2021-11-23

潘星雨

RVAT

[公司地址]

“Vostok564-D”核心技术手册

架构和机制

目录

[前言 5](#_Toc88568564)

[概览 6](#_Toc88568565)

[流水线结构 6](#_Toc88568566)

[内外不同步时钟处理 6](#_Toc88568567)

[分支预测 6](#_Toc88568568)

[流水线调度 7](#_Toc88568569)

[中断、异常处理 7](#_Toc88568570)

[队列化的存储器访问 7](#_Toc88568571)

[流水线 8](#_Toc88568572)

[级间信号定义 8](#_Toc88568573)

[命名规范 8](#_Toc88568574)

[功能分类 8](#_Toc88568575)

[级间握手 8](#_Toc88568576)

[空操作 8](#_Toc88568577)

[正常操作 9](#_Toc88568578)

[等待操作 9](#_Toc88568579)

[解决时序反压 9](#_Toc88568580)

[冲刷 9](#_Toc88568581)

[局部冲刷 9](#_Toc88568582)

[全局冲刷 9](#_Toc88568583)

[冲刷优先级 9](#_Toc88568584)

[指令前端 11](#_Toc88568585)

[分支目标缓冲器（BTB） 11](#_Toc88568586)

[程序指针（PC） 12](#_Toc88568587)

[地址转换单元/I-ATU 12](#_Toc88568588)

[流水线冲刷 13](#_Toc88568589)

[TLB冲刷 13](#_Toc88568590)

[TLB命中 14](#_Toc88568591)

[TLB未命中 14](#_Toc88568592)

[TLB表项 14](#_Toc88568593)

[4K页比对 14](#_Toc88568594)

[2M页比对 15](#_Toc88568595)

[1G页比对 15](#_Toc88568596)

[查询下一级TLB 15](#_Toc88568597)

[发出命令 16](#_Toc88568598)

[接收结果 16](#_Toc88568599)

[硬件页表查询/PTW 16](#_Toc88568600)

[页表查询 16](#_Toc88568601)

[页表置脏位 16](#_Toc88568602)

[Sv39页表查询流程 16](#_Toc88568603)

[地址区间比对 17](#_Toc88568604)

[在AQ中发出访问请求 17](#_Toc88568605)

[译码单元（IDU） 18](#_Toc88568606)

[行为分析 18](#_Toc88568607)

[冲刷 18](#_Toc88568608)

[解码失败 18](#_Toc88568609)

[正常解码 18](#_Toc88568610)

[指令一览表 18](#_Toc88568611)

[ITAG分配 21](#_Toc88568612)

[相关性检查 21](#_Toc88568613)

[地址计算 21](#_Toc88568614)

[分支预测 21](#_Toc88568615)

[当前预测为跳转（Predicted PC≠Current PC + 4） 22](#_Toc88568616)

[当前预测为不跳转（Predicted PC = Current PC + 4） 22](#_Toc88568617)

[指令派遣表（DITF） 23](#_Toc88568618)

[添加表项 23](#_Toc88568619)

[移除表项 23](#_Toc88568620)

[执行单元 24](#_Toc88568621)

[SIMD管线 24](#_Toc88568622)

[浮点管线 24](#_Toc88568623)

[数字协处理器Mcop/长整数管线 24](#_Toc88568624)

[存储管线LSU 24](#_Toc88568625)

[地址转换 24](#_Toc88568626)

[访问记录表 24](#_Toc88568627)

[结果单元 25](#_Toc88568628)

[算数逻辑管线ALU（短整数管线） 25](#_Toc88568629)

[提交(Commit)/写回(Write Back) 26](#_Toc88568630)

[写回顺序管理 26](#_Toc88568631)

[机器状态寄存器CSR 26](#_Toc88568632)

[异常管理 27](#_Toc88568633)

[返回 27](#_Toc88568634)

[Debug机制 29](#_Toc88568635)

[概览 29](#_Toc88568636)

[调试转换模块（Debug Transport Module） 29](#_Toc88568637)

[JTAG接口 29](#_Toc88568638)

[JTAG-Kernel时钟域同步 29](#_Toc88568639)

[TAP寄存器 29](#_Toc88568640)

[调试模块（Debug Module） 29](#_Toc88568641)

[概览 29](#_Toc88568642)

[调试模块接口（Debug Module Interface） 29](#_Toc88568643)

[抽象命令（Abstract Command） 29](#_Toc88568644)

[调试模块寄存器 30](#_Toc88568645)

[线程Debug单元（Hart Debug Unit） 33](#_Toc88568646)

[新增的Debug CSR 33](#_Toc88568647)

[调试器Halt请求 34](#_Toc88568648)

[单步模式（Step Mode） 34](#_Toc88568649)

[断点模式（Break Point） 34](#_Toc88568650)

[附录：核心架构图 35](#_Toc88568651)

[L1缓存 36](#_Toc88568652)

[组织方式 36](#_Toc88568653)

[附录：TLB间互联 36](#_Toc88568654)

[附录：队列化的总线访问接口 36](#_Toc88568655)

[发射单元（Issue） 36](#_Toc88568656)

[访问表（Access Table） 37](#_Toc88568657)

[结果单元（Result） 37](#_Toc88568658)

[操作和结果队列 37](#_Toc88568659)

[附录：灵活互联总线（FIB） 39](#_Toc88568660)

[信号规定 39](#_Toc88568661)

[发送数据包格式 41](#_Toc88568662)

[ID 41](#_Toc88568663)

[CMD 42](#_Toc88568664)

[BURST 42](#_Toc88568665)

[SIZE 42](#_Toc88568666)

[ADDR 43](#_Toc88568667)

[DATA 43](#_Toc88568668)

[接收数据包格式 43](#_Toc88568669)

[RPL 43](#_Toc88568670)

[传输 43](#_Toc88568671)

[单次传输（SIGR & SIGW） 43](#_Toc88568672)

[连续读传输（SEQR） 45](#_Toc88568673)

[连续写传输 46](#_Toc88568674)

[锁定传输 48](#_Toc88568675)

