CVA6: An application class RISC-V CPU core

Commit Stage

Translation Lookaside Buffer

Shared Translation Lookaside Buffer

Page Table Walker

文档链接

CVA6 项目的目标是创建一个具有生产质量的开源应用级 RISC-V CPU 内核系列。 CVA6 针对 ASIC 和 FPGA 实现，但单个内核可能针对特定的实现技术。 CVA6 采用 SystemVerilog 编写，具有很强的可参数化能力。 例如，参数可以将 ILEN 设置为 32 位或 64 位，还可以启用/禁用浮点支持。

CV 前缀表明它是 CORE-V 系列的成员，而 A6 则表明它是具有六级执行流水线的应用级处理器。 不过，CVA6 的 "原样 "并不是为了实现特定的生产内核。 相反，CVA6 将成为许多应用类内核的基础。 这些内核的命名规则是

```
CV <ILEN> <class> <# of pipeline stages> <product identifier>
```

因此，CV64A60 将是一个具有六级流水线的 64 位应用内核。 请注意，在本例中，产品标识符为 "0"。

The CVA6 Design Document describes in detail the **CVA6**, the code base that can be used to compile/synthesize a specific core instance (e.g. cv32a65x).

CVA6 Design Document (deprecated)

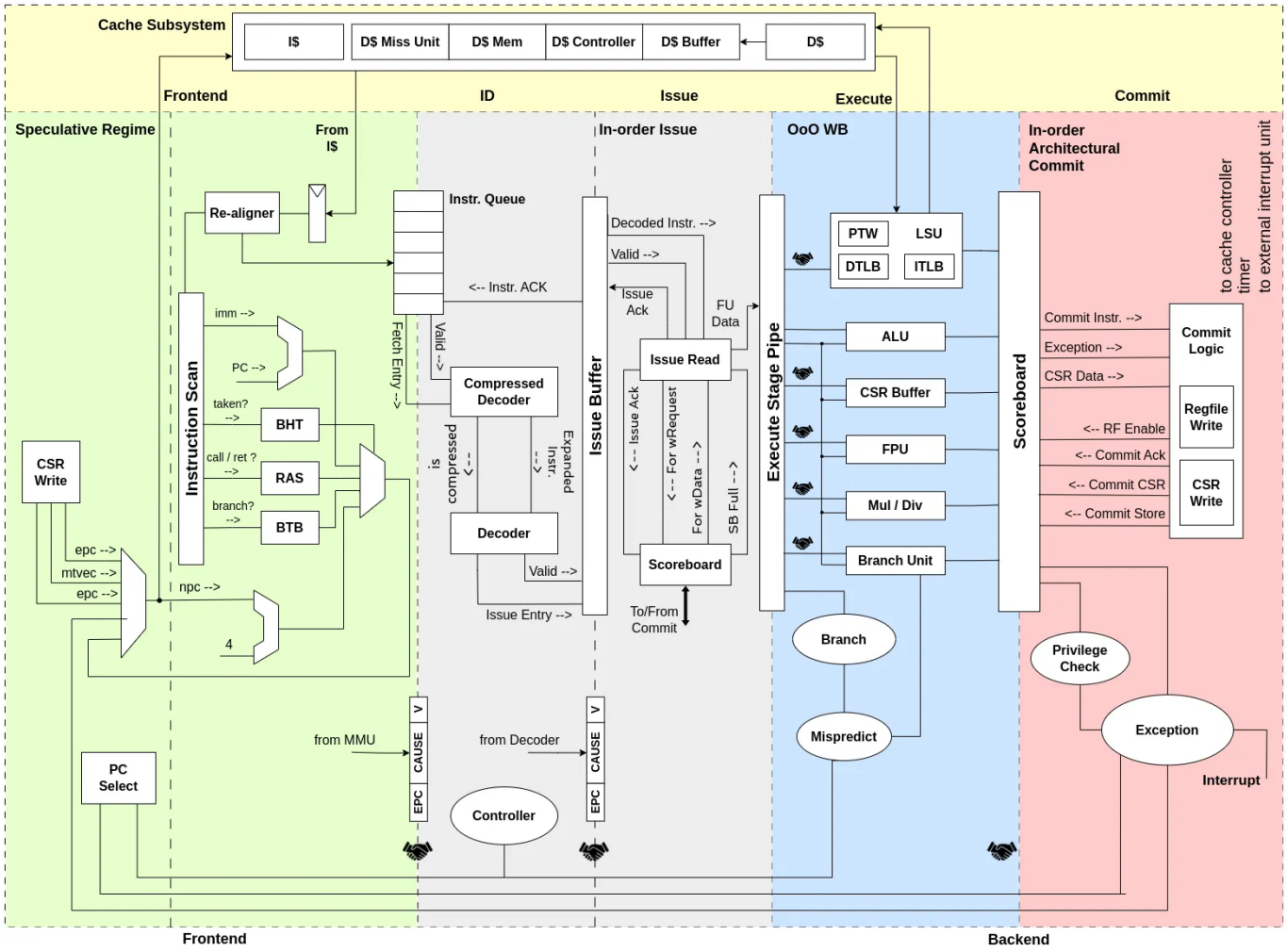
Introduction

本文件介绍了实现 64 位 RISC-V 指令集的 6 级单发 Ariane CPU。 它完全实现了《第一卷：用户级 ISA V 2.1》中规定的 I、M 和 C 扩展，以及特权扩展 1.10 草案。 它实现了 M、S、U 三个权限级别，完全支持类 Unix 操作系统。

Scope and Purpose

该内核的目的是以合理的速度和 IPC 运行完整的操作系统。 为了达到必要的速度，该内核采用了 6 级流水线设计。 为了提高 IPC，CPU 采用了记分板，通过发布与数据无关的指令来隐藏数据 RAM（高速缓存）的延迟。 指令 RAM（或 L1 指令高速缓存）的访问延迟为 1 个周期，而数据 RAM（或 L1 数据高

速缓存）的访问延迟较长，为 3 个周期。



PC Generation

PC gen 负责生成下一个程序计数器。 所有程序计数器都是逻辑寻址。 如果逻辑到物理映射发生变化， `fence.vm` 指令应清空流水线和 TLB。
该阶段包含对分支目标地址的推测以及分支是否被执行的信息。 此外，它还包含分支目标缓冲区 (BTB) 和分支历史表 (BHT)。

如果 BTB 将某个 PC 解码为跳转，则由 BHT 决定是否执行分支。 由于存在各种充满状态的内存组件，这一阶段被分为两个流水线阶段。 PC Gen 通过握手信号与 IF 通信。 指令取回通过断言 ready 信号来表示准备就绪，而 PC Gen 则通过断言 `fetch_valid` 信号来表示请求有效。

The next PC can originate from the following sources (listed in order of precedence):

