brupdate` 信息（即当计算出一条分支指令的结果时，我们需要清除所有指令的 `br_mask`）。唯一的冒险是读 `branch_mask` 的同时会有 `brupdate` 来清除 `branch_mask`。容易知道这里是必须要前递的，而 `BranchMaskGenerationLogic` 中已经前递过了。然后 brupdate 会传递给从 decode 开始的每个阶段，包括 rename，issue units，lsu，register read，exe unit，rob 这些。这就要求，每个阶段的寄存器，传递微指令给下一个阶段时或者因为 stall 仍停留在同一寄存器中时，必须有相应的组合逻辑来根据 `brupdate` 清除微指令 `br_mask` 的相应位。这就是为什么会看到铺天盖地 `GetNewUopAndBrMask`，`GetNewBrMask` 的使用。**另外，我认为`BranchMaskGenerationLogic` 中使用 `GetNewBrMask` 进行前递是多余的，因为 rename stage 中也同样处理了一次**
- 接下来我们考虑分支指令 mispredict 对整体结构的影响。
  - 同一周期多条分支指令 mispredict 时，我们只需要选择最早的那一条，然后忽略其它的就行了（因为在最早 mispredict 的指令后面的指令都应该被 kill 掉）
  - 首先是 `BranchMaskGenerationLogic` 模块中 `branch_mask` 的更新。因为 mispredict 可能 kill 掉后面的分支指令，我们需要释放掉这些分支指令占用的 `br_tag`（通常的释放方式是分支指令在 alu 中出了结果之后通过 `brupdate.resolve_mask` 更新）。那么怎么知道哪些分支指令被 kill 了呢？这里 boom 做得比较取巧，由于 `brupdate.b2` 中记录了上个周期 mispredict 的指令，它的 `br_mask` 则表示当前正在使用的 `br_tag`，我们只需要让它的 `br_mask` 和 `branch_mask` 做个与运算就行了
  - 然后是 rename stage 中的 `maptable` 和 `freelist` 模块。因为分支指令 mispredict 后，我们需要还原 `maptable` 中的映射并释放 `freelist` 中分配出去的寄存器。如何做到这一点呢？`maptable` 中的做法是保存 snapshot。即每条分支指令发射前，我们用一个 snapshot 记录恰好该分支指令修改 `maptable` 映射后的（例如 jalr 指令会修改映射）`maptable` 状态。因此我们需要 `maxBrCount` 个 `maptable` 存储。当分支预测错误时，使用该分支指令的 `br_tag` 对应的 snapshot 替换掉原来的 `maptable`。`freelist` 采取了类似的处理，不过它不是记录了分支指令发射时的 `freelist` 的 snapshot，而是每个周期更新 `br_alloc_lists` 寄存器，记录从该分支指令发射出去后分配出去的物理寄存器，然后 mispredict 时释放掉这些物理寄存器即可。当然代价是类似的，`br_alloc_lists` 是 `maxBrCount` 个 `freelist`。
  - **有意思的是，为什么 `freelist` 不采取 snapshot 这种方法呢？在 boom 文档中讨论了这个问题。如果采用 snapshot 方式，当发生 mispredict 时，直观上会认为 `freelist := freelist | snapshot`。这个表达式就是说，新的空闲物理寄存器由两部分构成，一部分来自 snapshot 发生时空闲的物理寄存器，另一部分来自从 snapshot 到 mispredict 这段时**

间里由于指令提交被释放的物理寄存器。但这里有一个隐蔽的漏洞，如果一个物理寄存器在 snapshot 到 mispredict 这段时间中被释放了，然后又被分配出去了，那么得到的新的 `freelist` 会认为该物理寄存器仍然处于被分配状态，但由于 mispredict ，使用该物理寄存器的指令被 kill 了，所以实际上该物理寄存器应该被释放才对，这会导致物理寄存器的泄露