[附录：FIB-A拓展 49](#_Toc88568676)

[新增命令 49](#_Toc88568677)

[操作格式 49](#_Toc88568678)

[互斥读LR 49](#_Toc88568679)

[条件写SC 50](#_Toc88568680)

[AMOSWAP等其他AMO指令 50](#_Toc88568681)

[附录：核心架构 51](#_Toc88568682)

[附录：问题记录 51](#_Toc88568683)

# 前言

PRV564是笔者设计的第5代RISC-V内核，PRV系列处理器经历了PRV32（2019.4开发，使用6T多周期设计）、PRV132（2019.7开发，在PRV32基础上增加了A指令拓展）、PV332（2019.12开发，在PRV132基础上增加了Sv32虚拟内存支持）、PRV464（2020.4开发，使用4级流水线，具有Sv39虚拟内存支持，4+4 L1缓存）、PRV464SXR（2020.8开发，在PRV464基础上改进了缓存结构，增加到8+8 L1缓存）的发展，笔者在设计的过程中汲取了宝贵的经验教训，其中不乏严重的设计失误，因此我决定将之前设计中吸取的教训一并写在前言中，希望后来人少走一些弯路。

首先需要明确的是，我们在这里这篇文章里谈论的架构设计要点并不仅仅针对RISC-V，而是适用于几乎所有通用处理器，处理器设计的思维是相通的，其解决问题的方法可能多种多样，但归根结底使用的思维方法是类似的。ISA（指令系统架构）只是设计目的、HDL语言只是设计工具、FPGA只是验证工具，需要时时刻刻铭记这一点，如果说某一个设计被特定的工具所局限，那就说明设计人员并没有理解其中的门道。

永远保持敬畏之心！相信读者对经典5级流水线架构的执行流程已经非常熟悉，你可能对本处理器采用的“如此基础”的流水线架构感到失望，笔者在设计这款处理器的时候提出了非常多的想法，其中不乏更长的流水线以及乱序写回的方案，但都被我否决了。处理器实现中的难题远不在指令正常执行的理想情况里面（在这种情况里，指令发生的行为是可以被预测的），而是在随时可能发生的异常里面，这些异常是不可能被预测的，例如内存访问指令中可能遭遇的内存访问失败和外部中断，如何让处理器能精确的执行每一条指令，精确的把控异常之后的边界点，这才是最大的难题。如果不解决这些问题那就只相当于设计了皮毛，大跃进式的开发只是运用皮毛一般的知识开发一个皮毛一般深度的核心，事物是需要遵守从小到大、从简单到复杂的发展规律的，在知识和实践水平储备不足的情况下搞大核心只能是摸黑走夜路，因此需要常对此持敬畏之心，比昨天强就是进步！

如何进行高效的开发？应当遵循自顶向下的开发方式，虽然说起来非常简单，但是实际却很鲜有人完全遵循，笔者在进行开发的早些时候非常不愿意写文档和注释，在草稿纸上打一个10%的草图就直接抱着Verilog代码就开始乱写一气，设计也是一边写一边修改，对于一些发现的bug笔者也只是单纯的继续在bug上叠床架屋，导致代码越来越难读，最后不得不推倒重来。为了避免这种看起来效率很高实际上全是豆腐渣工程的开发，应当完全严格的遵循自顶向下的过程，在开发文档中记录下自己的思路，深刻分析模块可能会遇到的每一个工况并记录下来。对于那些设计中考虑不周全的设计，应当有勇气推倒重来以避免之后花100倍的时间去继续叠床架屋。

如何设计一个复杂的模块？应当进行问题拆解，将一个复杂的问题拆解成一个一个的小过程。对于HDL设计而言，即拆分成多个子功能块。例如，在设计地址转换单元的过程中，首先分析地址转换单元所需的行为：地址转换和转换后的地址缓存，即MMU和TLB，因此将该单元拆分为TLB和MMU两个子功能块，二者通过自定义的接口进行连接。注意！当进行了模块拆解后，一定需要注明模块间连接接口的时序定义，方便维护。

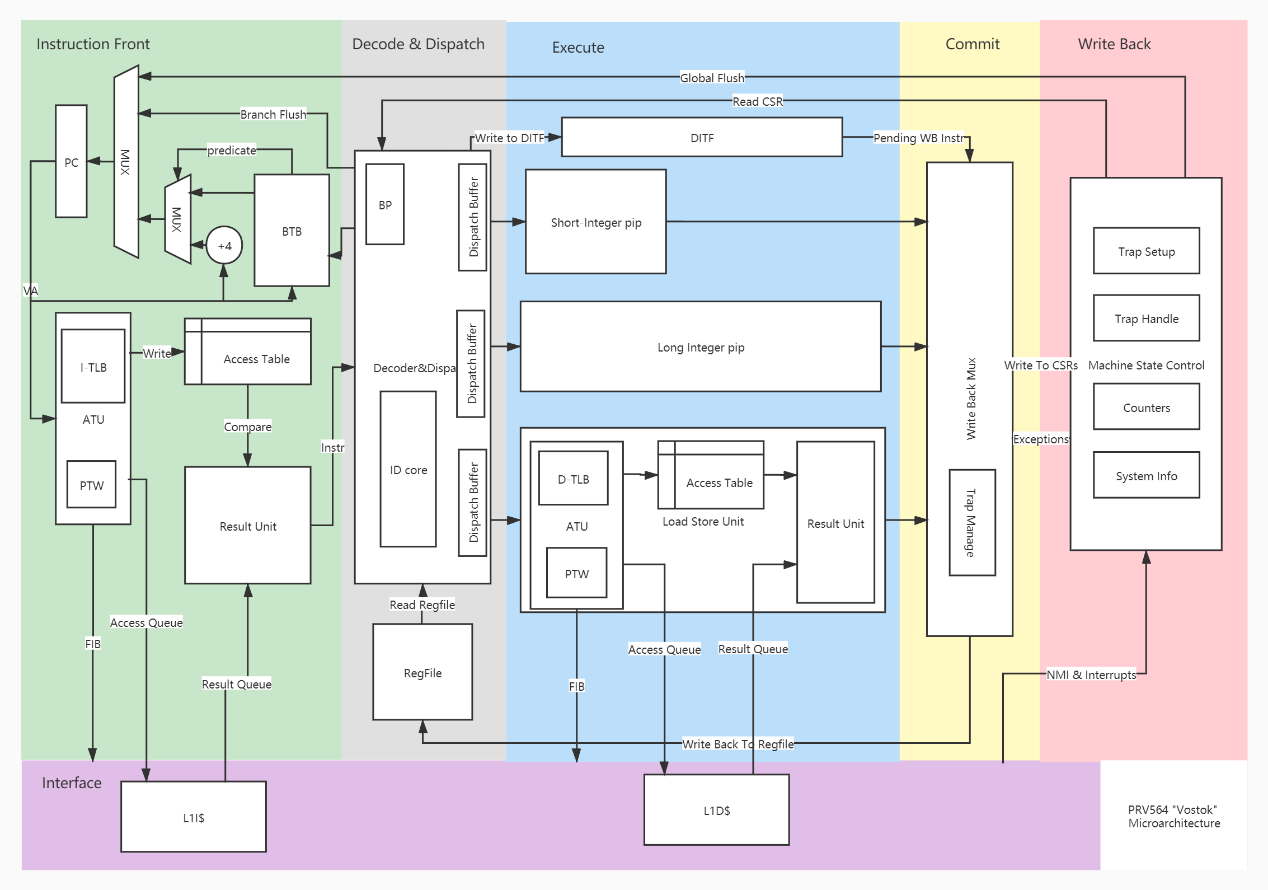
因为笔者水平十分有限，没有半点成功经验，上述经验教训仅基于本人大量的失败经验教训总结而来，如有不足，还请读者批评指正!