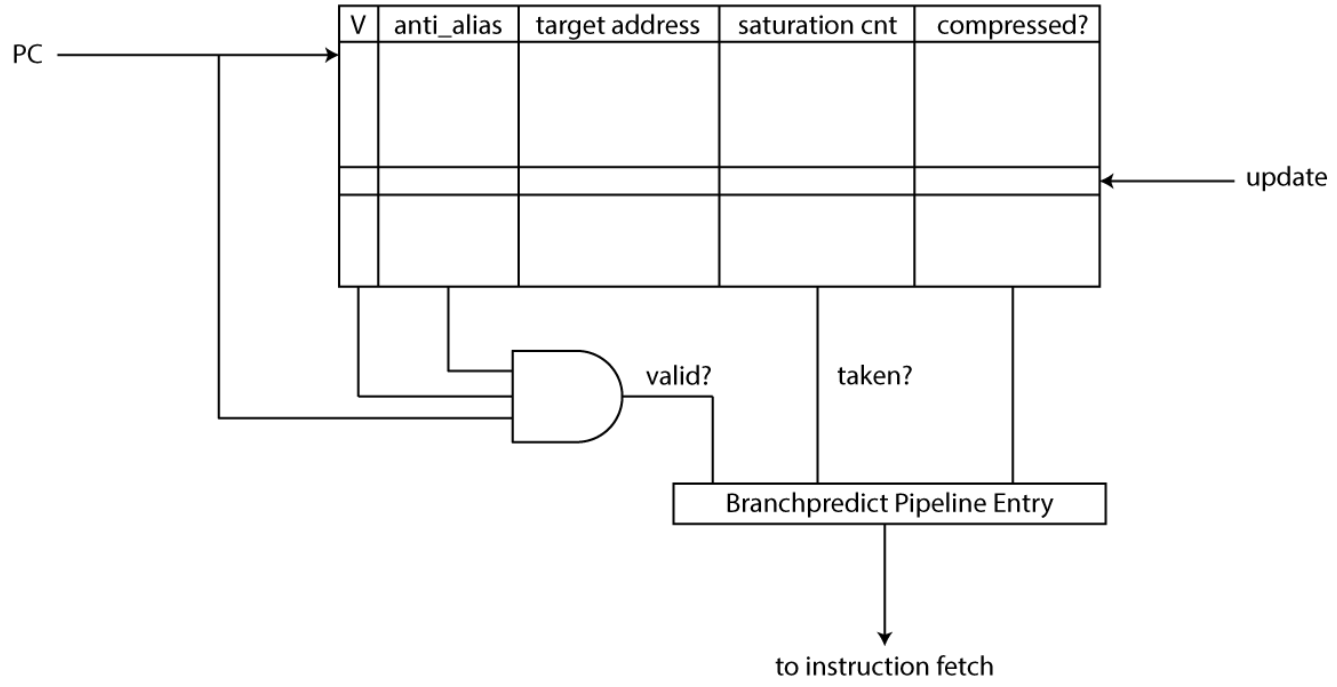
1. `PC+4`
2. 分支预测：如果 BHT 和 BTB 预测在某个 PC 上会出现分支，PC Gen 就会将下一个 PC 设置为预测的地址，同时通知 IF 阶段对该 PC 进行了预测。 在流水线下游的不同地方都需要这

样做（例如纠正预测）。 向下传递的分支信息封装在一个名为 `branchprrdict_sbe_t` 的结构中。 与此相反，在流水线上传递的分支预测信息只被称为 `bp_resolve_t` 。 它用于纠正操作（见下一要点）。 这种命名约定应能使源代码中的分支信息流更容易被发现。

3. 控制流更改请求： 控制流更改请求是由于分支预测器预测错误而产生的。 这可能是一个 "真正 "的错误预测，也可能是一个未被识别的分支。 无论如何，我们都需要纠正我们的操作，并开始从正确的地址获取数据。
4. 从环境调用返回： 从环境调用返回时，将对 PC 执行修正操作，将后续 PC 设置为 `[m|s]epc` 寄存器中存储的 PC。
5. 异常/中断： 如果出现异常（或中断，在 `RISC-V` 系统中非常相似）， `PC Gen` 将生成下一个 `PC` ，作为陷阱向量基地址的一部分。 捕获向量基地址可能会有所不同，这取决于异常是捕获到 S 模式还是 M 模式（目前不支持用户模式异常）。 `CSR` 单元的作用是找出陷阱的位置，并将正确的地址提交给 `PC Gen` 。
6. 因 `CSR` 副作用而刷新流水线： 当写入具有副作用的 `CSR` 时，我们需要刷新整个流水线，并重新从下一条指令开始获取，以便将最新信息考虑在内（例如虚拟内存基指针的变化）。
7. Debug： `调试的优先级最高，因为它可以中断任何控制流请求` 。 调试也是唯一能与其他强制控制流更改同时发生的控制流更改源。 调试单元报告更改 PC 的请求以及 CPU 应更改到的 PC。

该单元还负责处理一个名为 `fetch_enable` 的信号，该信号的作用是防止在未断言的情况下获取数据。 还需注意的是，该单元不进行刷新。 所有冲洗信息都由控制器发布。 实际上，控制器的唯一目的就是刷新不同的流水线阶段。

Branch Prediction



所有分支预测数据结构都位于一个类似寄存器文件的数据结构中。 它以 PC 中相应的位数为索引，包含有关预测目标地址的信息以及可配置宽度饱和计数器（默认为 2）的结果。 预测结果将用于后续阶段的跳转（或不跳转）。

除了提供预测结果之外，BTB 还会更新其错误预测的信息。它可以纠正饱和计数器或清除分支预测条目。后者是在分支单元发现预测的 PC 不匹配或提交具有特权更改副作用的指令时完成的。

分支结果和分支目标地址在同一个功能单元中计算，因此对目标地址的错误预测与对分支决策的错误预测一样代价高昂。由于分支单元（执行所有分支处理的功能单元）在时间方面已经非常关键，因此这是一个潜在的改进。

由于 Ariane 完全实现了压缩指令集，分支也可能发生在 16 位（或半字）指令上。由于这会显著增加 BTB 的大小，因此 BTB 用字对齐的 PC 进行索引。这带来了潜在的缺点，即分支预测总是会对包含两个压缩分支的指令提取字进行错误预测。然而，这种情况在现实中应该很少见。

我们在这里使用的一个技巧是，取未对齐指令的下一个 PC（例如：该指令高 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，它将获取另外两个未使用的字节，然后由指令重新对齐器丢弃。因此，我们还需要保留一个额外的位，无论指令是在低 16 位还是高 16 位。

对于分支预测，不必要的流水线气泡的潜在来源是混叠。为了防止发生混叠（或至少使其不太可能发生），每次访问时都会使用并比较几个标记位（索引 PC 的高位）。这是一种必要的权衡，因为我们缺乏足够快的 SRAM 来承载 BTB。相反，我们被迫使用对整体面积和功耗有更大影响的寄存器。

以前的下一个

Instruction Fetch Stage

指令提取阶段 (IF) 从 PC Gen 阶段获取信息。此信息包括有关分支预测（是否是预测分支？目标地址是哪个？是否预测会执行？）、当前 PC（如果是连续提取，则字对齐）以及此请求是否有效的信息。IF 阶段要求 MMU 对请求的 PC 进行地址转换并控制 I$（或只是指令存储器）接口。指令存储器接口在 中有更详细的描述。

指令取回的微妙之处在于它对时间的要求非常高。 由于这一原因，我们无法实施更复杂的握手协议（因为往返时间太长）。 因此，IF 阶段会向 I$ 接口发出信号，表示希望向内存提出取回请求。 根据高速缓存的状态，该请求可能被批准，也可能不被批准。 如果允许，指令获取阶段会将请求放入内部fifo。 之所以需要这样做，是因为它必须随时知道有多少事务尚未处理。 这主要是由于指令提取是在非常投机(very speculative)的基础上进行的，因为要进行分支预测。 在这种情况下，控制器可能会决定刷新指令取回阶段，从而需要丢弃所有未完成的事务。