- 另外值得一提的是，当 mispredict 时，完全没有对 busytable 进行一些操作。虽然误执行的指令可能对 busytable 中某些物理寄存器的置了 busy 位，但当 mispredict 发生时，这些物理寄存器必然被释放了，因此后续指令必然不会读这些物理寄存器对应的 busy 位（换句话说，空闲寄存器在 busytable 中的 busy 位是无关紧要的）。
- TODO：为什么 mispredict 后的两个周期都需要暂停 dec 和 rename 周期，如何 kill 掉这里的指令。一些暂时的猜测：暂停很好理解，因为此时处于这些周期的指令都需要 kill 掉，当然也可以选择增加额外的 `IsKilledByBranch` 判断，都差不多吧。具体的 kill 逻辑是，当 mispredict 时会拉高 `io.ifu.redirect_flush` 信号，这个信号会将 rename 阶段的寄存器 kill 掉
- 然后是 issue slot，register read，exe unit，rob 中每个周期使用 `IsKilledByBranch`，来清除掉依赖于 mispredict 分支的指令。这里在重申一下在 exe unit 阶段我没搞明白的问题，为什么用作 bypass 的 `r_val` 信号不会因为 mispredict 而被清除掉，我唯一能想到的理由是减少判断 `IsKilledByBranch` 等的时序损失？
- rob 中除了调用 `IsKilledByBranch` 清除掉依赖于 mispredict 分支的 rob 项，还需要调整 rob tail
- TODO：对 LSU 的影响
  - 每条 `MicroOp` 都有自己的 `dis_ldq_idx` 和 `dis_stq_idx`，表明自己在 ldq 和 stq 中的位置。这俩 idx 只有在碰到相应的访存指令的时候才涨。当发生 mispredict 时（注意这是在 mispredict 的第二个周期完成的，因为需要用到 `b2` 字段），`ldq_tail` 和 `stq_tail` 重置为该分支指令的 `dis_ldq_idx`，`dis_stq_idx`
  - 访问每个 ldq 和 stq 的 entry，如果该 entry 应该被 mispredict kill 掉，那么 invalid 该 entry
- TODO：对 frontend 的影响

LSU

- load / store 指令在进入 lsu 前的一些情况：整点 load 指令比较简单，发往 mem_iss_unit，最后在 `MemAddrCalc` 中计算出计算出地址，送往 lsu。boom 则分离了整点 store 指令，它也被发往 mem_iss_unit，但其中做了特殊处理（在 issue slot 中处于 `s_valid_2` 状态），除非地址和需要存储的数据同时来到，否则 store 会被分为两条指令进入 `MemAddrCalc`，最终可能有两条指令进入 lsu，一条携带数据，一条携带地址（`uop.ctrl.is_sta` 和 `uop.ctrl.is_std` 表征进入 lsu 的指令携带了什么）。另外是浮点 load / store 指令。浮点的 load 和 整点的情况一致。但浮点 store 会稍微麻烦一点，因为它需要的数据来源于浮点寄存器，但 mem_iss_unit 中的指令会被 issue 到 `iregister_read` 中。因此在 dispatch 时，浮点 store 就会被同时分发到 fp_pipeline 和 mem_iss_unit 中。然后这两条指令被发往 lsu
- lsu 的整体结构来说：
  - ldq：由 `LDQEntry` 组成的队列，其中 `youngest_stq_idx` 字段表明该 ldq entry 进队列时 `st_enq_idx` 的值。 `st_dep_mask` 则表明了该 ldq entry 依赖的 st entry（即程序顺序在该 load 前的 store）
  - stq