# 概览

564处理器是464PRO处理器的进一步升级型号，相对于464PRO处理器有更长的流水线、更高的主频、更强的可拓展性和异步总线等技术，564的核心架构图见附录。

## 流水线结构

564采用变长流水线，最小深度为6级，顺序单派遣顺序写回，核心结构框图见下图，高清图片见附录A，在这里推荐阅读本文的过程中对照此图进行阅读。



图：PRV564核心架构简图

## 内外不同步时钟处理

　　考虑到核心面积较大，如果让处理器时钟和总线时钟同步运行，可能会造成时序上的一些问题，因此设计了FIB总线让核心和外部总线完全独立开来。详见附录：FIB总线。

## 分支预测

本处理器采用静态分支预测，跳转地址在Decode阶段给出。

## 流水线调度

在Execute阶段，指令需要的执行周期可能是1个或者多个，在最好情况下是1个周期即可完成（例如基础运算指令），但是如果是乘除或者Load/Store类型的指令，可能需要消耗多个周期，因此指令即便是按照顺序的方式从ID派遣，也有可能是以不一样的顺序到达写回级，为了确保写回的时候按照顺序的方式写回，增设DITF（Dispatch Instruction Table FIFO）来对已派遣的指令进行记录，写回的时候按照此FIFO里先进先出的顺序进行写回。

同时，DITF中还记录了指令使用的RS1、RS2、和RD信息，ID在派遣指令的时候会查询此表以知晓是否会产生WAR相关性问题（详情可见DITF一节）。

## 中断、异常处理

为了简化设计，避免产生不必要的bug，统一在Write Back阶段对异常和中断进行处理。

## 队列化的存储器访问

在之前的处理器总线中，我们都只采用了简单的同步总线作为对外访问的接口，操作时序和AHB总线类似：第一个Cycle给出地址和操作方式，第二个Cycle给出访问结果，或者继续等待。这种访问方式虽然简单有效，但是在面对更长的缓存流水线时，就需要精心设计内部逻辑以确保从Cache中返回的读写结果是当前指令的。在简单同步接口上会遭遇的一个典型的情况是：指令A通过SRAM接口发出了读指令请求，但是指令A随后被取消了，新执行的指令B也发出读指令请求，这个时候指令A的访问结果被返回了，那么指令B就会错误的拿到指令A的数据。

本处理器使用了一种基于Score Board的访问调度方法，通过给每个访问分配一个ID的形式，配合访存单元里的Score Board对每个访问进行记录，在访存结果接收端对Cache访问的每个结果和Score Board里的记录进行对比，并决定这个结果是否会被当前正在等待结果的指令取走。

# 流水线

在开始整个技术文档之前，先介绍处理器的流水线中使用的信号，读者可以对比此节中介绍的流水线信号和概览中的架构简图以逐步理解该处理器的定量运行流程。

## 级间信号定义

### 命名规范

鉴于之前设计的大量经验，流水线间的信号规范的分组命名非常重要，本集中流水线相关信号的命名的格式为：PIP\_{模块名i/o}\_{功能}\_{信号名称}。

模块名i/o表示该信号相对当前模块的输入/输出情况，例如：PIP\_ATUi\_DATA\_ds1表示ATU模块输入的流水线信号，数据用途，data source 1。