目前的实现方式允许最多有两个未处理事务。 一旦从内存返回有效应答（且请求未因刷新而被视为过期），应答将与取回地址和分支预测信息一起放入 FIFO。

除了来自内存的应答，MMU 还会发出潜在异常信号。 因此，这是可能发生异常的第一个地方（总线错误、无效访问和指令页面故障）。

Fetch FIFO

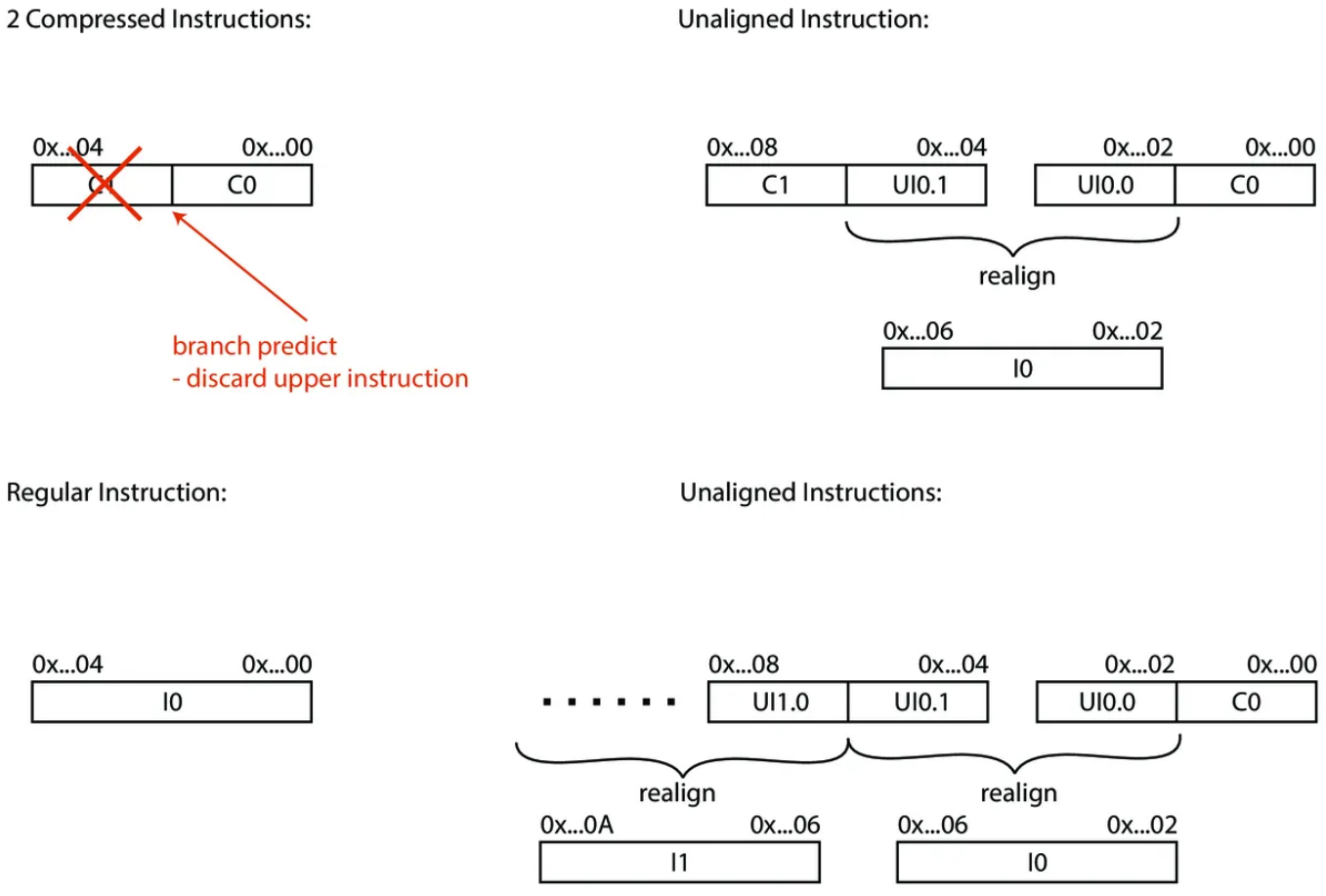
取指 FIFO 包含指令存储器中所有请求的（有效）取指。 FIFO 目前有一个写端口和两个读端口（其中只使用了一个）。 在未来的实现中，第二个读取端口有可能被用来实现宏操作融合，或扩大发射接口以涵盖两条指令。

Instruction Decode

指令解码是处理器后端的第一个流水线阶段。 它的主要目的是从 IF 阶段获得的数据流中提炼指令，对其进行解码并将其发送到发射阶段。

随着压缩指令（一般是可变长度指令）的引入，ID 阶段变得有点复杂： 它必须在输入的数据流中搜索潜在的指令，重新对齐，并（在压缩指令的情况下）解压缩。 此外，在这一阶段结束时，我们将知道解码后的指令是否是分支指令，因此它会将这一信息传递给发射阶段。

Instruction Re-aligner



如上所述，指令重新对齐器会检查输入数据流中是否有压缩指令。 压缩指令的最后一位不等于 11，而普通 32 位指令的最后两位设置为 11。 主要的复杂之处在于，压缩指令会使普通指令不对齐（例如：指令从半字边界开始）。 这可能（在最糟糕的情况下）要求在指令完全解码前进行两次内存访问。 因此，我们需要确保取指 FIFO 有足够的空间来保存指令的第二部分。 因此，指令重新对齐器需要跟踪前一条指令是未对齐的还是压缩的，以便正确决定如何处理即将到来的指令。

此外，分支预测信息只用于向问题阶段输出正确的指令。 由于我们只对字对齐的 PC 进行预测，因此在存在两条指令（压缩或未对齐）的情况下，需要对传递的分支预测信息进行调查，以确定我们实际需要哪条指令。 这意味着我们可能需要放弃两条指令中的一条（分支目标前的指令）。 因此，指令重新对齐器还需要检查该取值条目是否包含一个有效的分支。 根据是否在高 16 位上预测到分支，它必须相应地放弃低 16 位。 这一过程如图所示。

Compressed Decoder

如前所述，我们还需要解压缩所有压缩指令。 这是通过一个小型组合电路来完成的，该电路将 16 位压缩指令扩展为 32 位等效指令。 所有压缩指令都有一个 32 位等效指令。