- ld 指令和 st 指令的提交：当 stq 中的 st 指令获得物理地址和数据时，即可通知 rob 这条 st 指令完成了（对应 clr_bsy 相关信号）。当 rob 提交该 st 指令后，lsu 会设置 stq 中对应的 `STQEntry` 的 `committed` 字段为 true。此时该 st 指令可以发往 DCache。而 ld 指令的提交则做得更加激进，如果一条 ld 指令的地址翻译完成，那么它就可以向 DCache 发起访存请求，或者扫描 stq 尝试 forward st data，即使此时还有该 ld 指令依赖的 st 指令的地址未确定。一旦 ld 指令获得数据（来自 DCache 或者 st data），它就会告知 rob 该 ld 指令执行完成（数据会被写入到物理寄存器堆，并且依赖于该 ld 的指令可以发射），当 ld 指令到达 rob head，就会被提交
- 可以把 LSU 内做的事情分解为如下几部分
  - 接收 `dis_uop`，压入 ldq 或者 stq
  - 接收 `mem_units` 传来的 data 或者 addr，并为这些地址确定的指令发送 TLB 请求，尝试地址翻译，这对应代码中的 load_incoming，stad_incoming，sta_incoming，std_incoming。对于 load_incoming 类的指令，如果 TLB 命中了，那么会在同一周期发送 DCache 请求
  - 每个周期 LSU 会选择一个 ldq 和 stq 中还未获得物理地址的表项（即此前的 TLB 请求未命中），尝试为其发送 TLB 请求，这对应代码中的 load_retry 和 sta_retry，这两个请求的编号都固定为 `memWidth-1`，因此 load_retry 和 sta_retry 只有一个会发生
    - 现在假设可以从 ldq 和 stq 中选一个出来，应该选择哪个呢？最直观的做法当然是选最老的那条指令，boom 中为了加速 load 指令，ldq 中的指令优先级会更高。再然后，如果这个周期选过 ld 指令 A 了，那么至少要等若干个周期后才能重新选 A（我想这里的考虑是因为 L1TLB miss 时，访问 ptw 需要若干个 cycle 得到结果，所以我们可以在此期间试试其它的，但我很怀疑这是否有用，其它的请求也基本都是 miss，而 ptw 的带宽只有 1，因此这也不会产生新的 ptw 请求）。但对于 st 指令没有这样的限制，我猜是因为 retry st 的优先级最低？
    - ptw 带宽只有 1 也解释了为什么只从 ldq 和 stq 中选一个指令出来
    - 对于 load_retry，如果 L1TLB 命中了，会在同一周期发送 DCache 请求 （与 load_incoming 类似）
  - 另外，每个周期还要选择 load_wakeup 和 store_commit，前者从 ldq 中选择一条具有物理地址的指令，请求编号为 `memWidth-1`，后者从 stq 中选择一条 `committed` 字段为 true 的 st 指令，请求编号为 0。由于编号不同，这两条指令的访存请求可以同时发生
  - 现在我们每个周期的指令类有 load_incoming，stad_incoming ... load_wakeup 等等，这些指令类需要占据一定的端口（这里有四类端口资源：L1TLB，DCache，LCAM，ROB），例如 load_incoming 和 stad_incoming 都会占据 L1TLB 的端口。因此代码中存在一个优先级排序，例如当本周期有编号为 0 的 load_incoming 时，就不会再选择 load_wakeup 类指令，因为它们需要同一个 DCache 端口
  - 我认为这个 LCAM 端口代表一种 search 资源。当 `do_ld_search(w)` 为 true 时，表明该编号的 ld 指令执行了一种需要 search 的操作。类似地，如果 `do_st_search(w)` 为 true 时，表明该编号的 st 指令执行了一种需要 search 的操作。前面提到，boom 中 ld 指令的实现十分激进。这里 search 就是遍历 ldq 和 stq 中的表项，它的作用有两个
    - 对于已经确定物理地址的 ld 指令，搜索 stq 队列，寻找是否可以前递 st data
    - 遍历 ldq 和 stq，看是否 ld 指令的执行违反了程序语义（例如 ld 先于 st 指令执行，且 ld ）
    - 具体来讲，当一条 st 指令的物理地址确定后，就需要执行 search 操作（`do_st_search`，检测 ld-st 乱序）：遍历 ldq 中的每个 entry，检查是否有 ldq entry 与该 st 指令地址重叠，且该 entry 依赖于该 st 指令，并且该 ldq entry 已经执行了（通过访存或者 st data 前递），那么就说明 `order_fail`，引发 `MINI_EXCEPTION_MEM_ORDERING` 异常（当该 ld 指令提交时），通过回滚来重新执行