### 功能分类

MSC：Machine State Control，机器状态控制，带有此功能名称的信号均表示对机器状态会产生实际影响的。

INFO：Information，信息，带有此功能名称的信号均表示这是指令信息的，如pc、itag。

DATA： Data，数据，带有此功能名称的信号均表示传输的数据。

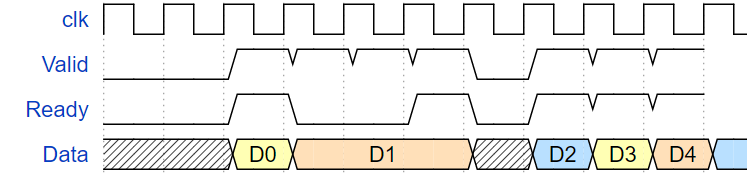
Opcode：opcode，操作码。

Opinfo：operation information，操作信息，标明当前操作是符号/无符号，32位/64位。

Opsize：operation size，操作大小，用于Load/Store单元确定大小是8/16/32/64位。

## 级间握手

流水线之间采用标准的Valid-Ready信号进行握手，流水线的上一级给出Valid和Data，流水线的下一级反馈Ready信号，具体的时序图如下图中描述：



图：流水线间握手信号

### 空操作

如果当前无操作派遣到下一级，则流水线的上一级的Valid信号为0，如图中的T0和T1，T6。

### 正常操作

如果派遣的指令可以在一个周期内完成，则Ready信号置1表示可以继续派遣下一个指令，如图中的T2，T7-T9。

### 等待操作

如果派遣的指令无法在一个周期内完成，则Ready信号置0表示当前派遣的指令需要Hold up，上一级的输出寄存器应当保持数据，如图中的T3-T5。

## 解决时序反压

本核心在IDU派遣级采用了乒乓工作的派遣缓冲器（Dispatch Buffer），截断IDU之后执行级握手信号到IDU之前指令前端的组合逻辑路径。

## 冲刷

由于本处理器在Execute阶段会给出详细的跳转地址，如果分支预测失败，则在Execute之前的所有流水线级都将被冲刷而之后的不冲刷，这个过程被称为“**局部冲刷**”。

和局部冲刷相对的，是“**全局冲刷**”，全局冲刷的信号由Write Back级直接发出，将整个流水线冲刷。

### 局部冲刷

因为跳转指令预测跳转失败引起局部冲刷，此举将清空Decode之前的所有指令，并且PC将会被设置成跳转地址。

### 全局冲刷

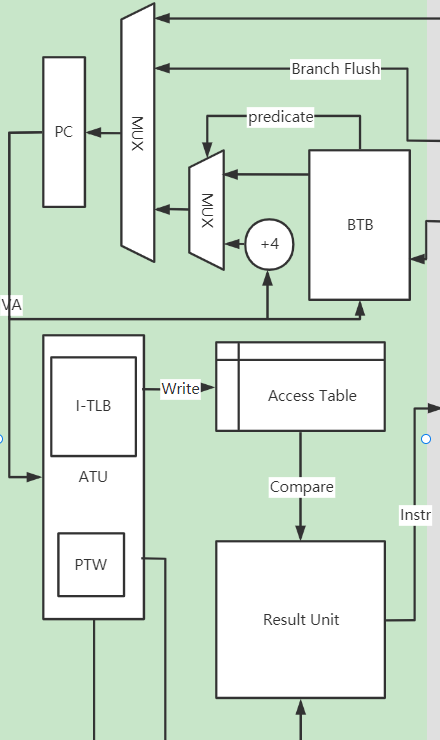
如果一条指令遭遇了异常、中断、或者是一条特殊的机器控制指令，则会在写回后进行全局冲刷。全局冲刷将清空流水线中所有的指令，并将PC设置为跳转地址或当前指令的下一条。

### 冲刷优先级

PC可以被全局冲刷和局部冲刷设置，显而易见的，全局冲刷具备更高的优先级因为全局冲刷可以将发起局部冲刷的指令一并冲掉。因此，在同时发生全局冲刷和局部冲刷的时候，PC会优先响应全局冲刷。

# 指令前端

指令前端包含了IATU（指令地址转换）、Access Table、和Fetch。经过ATU转换后的访存会直接进入Cache的访问队列里，同时，已经被送出的访问信息会被记录在Access Table中。



图：指令前端

## 分支目标缓冲器（BTB）

PC为输入，Predicted作为输出，如果预测为跳转，则Predicted位=1，且给出预测的PC，表示下一条指令的跳转地址。

## 程序指针（PC）

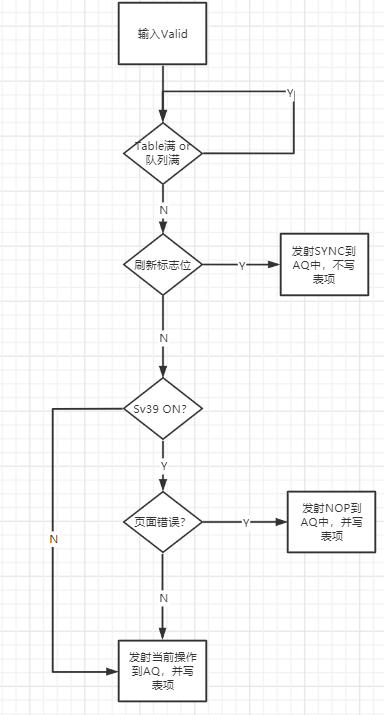
目前本处理器暂不支持C拓展，故每条指令的间隔是固定的4Byte，PC只需按照固定+4的速度进行增加。

如果分支预测成功，则Predicted位被置1，PC跳转到BTB中预测的地址处。

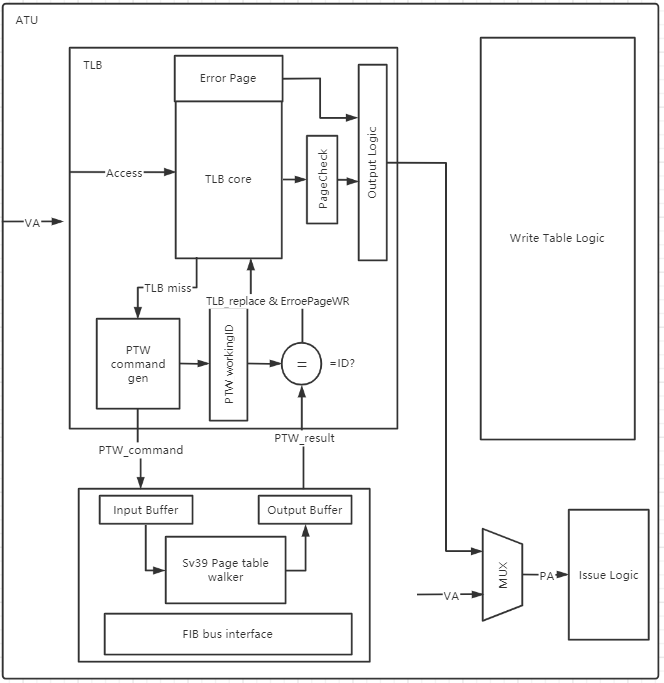
## 地址转换单元/I-ATU

564处理器使用可变长度的TLB方式，各级之间通过特定的协议交换数据，在本核心中使用的TLB包含可配置数量的表项，并且通过TLB间互联协议连接到MMU。

注意，只有当虚拟内存被打开之后，这个模块才会介入，否则是不介入的，地址转换单元的行为如下流程图表示：



图：地址转换单元流程图



图：ATU结构

### 流水线冲刷

流水线冲刷是模块响应的最高优先级，当此信号被Enable之后，ATU访问将无条件置0，同时清除PTW walking ID中记录的正在等待结果的传输，以避免后续某条倒霉的指令拿到了错误的PTW查询结果。