Decoder

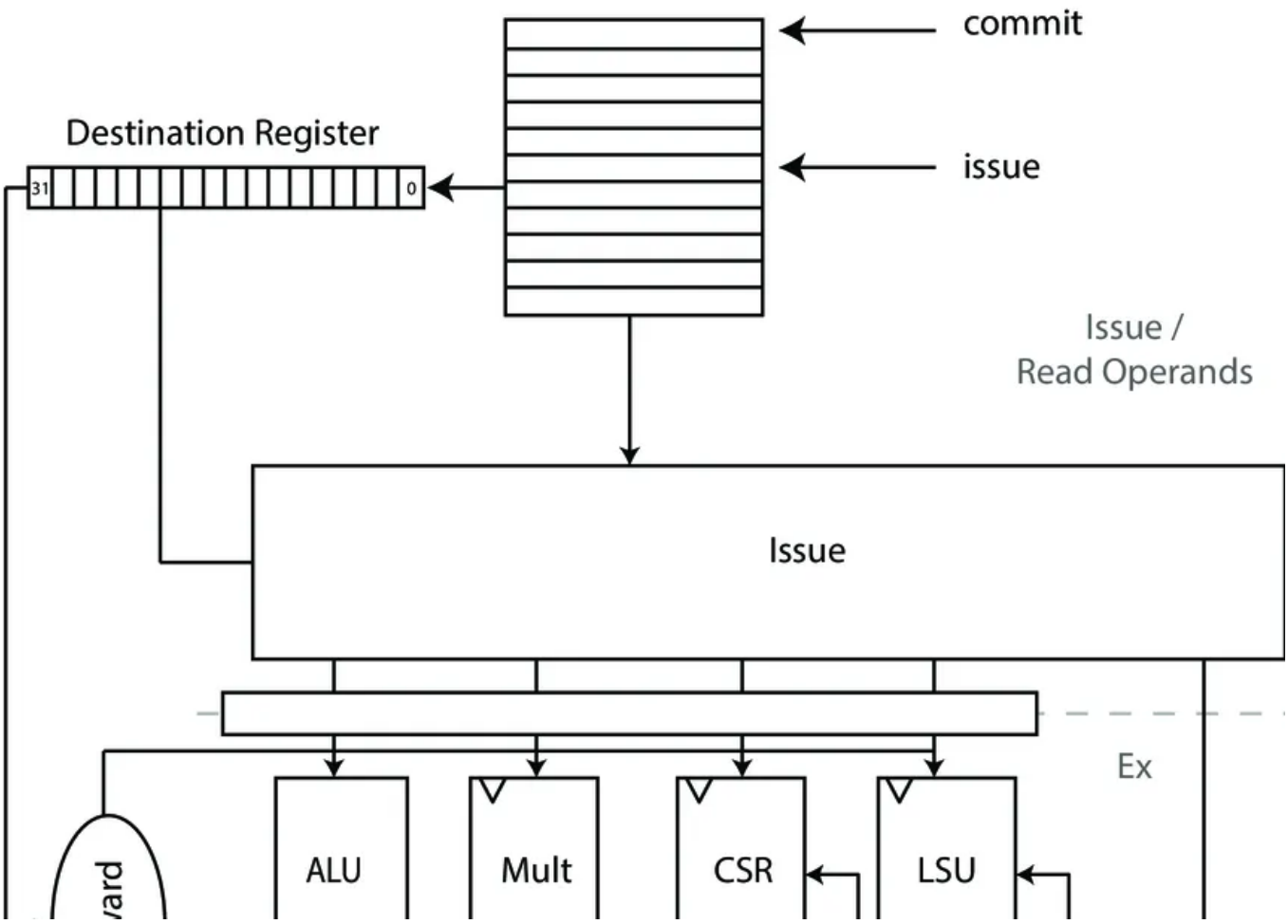
解码器接收原始指令数据或未压缩的 16 位指令等效数据，并进行相应解码。 它将原始比特转换为阿里安中最基本的控制结构，即记分板条目：

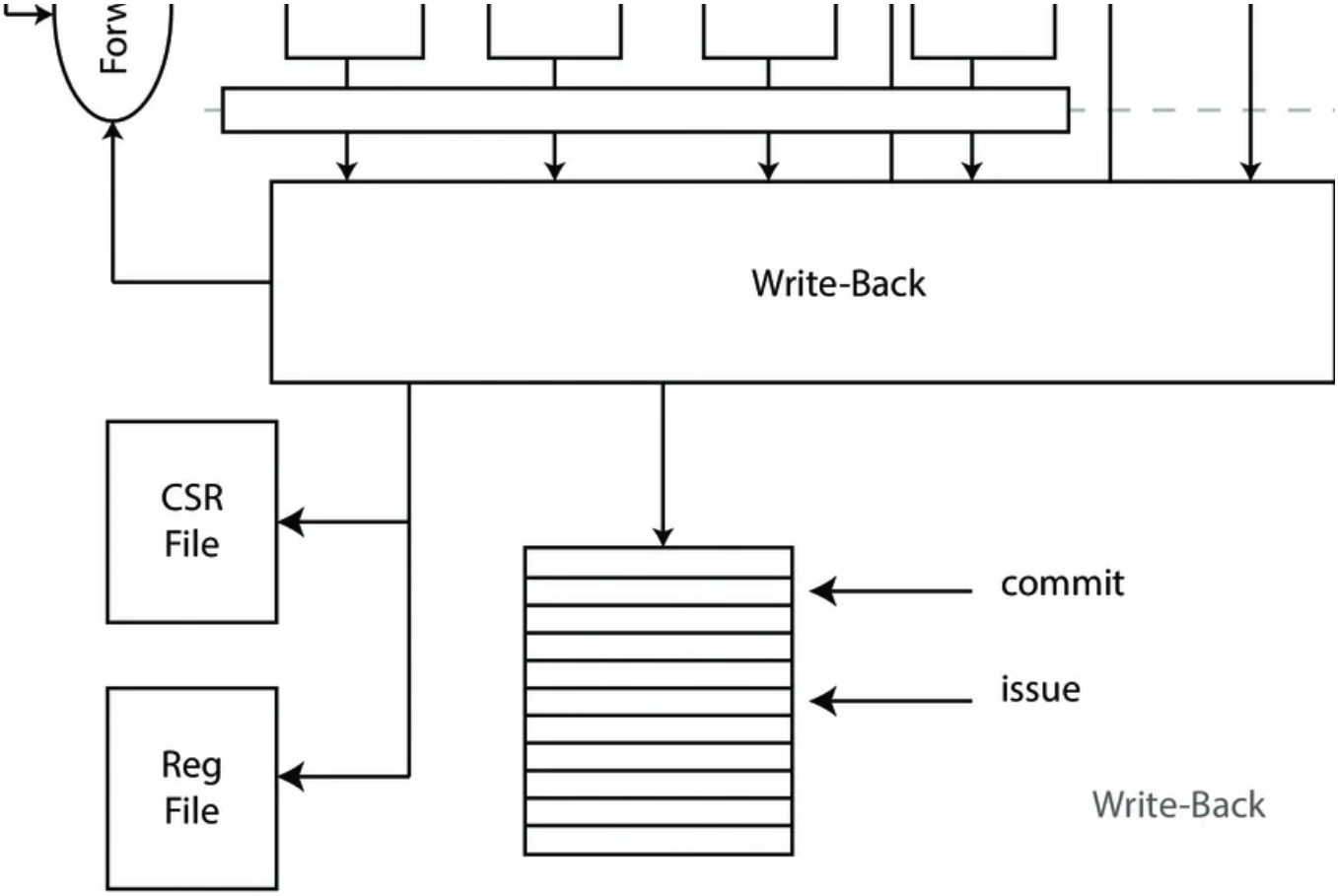
- **PC**: PC of instruction
- **FU**: functional unit to use
- **OP**: operation to perform in each functional unit
- **RS1**: register source address 1
- **RS2**: register source address 2
- **RD**: register destination address
- **Result**: for unfinished instructions this field also holds the immediate
- **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`

它在流水线的下游被逐步处理。 记分板条目控制着操作数的选择、分派和执行。 此外，它还包含一个异常条目，将特定指令与其潜在异常紧密联系在一起。 由于可能出现异常的第一次是在 IF 阶段，因此解码器也会确保将该异常记录到记分板条目中。 解码过程中可能会出现非法指令异常。 在这种情况下，如果之前没有发生过异常，解码器将设置相应的异常字段和故障位（在 `[s|m]tval` 中）。 由于这并不是唯一可能发生非法指令异常的地方，而且非法指令异常总是要求在 `[s|m]tval` 字段中设置故障地址，因此该字段会在这里被设置。 但前提是指令取回还没有为这条指令抛出异常。

Issue Stage

发射阶段的目的是接收解码指令，并将其发送给各个功能单元。 此外，发射级还负责跟踪所有已发布指令、功能单元状态，并接收执行级的回写数据。 此外，它还包含 CPU 的寄存器文件。 通过使用一种名为记分板的数据结构（参见），它可以准确地知道哪些指令被发出、指令在哪个功能单元中执行以及指令将回写到哪个寄存器中。 如前所述，执行过程大致可分为四个部分：1.发出指令；2.读取操作数；3.执行指令；4.回写指令。 发射阶段处理第一步、第二步和第四步。





Issue

当发射阶段收到一条新的解码指令时，它会检查所需功能单元是否空闲或在下一个周期内是否空闲。 然后检查其源操作数是否可用，以及是否没有其他当前已发出的指令会写入相同的目标寄存器。 此外，它还会跟踪是否有未解决的分支指令被发出。 后者主要是为了简化硬件设计。 通过只允许一个分支，如果我们后来发现在该分支上预测错误，就可以很容易地进行回溯。

通过确保记分板只允许一条指令写入某个目的寄存器，可以大大简化转发路径的设计。 记分板有一个组合电路，可以输出所有 32 个目标寄存器的状态以及产生结果的功能单元。 这个信号称为 `rd_clobber` 。

发射级与各功能单元独立通信。 这尤其意味着，它必须监控各功能单元的ready和valid，无条件接收和存储回写数据。 它总是有足够的空间，因为每发出一条指令，它都会在记分板上分配一个插槽。 这就解决了较小微处理器在结构上的潜在隐患。 这种模块化设计还有助于探索更先进的发行技术，如乱序发射（）。

指令的发射是按顺序进行的，这意味着程序流程的顺序会自然保持。 每个功能单元的回写可能会超出顺序。 举例来说，发射级发出一条加载指令，需要 n 个时钟周期才能产生有效结果。 在下一个周期，发

射级发出一条 ALU 指令，如加法运算。 加法运算只需要一个时钟周期，因此会在加载结果就绪之前返回。 因此，我们需要为各个发射阶段分配 ID。 ID 类似于记分板存储该指令结果的（唯一）位置。 该 ID（称为事务 ID）有足够的位数来唯一代表记分板中的每个插槽，需要与其他数据一起传递给相应的功能单元。

这种方案允许功能单元完全独立于发射逻辑运行。它们可以以不同的顺序返回不同的交易。只要相应的 ID 与结果一起发出信号，记分板就会知道将它们放在哪里。这种方案甚至允许功能单元缓冲结果并完全无序地处理它们（如果这对它们有意义的话）。这是如何有效地分离处理器的不同模块的另一个例子。

读取操作数

读取操作数在物理上与发射指令在同一周期内发生，但从概念上可以将其视为另一个阶段。由于记分板知道哪些寄存器正在被写入，因此它可以在必要时处理这些操作数的转发。设计目标是连续执行两个 ALU 指令（例如：中间没有气泡）。操作数来自寄存器文件（如果记分板中当前没有其他指令会写入该寄存器）或由记分板转发（通过查看信号 `rd_clobber`）。

操作数选择逻辑是一种经典的优先级选择，它优先考虑记分板的结果，而不是 `regfile` 的结果，因为功能单元将始终产生最新的结果。为了获得正确的寄存器值，我们需要轮询记分板中的两个源操作数。

记分牌

记分板被实现为一个 FIFO，带有一个读取端口和一个写入端口，并带有有效和确认信号。除此之外，它还提供上述信号，这些信号会告诉 CPU 的其余部分哪些寄存器将被先前安排的指令破坏。如果记分板尚未填满，指令解码会直接写入记分板。提交阶段会查找已经完成的指令并更新架构状态。这意味着要么发生异常，要么更新寄存器或 CSR 文件。

Execute Stage

执行阶段是一个逻辑阶段，它封装了所有功能单元 (FU)。目前，FU 不应具有单元间依赖性，例如：每个 FU 必须能够独立于其他每个单元执行其操作。每个功能单元都维护一个有效信号，它将使用该信号发送有效输出数据，并维护一个就绪信号，该信号告知发出逻辑是否能够接受新请求。此外，正如在关于指令发射 () 的部分中简要解释的那样，它们还接收一个唯一的事务 ID。功能单元应该将此事务 ID 与有效信号和结果一起返回。在撰写本文时，执行阶段包含一个 ALU、一个分支单元、一个加载存储单元 (LSU)、一个 CSR 缓冲区和一个乘法/除法单元。

算术逻辑单元

算术逻辑单元 (ALU) 是一个小型硬件，可执行 32 位和 64 位减法、加法、移位和比较。它始终在单个周期内完成其操作，因此不包含任何状态满元素。它的就绪信号始终处于置位状态，它只是将事务 ID 从其输入传递到其输出。除了两个操作数外，它还接收一个运算符，该运算符告诉它要执行哪个操作。

分支单元

分支单元的目的是管理所有类型的控制流变化，即：条件和无条件跳转。它通过提供一个加法器来计算目标地址，并提供一些比较逻辑来决定是否进行分支。此外，它还决定分支是否被错误预测，并向 PC Gen 阶段报告纠正措施。纠正措施包括更新 BHT 和设置 PC（如有必要）。由于任何指令（包括根本没有跳转的指令 – 请参阅 PC Gen 部分中的混叠问题）都可能预测到跳转，因此它需要知道何时向功能单元发出指令并监视分支预测信息。如果在非分支指令上意外预测到分支，它也会采取纠正措施并将 PC 重新设置为正确地址（取决于指令是否被压缩，它会添加 PC `+ 2` 或 PC `+ 4`）。

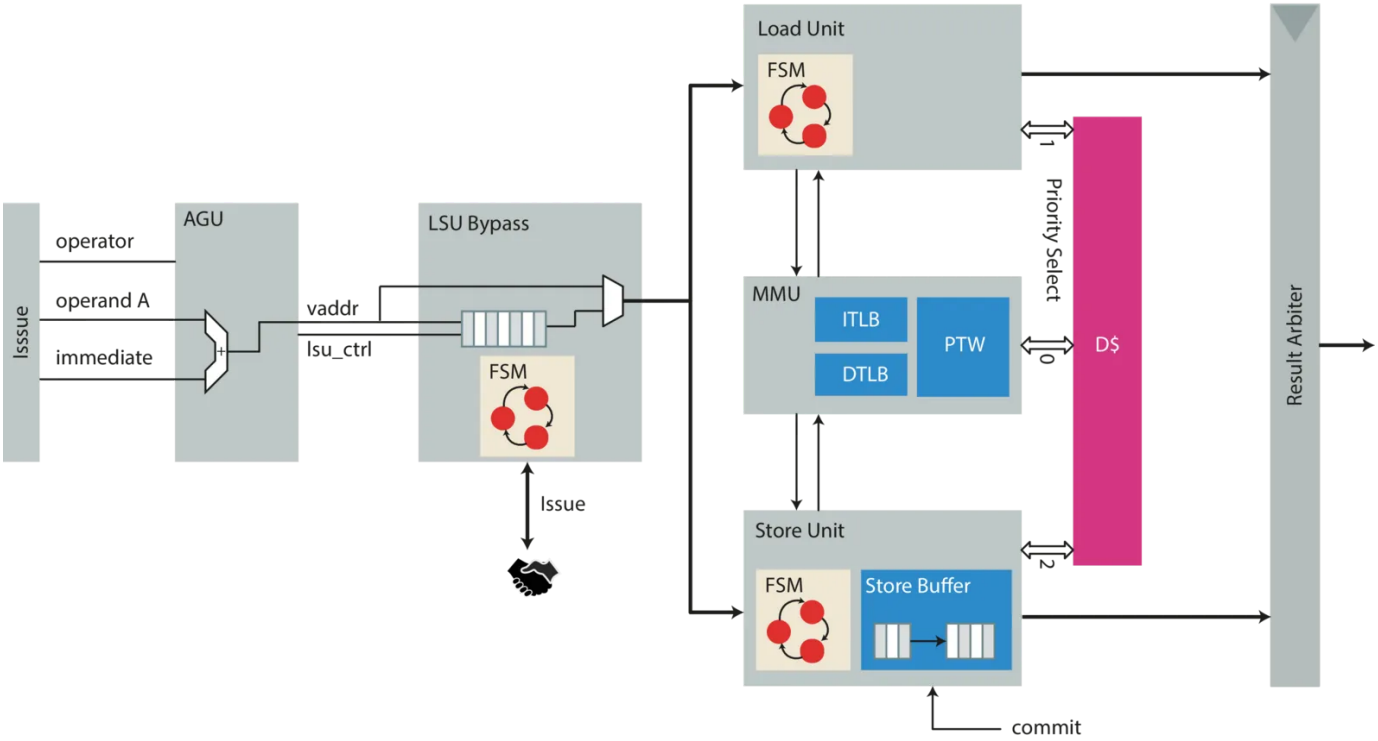
正如在关于指令重新对齐的部分中简要提到的，分支单元将未对齐的 32 位指令的 PC 放置在高 16 位上（例如：在新字边界上）。此外，如果指令被压缩，它也会对报告的预测产生影响，因为如果预测发生在低 16 位上（例如：低压缩指令），它需要设置一个位。

可以看出，这一切都为这一阶段增加了很多昂贵的操作，主要是比较和添加。因此，分支单元处于整体设计的关键路径上。尽管如此，我们还是选择将分支保持在单周期操作中。不过，在未来的版本中，拆分这条路径可能是有意义的。这将给整体设计带来一些昂贵的 IPC 影响，主要是因为目前的限制是，如果没有未解决的分支，记分板只会接受新指令。使用单周期操作，所有分支都会在

同一个问题周期内得到解决，这不会引入任何管道停顿。
正如在关于指令重新对齐的部分中简要提到的，分支单元将未对齐的 32 位指令的 PC 放置在高 16 位上（例如：在新字边界上）。此外，如果指令被压缩，它也会对报告的预测产生影响，因为如果预测发生在低 16 位上（例如：低压缩指令），它需要设置一个位。

可以看出，这一切都为这一阶段增加了很多昂贵的操作，主要是`comparison and additions`。。因此，分支单元处于整体设计的关键路径上。尽管如此，我们还是选择将分支保持在单周期操作中。不过，在未来的版本中，拆分这条路径可能是有意义的。这将给整体设计带来一些昂贵的 IPC 影响，主要是因为目前的限制是，如果没有未解决的分支，记分板只会接受新指令。使用单周期操作，所有分支都会在同一个问题周期内得到解决，这不会引入任何流水线停顿。

Load Store Unit (LSU)



加载存储单元与其他功能单元类似。 此外，它还必须管理与数据存储器 (D$) 的接口。 特别是，它包含 DTLB（数据转换旁侧缓冲区）、硬件页表走行器（PTW）和内存管理单元（MMU）。 它还负责在加载、存储和 PTW 之间仲裁对数据内存的访问--优先处理 PTW 查找。 这样做是为了尽快解决 TLB 未命中问题。
只要记分板不发出提交信号，LSU 就可以立即发出加载请求，而存储则需要保留： 这样做是因为整个处理器的设计只有一个提交点。 由于向内存层次结构发布加载操作不会产生任何语义副作用，因此 LSU 可以立即发射加载操作，这与存储的性质完全不同。 存储会改变架构状态，因此会

被放置在存储缓冲区中，并在稍后的提交阶段进行提交。 有时，这也被称为 "发射存储"（posted-store），因为存储请求会发布到存储队列中，一旦提交信号变为高电平，且内存接口未被使用，存储请求就会等待进入内存层次结构。
因此，在加载时，LSU 还需要检查存储缓冲区是否存在潜在的别名。 如果发现未提交的数据，它就会停滞，因为它无法满足当前的请求。

这意味着：

允许向同一地址装载两批货物，并按发发射顺序返回。

允许对同一地址进行两次存储。 只要记分板没有发出提交信号，它们就会按顺序由记分板发射，并按顺序存储在存储缓冲区中。 只有在存储已经提交（在存储缓冲区中标记为已提交）的情况下，才能执行存储后加载到同一地址的指令。 否则，LSU 就会停滞，直到记分板提交指令。 我们无法保证存储最终会被提交（例如：出现异常）。

目前，LSU 不处理错位访问。 具体来说，对于双字访问，这意味着访问边界未对齐至 64 位；对于字访问，这意味着访问边界未对齐至 32 位；对于半字访问，这意味着访问边界未对齐至 16 位。 如果遇到这样的加载或存储，它将抛出错位异常，并让异常处理程序解决加载或存储问题。 除了错位异常外，它还会抛出页面故障异常。

为便于通量核实系统的设计，该系统分为 6 个主要部分，下文将对每个部分进行详细介绍：

LSU Bypass {#par:lsu_bypass}

LSU 旁路模块是一个辅助模块，用于管理 LSU 状态信息（满标志等），并将其提交至发射阶段。 这是必要的，原因如下： LSU 的设计在大多数方面都至关重要，因为它直接连接相对较慢的 SRAM。 此外，它还需要依次执行一些代价高昂的操作。 成本最高的操作（就时间而言）是地址生成、地址转换和检查存储缓冲区是否存在潜在的别名。 因此，只有在很晚的时候才能知道当前的加载/存储是否能进入内存，或者是否需要额外的周期。 其中，存储缓冲区的别名和 TLB 错失是最突出的问题。 由于问题阶段依赖于就绪信号来分派新指令，这将导致路径过长，从而在某些情况下大大降低整个设计的速度。

此外，加载单元还需要执行地址转换。 它使用虚拟索引和物理标记的 D$ 访问方案，以减少负载访问所需的周期数。 由于负载可能会阻塞 D$，因此必须终止内存接口上的当前请求，以便为高速缓存侧的硬件 PTW 让路。 一些更先进的缓存基础设施（如无阻塞缓存）可以缓解这一问题。

Store Unit {#par:store_unit}

存储单元管理所有存储。 它通过计算目标地址和设置适当的字节启用位来实现。 此外，它还执行地址转换，并与加载单元通信，以查看是否有任何加载与其缓冲区中的未完成存储相匹配。 存储单元的大部分业务逻辑都位于存储缓冲区中，下一节将详细介绍存储缓冲区。

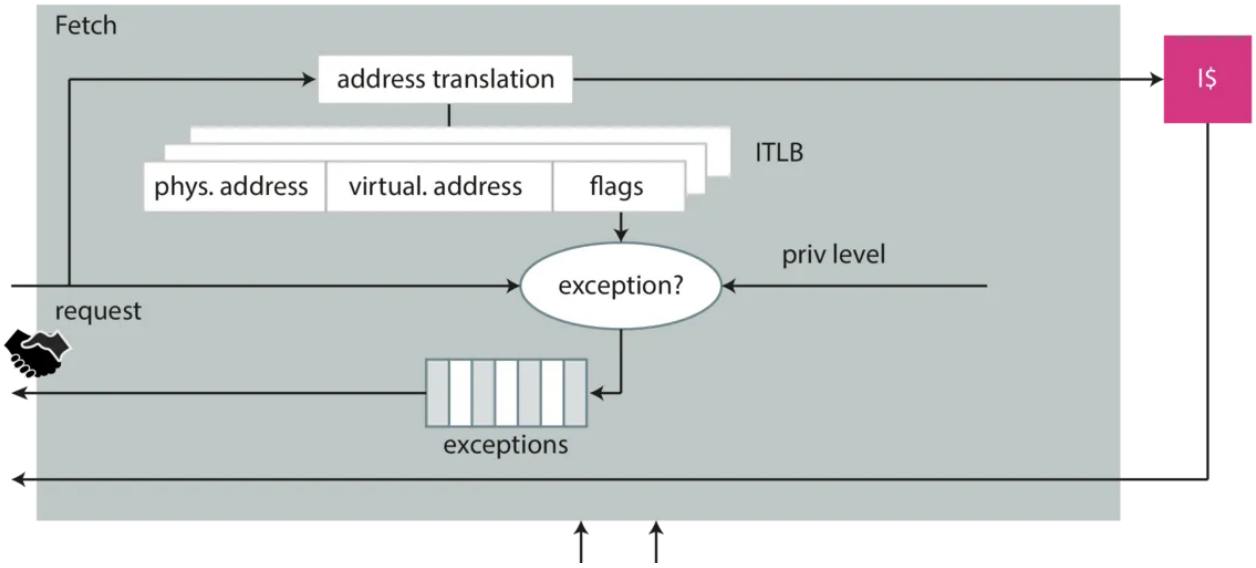
Store Buffer {#par:store_buffer}

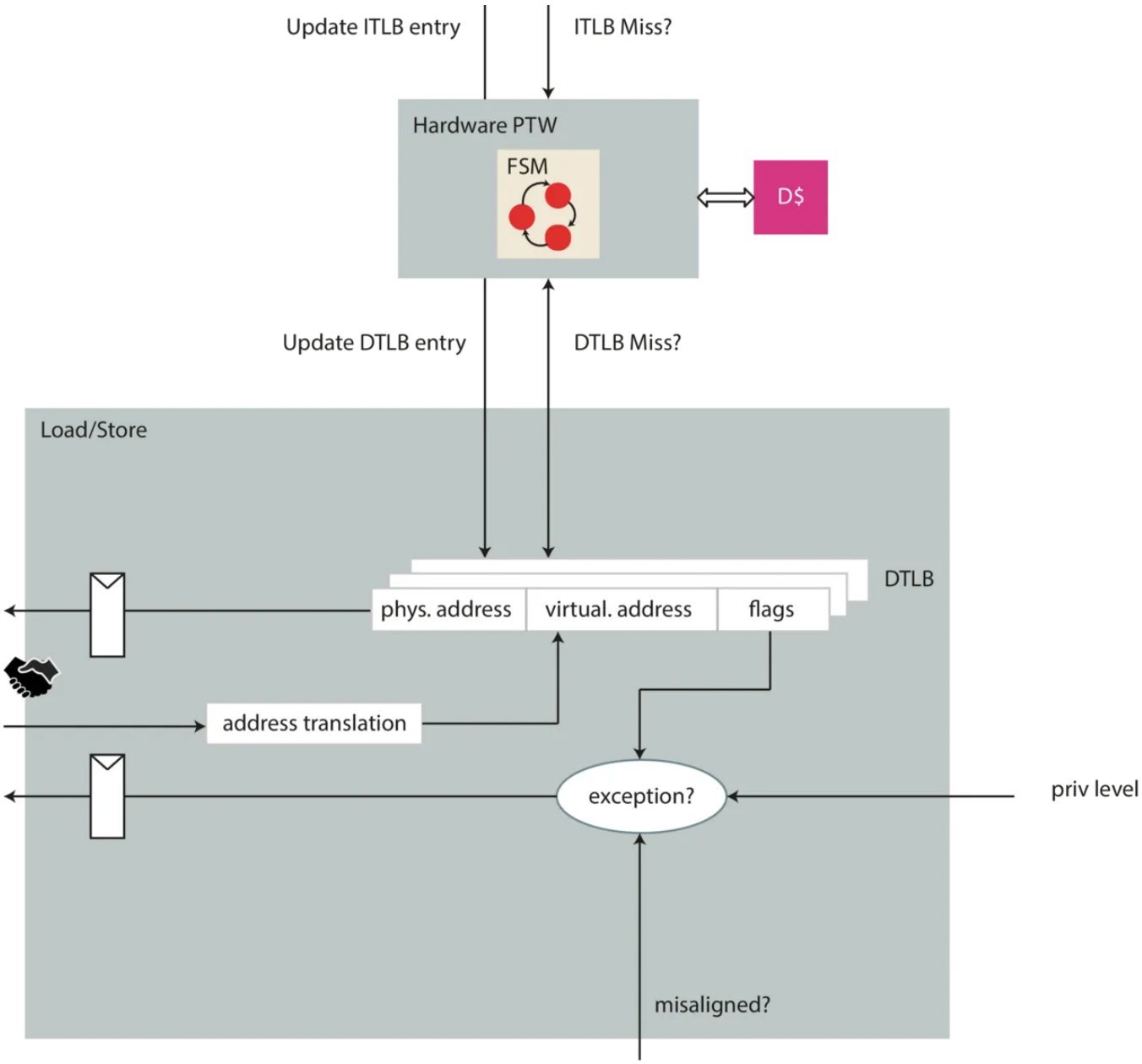
存储缓冲区跟踪所有存储。 它实际上由两个缓冲区组成： 一个是已提交指令的缓冲区，另一个是仍在推测中的未提交指令的缓冲区。 刷新时，只有已提交的指令会被保留，而 speculative 队列则会被完全清空。 为了防止缓冲区溢出，两个队列都会保留一个满标志。 speculative队列的满标志直接传递给存储单元，存储单元将停止 LSU 旁路模块的运行，从而不再接收任何请求。 与此相反，提交队列的 "满 "信号会转到提交阶段。 如果提交队列不能接受任何新数据项，提交阶段就会停滞。 在每次提交存储时，提交阶段也会发出 `lsu_commit` 信号，将特定条目从推测队列放入 non-sepculative（提交）队列。

一旦某个存储空间进入提交队列，队列就会自动尝试将队列中最老的存储空间提交到内存中。

存储缓冲区只对物理地址有效。 在提交时，地址转换已经正确。 对于 speculative 队列中的存储，地址可能并不正确，但如果更新地址转换数据结构，这一问题就会迎刃而解，因为这些指令也会自动刷新整个推测缓冲区。