该 ld 指令和之后的指令。因为此时错误的 ld 值可能已经传播到了许多地方，不知道除了触发异常外还有没有什么更高性能的修复手段
        - 当一条 ld 指令的地址确定后，或者一条 ld 指令会进行访存时（load_wakeup 为 true），就需要执行 search 操作（`do_ld_search`）：这个 search 操作有检测 ld-ld 乱序和 st data forward 两个目的，先说 ld-ld 乱序。RISC-V 内存模型规定，如果两个 ld 指令地址重叠，那么这两条 ld 指令的执行不能够交换顺序。因此这里的 search 操作会遍历 ldq 中的指令，与 trigger `do_ld_search` 的 ld 指令进行对比，如果发现地址重叠的 ld-ld 乱序，且 ld 指令被 `observe` 了（从 cache 一致性的角度讲，一个 block 的读权限被收回了，如果 ld 指令没有被 `obersve`，那么这个乱序当然无关紧要，因为其它处理器不会观测到），需要触发 `MINI_EXCEPTION_MEM_ORDERING` 异常。然后是 st data forward，search 操作会遍历 stq 中的 entry，寻找是否 st data 能够前递给 trigger `do_ld_search` 的 ld 指令，如果有，前递最新的那个 st data
- 这是一些无关紧要的 TODO：
    - TODO：speculative store data forwarding 会导致 FastPNR 出问题？因为即使 load 指令通过 speculative store data forwarding 写回数据后，依然可能引发异常
    - 如果 retry st 地址翻译和 st 指令的数据同时发生了，那么 retry st 会被 delay 一个 cycle，为啥？
- TODO：查看 DCache 是否会存在乱序的情况，LSU 与 DCache 交互的 `ordered` 和 `force_order` 等信号是啥意思
- TODO：查看 DCache 和 LSU 中的 nack 逻辑
- TODO：为什么提交 sfence 时要 flush pipeline？
- 接上一条，在 lsu 中，sfence 会盖过其它所有的 exe_req，就算 sfence 会 flush pipeline，但这些 exe_req 中有排在它前面的 load / store 呢？另外，这里也没有考虑多个 sfence 之间的顺序？

```
val exe_req = WireInit(VecInit(io.core.exe.map(_.req)))
// Sfence goes through all pipes
for (i <- 0 until memWidth) {
  when (io.core.exe(i).req.bits.sfence.valid) {
    exe_req := VecInit(Seq.fill(memWidth) { io.core.exe(i).req })
  }
}
```

- TODO：支持原子指令（尤其关注 aq，rl 语义的实现），fence，sfence 对整个 lsu 结构的影响
- TODO：支持 hellacache 对整个 lsu 结构的影响
- **TODO：memory model 对 ld，st 顺序的影响，最关键的问题是搞明白 memory model 对存储设备的影响，设想如果 memory model 不支持 st 交换顺序，这是否意味着内存设备也需要按序处理收到的 st 请求？**
    - **应该如何理解全局序，例如，处理器顺序发射了 st1，st2，但是存储设备是先处理了 st2，再处理了 st1**
        - **在这种情况下是否存在全局序？因为处理器认为序是 st1 -> st2，但存储设备认为序是 st2 -> st1。我目前的观点是认为这种情况下是有全局序的，且全局序是 st2 -> st1。即序约束和处理器本身的发射顺序无关。因为我们终究关心的问题是 load 指令获得的值，因此处理器发射存储指令的顺序并不对全局序构成约束，但处理器执行 load 指令观测到的结果会对全局序构成约束**
- TODO：重新看下 mips r10000 的 address queue 章节

TLB

- RISC-V 的页表表项包含 A 和 D 两个 bit，前者该表项是否被访问过，后者表明该页表对应的地址是否被写过。这在 OS 的页表替换中是极其重要的。RISC-V 规定了两种可能的设置 A，D 的方案
  - **一种是由软件实现，如果表项没有设置 A bit，那么一旦访问该页表则触发异常。或者表项没有设置 D bit，一旦写该页表的内存也触发异常。在软件的异常处理中来设置相应的 A，D bit**
  - **另一种是由硬件实现，每次内存访问时由硬件来设置 A，D bit。这样的实现显然效率更高，但是硬件会更复杂。复杂的原因在于设置 A，D bit 是需要写内存的。这也涉及到内存序的问题，要求在全局序中，设置 A，D bit 的写内存操作一定在实际的内存操作之前**
  - BOOM 以及 cva6 都采取的是软件实现的方式
- 在 lsu 中包含 L1TLB，与 L1TLB 大小相关的 3 个配置参数 `nSets`，`nWays`，`nSectors`。L1TLB 是完全的全相连存储，总共的表项数目为 `nSets x nWays`，`nSectors` 参数决定了多少个相邻的 entry 作为一个共同的 sector，一个 sector 内部的表项只能存储相邻的虚拟地址的 PTE（cache 中也可以做类似的事情）。这样做的好处是可以节省 tag，因为一个 sector 只需要一个 tag。坏处是替换时是以 sector 为单位的，需要同时 invalid 一个 sector 中所有表项
  - L1TLB 的访问宽度是 `memWidth`。为了支持多端口访问， L1TLB 的表项是由寄存器构成的，但 ptw 的访问宽度貌似只有 1。如果同时有多个 L1TLB 的访问 miss 了，那么只有编号最大的 TLB 访问被转发到 ptw 中（因为 LSU 中 retry 的 TLB 请求一定是 `memWidth-1` 编号，这保证了 retry 类型的请求的优先级是最高的）
- TODO：没看懂如果在 ptw 中触发了 pmp 异常后如何正确传递到 L1TLB 这边
- TODO：ptw 中的 L2TLB
- TODO：理解 `EntryData` 中的 `pr` 等字段的含义