### TLB冲刷

冲刷信号由LSU直接给出，当没有流水线冲刷时候TLB模块执行此操作，执行此操作时TLB输出插空并清零页表，执行完成后TLB模块将使能TLB\_FlushOK信号，或者TLB刷新错误后发出TLB\_FlushErr信号。

### TLB命中

如果TLB命中，且当前的访问不需要更改页表项，则直接给出转换后的地址和页面检查情况，在这个情况下不会向PTW发出访问请求。

### TLB未命中

如果TLB没有命中，则ATU输出将插空，并且将向PTW发出访问请求，会执行以下操作：

1. 为当前的访问分配一个ID，并记录当前访问ID到PTW working ID中，并置有效，标明有一个访问正在等待。
2. 如果有一个结果从PTW返回了，首先对比返回的ID与当前PTW working ID中记录的ID是否相同，如果不相同则丢弃该结果，如果相同则替换掉当前TLB中的一项。

如果在等待PTW结果的过程中，流水线冲刷发生了，则PTW working ID和PTW working寄存器将会被清零，标明当前没有正在等待的访问。

### TLB表项

进行地址转换的过程中，TLB表项将对比给出的Virtual Address和表项中的VPN来确定是否命中，如果命中，则将自身的NRU bit置0。同时，当所有的NRU bit都被置0后，NRU bit将会自动全部复位为1。

每个TLB表项包含的内容如下表所示：

表：TLB表项结构

|  |  |  |
| --- | --- | --- |
| 名称 | 位宽 | 描述 |
| V | 1bit | 当前页表项有效 |
| PAGESIZE | 2bit | 描述当前存储的页面大小，00为4K，01为2M，10为1G。 |
| NRU | 1bit | Not Recent Used 位，当页表被使用后置0， |
| PTE | 10bit | 该页表的页面描述符 |
| PPN | 44bit | 该页表项对应的物理页号 |
| VPN | 27bit | 该页表项对应的虚拟页号 |

表：TLB表项内容

### 4K页比对

如果转换完成后，从MMU返回的i=2’b00，则当前页面的大小是4K字节，则虚拟地址（VA）的bit0~bit12作为Page Offset，bit12~bit38作为虚拟页号，如下图：

|  |  |
| --- | --- |
| 38：12 | 11：0 |
| VPN | Page Offset |

在这个页面大小下，TLB的每一项的VPN都会和输入的虚拟地址的VPN段完全比对，如果命中，则给出PPN，和虚拟地址的低12位拼凑成一个56bit的物理地址（PA）

|  |  |
| --- | --- |
| 55：12 | 11：0 |
| PPN | VA[11:0] |

### 2M页比对

如果转换完成后，从MMU返回的i=2’b01，则当前页面的大小是2M字节，则虚拟地址（VA）的bit0~bit20作为Page Offset，bit21~bit38作为虚拟页号，如下图：

|  |  |
| --- | --- |
| 38：21 | 20：0 |
| VPN | Page Offset |

在这个页面大小下，因为输入的虚拟地址的VPN的一部分也被作为页内偏移，因此仅有输入虚拟地址的高18位会参与比对。如果命中，则给出PPN的高35位，和虚拟地址的低21位拼凑成一个56bit的物理地址（PA）

|  |  |
| --- | --- |
| 55：21 | 20：0 |
| PPN | VA[20:0] |

### 1G页比对

如果转换完成后，从MMU返回的i=2’b10，则当前页面的大小是1G字节，则虚拟地址（VA）的bit0~bit29作为Page Offset，bit30~bit38作为虚拟页号，如下图：

|  |  |
| --- | --- |
| 38：29 | 28：0 |
| VPN | Page Offset |

在这个页面大小下，因为输入的虚拟地址的VPN的一部分也被作为页内偏移，因此仅有输入虚拟地址的高9位会参与比对。如果命中，则给出PPN的高27位，和虚拟地址的低29位拼凑成一个56bit的物理地址（PA）

|  |  |
| --- | --- |
| 55：29 | 28：0 |
| PPN | VA[29:0] |

## 查询下一级TLB

TLB间互联采用自定义的协议，从上一级发往下一级采用如下格式：

|  |  |  |  |
| --- | --- | --- | --- |
| 26：0 | 7：0 | 7:0 | 0 |
| VPN | ID | CMD | V |

CMD和VADDR在一个周期内完成传输，考虑到各级TLB间会跨FIFO传递数据，因此当V位为1时将命令传输到下一级。

8位的CMD表示了当前需要下一级进行什么操作，其指令表如下：

|  |  |  |
| --- | --- | --- |
| CMD | 助记符 | 指令行为 |
| 00H | NOP | 空，NOP |
| 01H | rLUT | （以读的方式）查询页面 |
| 02H | wLUT | （以写的方式）查询页面 |
| 03H | xLUT | （以执行的方式）查询页面 |
| 04-FF | xxx | 保留 |

返回值格式如下：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 9：0 | 43：0 | 7:0 | 7：0 | 0 |
| PTE（低10位） | PPN | ID | RPL | V |

8位的RPL为返回的控制信息，表示当前的页面转换操作是否完成。

|  |  |  |
| --- | --- | --- |
| RPL | 助记符 | 含义 |
| 11H | rPERR | 读取页表失败 |
| 12H | wPERR | 写入页表失败 |
| 13H | xPERR | 执行页表失败 |
| 80H | RDY | 页面完成 |

### 发出命令

如果当前没有正在等待结果的查询（PTW working ID中没有有效的记录），且PTW\_FULL=0，则可以将命令写入到下一级的Buffer中（图中的Input Buffer），并给当前查询分配一个ID，写入到Walking ID寄存器中。

### 接收结果

从PTW返回的结果是PPN和Page Size，以及当前ID，如果返回的ID和当前Walking ID记录等待的ID相同，则标明这个结果是当前正在等待的，就可以替换表项。

如果返回的ID和记录中等待的ID不相同，则直接丢弃该结果（Read enable=1，从PTW的output buffer中移除该结果）。

## 硬件页表查询/PTW

### 页表查询

### 页表置脏位

### Sv39页表查询流程

一个标准的Sv39的虚拟地址的构成如下表：

|  |  |  |  |
| --- | --- | --- | --- |
| 38:30 | 29:21 | 20:12 | 11：0 |
| VPN[2] | VPN[1] | VPN[0] | Page offset |

表：Sv39的虚拟地址结构

一个标准的分页流程如下：

1. 中间地址A=SATP.PPN\*PAGE SIZE, I = LEVEL–1

( RV64的LEVEL是3，PAGESIZE=2^12)

1. 从地址A+VA.VPN[i]\*PTESIZE中取得第页表（Sv39的PTESIZE=8）
2. 如果V=0，或者R=0且W=1，造成页面错误。
3. 其余情况下，这个页表是有效的，如果R=1或者X=1，转到第五步，否则这个页是一个指针，它指向下一级页表。将I =I-1，如果I=0则造成页面错误，其他情况下 中间地址A=PTE.PPN\*PAGESIZE,然后到第二步。
4. 当前页表是一个末端页表，MMU会根据该页表中R，W，X位的状态和当前访问的类型来决定是否可以访问，如果不允许访问，则会引起页面错误。
5. 如果I > 0 但是PTE.PPN[i-1:0]不等于0，这是一个不对齐的超页面，这将会引起页面错误，如果没有发生错误前往下一步。
6. 这个页面被成功的访问了，如果A=0，那么MMU自动的将A位置1表示这个页面被访问了，如果这个页面是写数据且D位不为1，那么还会把D位置1
7. 地址已经转换完成

注意：

\*PA.PGOFF = VA.PGOFF（VA和PA的地址偏移相同）

\*如果I> 0，那么这是一个超页面，PA.PPN[i-1:0]=VA.VPN[i-1:0]

\*最后，PA.PPN[LEVEL-1:0]=PTE.PPN[LEVELS-1:0]

## 地址区间比对

RISC-V将内存空间分为可缓存的和MMIO（Memory Mapped Input Output），本处理器的可缓存区段是可以配置的，配置参数有cacheable MASK和cacheable ADDR。Cacheable MASK标明了当前可以缓存区段的地址掩码，cacheable ADDR为可缓存区段的段地址。

判断当前地址ADDR是否处于可缓存区段的方式如下：

如果ADDR & Cacheable MASK = Cacheable ADDR，则当前地址是可以被缓冲的，AQ中的CI位就为0，否则为1。例如，cacheable mask为0xffff ffff 8000 0000，cacheable ADDR为0x0000 0000 8000 0000，经过上述运算后，可以缓冲的地址区间为0x0000 0000 8000 0000 ~ 0x0000 0000 8fff ffff。

\*注：此部分内容可以参考TCP/IP协议中的地址掩码帮助理解。

## 在AQ中发出访问请求

如果当前访问经过TLB后被正确转换了地址，则可以将访问发送到AQ中进行访问（同时将当前访问的信息记录在Access Table中）。

# 译码单元（IDU）

指令解码单元负责将上一级：取指令单元送来的指令解码，检查数据相关性、译出操作数、译出操作码、然后派遣给对应的执行单元执行。每一条被派遣的指令都有一个独有的TAG，该TAG会随着指令一同在管线中前进。同时，IDU会将指令解码完成后的PC、RS1、RS2、RD压入DITF。

因为本设计的Execute阶段设计了多条管线，目前使用到的有：短整数管线（ALU）、存储管线（LSU）、长整数/协处理器管线（COP），正在规划中的有：浮点管线（FPU）、向量单元（VPU）。所有的RISC-V基础指令（不含存储指令）都会被派遣到短整数管线执行；对于译码阶段或在之前就发射错误的，也会被派遣到短整数管线，通过短整数管线到达写回级。所有和存储器相关的指令都会被派遣到存储管线。M拓展将会派遣给长整数管线进行处理，浮点将会派遣给FPU处理、向量将会派遣到向量单元VPU处理。

## 行为分析

当DITF满时，所有派遣都将被暂停，直到DITF恢复不满状态。避免派遣出去的指令在DITF中找不到相关项。

### 冲刷

若冲刷信号被使能，解码单元中正在译码的指令将无效化，已经在ID输出流水线寄存器的指令不会受到影响。

### 解码失败

如果指令无法解码到一条指令，则派遣到短整数单元，作为异常指令写回。

### 正常解码

如果指令正常解码，且目标管线当前空闲，则将指令派遣到对应管线，并将指令相关信息压入DITF。

## 指令一览表

PRV564处理器支持RISC-V 64位I、M、A指令，其中启动Linux系统必备的指令为I和A，将所有支持的指令做一览表：

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 指令助记符 | 数据源1 | 数据源2 | 写回GPR | 写回CSR | 派遣目标 | 派遣操作码 | 信息 | 大小 |
| RV32I/RV64I | | | | | | | | |
| LUI | imm20 | X | \* |  | Short-Integer | NOP | X | X |
| AUIPC | imm20 | pc | \* |  | Short-Integer | ADD | Sign64 | 8 |
| JAL | pc | JADDR | \* |  | Short-Integer | JAL | Sign64 | 8 |
| JALR | pc | JRADDR | \* |  | Short-Integer | JAL | Sign64 | 8 |
| BEQ | X | BADDR |  |  | Short-Integer | NOP | X | X |
| BNE | X | BADDR |  |  | Short-Integer | NOP | X | X |
| BLT | X | BADDR |  |  | Short-Integer | NOP | X | X |
| BLTU | X | BADDR |  |  |  | NOP | X | X |
| BGE | X | BADDR |  |  | Short-Integer | NOP | X | X |
| BGEU | X | BADDR |  |  |  | NOP | X | X |
| LB | LADDR | X | \* |  | Load-Store | READ | Sign64 | 1 |
| LBU | LADDR | X | \* |  | Load-Store | READ | Unsign64 | 1 |
| LH | LADDR | X | \* |  | Load-Store | READ | Sign64 | 2 |
| LHU | LADDR | X | \* |  | Load-Store | READ | Unsign64 | 2 |
| LW | LADDR | X | \* |  | Load-Store | READ | Sign64 | 4 |
| LWU | LADDR | X | \* |  | Load-Store | READ | Unsign64 | 4 |
| LD | LADDR | X | \* |  | Load-Store | READ | Sign64 | 8 |
| SB | SADDR | rs2 |  |  | Load-Store | WRITE | X | 1 |
| SH | SADDR | rs2 |  |  | Load-Store | WRITE | X | 2 |
| SW | SADDR | rs2 |  |  | Load-Store | WRITE | X | 4 |
| SD | SADDR | rs2 |  |  | Load-Store | WRITE | X | 8 |
| ADDI | rs1 | imm12 | \* |  | Short-Integer | ADD | Sign64 | X |
| ADDIW | rs1 | imm12 | \* |  | Short-Integer | ADD | Sign32 | X |
| SLTI | rs1 | imm12 | \* |  | Short-Integer | SLT | Sign64 | X |
| SLTIU | rs1 | imm12 | \* |  | Short-Integer | SLT | Unsign64 | X |
| XORI | rs1 | imm12 | \* |  | Short-Integer | XOR | Sign64 | X |
| ORI | rs1 | imm12 | \* |  | Short-Integer | OR | Sign64 | X |
| ANDI | rs1 | imm12 | \* |  | Short-Integer | AND | Sign64 | X |
| SLLI | rs1 | imm12 | \* |  | Short-Integer | SL | Unsign64 | X |
| SRLI | rs1 | imm12 | \* |  | Short-Integer | SR | Sign64 | X |
| SRAI | rs1 | imm12 | \* |  | Short-Integer | SR | Sign64 | X |
| SLLIW | rs1 | imm12 | \* |  | Short-Integer | SL | Unsign32 | X |
| SRLIW | rs1 | imm12 | \* |  | Short-Integer | SR | Sign32 | X |
| SRAIW | rs1 | imm12 | \* |  | Short-Integer | SR | Sign32 | X |
| ADD | rs1 | rs1 | \* |  | Short-Integer | ADD | Sign64 | X |
| ADDW | rs1 | rs2 | \* |  | Short-Integer | ADD | Sign32 | X |
| SUB | rs1 | rs2 | \* |  | Short-Integer | SUB | Sign64 | X |
| SUBW | rs1 | rs2 | \* |  | Short-Integer | SUB | Sign32 | X |
| SLL | rs1 | rs2 | \* |  | Short-Integer | SL | Sign64 | X |
| SLLW | rs1 | rs2 | \* |  | Short-Integer | SL | Sign32 | X |
| SLT | rs1 | rs2 | \* |  | Short-Integer | SLT | Sign64 | X |
| SLTU | rs1 | rs2 | \* |  | Short-Integer | SLT | Unsign64 | X |
| XOR | rs1 | rs2 | \* |  | Short-Integer | XOR | Sign64 | X |
| SRL | rs1 | rs2 | \* |  | Short-Integer | SR | Unsign64 | X |
| SRLW | rs1 | rs2 | \* |  | Short-Integer | SR | Unsign32 | X |
| SRA | rs1 | rs2 | \* |  | Short-Integer | SR | Sign64 | X |
| SRAW | rs1 | rs2 | \* |  | Short-Integer | SR | Sign32 | X |
| OR | rs1 | rs2 | \* |  | Short-Integer | OR | Sign63 | X |
| AND | rs1 | rs2 | \* |  | Short-Integer | AND | Sign64 | X |
| FENCE | X | X |  |  | Short-Integer | NOP | X | X |
| FENCE.I | X | X |  |  | Short-Integer | NOP | X | X |
| ECALL | X | X |  |  | Short-Integer | NOP |  |  |
| EBREAK | X | X |  |  | Short-Integer | NOP |  |  |
| CSRRWI | imm5 | csr | \* | \* | Short-Integer | CSRW | X | X |
| CSRRSI | imm5 | csr | \* | \* | Short-Integer | CSRS | X | X |
| CSRRCI | imm5 | csr | \* | \* | Short-Integer | CSRC | X | X |
| CSRRW | rs1 | csr | \* | \* | Short-Integer | CSRW | X | X |
| CSRRS | rs1 | csr | \* | \* | Short-Integer | CSRS | X | X |
| CSRRC | rs1 | csr | \* | \* | Short-Integer | CSRC | X | X |
| MRET | X | X |  |  | Short-Integer | NOP | X | X |
| SRET | X | X |  |  | Short-Integer | NOP | X | X |
| URET | X | X |  |  | Short-Integer | NOP | X | X |
| RV32/RV64 A拓展 | | | | | | | | |
| LRW | rs1 | rs2 | \* |  | Load-Store | READ\_L | Sign32 | 4 |
| LRD | rs1 | rs2 | \* |  | Load-Store | READ\_L | Sign64 | 8 |
| SCW | rs1 | rs2 | \* |  | Load-Store | WRIT\_U | Sign32 | 4 |
| SCD | rs1 | rs2 | \* |  | Load-Store | WRIT\_U | Sign64 | 8 |
| AMOSWAPW | rs1 | rs2 | \* |  | Load-Store |  | Sign32 | 4 |
| AMOSWAPD | rs1 | rs2 | \* |  | Load-Store |  | Sign64 | 8 |
| AMOADDW | rs1 | rs2 | \* |  | Load-Store |  | Sign32 | 4 |
| AMOADDD | rs1 | rs2 | \* |  | Load-Store |  | Sign64 | 8 |
| AMOXORW | rs1 | rs2 | \* |  | Load-Store |  | Sign32 | 4 |
| AMOXORD | rs1 | rs2 | \* |  | Load-Store |  | Sign64 | 8 |
| AMOANDW | rs1 | rs2 | \* |  | Load-Store |  | Sign32 | 4 |
| AMOANDD | rs1 | rs2 | \* |  | Load-Store |  | Sign64 | 8 |
| AMOORW | rs1 | rs2 | \* |  | Load-Store |  | Sign32 | 4 |
| AMOORD | rs1 | rs2 | \* |  | Load-Store |  | Sign64 | 8 |
| AMOMINW | rs1 | rs2 | \* |  | Load-Store |  | Sign32 | 4 |
| AMOMIND | rs1 | rs2 | \* |  | Load-Store |  | Sign64 | 8 |
| AMOMAXW | rs1 | rs2 | \* |  | Load-Store |  | Sign32 | 4 |
| AMOMAXD | rs1 | rs2 | \* |  | Load-Store |  | Sign64 | 8 |
| AMOMINUW | rs1 | rs2 | \* |  | Load-Store |  | Sign32 | 4 |
| AMOMINUD | rs1 | rs2 | \* |  | Load-Store |  | Sign64 | 8 |
| AMOMAXUW | rs1 | rs2 | \* |  | Load-Store |  | Sign32 | 4 |
| AMOMAXUD | rs1 | rs2 | \* |  | Load-Store |  | Sign64 | 8 |
| RV32/64 M拓展 | | | | | | | | |
| MUL | rs1 | rs2 | \* |  | Long-Integer | MUL | Sign64 | X |
| MULH | rs1 | rs2 | \* |  | Long-Integer | MULH | Sign64 | X |
| MULHSU | rs1 | rs2 | \* |  | Long-Integer | MULHS | Unsign64 | X |
| MULHU | rs1 | rs2 | \* |  | Long-Integer | MULH | Unsign64 | X |
| DIV | rs1 | rs2 | \* |  | Long-Integer | DIV | Sign64 | X |
| DIVU | rs1 | rs2 | \* |  | Long-Integer | DIV | Unsign64 | X |
| REM | rs1 | rs2 | \* |  | Long-Integer | REM | Sign64 | X |
| REMU | rs1 | rs2 | \* |  | Long-Integer | REM | Unsign64 | X |
| MULW | rs1 | rs2 | \* |  | Long-Integer | MUL | Sign32 | X |
| DIVW | rs1 | rs2 | \* |  | Long-Integer | DIV | Sign32 | X |
| DIVUW | rs1 | rs2 | \* |  | Long-Integer | DIV | Unsign32 | X |
| REMW | rs1 | rs2 | \* |  | Long-Integer | REM | Sign32 | X |
| REMUW | rs1 | rs2 | \* |  | Long-Integer | REM | Unsign32 | X |

\*注：JADDR：jal指令的跳转地址。JRADDR：jalr指令的跳转地址。BADDR：Branch指令用的跳转地址。LADDR/SADDR：Load/Store指令的地址

## ITAG分配

每一条派遣的指令都有一条不同于其他指令的标签，即指令标签ITAG，ITAG为8位，因此最大可以给256条指令打上标签。

为了实现简单，每发射一条指令，ITAG自增1，如此循环进行分配。因此至少需要派遣256条指令才会出现相同的ITAG，只需要保证Execute阶段没有挤超过256条正在执行的指令即可避免混淆。

## 相关性检查

当一条指令需要使用rs1或rs2时候，IDU会将指令所用的rs1index和rs2index送往DITF进行相关性检查，如果检查到了有相关性，则将流水线stall住。

## 地址计算

我们选择在ID阶段计算跳转地址、内存访问地址，并且在ID直接得到跳转信息，这样就可以在ID直接跳转了。

## 分支预测

指令前端送到IDU的数据同时包含Predicted PC，即预测的下一条指令的地址。IDU会对这条指令进行解码，判断下一条指令的地址是否和预测的地址相同，如果不相同则进行分支冲刷确保取得的下一条指令是正确的。

根据当前指令和预测的情况可以分为以下几种情况：

### 当前预测为跳转（Predicted PC≠Current PC + 4）

**If（当前指令是分支指令）**

**If（当前分支指令的跳转地址和预测地址一致）**

分支预测成功，

**Else（当前分支指令的跳转地址和预测不一致）**

分支预测失败，冲刷，将PC置为要跳转的PC。

**Else（当前指令不是分支指令）**

分支预测失败，冲刷，并将PC置为当前PC+4。

### 当前预测为不跳转（Predicted PC = Current PC + 4）

**If（当前指令是分支指令）**

**If（不跳转）**

分支预测成功，

**Else（跳转）**

分支预测失败，冲刷，将PC置为要跳转的PC。

**Else（当前指令不是分支指令）**

分支预测成功。

# 指令派遣表（DITF）

DITF（Dispatched Instructions Table FIFO）。派遣指令FIFO，记录当前派遣到执行单元执行的指令的信息，记录了指令的PC、RS1\_index、RS2\_index、RD\_index、CSRindex、以及指令的异常信息，以及最重要的：指令的ITAG。

一个DITF表项目包含的内容如下表所示：

|  |  |  |
| --- | --- | --- |
| 名称 | 位宽 | 描述 |
| ITAG | 8 | 指令标签 |
| Rs1\_index | 5 | 指令Rs1寄存器索引 |
| Rs2\_index | 5 | 指令Rs2寄存器索引 |
| Rd\_index | 5 | 指令Rd寄存器索引 |
| GPR\_wren | 1 | 指令写回使能 |
| CSR\_index | 12 | 指令CSR寄存器索引 |
| CSR\_wren | 1 | 写CSR寄存器使能 |
| