Ariane 代码说明文档

阿航

Ver 2.0

Revision History

		……..
Ver 1.0	2018.09.06	初始版本……..
Ver 2.0	2018.11.23	第 2 遍梳理，完成了流水线前 4 级

1 总体功能概述

ariane 是一个六级流水线单发射 CPU，实现了 RISCV 官方用户指令集 V2.1 版本规定的 64 位的 RISCV 基本指令集（RV64I）和 I/M/C 扩展指令集（整数乘除法和压缩指令），以及特权指令集 1.10 版本规定的 M/S/U（用户级、管理员级、机器级）。这些实现便于支持 unix-like 操作系统。

以下内容是对流水线结构的解读。

在目前的的 ariane 核中，已经集成了 icache 和 dcache。

首先看前端 Frontend，包括 pcgen 和 IF。

pcgen 用一个多路选择器从 7 个源中选 1 个作为 npc，默认是 pc+4，还有分支预测的 npc，这 2 个选择源来自 IF 级。有 3 个选择源来自“CSR Write”模块，标注 mtvec 的源指的是 mtvec 寄存器中的自陷跳转地址，标注 epc 的源指的是 mepc 寄存器中的自陷返回地址，另一个标注 epc 的源应该指的是由于写 CSR 的副作用引发 flush 流水线后的 pc 值（这个地址来自 commit 级），最后一个选择源是发生分支预测出错后，ex 级的 mispredict 信号触发 commit 级的 exception 模块递送的地址。

得到的 npc 发给 icache 取指（如果使能地址翻译，则先经过 MMU 中的 ITLB，这是一个组合逻辑），经过一个时钟周期，进入 IF 级（暂不考虑各种 miss 情况）。把得到的指令给 Instr Scan 模块“预译码”，如果识别到跳转指令，则提前计算出跳转地址（可能用到预译码出的立即数 imm），并参考 BHT 和 BTB 两个预测模块的结果（这个结果在前一周期 pcgen 阶段已经得到），给出最终预测结果。这个预测结果和指令一起放入 fetch_fifo 模块中打包，等待发给下一级。取指可能发生异常，这个异常由 MMU 发出，读写相关 CSR 寄存器，并进行处理。

ID 级中，由于 ariane 核实现了压缩指令，因此，从 fetch_fifo 中读取的指令首先进入 Realigner 模块处理由于压缩指令导致非对齐指令的情况，然后给 Compressed Decoder 模块把压缩指令翻译为正常指令（16 位->32 位），最后送入 Decoder 模块译码。这 3 个模块中，只有 Realigner 模块包含时序逻辑，且代码只包含一个时序语句块。译码过程也会出现异常。

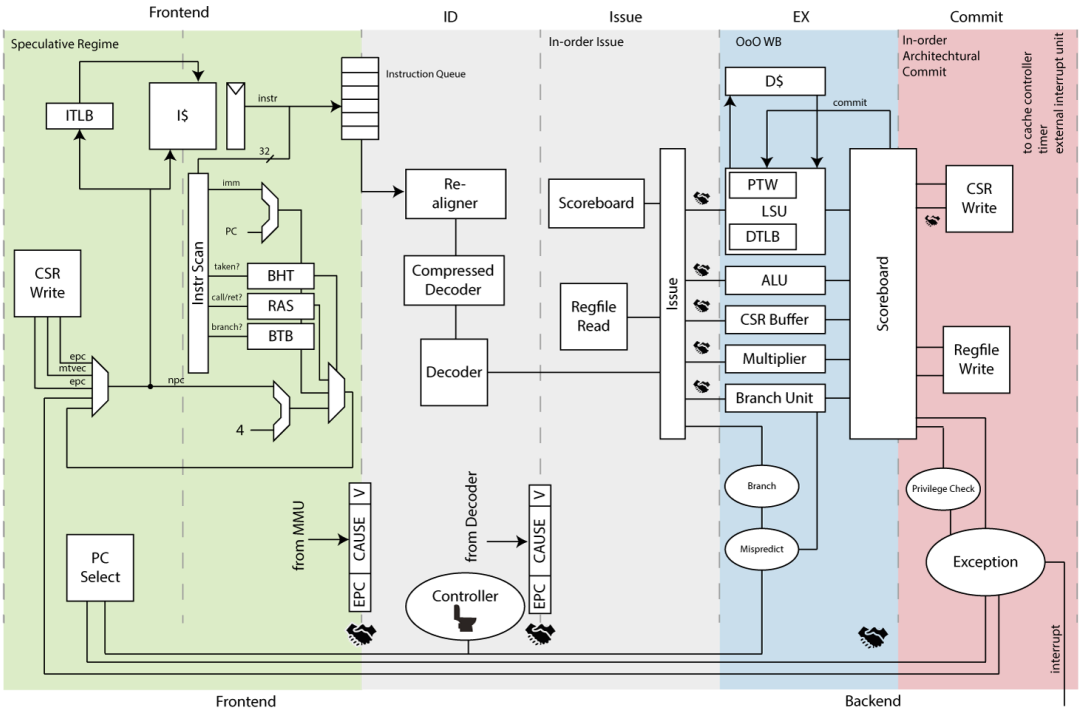
issue 级中完成对 32 个通用寄存器的读写。其中读寄存器无条件执行。issue 级的主要含义是根据冒险情况决定一个指令是否可以发给 ex 级执行。是否有冒险由 ScoreBoard 判断并处理。为了做到这一点，ScoreBoard 不仅要跟踪 issue 的指令，还要跟踪 ex 级指令的计算结果。

issue 级与 ex 级的各个功能单元单独交互。其中，数据流比较简单的逻辑计算指令，由 ALU 和 Multiplier 功能单元完成，所需操作数全部可以由 issue 级直接得到，两个功能单元计算出结果后，发给 commit 级等待提交。对于分支指令，ex 级已经可以得到是否跳转的确定结果，如果预测出错，交回 Frontend 修改（通过 controller 模块 flush 相关指令）。对于 CSR 指令，它要去 csr_regfile 模块读写数据，而 CSR buffer 只是一个暂存数据的地方，并且是否被允许写入（提交）CSR 寄存器，也得等待 commit 级的信号。对于 load 和 store 指令，由 LSU 模块完成。LSU 模块中，有 MMU 和 TLB 来运行与 cache 的交互过程，store 指令的提交也由 commit 级控制。

commit 级只控制指令是否能提交，而提交过程即写回过程还是由相关模块完成（比如，csr_regfile 模块，LSU 模块）。所有指令的提交请求是由 issue 级的 ScoreBoard 发给 commit 级的，这样能够保证 issue 指令和 commit 指令都是相同顺序。

commit 级的另一个功能是处理异常，如果异常的处理需要 flush 流水线中的一些模块，就由 controller 模块具体执行。

2 流水线结构



来自 https://pulp-platform.github.io/ariane/docs/home/

注：以下 2.1 到 2.6 内容，几乎是 ariane 文档的原文翻译。我自己的补充以【】标识，个别不明含义的语句以英文红字加粗或者中文红字加粗标识，个别重点词句以**加粗**标识。

2.1 frontend（pcgan）

pcgen 级负责产生下一个 pc 值。所有 pc 值都是逻辑地址（虚拟地址）。如果逻辑到物理映射更改，**fence.vm** 指令应该刷新流水线和 TLB。

pcgen 级包括了对是否发生分支以及分支地址的预测。另外，它包括一个 branch target buffer (BTB)和一个 branch history table (BHT)。

如果 BTB 确定一个 pc 值是指向跳转指令的话，就由 BHT 来决定是否跳转 **Because of the various state-full memory components this stage is split into two pipeline stages.** pcgen 级和 IF 级通过握手信号交互，即 ready 和 valid 信号。

下一个 pc 值（**npc**）从如下来源中选择（按优先顺序列出）：

1.默认：pc+4。即，总是以 word 边界对齐（32 bit），压缩指令会在流水线中处理。

2.分支预测：即来自 BHT 和 BTB 的预测结果。如果将分支预测地址作为下一个 pc 值，则还应该通知 IF 级当前执行了分支预测，这对于后续流水线中的分支预测执行过程非常有必要（比如纠正错误的预测）。分支预测的信息以一个 branchpredict_sbe_t 结构体的形式向后传递，注意与 branchpredict_t 区分，branchpredict_t 传递的是来自 ex 级的分支预测是否正确的信息。这种命名方式可以方便的检测不同的分支信息流。

3.控制流请求：这在预测错误（mis-prediction）的情况下会发生。此时，就需要将 pc 值纠正回正确的地址。

4.返回调用：环境调用的返回地址，npc 将被设置为[m|s]epc 寄存器中的值。

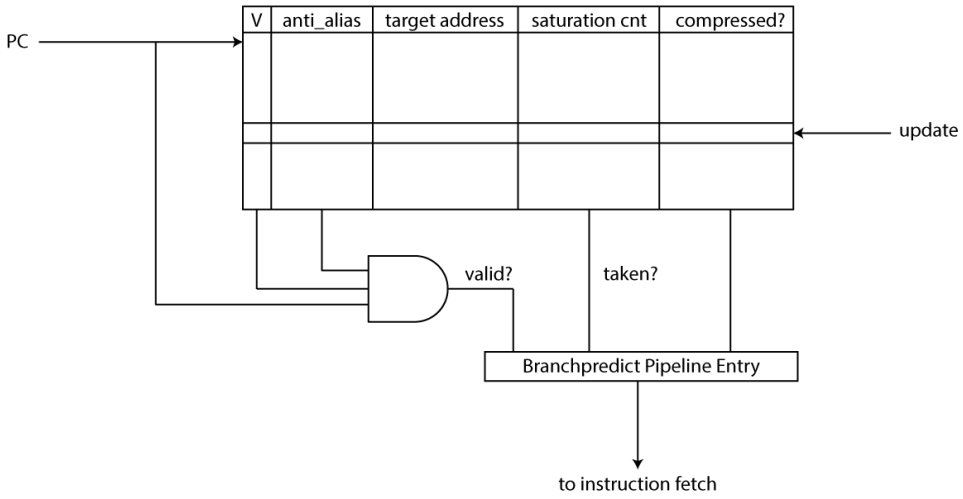
5.异常和中断：异常或中断（在 RISCV 中这两者基本相同）发生时，下一个 pc 值应该

指向自陷向量基址（trap vector base address）。它的具体值取决于是 S-Mode 还是 M-Mode（目前还不支持 U-mode）。CSR 单元的目的是找出自陷的位置并向 pcgen 提供正确的地址。

6.流水线 flush：当带有副作用的 CSR 寄存器发生写操作时【一条指令的副作用指的是，除了执行指令本身的内容，还有别的操作，比如更新 pc 值，更新 minstret 寄存器】，我们需要 flush 整个流水线，然后再次从下一个指令开始取指。这是为了考虑更新后的信息，比如虚拟内存基地址的变化。

7.debug：Debug 具有最高优先级，因为它可以中断任何控制流请求。**It also the only source of control flow change which can actually happen simultaneously to any other of the forced control flow changes.**

pcgen 级还要关注一个信号 fetch_enable，它使能 fetch 过程。另外，本模块不发生 flush 请求。所有的 flush 命令由 controller 模块下达。controller 模块的主要目的就是 flush 不同的流水级。



所有的分支预测数据结构都保留在一个类似寄存器文件的结构中，它使用 pc 的适当位数索引，并包含有关预测目标地址的信息以及一个可配置宽度的饱和计数器的计数结果（默认情况下为两位）。这个计数结果决定了分支是否跳转。发生错误预测时，分支单元（包括 BTB 和 BHT）也需要及时更新，具体更新方式是修改计数器值或者清除分支预测条目。**当发生具有改变特权级的副作用的指令被提交时，BTB 也要更新。**是否分支的预测和分支地址的预测，在发生预测错误时，代价通用昂贵。对于 timing 来说，这是 critical 的，有待改进。

BTB 中的索引目录是以 word 长度对齐的，而 ariane 实现了压缩指令（half word），这导致对于包含 2 个压缩指令的取指地址的分支预测常常会出错。然而，这种情况在实际中会很少发生。【BTB 和 BHT 是以指令地址信息即 pc 值来做预测的。同时该地址送到 icache 请求指令。取回指令后，frontend 主模块检测指令是否压缩，并调用 instr_scan 子模块做了简单译码，来进一步校正分支预测的结果，最后打包发给下一级。frontend 主模块检测指令，如果发现取到的是 1 个压缩指令和半个正常指令，这就产生了取到**非对齐指令**的情况。】**我们在这里玩的一个技巧是采用未对齐指令的下一个 PC（例如，该指令的高 16 位的字对齐 PC）来索引 BTB。**这自然允许 IF 阶段获取所有必要的指令数据。实际上，它将获取两个未

使用的字节，然后由 the instruction re-aligner 丢弃。因此，我们还需要保留一个额外的位，来指明压缩指令在高 16 位还是低 16 位。【虽然 frontend 级已经识别并重整了压缩指令，但这只是为了做分支预测。它传递给 id 级的仍然是从 icache 得到的原始数据包，没有标记是否是压缩指令。**但原始的指令数据和分支预测结果是如何对应的**】

对于分支预测，不必要的流水线气泡的潜在来源是混叠（aliasing）。为了防止混叠发生（或者至少使其不太可能），使用几个标记位（来自索引 PC 的高位）并在每次访问时进行比较。这是必要的权衡，因为我们缺乏可用于托管 BTB 的足够快的 SRAM。相反，我们被迫使用寄存器，这对整个面积和功耗的影响都要大得多。

2.2 frontend（Instruction Fetch）

IF 级得到 pcgan 级发来的关于 pc 值的所有相关信息（当前 pc 值，分支预测情况，pc 是否有效），请求 MMU 翻译地址，然后去和 icache 交互，去取指。

IF 的一个关键问题是它是 timing critical 的。这使我们无法实现更精细的握手协议。因此，IF 阶段【只能简单的】向 icache 接口发信号通知它要对内存进行获取请求。根据 cache 的状态，可以授予或不授予此请求。如果它被授权，则将请求放入内部 FIFO。它需要这样做，因为它必须在任何时间点知道有多少交易（transactions）未完成。这主要是由于分支预测而使得 IF 是猜测执行的。控制器随时可能 flush 掉 IF 级，在这种情况下，它需要丢弃所有未完成的事务。

当前实现允许最多两个未完成的事务。如果有两个以上，IF 阶段将不会确认（acknowledge）来自 pcgan 的任何新请求。一旦返回来自内存的有效答案（并且该请求不是因为 flush 而过时），答案将与提取地址和分支预测信息一起被放入 FIFO。这也是可能发**生异常的第一个地方**(bus errors, invalid accesses and instruction page faults)。

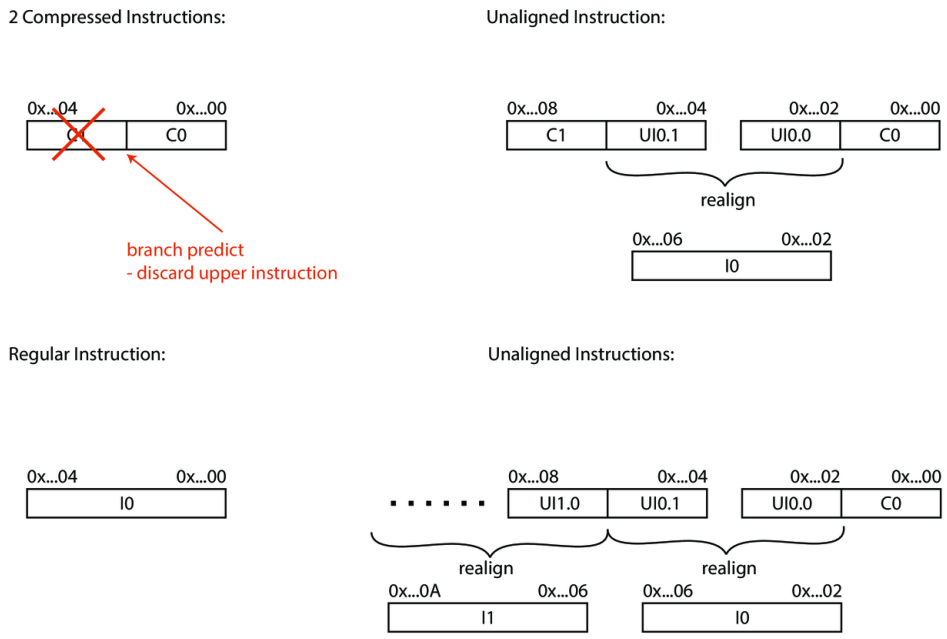
取指 FIFO 包含来自指令存储器的所有请求（有效）提取。 FIFO 当前有一个写端口和两个读端口（其中只有一个使用）。**在未来的实现中，第二个读端口可能用于实现宏操作融合或扩展问题接口以覆盖两个指令。**

取指 FIFO 还完全解耦处理器的前端及其后端。发生 flush 时，整个取指 FIFO 被重置。

2.3 id_stage

id（instruction decode）级是流水线后端（backend）的第一级，主要功能是译码。由于压缩指令的存在，id 级的硬件实现稍微复杂一些。包括 3 个主要模块：instr_realigner，compressed_decoder 和 decoder。

instr_realigner 模块首先从指令流中检索出压缩指令（压缩指令的最低位不是 11，而正常 32 位指令的最低位是 11）。压缩指令的存在使得常规指令出现对不齐的情况，导致一条完整的常规指令需要 2 次访存，需要 2 倍的指令 FIFO 空间。因此 instr_realigner 模块需要跟踪前一条指令的对齐情况，来决定下一条指令如何处理。**另一方面，分支预测的结果可能要求我们舍弃这个非对齐指令的前半部分或者后半部分。。。**如图。



来自./docs/img

compressed_decoder 用于对 16 位指令进行解压缩，把它扩展成标准的 32 位指令。由一个小的组合电路完成。

decoder 是完成译码工作的主要模块，将 32 位指令转换成各种控制信息和分立数据，这些也组成了一条指令在计分板（scoreboard）中的内容，包括如下：

PC: PC of instruction

FU: functional unit to use

OP: operation to perform in each functional unit

RS1: register source address 1

RS2: register source address 2

RD: register destination address

Result: for unfinished instructions this field also holds the immediate

Valid: is the result valid

Use I Immediate: should we use the immediate as operand b?

Use Z Immediate: use zimm as operand a

Use PC: set if we need to use the PC as operand a, PC from exception

Exception: exception has occurred

Branch predict: branch predict scoreboard data structure

Is compressed: signals a compressed instructions, we need this information at the commit stage if we want jump accordingly e.g.: +4, +2

计分板中的这些信息会在流水线的后续阶段进一步处理。

2.4 issue_stage

issue 级根据译码结果将等待被执行的指令发往各个功能单元（FU），还要接收 execute 级的回写数据。issue 级跟踪了每一条发射指令的执行状态，各个功能单元的状态，这些通

过 issue 级的一个重要模块计分板（scoreboard）来实现。它可以明确知道哪个指令被发射，哪个功能单元正在被使用，哪个寄存器被写回。如果我们把指令执行过程看做由 issue，read operands，execute，write back 四个子过程构成，那么 issue 级完成了其中的第 1/2/4 过程。

issue 模块代码中例化了寄存器堆的模块。

issue 过程。

当收到一条新的译码后的指令时，issue 级要检查执行这条指令需要的功能单元是否可用或者下一个时钟周期可用，然后检查指令的源操作数是否可用，以及当前是否有其他指令要写入相同的目标寄存器。**Furthermore it keeps track that no unresolved branch gets issued. The latter is mainly needed to simplify hardware design. By only allowing one branch we can easily back-track if we later find-out that we've mis-predicted on it.**

计分板一次只允许一条指令写某一个目的寄存器，这样可以大大简化前递路径的设计。计分板有一个组合电路，输出所有 32 个目的地寄存器的状态以及将产生结果的功能单元。这个信号叫 rd_clobber。

计分板与各个功能单元单独交互。这意味着，它要监测它们的 valid 和 ready 信号，无条件的接收和保存它们的写回数据。记分板中为每个已发射的指令分配一个槽，以确保足够的存储空间，也解决了较小微处理器的潜在结构冒险。这种模块化的设计也可以用来探索更先进的 issue 技术，比如乱序发射。

当前的设计中，指令的 issue 过程是顺序（in order）发生的，保持了程序流（program flow）的顺序。各个功能单元的写回是乱序的。比如，紧接着乘法指令的加法指令会在乘法指令之前写回数据。因此，计分板给每条发射的指令标记了 ID（transaction ID），可以唯一表示计分板中对应的槽，并且会和其他数据一起传递到下一级的功能单元。

这种方案允许功能单元的执行过程独立于 issue 逻辑。各个功能单元可以乱序写回结果。通过关联的 ID，计分板可以识别这些写回结果。这种方案甚至允许各个功能单元缓冲计算结果，乱序执行。这是如何有效地解耦处理器的不同模块的另一个例子。

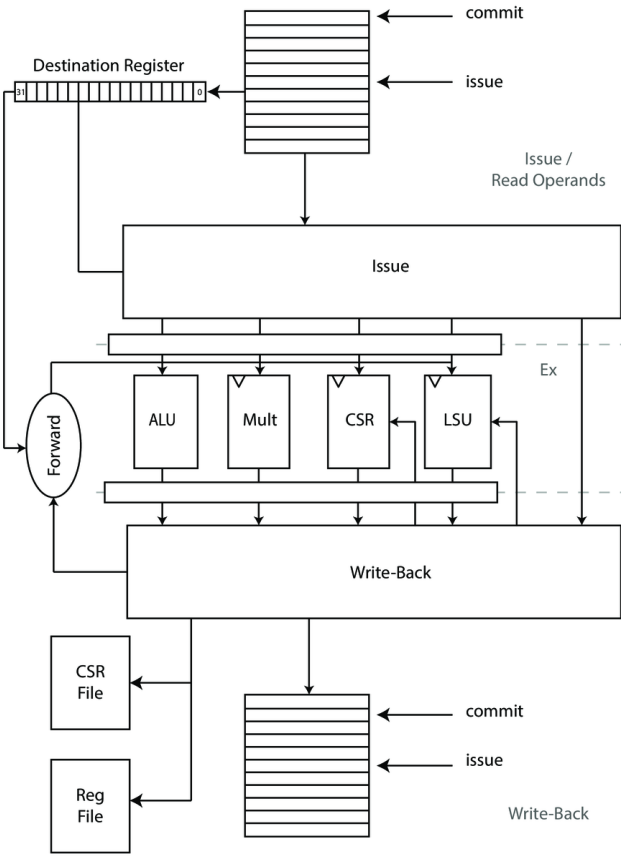
read operands 过程。

读操作数的过程实际上和 issue 是同时发生的，只是概念上可以看做另一个过程。读操作数的过程中，由于计分板清楚的知道哪个寄存器将要被写入数据，所以可以方便的通过前递来处理数据冒险。设计目标是背靠背（back to back）地执行两个 ALU 指令（例如：两者之间没有气泡）。操作数来自寄存器文件（如果记分板中当前没有其他指令将写入该寄存器）或者由记分板前递（通过查看 rd_clobber 信号）。

操作数优先从计分板而不是寄存器中选择，因为计分板中会记录功能单元得到的最新结果。要获得正确的寄存器值，我们需要轮询计分板获得两个源操作数。

计分板。

计分板是一个 FIFO，有一个读端口和一个写端口（通过 valid 和 acknowledge 信号）。计分板还会记录当前每一个寄存器被使用的情况。如果计分板未满的话，id 级的译码结果可以直接写入。commit 级根据计分板中已经完成的指令来更新系统状态（更新寄存器或者触发异常）。



来自./docs/img

2.5 ex_stage

ex 级包含了 5 个相互独立的 FU（功能单元）：ALU，branch unit，LSU，CSR buffer，乘除单元。每个 FU 维护 1 个 valid 信号和 1 个 ready 信号，valid 信号指示当前输出的数据是否有效，ready 信号指示是否准备好接受下一个计算请求。也正如 issue 逻辑中说明的，ex 级的功能单元从 issue 级接受一个唯一的事件 ID 和相关数据，计算后把有效结果和 ID 一同给出。

ALU。执行 32 位或者 64 位的加法，减法，移位，比较。可以单周期完成，因此 ready 信号始终有效，事件 ID 直接从输入连到输出。

branch unit。这个功能单元的目的是管理各种跳转情况。需要有加法器和比较器去判断分支跳转指令是否预测正确，如果错误，会及时通知 pcgan 级纠正。由于非对齐指令的存在，跳转也可能发生在非分支指令中。对于这种情况，branch unit 也能处理。

As briefly mentioned in the section about instruction re-aligning the branch unit places the PC from an unaligned 32-bit instruction on the upper 16-bit (e.g.: on a new word boundary). Moreover if an instruction is compressed it also has an influence on the reported prediction as it needs to set a bit if the prediction occurred on the lower 16 bit (e.g.: the lower compressed instruction).

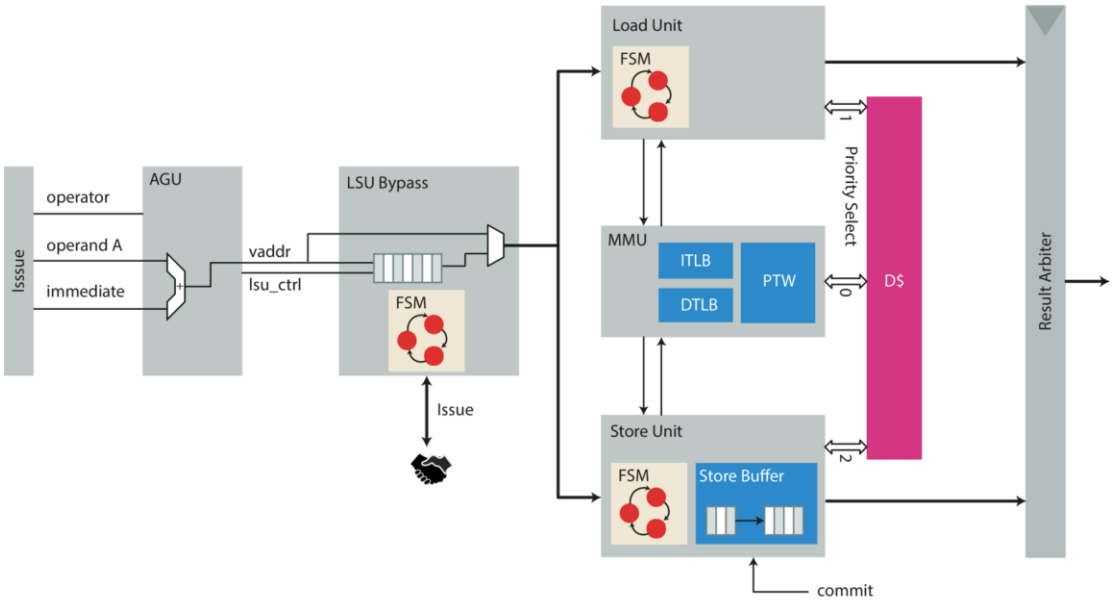
由于加法器和比较器的使用，branch unit 也成为了一个 critical path。然而，在我们的设计中，branch unit 也是单周期完成的。因为计分板只有在没有 unresolve branches 的情况下才

接受新指令，单周期完成的 branch unit 可以保证不会引起流水线暂停。

LSU。完成 load 和 store 指令，涉及与 dcache 的交互。它还要组织 DTLB (Data Translation Lookaside Buffer)，PTW（the hardware page table walker）和 MMU（and the memory management unit）。LSU 还需要仲裁 load、store、PTW 三者对 **memory** 的访问顺序（优先给 PTW 访问权限，这样可以尽快解决 TLB 缺失的问题）。

LSU 可以马上发出 load 请求，但是发出 store 请求必须先等待来自计分板的 commit 信号。因为 store 涉及对 memory 关键位置的重写。在 commit 信号到来前，store 只能先放入 buffer 中等待。但发出 load 之前也要检查这个 buffer 中 store 的情况，检查 store 缓冲区是否存在潜在的混叠。有如下 3 种情况：允许 2 个针对同一个内存地址的 load 指令；允许两个针对同一个内存地址的 store 指令；不允许对于同一个内存地址“写后读”的情况，如果出现这种情况，那么这个 load 指令得暂停，等待 buffer 中的 store 先提交。

LSU 还不能处理地址非对齐访存的情况。如果遇到未对齐 64 位的双字访存请求，未对齐 32 位的单字访存请求，未对齐 16 位的半字访存请求，LSU 会抛出一个未对齐异常或者页面错误异常。



来自./docs/img

如上图，为了减轻 LSU 的设计难度，将它分成了 6 个部分：

1. **LSU Bypass**。用来辅助管理 LSU 的状态信息，这个状态是要与 issue 级交互的。因为 LSU 要先产生地址，地址翻译，检查写后读，才能知道是否可以访存，而且访存的 SRAM 也比较慢，因此路径也是 critical 的（其中，检查写后读和 TLB 缺失是时间开销最大的两块）。因为 issue 级根据 ex 级的 ready 信号来发新指令，所以 LSU 较长的时序路径会拖慢整个设计。

为了减轻时序压力，加入了一个 FIFO，它可以保存来自 issue 级的另一个请求，功能单元的 ready 信号可以延迟一个周期，这减轻了 timing 要求。也解耦了 LSU 和 issue 级的关系。**In particular the LSU bypass is called that way because it is either bypassed or serves the unit from its internal FIFO until they signal completion to the LSU bypass module.**

2. **D\$ Arbiter**。

3. **Load Unit**。完成 load 操作。需要计算地址，并发出地址翻译请求，需要与 store

buffer 中的地址比较来检查是否存在写后读的情况。为了减轻时序压力，其中的比较器只比较低 12 位而不是全部 64 位（低 12 位正好是物理地址和虚拟地址相同的 page offset）。这样可以加快运算速度，也不必等待地址转换。

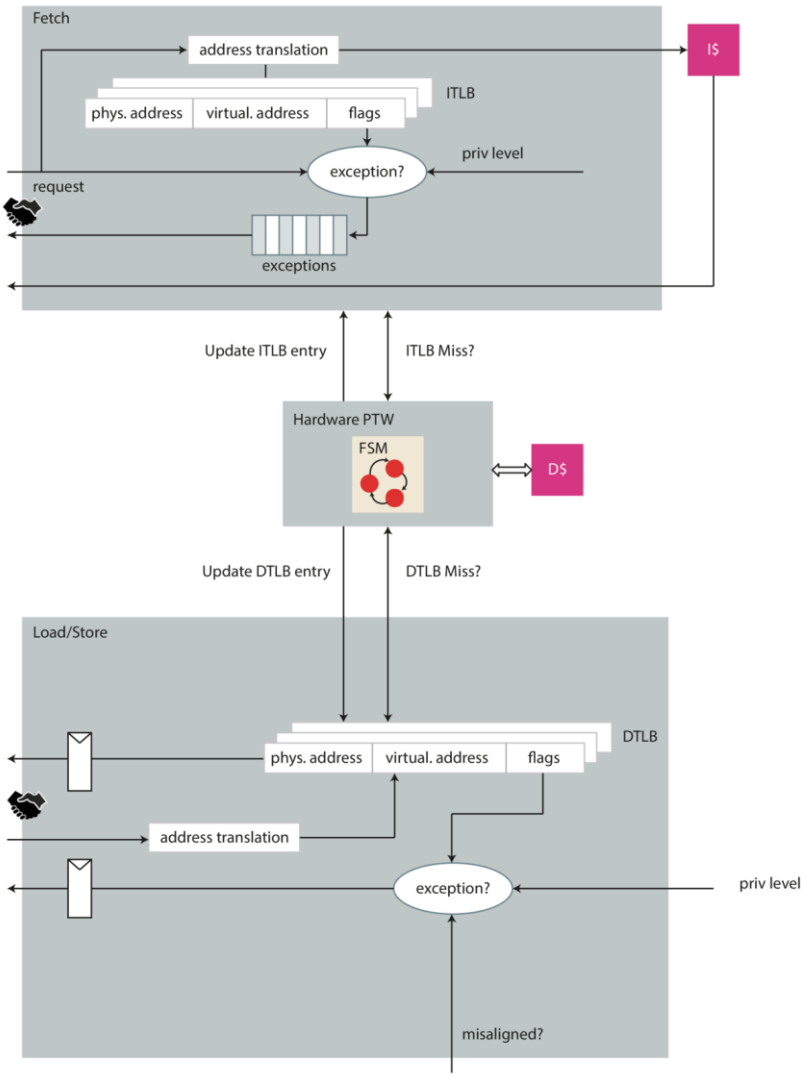
4. **Store Unit**。完成 store 操作。其中的关键结构是 **store buffer**，实际上是 2 个 buffer，一个装准备提交的 store 指令，另一个装仍然是推测状态的 store 指令。flush 信号只能清空装推测 store 指令的 buffer。推测 store 的队列的满信号控制暂停 LSU bypass 单元，不再接受新的 store 请求。提交 store 的队列的满信号发往 commit 级，表示队列已满无法接收新数据，commit 级就会暂停。commit 级发出 lsu_commit 信号，表示允许提交一个 store 指令，则该指令由推测 store 队列转入提交 store 队列。只要 cache 授权了请求，那么提交 store 队列会自动向 memory 提交请求。store buffer 只针对物理地址。当 store 处于提交状态时，它的地址转换已经正确。**For stores in the speculative queue addresses are potentially not correct but this fact will resolve if address translation data structures are updated as those instructions will also automatically flush the whole speculative buffer.**

5. **MMU (including TLBs and PTW)**。MMU 关注地址转换和访存。是否激活地址转换功能，与相关的 CSR 寄存器有关。MMU 包括一个 ITLB，一个 DTLB，和一个 HPTW（hardware page table walker）。取指阶段与 icache 的交互，执行阶段与 dcache 的交互，都要经过 MMU。但具体控制过程完全不同。

取指阶段和 icache 这里，取指请求的往往是虚拟地址，如果激活了地址转换功能，就需要通过 ITLB 转换成物理地址，然后发给 icache。ITLB 的实现是纯组合逻辑。这个 TLB 是基于触发器实现的组相联 cache。这条经过地址转换的数据通路往往也是 critical 的。如果地址转换失败，MMU 会抛出一个异常。产生的异常会和 valid 信号（而不是 grant 信号）一起送到取指阶段。为了支持多个异常的指令，MMU 还设置了专用的 buffer 去存储这些异常事件。如果没有激活地址转换功能，取指的请求就直接送给 icache。

dcache 这里，load 单元或者 store 单元也会请求地址转换，但这个过程不是纯组合逻辑。而且，由于在 ex 级，地址转换前要先由 LSU Bypass 单元计算出地址，因此这个路径肯定是 critical 的。

PTW 监听 ITLB 和 DTLB 的工作，如果地址转换出了问题，PTW 会保存这个虚拟地址然后从内存的页表中检索物理地址。PTW 是两个 TLB 共用的，但会优先处理 DTLB 的缺失。详细信息参考 RISCV 特权指令集手册。



来自./docs/img

6. ** Multiplier **

乘除单元。乘法需要 2 个时钟周期完成，而且是全流水的，除法是一个序列除法器，最坏情况下需要 64 个时钟周期。

CSR buffer。它的唯一功能是保存将要被指令读写的 CSR 寄存器的地址。原因有二：其一，CSR 指令会更改架构状态，因此应该被缓存，等待 commit 级的提交信号，尽快提交；其二，计分板中只能保存指令执行的一个结果，但 CSR 指令需要保存写入的数据和写入的寄存器地址等多个结果。CSR buffer 的缺点是，它只能保存一个 CSR 指令的结果。如果遇到连续执行 2 个 CSR 指令，只能暂停流水线。所幸，这种情况不多发生。

2.6 commit_stage

commit 级是流水线的最后一级，它的主要作用是根据指令的执行结果更新系统结构状态，具体有写回到寄存器堆，写回 CSR 寄存器，写回内存（store 指令）。除了 commit 级，流水线其他级都不允许执行这些操作。**If it keeps an internal state it must be re-settable**

commit 级还要管理多种异常情况，这些异常主要有 3 种来源：一是发生在流水线前几

级，包括 IF 级、ID 级、issue 级、ex 级（不包括 pcgan 级）；二是发生在 commit 级，commit 级的异常只可能来自 CS 寄存器；三是来自中断。

此外，commit 级还负责决定流水线是否暂停。commit 级还和 controller 模块一起执行 fence 指令（cache flushes）和其他流水线重置操作。

注：

这些说明材料来自../docs/_docs

另外还要一些辅助说明图片在../docs/img

3 frontend 接口信号与代码说明

前言：ariane 设计的一个代码风格。为了突出并简化时序语句块，为同一个变量 x 设计了两个赋值版本 x_d 和 x_q。主要赋值操作尽可能在组合逻辑赋值语句中用 x_d 完成，时序逻辑赋值语句中实现 x_q<=x_d。

frontend 模块的功能和流程主要是：

选出 npc 值。7 种来源中，一部分直接由输入信号提供，比如 pc_commit_i，epc_i，trap_vector_base_i，另一部分是直接加 4，或者指向固定的 debug 地址，或者是由 RAS 模块提供的函数返回地址。

把选出的 npc 值作为取指地址发往 icache。发送包（结构体）的信息除了 npc，还有 kill 的信息：kill_s1 和 kill_s2。如果输入的 flush 有效，或者输入信号显示之前的分支预测出错，则 kill_s1 和 kill_s2 有效；如果当前模块内做出的分支预测有效，则 kill_s2 有效（因为 icache 中的预取指没有考虑分支预测）。如果系统配置使能地址翻译的话，这里的取指地址就是虚拟地址，icache 处理后会转发给 ex 级的 mmu 子模块去寻找对应的物理地址，然后返回给 icache 寻址，最后再由 icache 把取到的指令返回给 frontend。

分支预测。（仅用 pc 值通过 BHT 和 BTB 做出的预测结果是不完整的，只能起到辅助作用，所以又加了一种对指令预译码的分支预测方式。）具体是，接受从 icache 中读到的指令信息后，交给 instr_scan 模块，判断是否是分支或跳转指令，并解码出相关的立即数。

如果是 jalr 指令：而且是 return 的功能，则使用 RAS 模块提供的返回地址；如果是一般的 jalr 指令，而且 BTB 模块有有效的预测，则使用 BTB 模块的地址。如果是 jump 指令，则用立即数计算跳转地址。如果是 branch 指令，可选择采用动态预测（使用 BHT 的结果）或者静态预测方法，再用立即数计算跳转地址。

最终结果发给 id 级。把从 icache 得到的指令信息和分支预测的信息打包在一起，放入取指 fifo。依次弹出 fifo 信息，发往 id 级。

关于压缩指令的处理。如果取到的一条 32 位指令是两条连续的压缩指令（16 位），则通过高 16 位补 0 的方式重组为两条 32 位指令，发给 instr_scan 模块。如果取到的是一条压缩指令和半条正常指令，则只发给 instr_scan 模块那个压缩指令，剩下半条指令等待下一个 clk 周期新的指令到达后，重组为完整指令。注意，这些操作只是为了做分支预测。发给 id 级的指令仍然是从 icache 取回的原始形式。

Table-1 signals

singal	I/O	Description
clk_i，rst_ni，flush_i	Input	
flush_bp_i		与 btb 和 bht 的输入信号 flush_i 连接
resolved_branch_i		npc 的来源之一

		branchpredict_t 类型结构体，来自 controller（或 ex 级），是分支跳转的确切结果，据此更新 btb 和 bht，并 kill 掉 icache 已有的取指请求
set_pc_commit_i		npc 的来源之一，来自 commit 级，流水线 flush 时用提交阶段的 pc 作为 npc
eret_i		npc 的来源之一，CSR 寄存器的函数返回地址
ex_valid_i		npc 的来源之一，异常和中断
set_debug_pc_i		npc 的来源之一，指向一个固定地址
pc_commit_i		64 位，与 set_pc_commit_i 相关
epc_i		64 位，与 eret_i 相关
trap_vector_base_i		64 位，与 ex_valid_i 相关
boot_addr_i		64 位，是系统复位后的第一个 pc 值
icache_dreq_i	与 icache 交互	icache_dreq**_o_t** 类型结构体，来自 icache 的取指结果
icache_dreq_o		icache_dreq**_i_t** 类型结构体，输往 icache，请求取指。主要增加了 kill_s1 和 kill_s2 的信息。如果预测错误或者 flush，就应该 kill 掉 icache 请求 s1 和 s2。如果分支预测有效，就 kill 上一个 cache 请求 s2.
fetch_ack_i	与 id 级交互	来自 id（译码）级，表示 id 级确认收到指令，发往取指 fifo 模块，fifo 中读指针就能移动到下一个位置，即删除这一条指令
fetch_entry_o		fetch_entry_t 类型结构体，由取指 fifo 得到，包含指令相关的信息，发送给 id 级
fetch_entry_valid_o		与 fetch_entry_o 相关

向 icache 取指的结构体

```
typedef struct packed {
        logic                       req;                 // we request a new word
        logic                       kill_s1;             // kill the current request
        logic                       kill_s2;             // kill the last request
        logic [63:0]                vaddr;
} icache_dreq_i_t;
```

从 icache 返回的结构体

```
typedef struct packed {
        logic                       ready;               // icache is ready
        logic                       valid;               // signals a valid read
        logic [FETCH_WIDTH-1:0]     data;                // 2+ cycle out: tag
        logic [63:0]                vaddr;               // virtual address out
        exception_t                 ex;
} icache_dreq_o_t;
```

处理后，发往 id 级的结构体

```
typedef struct packed {
        logic [63:0]            address;
        logic [31:0]            instruction;
        branchpredict_sbe_t     branch_predict;
        exception_t             ex;
} fetch_entry_t;
```

4 frontend 子模块

4.1 子模块 1——BTB

BTB：branch target buffer

一种动态分支预测方法。在取指阶段就给出分支预测结果。BTB 是一个固定大小的哈希表，映射了分支指令地址和它对应的跳转地址。取值阶段，将得到的指令地址拿到 BTB 检索，如果 BTB 中有相同的地址，我们就可以在流水线译码阶段之前就知道当前指令是分支指令，并得到 BTB 中保存的改分支指令历史记录的跳转指令地址。当然，跳转指令地址仍然不一定是预测正确的。如果预测出错，要及时更新 BTB 中的记录。

当前设计中，BTB 只有 8 个跳转地址和当前 pc 值的映射的条目，截取输入的 vpc_i 的低 3 位查询映射。组合逻辑完成。

更新逻辑是用输入的 btb_update_i 更新映射。需要消耗一个 clk 周期。

接口信号与代码说明

Table-1 signals

singal	**I/O**	**Description**
clk_i，rst_ni，flush_i	Input	
vpc_i	Input	64 位，**虚拟地址**
btb_update_i	Input	结构体，包含 pc 值和跳转地址等信息，用于更新 btb 表
btb_prediction_o	output	结构体，包含跳转地址等信息，用于输出预测结果

两个结构体的具体信息

```
typedef struct packed {
        logic           valid;
        logic [63:0]    pc;
        logic [63:0]    target_address;
        logic           is_lower_16;
        logic           clear;
} btb_update_t;

typedef struct packed {
        logic           valid;
        logic [63:0]    target_address;
        logic           is_lower_16;
} btb_prediction_t;
```

其中，is_lower_16 表明是否是压缩指令，clear 可以使得 btb 中相应的条目 valid 为 0。

4.2 子模块 2——BHT

BHT：branch history table

一种动态分支预测方法。在译码阶段确定是分支指令后，通过检索当前分支指令对应的过去分支跳转的历史，来决定是否跳转。这个跳转历史通过一个 2 位计数器来表达，值越高，表示过去实际发生跳转的情况越多。

当前设计中，BHT 有 1024 个跳转预测记录，需要截取输入的 vpc_i 的低 10 位查询映射。组合逻辑完成。

更新逻辑是用输入的 bht_update_i 更新映射。需要消耗一个 clk 周期。

接口信号与代码说明

Table-1 signals

singal	**I/O**	**Description**
clk_i，rst_ni，flush_i	Input	
vpc_i	Input	64 位，**虚拟地址**
bht_update_i	Input	结构体，包含 pc 值和预测结果等信息，用于更新 bht 表
bht_prediction_o	output	结构体，包含预测结果等信息，用于输出预测结果

两个结构体的具体信息

```
typedef struct packed {
        logic           valid;
        logic [63:0]    pc;
        logic           mispredict;
        logic           taken;
} bht_update_t;

typedef struct packed {
        logic           valid;
        logic           taken;   //可能跳转
        logic           strongly_taken;   //非常可能跳转
} bht_prediction_t;
```

其中，bht_update_t 中的 taken 值用于更新 bht 表中相应条目的计数器值。

4.3 子模块 3——RAS

RAS：return address stack

一个固定大小的返回地址的栈，保存跳转指令的返回地址。

当前设计中，栈的深度只有 2，栈中每个条目的内容只有 pc 值和 valid 值。压栈出栈需要输入信号 pop_i 和 push_i，同时消耗 1 个 clk 周期。

接口信号与代码说明

Table-1 signals

singal	**I/O**	**Description**
clk_i，rst_ni	Input	

data_i	Input	64 位
pop_i	Input	栈操作
push_i	Input	栈操作
data_o	output	结构体，包含 64 位地址及 valid 信息

结构体的具体信息

```
typedef struct packed {
        logic           valid;
        logic [63:0]    ra;
} ras_t;
```

4.4 子模块 4——instr_scan

初步判断当前指令的类型。

纯组合逻辑。

判断 rvi_call_o 的语句是 assign rvi_call_o = (rvi_jalr_o | rvi_jump_o) & instr_i[7]；其中 instr_i[7]是指指令中的目的寄存器尾号是否是 1，可用来判别目的寄存器是否是 x1 或者 x5 。因为 x1 被编译器用来做返回地址寄存器 ra，x5 被用来做链接寄存器 t0 。rvi_call_o 信号用来通知 RAS 模块压栈。

判断 rvi_return_o 的语句是 assign rvi_return_o = rvi_jalr_o & ~instr_i[7] & ~instr_i[19] & ~instr_i[18] & ~instr_i[16] & instr_i[15]；这个语句判断是否是用到 x1 或者 x5 作为基址寄存器的 jalr 指令。

使用了 x1 或者 x5 作为目的寄存器或者基址寄存器说明当前跳转指令需要（从 RAS）保存或者读取返回地址。详见 User-Level ISA 2.2 page 16.

接口信号与代码说明

Table-1 signals

singal	**I/O**	**Description**
		无时钟和复位信号
instr_i	Input	32 位
is_rvc_o	output	判断是否是压缩指令
rvi_return_o, rvi_call_o, rvi_branch_o, rvi_jalr_o, rvi_jump_o,	output	判断是哪种类型的非压缩指令，这里只识别是否是 jal，jalr，branch 三种类型之一
rvi_imm_o	output	64 位，对于上述 3 种跳转类型，imm 有 2 种译码形式，第一种，uj_imm，指的是 U 型和 J 型指令的 imm 编码形式；第二种，sb_imm，指的是 S 型和 B 型指令的 imm 编码形式
rvc_return_o, rvc_call_o, rvc_branch_o, rvc_jalr_o,	output	判断是哪种类型的压缩指令，同上

rvc_jump_o,		
rvc_imm_o	output	64 位

4.5 子模块 5——fetch_fifo

该模块维护一个 FIFO 单元，其中每一个条目是一个 fetch_entry_t 类型的结构体，当前实现中，FIFO 深度是 8 。该模块把输入的（从 icache 读取的信息，以及 Frontend 模块内的产生的分支预测信息）各种与指令相关的信息打包，保存。出栈的指令直接给流水线下一级的 id 级。

```
typedef struct packed {
        logic [63:0]            address;
        logic [31:0]            instruction;
        branchpredict_sbe_t     branch_predict;
        exception_t             ex;
} fetch_entry_t;
```

接口信号与代码说明

Table-1 signals

singal	**I/O**	**Description**
clk_i，rst_ni，flush_i	Input	
rdata_i		32 位
addr_i		64 位
valid_i		如果为 1，则将 rdata_i，addr_i 等输入信息打包压入 fifo 中
ex_i		exception_t 类型结构体
branch_predict_i		branchpredict_sbe_t 类型结构体，是 branchpredict scoreboard entry
fetch_ack_i		请求 pop
ready_o	output	为 1，表示当前 fifo 内条目数小于 fifo 总容量减 3
fetch_entry_o		fetch_entry_t 类型结构体
fetch_entry_valid_o		

结构体 exception_t 的具体信息

```
typedef struct packed {
        logic [63:0]    cause;      // cause of exception
        logic [63:0]    tval;       // additional information of causing exception
        logic           valid;
} exception_t;
```

结构体 branchpredict_sbe_t 的具体信息

```
typedef struct packed {
        logic           valid;              // this is a valid hint
        logic [63:0] predict_address; // target address at which to jump, or not
        logic           predict_taken;  // branch is taken
        logic           is_lower_16;    // branch instruction is compressed and resides
                                        // in the lower 16 bit of the word
        cf_t            cf_type;        // Type of control flow change
```

} branchpredict_sbe_t;

5 std_cache_subsystem 接口信号与代码说明

主模块内除了例化 icache 和 dcache 外，没有别的功能语句。因此接口信号与代码说明放入 2 个子模块中介绍。

6 std_cache_subsystem 子模块

6.1 子模块 1——icache

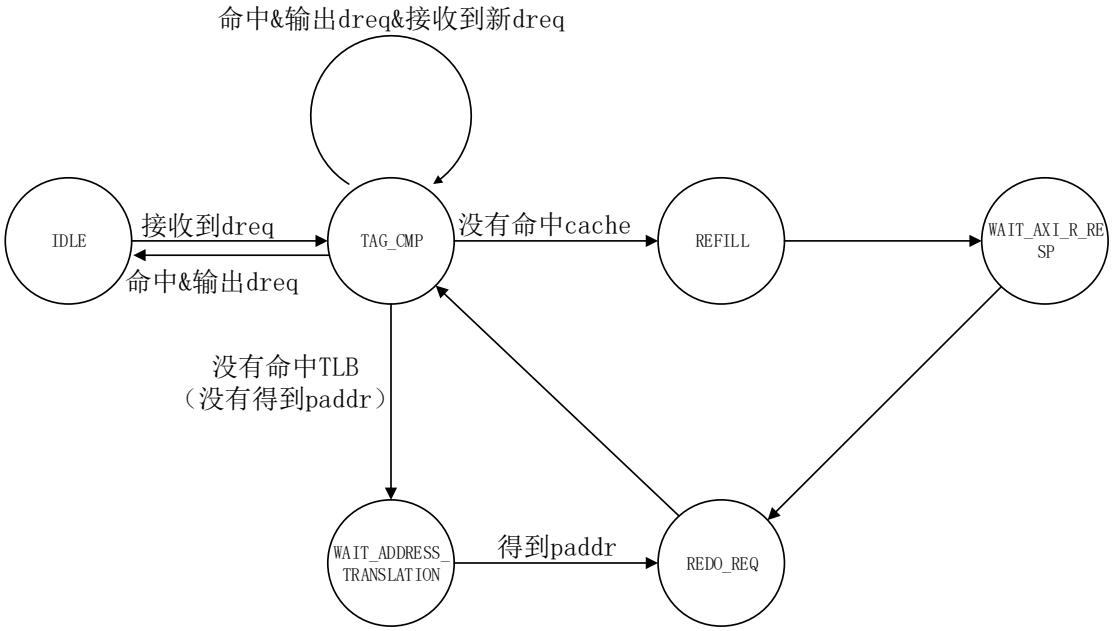
icache 定义了 DATA SRAM 和 TAG SRAM，分别存储指令数据和它在物理内存中的地址。根据物理内存的一个数据被允许放入 cache 中数据位置的多少，分为三种实现方式：①如果 cache 中只有一个位置可以容纳这个数据，则为直接映射的 cache；②如果 cache 中有多个地方可以放置这个数据，它就是组相联的 cache；③如果 cache 中的任何地方都可以放置这个数据，那么它就是全相连的 cache。

在 ariane_pkg.sv 中定义了 ICACHE_SET_ASSOC=4，即组相联是 4，就是说 1 个 set 有 4 个 cacheline（每个 cacheline 只能放一个数据），根据物理地址的字段索引数据放入了哪一个 set 中，数据可以放入这个 set 的任何一个 cacheline 中，所以再对比 tag 信息确定数据所在 cacheline 的具体位置。

在 ariane 的程序实现中，icache 模块例化了 4 组存贮阵列，各包括 1 个 tag array 和 1 个 data array。tag array 中 1 个条目包括 1 个 44 位的 tag 和一个 1 位的有效标志。输入信号 dreq_i 携带 vaddr 请求取指，icache 模块一方面向 MMU 的 TLB 发出地址翻译请求 areq_o（通过状态机控制），等待得到 paddr，另一方面利用 vaddr 和 paddr 的相同字段索引 4 个 tag array，得到 4 个备选 tag 值，再与从 TLB 得到的 paddr 的 tag 比较，最终确定命中结果。假如命中的是 4 组 tag array 的第 3 组的 tag 值，则输出其对应的 data 指令。

如果没有命中，由状态机逻辑决定替换策略是替换失效的 cacheline 还是随机替换（当 4 个 cacheline 都有效时）。由 axi 接口信号与顶层模块 ariane 外的 memory 交互，取指。

一般来说，tag 应该用 CAM，即内容寻址存储器来实现。



这里，icache 模块例化了 LFSR 和 ff1 。LFSR（线性反馈移位寄存器）用于产生随机值，ff1 用于 find first one。

接口信号与代码说明

Table-1 signals

singal	I/O	Description
clk_i，rst_ni，flush_i	Input	
en_i		使能 icache
areq_i		icache_areq_i_t 类型结构体，包含物理地址信息
dreq_i		icache_dreq_i_t 类型结构体，来自 frontend，请求取指
miss_o	output	
areq_o		icache_areq_o_t 类型结构体，输出翻译成虚拟地址
dreq_o		icache_dreq_o_t 类型结构体，发往 frontend，取指结果
axi	AXI_BUS.Master	在../include/axi_if.sv 中定义

结构体的信息

```
typedef struct packed {
        logic                       fetch_valid;        // address translation valid
        logic [63:0]                fetch_paddr;        // physical address in
        exception_t                 fetch_exception; // exception occurred during fetch
} icache_areq_i_t;

typedef struct packed {
        logic                       fetch_req;          // address translation request
        logic [63:0]                fetch_vaddr;        // virtual address out
} icache_areq_o_t;

typedef struct packed {
        logic                       req;                    // we request a new word
        logic                       kill_s1;                // kill the current request
        logic                       kill_s2;                // kill the last request
        logic [63:0]                vaddr;          // 1st cycle: 12 bit index is taken for lookup
} icache_dreq_i_t;

typedef struct packed {
        logic                       ready;                  // icache is ready
        logic                       valid;                  // signals a valid read
        logic [FETCH_WIDTH-1:0]     data;                   // 2+ cycle out: tag
        logic [63:0]                vaddr;                  // virtual address out
        exception_t                 ex;                 // we've encountered an exception
} icache_dreq_o_t;
```

6.2 子模块 2——nbdcache

该模块注明是非阻塞的私有的 L1 dcache。非阻塞 cache 的含义是，当一条访存请求在 cache 发生 miss 时，cache 还可以接收处理别的访存请求。而阻塞 cache 一次只能接收处理

一条访存请求。在 ariane 的设计中，nbdcache 可以同时接受 3 个请求，分别来自 load，store 和 ptw。相应的，nbdcache 模块例化了 3 个 cache_ctrl。

相比 icache，dcache 的控制逻辑更加复杂，主要在于 dcache 涉及到写操作，所以对于 load 和 store 指令发出的 dcache 请求有不同的控制状态切换，而且要考虑 cache 一致性的问题。但来自 ex 级的 dcache 请求已经在 ex 级模块内做了 TLB 处理，因此 dcache 控制逻辑不需要考虑物理地址和虚拟地址的转换。

在 ariane_pkg.sv 中定义了 DCACHE_SET_ASSOC=8，该模块也例化了 8 组 data+tag 的 SRAM。还例化了 1 个 SRAM 作为 dirty_sram。此外，该模块例化了多个小模块：cache_ctrl，miss_handler，tag_cmp。

子模块 tag_cmp 较为简单。tag_cmp 选出一个 addr 发往 tag sram，获取标签一致的 8 个 tag 值，然后对比目标地址的 tag 值，给出是否命中的结果，发给 cache_ctrl 子模块。子模块 tag_cmp 的一个重要作用是仲裁，即决定优先处理哪一个 dcache 请求。

子模块 miss_handler 处理 dcache 没有命中的情况。内部例化了子模块 axi_adapter 和 arbiter（什么情况 bypass），用于和外部的 memory 交互。主体是一个有限状态机：

IDLE：检查如果发现有一个 dcache 请求发生 miss，则把请求存入 mshr 寄存器，并转入状态 MISS。mshr 是 Miss-status Handling Registers 的缩写。

MISS：发出 miss 请求和 miss 地址，并转入 MISS_REPL 状态。miss 信息经过子模块 tag_cmp 时会被优先处理，发往？？

MISS_REPL：检查是否有空的 cacheline。如果没有，通过 lfsr 随机决定替换一个 cacheline。并检查这个被替换掉的 cacheline 是否 dirty（是否需要先写回 memory），如果是，则进入状态 WB_CACHELINE_MISS。如果不需要，则进入状态 REQ_CACHELINE。如果有空的 cacheline，也进入状态 REQ_CACHELINE。

REQ_CACHELINE：将 miss 请求地址通过 axi_adapter 模块的接口发给外部的 memory。等待，如果收到响应信号，就进入状态 SAVE_CACHELINE。

SAVE_CACHELINE：接收数据，发给子模块 tag_cmp 仲裁后转给 sram，替换 cacheline。并且标记 mshr 使其失效。如果这个请求是写请求，还要标记 dirty。最后无条件转入状态 IDLE。

WB_CACHELINE_MISS：通过 axi_adapter 模块的接口与 memory 交互，先写回要被替换掉的脏的 cacheline。再转回状态 MISS。

子模块 cache_ctrl 是 dcache 控制逻辑的主体模块，同样设计了一个有限状态机。

IDLE：检测是否有 dcache 请求，如果有，转入状态 WAIT_TAG。

WAIT_TAG：向 tag_cmp 模块发出 tag 请求，这个 tag 比较过程在 tag_cmp 模块内由组合逻辑执行，单个周期内即可完成。WAIT_TAG 状态接收 tag 的比较结果。如果是 load 指令的请求命中 dcache，则给出 dcache 的输出结果，下一个状态继续停留在状态 WAIT_TAG，接收新的 dcache 请求。如果是 store 指令的请求命中 dcache，则转入状态 STORE_REQ。如果 load 或者 store 没有命中，检测当前这个 dcache 请求和 mshr 寄存器中的请求是否相同，这个检测过程由模块 miss_handler 中的组合逻辑单周期内完成。如果相同，转入 WAIT_MSHR 状态。如果不相同，转入 WAIT_REFILL_GNT 状态。

STORE_REQ：这个状态主要还是检测当前这个 store 的 dcache 请求和 mshr 寄存器中的请求是否相同。检测结果如果不相同，就正常通过 tag_cmp 模块把请求写的数据送入 sram，并标记 dirty 位。如果相同，则转入状态 WAIT_MSHR。（store 的 dcache 请求和 mshr 寄存器中的请求相同，表示写 dcache 的地址没有命中，则与 mshr 中的请求合并，发送给模块 miss_handler 处理。）

WAIT_MSHR：再检测一遍 mshr，等到不再相同（即这个 miss 处理已经结束），下一状

态转回 WAIT_TAG_SAVED（WAIT_TAG）。

WAIT_REFILL_GNT：再检测一遍 mshr，如果此时的 mshr 寄存器的请求与当前请求匹配，就转入状态 WAIT_MSHR。如果不匹配，向 miss_handler 模块发出 miss 请求，等待完成。如果完成，就转回 IDLE 状态。如果是 load，转入 WAIT_CRITICAL_WORD。

WAIT_CRITICAL_WORD：

关于 dirty sram，，

接口信号与代码说明

Table-1 signals

singal	**I/O**	**Description**
clk_i，rst_ni，flush_i	Input	
enable_i	Input	
flush_ack_o	output	
miss_o	output	
amo_commit_i	与外部信号暂无连接	
amo_valid_o		
amo_result_o		
amo_flush_i		
req_ports_i		dcache_req_i_t [2:0]
req_ports_o		dcache_req_o_t [2:0]
data_if		AXI_BUS.Master
bypass_if		AXI_BUS.Master

7 id_stage 接口信号与代码说明

指令先后经过 instr_realigner，compressed_decoder 和 decoder 三个模块进行译码处理。id_stage 主模块中还有几条语句，实现与流水线前后级的交互。

接口信号与代码说明

Table-1 signals

singal	**I/O**	**Description**
clk_i，rst_ni，flush_i	Input	
fetch_entry_i		fetch_entry_t 类型结构体，来自 frontend
fetch_entry_valid_i		来自 frontend
issue_instr_ack_i		来自 issue 的指令确认信号
priv_lvl_i	Input（来自 CSR，用于 decoder 子模块）	
debug_mode_i		
tvm_i		
tw_i		
tsr_i		
decoded_instr_ack_o	output	发给 frontend 的 fetch_ack_i，表示是否接收到指令

issue_entry_o		scoreboard_entry_t 类型结构体，包含译码结果，来自 decoder 子模块，发给下一级
issue_entry_valid_o		
is_ctrl_flow_o		来自 decoder 子模块，发给下一级

8 id_stage 子模块

8.1 子模块 1——instr_realigner（细节）

该模块检测非对齐指令，并做对齐处理（对压缩指令的高 16 位补 0）。
其中的逻辑细节，目前还不清楚。

接口信号与代码说明

Table-1 signals

singal	I/O	Description
clk_i，rst_ni，flush_i	Input	
fetch_entry_0_i	Input	fetch_entry_t 类型结构体，来自 frontend 的 fetch_fifo
fetch_entry_valid_0_i		
fetch_entry_o	output	fetch_entry_t 类型结构体
fetch_entry_valid_o		
fetch_ack_i	Input	
fetch_ack_0_o	output	与 id_stage 的 decoded_instr_ack_o 输出接口相连 如果发现从 frontend 收到的指令的高 16 位是一个完整的压缩指令，那么此信号置 0，表示要求 frontend 级中的取指 fifo 不要删除这条指令

```
typedef struct packed {
        logic [63:0]            address;
        logic [31:0]            instruction;
        branchpredict_sbe_t     branch_predict;
        exception_t             ex;
} fetch_entry_t;
```

8.2 子模块 2——compressed_decoder

该模块将对齐后的 32 位压缩指令（高 16 位是 0）解压为真正的 32 位指令。

接口信号与代码说明

Table-1 signals

singal	I/O	Description
		无 clk 信号，是纯组合电路
instr_i	Input	32 位
instr_o	output	32 位
illegal_instr_o		标志指令是否合法
is_compressed_o		标志指令原来是否是压缩的

8.3 子模块 3——decoder

该模块完成指令译码，内部逻辑比较简单。

接口信号与代码说明

Table-1 signals

singal	**I/O**	**Description**
		无 clk 信号，是纯组合电路
pc_i	Input（直接添加到 instruction_o 输出中）	64 位
is_compressed_i		来自前一子模块
instruction_i		32 位，来自前一子模块
branch_predict_i		branchpredict_sbe_t 类型结构体
ex_i		exception_t 类型结构体
priv_lvl_i	Input（帮助判断是否是非法指令）	riscv::priv_lvl_t 类型结构体，决定了某种特权模式下，一些指令是否允许被执行，来自 CSR。如果不允许，则标记是非法指令
is_illegal_i		来自前一子模块
debug_mode_i		来自 CSR
tvm_i		来自 CSR，trap virtual memory
tw_i		来自 CSR，timeout wait
tsr_i		来自 CSR，trap sret
instruction_o	output	scoreboard_entry_t 类型结构体
is_control_flow_instr_o		表示是否是 Control Flow Instructions（即，是否是 branch，jalr 或者 jal）

scoreboard_entry_t 结构体

```
typedef struct packed {
        logic [63:0]                    pc;
        logic [TRANS_ID_BITS-1:0] trans_id;     //id 级统一标记为 0

        fu_t                            fu;     //使用的功能单元
        fu_op                           op;     //具体运算符
        logic [REG_ADDR_SIZE-1:0] rs1;     //这里 REG_ADDR_SIZE=6（为了 rename）
        logic [REG_ADDR_SIZE-1:0] rs2;
        logic [REG_ADDR_SIZE-1:0] rd;
        logic [63:0]                    result;      //已经扩展好的立即数
        logic                           valid;
        logic                           use_imm;
        logic                           use_zimm;
        logic                           use_pc;
        exception_t                     ex;
        branchpredict_sbe_t             bp;
```

```
        logic                   is_compressed;
} scoreboard_entry_t;
```

9 issue_stage 接口信号与代码说明

主模块内没有功能语句，完全由 3 个子模块互联而成。

基本数据流：一条指令 scoreboard_entry_t 条目输入 issue 级后，先进入 rename 子模块，做初步检查，判断是否需要寄存器换名。然后把 scoreboard_entry_t 送入计分板模块，如果指令有效而且计分板队列未满，就可以存入队列，并发往 issue 子模块。issue 子模块检查如果没有冲突冒险的情况，就正式把指令对应的数据和控制信号发往 ex 级。

（以下资料参考网页 https://blog.csdn.net/whutxinriyue/article/details/5485171）

Scoreboard 算法能够在按序发射-无序完成的实现中，自动动检测和消除相关性。在这一实现中，指令按照源二进制程序的顺序发射到 Scoreboard 单元，Scoreboard 有一个指令队列，保存译码后的指令，当 Scoreboard 检测到某条指令可以执行时，就立刻调度该指令，而忽略它们的原始顺序。在 Scoreboard 算法中，指令执行需要经过以下阶段：

发射(Issue)

指令译码之后，根据指令的操作确定该条指令所需要的功能部件，如果当前功能部件是空闲的(保证了没有资源冲突(Structural Hazards)，并且当前处于执行阶段的指令中(可能存在多条指令乱序执行)，没有任何一条指令的目的操作数(写入对象)与当前待发射指令的目的操作数相同(保证了不会导致 WAW 冒险(输出相关性)。)，就把译码后的指令提交到 Scoreboard 指令队列，更新 Scoreboard 的相关数据结构。否则停止发射，在这种情况下，如果取指队列还没有满，取指工作会继续。

读操作数(Read Operands)

Scoreboard 监视到指令队列中的每一个源操作数寄存器，如果没有任何一条处于运行阶段(多条指令)的指令对该寄存器进行写入，或者写入操作已经完成时，Scoredboard 通知相关功能单元，可以为该条指令读取源操作数。在这个过程中，Scoreboard 保证了当前读取到的源操作数是最新的，消除了 RAW 冒险的可能。

执行(Execution)

源操作数准备就绪之后，进入执行阶段，不同的功能部件的执行时间可能不一致，当执行完成后，会通知 Scoreboard.

回写(Write Result)

当 Scoreboard 接到执行单元的执行完毕的通知之后，Scoreboard 检测回写的目标寄存器当前某条指令的源寄存器一致，并且该条指令还没有完成读源操作数(消除 WAR 冒险)

（以上资料参考网页 https://blog.csdn.net/whutxinriyue/article/details/5485171）

一些普通指令在 ex 级模块计算出结果后，先返回给 issue 级模块的计分板登记，再发给 commit 级模块决定是否能提交，最后再返回给 issue 级模块写回寄存器。

load 指令的提交决定是最后发给 ex 级的（和 cache 交互），而不是 issue 级（和寄存器堆交互）。

csr 指令的计算结果发给 csr 模块，同时发给 commit 级提交请求，最终提交工作在 csr 模块内完成。

接口信号与代码说明

Table-1 signals

singal	I/O	Description
clk_i，rst_ni，flush_i	Input	
test_en_i	Input	在寄存器堆模块中使用，用于 cluster_clock_gating
flush_unissued_instr_i		来自 controller
decoded_instr_i		来自 id 级，scoreboard_entry_t 类型结构体
decoded_instr_valid_i		来自 id 级
is_ctrl_flow_i		来自 id 级，**无内部连线？**
alu_ready_i	Input 来自 ex 级 表示 FU 空闲	
branch_ready_i		
lsu_ready_i		
mult_ready_i		
csr_ready_i		
resolve_branch_i		来自 ex 级，**无内部连线？**
trans_id_i	Input 来自 ex 级的各 FU 计算结果	把计算结果写入计分板的 FIFO 中，然后依次放入 commit_instr_o 端口，准备提交 这里的计算结果也可用于数据冒险时的前递
wbdata_i		
ex_i		
wb_valid_i		
waddr_i	Input 来自 commit 级的写回结果	NR_COMMIT_PORTS 位宽，5 深度
wdata_i		NR_COMMIT_PORTS 位宽，64 深度
we_i		NR_COMMIT_PORTS 位宽
decoded_instr_ack_o	output 发往 id 级	表示确认接收指令
fu_o	output 发往 ex 级 继续传递 id 级的结果	fu_t 类型结构体
operator_o		fu_op 类型结构体
operand_a_o		64 位
operand_b_o		64 位
imm_o		64 位
pc_o		64 位
trans_id_o		TRANS_ID_BITS 位宽
is_compressed_instr_o		
alu_valid_o	output 发往 ex 级 根据指令类型使能 ex 级中相应的 FU 单元	
branch_valid_o		
branch_predict_o		branchpredict_sbe_t 类型结构体
lsu_valid_o		
mult_valid_o		
csr_valid_o		
commit_ack_i	与 commit 级交互	NR_COMMIT_PORTS 位宽
commit_instr_o		scoreboard_entry_t 类型结构体 包含准备提交的计算结果

10 issue_stage 子模块

10.1 子模块 1——re_name

从 id 级得到的译码数据，先进入 re_name 模块处理。该模块（硬件）实现寄存器重命名，目的是消除数据相关。

本设计应该是在 32 个结构寄存器之外，增设了 32 个对应的重命名寄存器（程序员不可见的微结构寄存器）。为了实现重命名，scoreboard_entry_t 条目中是用 6 位二进制来索引 32 个结构寄存器的。重命名需要替换索引值的最高位。这也是 re_name 模块对输入数据的唯一修改。

寄存器重命名技术在乱序执行流水线中有两个作用。一是消除指令之间的寄存器读后写相关（WAR），和写后写相关（WAW）；二是当指令执行发生例外或转移指令猜测错误而取消后面的指令时可以保证现场的精确。寄存器重命名的思路很简单：就是当一条指令写一个结果寄存器时不直接写到这个结果寄存器，而是先写到一个中间寄存器过渡一下，当这条指令提交的时候再写到结果寄存器（https://blog.csdn.net/edonlii/article/details/8771023 ）。

接口信号与代码说明

Table-1 signals

singal	**I/O**	**Description**
clk_i，rst_ni，flush_i	Input	
issue_instr_i	Input	scoreboard_entry_t 类型结构体
issue_instr_valid_i		
issue_ack_i		
issue_instr_o	output	scoreboard_entry_t 类型结构体
issue_instr_valid_o		与 issue_instr_valid_i 直接相连
issue_ack_o		与 issue_ack_i 直接相连

10.2 子模块 2——issue_read_operands

该模块接收指令的译码结果，完成如下工作：

第一，确定源操作数。这个工作由 operands_available 和 forwarding_operand_select 两个逻辑块完成。一般情况下，源操作数从例化的 ariane_regfile 读取。如果通过查表 rd_clobber_i，发现源操作数要从前面指令的写回结果得到，则由 rs1_o/rs2_o 和 rs1_i/rs2_i 两组信号取计分板中的最新结果。还有可能是源操作数由立即数得到。

第二，确定要用到的功能单元。这个工作由 unit_valid 逻辑块完成。

第三，最终确定指令是否可以 issue 给下一级。这个工作由 issue_scoreboard 和 unit_busy 两个逻辑块完成。如果用到的寄存器数据不冲突，用到的功能单元有空闲，就可以置位 issue_ack_o 信号。

另外，issue_read_operands 子模块还接收从 commit 级发来的写回数据，因为要写回的通用寄存器组 ariane_regfile 是在该模块中例化的。这个寄存器组模块，包含 32 个 64 位寄存器。寄存器组模块的输入输出端口很简单，主要就是 2 组读端口和 2 组写端口，但是寄存器地址都**是用 5 位二进制数索**引的。

接口信号与代码说明

Table-1 signals

singal	**I/O**	**Description**
clk_i，rst_ni，flush_i	Input	
test_en_i	Input	在寄存器堆模块中使用，用于 cluster_clock_gating
issue_instr_i		来自计分板，scoreboard_entry_t 类型结构体
issue_instr_valid_i		来自计分板
rs1_i		64 位，与计分板交互，检查是否需要前递数据
rs1_valid_i		
rs2_i		64 位，与计分板交互，检查是否需要前递数据
rs2_valid_i		
alu_ready_i		来自 ex 级
branch_ready_i		来自 ex 级
lsu_ready_i		来自 ex 级
mult_ready_i		来自 ex 级
csr_ready_i		来自 ex 级
rd_clobber_i		fu_t 类型结构体，2**REG_ADDR_SIZE+1 深度
waddr_i		NR_COMMIT_PORTS 位宽，5 深度 来自 commit 级，与寄存器堆交互，完成回写
wdata_i		NR_COMMIT_PORTS 位宽，64 深度 来自 commit 级，与寄存器堆交互，完成回写
we_i		NR_COMMIT_PORTS 位宽 来自 commit 级，与寄存器堆交互，完成回写
rs1_o	output	REG_ADDR_SIZE 位宽，与计分板交互，检查是否前递
rs2_o		REG_ADDR_SIZE 位宽，与计分板交互，检查是否前递
alu_valid_o		发往 ex 级
branch_valid_o		发往 ex 级
branch_predict_o		branchpredict_sbe_t 类型结构体
lsu_valid_o		发往 ex 级
mult_valid_o		发往 ex 级
csr_valid_o		发往 ex 级
fu_o	output 发往 ex 级	fu_t 类型结构体
operator_o		fu_op 类型结构体
operand_a_o		64 位
operand_b_o		64 位
imm_o		64 位
pc_o		64 位
trans_id_o		TRANS_ID_BITS 位宽
is_compressed_instr_o		
issue_ack_o	output	请求发射，发往计分板

10.3 子模块 3——scoreboard

该模块跟踪所有 decoded、issued、committed 指令的状态。所以该模块与 ex 级、commit 级都有交互。

当前计分板是一个深度为 8 的的 FIFO，其中每一个条目是一个 scoreboard_entry_t 结构体，里面存储一条指令译码后的各种信息。一般情况下，从 id 级发来的指令译码信息，直接压入计分板，并根据在计分板中位置标记 trans_id，然后发给 issue_read_operands 子模块。

对于 ex 级执行完成的指令，计分板获取它们的写回结果，改写其对应的 result 信息。这样，issue_read_operands 子模块发来的操作数前递请求，就可以通过查看计分板实现。

对于 commit 级发来的提交信号，计分板检索对应信息，把要提交的数据返给 commit 级模块。之后，删除计分板中的对应信息。

为了监测数据冒险，计分板模块维护了一个 rd_clobber_o 结构，它记录了 65 个寄存器（32 个结构寄存器和 32 个重命名寄存器）正在参与哪些功能单元（作为目的寄存器）的运算的信息。可作为 issue_read_operands 子模块检测数据冒险的参考。

接口信号与代码说明

Table-1 signals

singal	**I/O**	**Description**
clk_i，rst_ni，flush_i	Input	
flush_unissued_instr_i	Input	
unresolved_branch_i		**指定为 0**
rs1_i		REG_ADDR_SIZE 位宽，与 issue 子模块交互
rs2_i		REG_ADDR_SIZE 位宽，与 issue 子模块交互
commit_ack_i		与 commit 级交互 NR_COMMIT_PORTS 位宽
decoded_instr_i		与 id 级交互 scoreboard_entry_t 类型结构体
decoded_instr_valid_i		与 id 级交互
issue_ack_i		与 issue 子模块交互
trans_id_i	Input 来自 ex 级的各个 FU	[NR_WB_PORTS-1:0][TRANS_ID_BITS-1:0]
wbdata_i		[NR_WB_PORTS-1:0][63:0]
ex_i		exception_t [NR_WB_PORTS-1:0]
wb_valid_i		[NR_WB_PORTS-1:0]
rd_clobber_o	output	fu_t 类型结构体，2**REG_ADDR_SIZE+1 深度
rs1_o		与 issue 子模块交互，64 位
rs1_valid_o		与 issue 子模块交互
rs2_o		与 issue 子模块交互，64 位
rs2_valid_o		与 issue 子模块交互
commit_instr_o		与 commit 级交互 scoreboard_entry_t 类型结构体，NR_COMMIT_PORTS 位宽
decoded_instr_ack_o		与 id 级交互
issue_instr_o		与 issue 子模块交互 scoreboard_entry_t 类型结构体

issue_instr_valid_o		与 issue 子模块交互

11 ex_stage 接口信号与代码说明

主模块内没有功能语句，完全由多个功能单元子模块互联而成。其中比较复杂的是 lsu 模块。

接口信号与代码说明

Table-1 signals

singal	**I/O**	**Description**
clk_i，rst_ni，flush_i	input	
fu_i,	input 来自 issue 级	fu_t
operator_i,		fu_op
operand_a_i,		[63:0]
operand_b_i,		[63:0]
imm_i,		[63:0]
trans_id_i,		[TRANS_ID_BITS-1:0]
pc_i,		[63:0]
is_compressed_instr_i,		
alu_valid_i,	input 来自 issue 级	
branch_valid_i,		
branch_predict_i,		branchpredict_sbe_t
lsu_valid_i,		
csr_valid_i,		
mult_valid_i,		
alu_valid_o,	output 统一返回给 issue 级的 wb_valid_i	
branch_valid_o,		
lsu_valid_o,		
csr_valid_o,		
mult_valid_o,		
alu_trans_id_o,	output 统一返回给 issue 级的 trans_id_i	[TRANS_ID_BITS-1:0]
branch_trans_id_o,		[TRANS_ID_BITS-1:0]
lsu_trans_id_o,		[TRANS_ID_BITS-1:0]
csr_trans_id_o,		[TRANS_ID_BITS-1:0]
mult_trans_id_o,		[TRANS_ID_BITS-1:0]
alu_result_o,	output 统一返回给 issue 级的 wbdata_i	[63:0]
branch_result_o,		[63:0]
lsu_result_o,		[63:0]
csr_result_o,		[63:0]
mult_result_o,		[63:0]
branch_exception_o,	output	exception_t

lsu_exception_o,	统一返回给 issue 级的 ex_ex_i	exception_t
alu_ready_o,	output 表示 FU 已经空闲返回给 issue 级	连接 issue 级的 alu_ready_i，指定为 1
branch_ready_o,		连接 issue 级的 branch_ready_i，指定为 1
lsu_ready_o,		连接 issue 级的 lsu_ready_i
csr_ready_o,		连接 issue 级的 csr_ready_i
mult_ready_o,		连接 issue 级的 mult_ready_i
resolved_branch_o,	output	branchpredict_t
resolve_branch_o,	output	发往 issue 级，**但没有连线**
lsu_commit_i,	与 commit 级交互	load 指令的提交决定是发给 ex 级的，而不是 issue 级
lsu_commit_ready_o,		
no_st_pending_o,	output	
csr_addr_o,	output	[11:0]
csr_commit_i,	input	
enable_translation_i,	input	
en_ld_st_translation_i,	input	
flush_tlb_i,	input	
priv_lvl_i,	input	riscv::priv_lvl_t
ld_st_priv_lvl_i,	input	riscv::priv_lvl_t
sum_i,	input	
mxr_i,	input	
satp_ppn_i,	input	[43:0]
asid_i,	input	[ASID_WIDTH-1:0]
icache_areq_i,	与 icache 交互	icache_areq_o_t，地址翻译请求，包含虚拟地址信息
icache_areq_o,		icache_areq_i_t，返回物理地址信息
dcache_req_ports_i,	与 dcache 交互	dcache_req_o_t [2:0]
dcache_req_ports_o,		dcache_req_i_t [2:0]
itlb_miss_o	output	发给 pref_counters 计数
dtlb_miss_o	output	发给 pref_counters 计数

对于条件分支指令，在子模块 branch 中处理。输出的 resolved_branch_o 结构体包含了经过实际计算后分支跳转的真实情况。这个信息包发给 frontend 级和 controller 模块。

对于 load&store 指令，它们与 dcache 的交互都通过 dcache_req_ports_i& dcache_req_ports_o 端口。发给 dcache 的是一个 dcache_req_i_t 结构体，从 dcache 返回一个 dcache_req_o_t 结构体。

```
typedef struct packed {
    logic [DCACHE_INDEX_WIDTH-1:0] address_index;
    logic [DCACHE_TAG_WIDTH-1:0]     address_tag;
    logic [63:0]                     data_wdata;
    logic                              data_req;
    logic                              data_we;
    logic [7:0]                      data_be;
    logic [1:0]                      data_size;
```

```
        logic                           kill_req;
        logic                           tag_valid;
        amo_t                            amo_op;
    } dcache_req_i_t;
    typedef struct packed {
        logic                       data_gnt;
        logic                       data_rvalid;
        logic [63:0]                data_rdata;
    } dcache_req_o_t;
```

其中 store 指令使用端口 2，load 指令使用端口 1，tlb 命中失败时，由子模块 mmu 的 ptw 使用端口 0 与 cache 交互（这个时候应该是从 cache 中去找物理地址，找到物理地址返回给 mmu，用于更新 tlb，以及第 2 次给 cache 发出读数据的访存请求）。

12 ex_stage 子模块

12.1 子模块 1——alu

该模块实现加减，移位，比较运算，以及与、或、异或逻辑。

接口信号与代码说明

Table-1 signals

singal	**I/O**	**Description**
		无时钟信号
trans_id_i	Input	TRANS_ID_BITS 位宽，与输出 alu_trans_id_o 直接相连
alu_valid_i		与输出 alu_valid_o 直接相连
operator_i		fu_op 枚举类型
operand_a_i		64 位
operand_b_i		64 位
result_o	Output	64 位
alu_branch_res_o		帮助 branch 单元计算，结果发往 branch 单元
alu_valid_o		与输入 alu_valid_i 直接相连
alu_ready_o		指定为 1
alu_trans_id_o		TRANS_ID_BITS 位宽，与输入 trans_id_i 直接相连

12.2 子模块 2——branch_unit

该模块实现条件分支和跳转指令。

对于 BEQ，BNE，BLT，BGE，其中的加法和比较操作直接利用 ALU 子模块的结果，因而 branch 子模块里输入 2 个 64 位操作数实际没有任何作用。

对于跳转指令，输入的 operand_a_i 可用于计算跳转地址。

输入的 branch_predict_i 结构体包含了我们对分支跳转的预测，输出的 resolved_branch_o 结构体包含了经过实际计算后分支跳转的真实情况，包括实际是否跳转（is_taken），是否预测错误（is_mispredict），真实的跳转地址（target_address）。branch 子模块要比较两种信息，给出是否预测失败的结论。

接口信号与代码说明

Table-1 signals

singal	I/O	Description
		无时钟信号
trans_id_i	Input	TRANS_ID_BITS 位宽，与输出 branch_trans_id_o 相连
branch_valid_i		与输出 branch_valid_o 相连
operator_i		fu_op 枚举类型
operand_a_i		64 位
operand_b_i		64 位
imm_i		64 位
pc_i		64 位
is_compressed_instr_i		
fu_valid_i		
branch_comp_res_i		来自 ALU 模块的计算结果
branch_predict_i		branchpredict_sbe_t 类型结构体
branch_valid_o	Output	与输入 branch_valid_i 相连
branch_ready_o		指定为 1
branch_result_o		64 位
branch_trans_id_o		TRANS_ID_BITS 位宽，与输入 trans_id_i 相连
resolved_branch_o		branchpredict_t 类型结构体
resolve_branch_o		
branch_exception_o		exception_t 类型结构体

结构体

```
typedef struct packed {
        logic              valid;              // this is a valid hint
        logic [63:0]       predict_address;    // target address at which to jump, or not
        logic              predict_taken;      // branch is taken
        logic              is_lower_16;
        cf_t               cf_type;            // Type of control flow change
} branchpredict_sbe_t;
```

结构体

```
typedef struct packed {
        logic [63:0]       pc;                 // pc of predict or mis-predict
        logic [63:0]       target_address;     // target address at which to jump, or not
        logic              is_mispredict;      // set if this was a mis-predict
        logic              is_taken;           // branch is taken
        logic              is_lower_16;
        logic              valid;              // prediction with all its values is valid
        logic              clear;              // invalidate this entry
        cf_t               cf_type;            // Type of control flow change
} branchpredict_t;
```

12.3 子模块 3——mult

该模块实现乘除法。

乘法操作例化了子模块 mul。

除法操作例化了子模块 ff1 和 serial_divider。

接口信号与代码说明

Table-1 signals

singal	**I/O**	**Description**
clk_i，rst_ni，flush_i	Input	
trans_id_i	Input	TRANS_ID_BITS 位宽
mult_valid_i		
operator_i		fu_op 枚举类型
operand_a_i		64 位
operand_b_i		64 位
result_o	Output	64 位
mult_valid_o		
mult_ready_o		
mult_trans_id_o		TRANS_ID_BITS 位宽

12.4 子模块 4——lsu

该模块例化了 mmu，store_unit，load_unit，lsu_arbiter，lsu_bypass 等 5 个子模块。其中：

lsu_arbiter 又例化了 rrarbiter 和 2 个 FIFO；

store_unit 又例化了 store_buffer；

mmu 又例化了 2 个 tlb 和 1 个 ptw。

mmu 主要处理虚拟地址和物理地址的映射。引入虚拟内存机制的 CPU，它初始发出的访存地址是虚拟地址，即访存地址范围超过了内存空间。先根据虚拟地址在 TLB 中查有没有对应的页表映射，如果有，即 TLB 命中，就找到了对应的物理地址，就可以继续去 cache 中查有没有缓存对应的数据了。如果没有 TLB 命中，就只能去内存中从全部页表的映射中查物理地址了。TLB 模块也要有自己的页面替换机制，以便及时更新。

要读写的目标地址在 lsu 主模块内计算得到 vaddr_i，把它与 lsu_valid_i、vaddr_i、operand_b_i、be_i、fu_i、operator_i、trans_id_i 一起组成一个信息包 lsu_req_i。其中 fu_i 指功能单元，operator_i 指具体指令类型，operand_b_i 是 store 指令中写入数据的来源，be_i 是字节使能，用于 load 双字、单字、半字时的控制信号。

信息包 lsu_req_i 发往 **lsu_bypass**。lsu_bypass 主要维护了一个 FIFO，将 issue 级发来的内存读写请求（即 lsu_req_i）先放入 FIFO 中缓存，因为当前还不确定这个读写请求能否命中 TLB。所以，可以说 lsu_bypass 解耦了 issue 级和 lsu 模块的数据关联。lsu_bypass 的输入信号 pop 控制从 FIFO 中读取数据（同样结构的信息包 lsu_req_o）。

接下来，信息包同时发往 **load_unit** 和 **store_unit**，并在 lsu 主模块内做初步解析。这个初步解析包括，判断是 load 还是 store，判断读写地址是否要触发非对齐异常（双字、单字、半字时）。

在 load_unit 中，先剥离出虚拟地址发给 **mmu**，同时把虚拟地址的低 12 位单独拎出来作为页面偏移地址 page_offset_o 发往 store_unit。mmu 中，发给 **tlb** 和 **ptw**，如果命中则从 tlb 中得到物理地址。如果没有命中，就交由 ptw 去内存或者 cache 中做一次完整的页表查询。store_unit 中，把 page_offset_o 交给 store_buffer 检查是否有 store 地址与它冲突，并输出 page_offset_matches_o 标志信号。一切顺利的话，load_unit 发出 load 访存请求，离开 ex 级，发到 **dcache**。从 dcache 得到数据 result，发往 **lsu_arbiter** 子模块。lsu_arbiter 从 load_unit 和 store_unit 的结果中选择一个，作为整个 lsu 单元的结果。

在 store_unit 中，，

接口信号与代码说明

Table-1 signals

singal	**I/O**	**Description**
clk_i，rst_ni，flush_i	Input	
trans_id_i	Input	TRANS_ID_BITS 位宽
lsu_valid_i		
operator_i		fu_op 枚举类型
operand_a_i		64 位
operand_b_i		64 位
no_st_pending_o		
fu_i		fu_t 类型
imm_i		64 位
commit_i		
enable_translation_i		
en_ld_st_translation_i		
icache_areq_i		
priv_lvl_i		riscv::priv_lvl_t 类型
ld_st_priv_lvl_i		riscv::priv_lvl_t 类型
sum_i		
mxr_i		
satp_ppn_i		44 位
asid_i		ASID_WIDTH 位宽
flush_tlb_i		
dcache_req_ports_i		dcache_req_o_t [2:0]
lsu_result_o	Output	64 位
lsu_valid_o		
lsu_ready_o		
lsu_trans_id_o		TRANS_ID_BITS 位宽
commit_ready_o		
icache_areq_o		
itlb_miss_o		
dtlb_miss_o		
dcache_req_ports_o		dcache_req_i_t [2:0]
lsu_exception_o		exception_t 类型

12.4.1 mmu

该模块

12.4.1 store_unit

12.4.1 load_unit

12.4.1 lsu_arbiter

12.4.1 lsu_bypass

12.5 子模块 5——csr_buffer

该模块实

接口信号与代码说明

Table-1 signals

singal	**I/O**	**Description**
clk_i，rst_ni，flush_i	Input	
trans_id_i	Input	TRANS_ID_BITS 位宽
csr_valid_i		
operator_i		fu_op 枚举类型
operand_a_i		64 位
operand_b_i		64 位
commit_i		
csr_result_o	Output	64 位
csr_valid_o		
csr_ready_o		
csr_trans_id_o		TRANS_ID_BITS 位宽
csr_addr_o		12 位

13 commit_stage 接口信号与代码说明

所有指令的提交请求是由 issue 级发给 commit 级的，这样能够保证 issue 指令和 commit 指令都是相同顺序。

在 commit 级的仲裁逻辑中，只要没发生异常中断或者 debug，则允许正常的提交请求。

FENCE 和 FENCE.I 指令在 commit 级执行，因为它们涉及到 flush cache 操作，也因此这种操作是时间开销高昂的。

commit 级的异常处理就是，标记异常原因和对应的**处于 commit 阶段**的指令 pc 值，转发给 controller 模块和 csr regfile 模块。这个 pc 值很重要，异常结束后，要从这个 pc 值恢复流水线。

接口信号与代码说明

Table-1 signals

singal	I/O	Description
clk_i,	Input	只有时钟信号
halt_i,	Input	由 controller 发出，要求暂停 cpu 工作，也不再提交
flush_dcache_i,		由 controller 发出
exception_o,	Output	输出异常信息
debug_mode_i,	Input（debug）	debug 时，不允许提交指令
debug_req_i,		
single_step_i,		
commit_instr_i,	与 issue 级交互	[NR_COMMIT_PORTS-1:0]，scoreboard_entry_t 类型来自 issue 级的请求提交的指令执行结果
commit_ack_o,		[NR_COMMIT_PORTS-1:0]
waddr_o,	Output（写回通用寄存器）	[NR_COMMIT_PORTS-1:0][4:0]
wdata_o,		[NR_COMMIT_PORTS-1:0][63:0]
we_o,		[NR_COMMIT_PORTS-1:0]
pc_o,		[63:0]
commit_csr_o,	与 csr regfile 交互	允许 csr 指令提交
csr_op_o,		
csr_wdata_o,		[63:0]
csr_rdata_i,		[63:0]
csr_exception_i,		
commit_lsu_o,	与 ex 级交互	允许 lsu 指令提交
commit_lsu_ready_i,		
no_st_pending_i,		
fence_i_o,		
fence_o,		
sfence_vma_o	Output	

14 csr_regfile 接口信号与代码说明

定义了一系列必要的 CSR 寄存器，并规定了它们的读写规则。

接口信号与代码说明

Table-1 signals

singal	I/O	Description
clk_i，rst_ni	Input	
flush_o		
time_irq_i,	input	
halt_csr_o,	output	
commit_instr_i	input 来自 issue 级	scoreboard_entry_t
commit_ack_i	input 来自 commit 级	
core_id_i,	input	[3:0]
cluster_id_i,	input	[5:0]

boot_addr_i,	input	[63:0]
ex_i,	input	exception_t
csr_op_i,	input	fu_op
csr_addr_i,	input	[11:0]
csr_wdata_i,	input	[63:0]
csr_rdata_o,	output	[63:0]
pc_i,	input	[63:0]
csr_exception_o,	output	exception_t
epc_o,	output	[63:0]
eret_o,	output	
trap_vector_base_o,	output	[63:0]
priv_lvl_o,	output	riscv::priv_lvl_t
en_translation_o,	output	
en_ld_st_translation_o,	output	
ld_st_priv_lvl_o,	output	riscv::priv_lvl_t
sum_o,	output	
mxr_o,	output	
satp_ppn_o,	output	[43:0]
asid_o,	output	[ASID_WIDTH-1:0]
irq_i,	input	[1:0]
ipi_i,	input	
debug_req_i,	input	
set_debug_pc_o,	output	
tvm_o,	output	
tw_o,	output	
tsr_o,	output	
debug_mode_o,	output	
single_step_o,	output	
icache_en_o,	output	
dcache_en_o,	output	
perf_addr_o,	output	[11:0]
perf_data_o,	output	[63:0]
perf_data_i,	input	[63:0]
perf_we_o	output	

15 perf_counters

IO 口中只有一个输出连到 CSR 寄存器，其余全部是输入信号，有时钟和复位信号。

主要功能是记录各类型指令执行次数，异常次数，cache 和 tlb 的 miss 次数。

16 controller 接口信号与代码说明

主要功能是控制 flush 流水线中的哪些模块，发出 flush 信号。

对于条件分支指令，来自 ex 级的输入信号 resolved_branch_i 如果表示分支预测错误，controller 就发出针对 if 级和未发射指令的 flush 信号。

接口信号与代码说明

Table-1 signals

singal	**I/O**	**Description**
clk_i，rst_ni	Input	
flush_dcache_ack_i		
halt_csr_i		
eret_i		
ex_valid_i		
set_debug_pc_i		
resolved_branch_i		branchpredict_t
flush_csr_i		
fence_i_i		
fence_i		
sfence_vma_i		
set_pc_commit_o	output	
flush_if_o		接入到 frontend 级和 id 级的 flush_i
flush_unissued_instr_o		接入到 issue 级的 flush_unissued_instr_i
flush_id_o		
flush_ex_o		
flush_icache_o		
flush_dcache_o		
flush_tlb_o		
halt_o		

17 源代码中的其他部分

关于核的外围部分，它通过 AXI 总线模块外联了 memory 和 ROM。参照 SiFive 公司设计的 debug 标准，设计了 debug 硬件模块。

2018 年 10 月份左右，设计团队开源了浮点运算单元的设计。