Frontend

- TODO：重新看下 mips r10000 的 instruction fetch 等章节
  - TODO：了解了 boom 中 speculative load wakeup 相关代码后重新看下 mips r10000 的 instruction queue 章节，它也提到了 speculative load wakeup

DCache and ICache

Exception, Interrupt, and Pipeline Flush

TODO

- 结合文档 rename-stage 理解 boom 是如何处理分支预测错误的问题的（除了需要 flush 已经发射的指令，还需要考虑 rename table，freelist 如何重置等等）。
- 当 corewidth 中前面指令可以发射，后面指令不能发射时如何处理？（我猜测 decode, rename 这些阶段应该是按序的，也就是说不可能出现前面的指令 `dec_fire` 还没被拉高，后面的指令就被拉高了）
- commit.rollback 是干嘛的，为什么可能需要回滚？
- 为什么 freelist 中要保留 0 寄存器（保留x0可以理解，为啥要保留 f0），即 reset 时默认 0 寄存器已经被分配了

- 结合文档 The Reorder Buffer (ROB) and the Dispatch Stage 理解 boom 是如何确定 rob 中的表项的 pc 值的，疑惑的地方在于每一行为什么只需要一个 pc，如果一行中有压缩指令如何计算表项的 pc 值？
- uop 中的 is_sfb 位用于 short-forwards branch，理解这个的具体实现
  - 参考这个 SonicBoom SFB（short-forwards branch）源码分析
- fetch target queue（FTQ）是干嘛的
- speculative load 是什么，相关的 iw_p1_poisoned, iw_p2_poisoned 是干嘛的
- 在 `issue-unit-unordered` 中阻止了 fence 类指令发送过来，为什么？fence 指令会被发到哪呢？
- `core` 中 对 `exe_unit.supportedFuncUnits.muld` 进行判断，然后当发射 FU_DIV 类指令时 supress 下个周期的 FU_DIV 的指令（拉低对应 fu_type），为啥要这么做？
- 大部分指令都只占用一个 issue queue，然后占用一个 function unit，但是 fsw, fsd 会被发往两个 issue queue，并占用两个 function unit，具体是怎么处理的？
- bypass 网络，梳理清楚哪些地方有 bypass，bypass 了什么，如何与 rename 网络对接
  - brupdate 的传播？感觉指令每个周期都会考虑这个
  - registerfile 中的 bypassableArray 参数是干嘛的 --> 就是表示哪些 writeport 可以在同一周期被 bypass 到 readport 中，使其被读到
  - ~~execution unit 中的 numBypassStages 是啥意思~~ --> 就是你在内部的流水线中可能有多个周期，每个周期都有 bypass 的数据传出去。为什么 numTotalBypassPorts 和 bypassable_write_port_mask 不一致
  - register read 中的 bypass 数据从哪传过来的？--> 从 execution unit 中的 alu 来的（仅有 alu 的数据做 bypass）
- 为什么 fregfile 里寄存器的宽度是 flen + 1
- TODO：为什么 store 指令不直接发给 lsu，而是先经过 aluexeunit，但 load 指令是直接发给 lsu？
- 为什么 `slow_wakeup.valid` 拉高要求 uop 不是 bypassable 的？
- TODO：考察不同类型指令的执行情况：csr 指令？fp 指令？load / store 指令？分支指令？
- 哪些指令的 flush_on_commit 为真，为什么？
- **复杂的信号可压缩，这么多多余的信号（例如满天飞的 MicroOp）都能够被优化掉吗**

Ideas

- 什么东西依赖于指令顺序：提交的时候（ROB），指令依赖分析 --> 这又依赖于译码（导致这两个阶段都得顺序完成）