Memory Management Unit (MMU) {#par:mmu}





内存管理单元（MMU）负责地址转换（）和一般内存访问。 地址转换需要单独激活，方法是写入相应的控制和状态寄存器，并切换到比机器模式更低的权限模式。 一旦启用地址转换，它还将处理页面故障。 MMU 包含 ITLB、DTLB 和 `hardware page table walker` (HPTW)。 虽然在逻辑上并没有真正纠缠在一起，但取数接口也是通过 MMU 路由的。 一般来说，取数据接口和数据接口的处理方式是不同的。 它们只相互共享 HPTW。

通过 MMU 主要有两条根本不同的路径：一条来自指令获取阶段，另一条来自 LSU。 让我们从指令获取接口开始： IF 阶段请求获取特定地址的内存内容。 指令取回总是要求虚拟地址。 根据是否启用地址转换，MMU 要么以透明方式让请求直接进入 I$，要么进行地址转换。

如果激活了地址转换，对指令缓存的请求将被延迟，直到找到有效的转换。 如果找不到有效的转换，MMU 将发出异常信号。 此外，如果地址转换可以在 ITLB 上命中，则是一个纯粹的组合路

径。 TLB 是作为由触发器组成的全集关联缓存来实现的。 这反过来又意味着，请求到内存的路径相当长，而且很容易成为关键路径。

如果出现异常，异常将与有效信号一起返回到指令获取阶段，而不是授予信号。 这意味着我们还需要在异常路径上支持多个未完成的事务（见）。 MMU 有一个专门的缓冲区（FIFO）来存储这些异常，并在答案有效时立即返回。

数据存储器侧（D$）的 MMU 接口则完全不同。 它有一个由握手信号保护的简单请求–响应接口。 加载单元或存储单元都会要求 MMU 执行地址转换。 然而，地址转换过程并不像获取接口那样是组合式的。 额外的寄存器组会将 MMU 的应答（TLB 命中）延迟一个额外的周期。 如上段所述，地址转换是一个非常关键的计时过程。 数据接口的特殊问题在于 LSU 需要事先生成地址。 地址生成涉及另一个昂贵的附加环节。 与地址转换一起，这条路径无疑变得至关重要。 由于数据高速缓存是虚拟索引和物理标记的，因此这个额外的周期不会对 IPC 造成任何损失。 但是，它使内存请求过程变得复杂了一些，因为我们可能需要因为异常而中止内存访问。 如果加载请求出现异常，加载单元需要终止之前发送的内存请求。 异常加载（或存储）永远不会进入内存。

两个 TLB 都是完全关联设置，大小可配置。 此外， `application specifier ID` 应用程序指定标识符（ASID）的大小也可以改变。 ASID 可以防止刷新 TLB 中的某些区域（例如在切换应用程序时）。 这一点目前尚未实现。

Page Table Walker (PTW)

在 `` 中已经介绍了走页器的用途。 `The page table walker` 在 ITLB 和 DTLB 上监听传入的转换请求。 如果它发现 TLB 上缺少其中一个请求，就会保存虚拟地址并开始走页表。 如果 `The page table walker` 遇到任何错误状态，它将抛出一个页面故障异常，该异常会被 MMU 捕获，并传播到获取接口或 LSU。

The page table walker 走表程序优先处理 DTLB 未命中。 RISC-V 特权架构中对走表过程有更详细的描述

PMA/PMP Checks

内核支持物理模式下的 PMA 和 PMP 检查，也支持启用虚拟内存的情况下的 PMA 和 PMP 检查。 PMA 检查仅在最终访问（转换后的）物理地址时执行。 不过，在走页表期间也必须检查 PMP。 在走页过程中，所有内存访问都必须通过 PMP 规则。

条目的数量可通过 `CVA6Cfg.NrPMPEntries` 参数进行参数化。 不过，内核只支持粒度 8

(G=8)。 这简化了实现过程，因为我们不必担心任何未对齐的访问。 设计中共有三个不同的 PMP 单元。 它们分别验证指令访问、数据加载和存储以及 `page table walk` 。

MMU Implementation Details

1. MMU 优先处理指令地址转换和数据地址转换。 MMU 的行为描述如下：
   一旦指令取回阶段的请求到达，ITLB 就会检查缓存条目（组合路径）。 一旦缓存未命中，就会调用 PTW;
2. PTW 会在多个周期内执行页表走行。 在走页过程中，PTW 将更新 ITLB 的内容。 MMU 每个周期都会检查 ITLB 中是否存在缓存命中，因此页表走行已经结束。

Multiplier

乘法器包含一个除法和乘法单元。 乘法运算在两个周期内完成，完全流水线操作（需要重新计时）。 除法是一个简单的串行除法器，在最坏情况下需要 64 个周期。

CSR Buffer

CSR 缓冲区是一个功能单元，其唯一目的是存储指令要读写的 CSR 寄存器的地址。 我们需要这样做有两个原因。 第一个原因是 CSR 指令会改变架构状态，因此必须对该指令进行缓冲，只有在提交阶段决定提交该指令后才能执行。 第二个原因是记分板条目的结构方式： 它只有一个结果字段，但对于任何 CSR 指令，我们都需要保留要写入的数据和该指令要更改的 CSR 地址。 为了避免记分板上出现一些特殊情况的位字段，CSR 缓冲区开始发挥作用。 它只需保存地址，如果要执行 CSR 指令，就会使用存储的地址。

明显的缺点是，由于缓冲区只有一个元素，我们无法在不出现流水线停滞的情况下背靠背执行多条 CSR 指令。 由于 CSR 指令非常罕见，这并不是什么大问题。 反正有些 CSR 指令会导致管道刷新。

Commit Stage

提交阶段是处理器流水线的最后一个阶段。 其目的是接收传入指令并更新架构状态。 这包括写入 CSR 寄存器、提交存储并将数据写回寄存器文件。 黄金法则是，在任何情况下，其他流水线阶段都不得更新架构状态。 如果要保持内部状态，则必须可以重新设置（例如：通过刷新信号，参见）。

我们可以将退订指令分为两类。 第一类只是写入架构寄存器文件。 第二类也可以写寄存器文件，但需要一些进一步的业务逻辑。 在本次写入时，只有两个地方需要这样做，一个是存储单元，提交阶段需要告诉存储单元将存储提交到内存中，另一个是 CSR 缓冲区，一旦相应的 CSR 指令退出，就需要释放该缓冲区。

除了退订指令，提交阶段还管理各种异常源。 特别是在提交时，异常可能来自三个不同的来源。 首先，异常发生在之前四个流水线阶段中的任何一个（只有四个，因为 PC Gen 不能抛出异常）。 其次，提交过程中出现异常。 提交期间发生异常的唯一来源是 CS 寄存器文件和中断。

为了实现精确的中断，中断仅在提交期间被考虑，并与该特定指令相关联。 由于我们需要特定的 PC 才能将中断与之关联，因此中断可能需要推迟到另一条有效指令进入提交阶段时才发生。

此外，提交阶段还能控制处理器的整体停滞。 如果出现停止信号，它将不会提交任何新指令，这将产生反向压力，最终导致流水线停滞。 提交阶段还与控制器进行大量通信，以执行栅栏指令（刷新缓存）和其他流水线重置。

Translation Lookaside Buffer

Shared Translation Lookaside Buffer

Page Table Walker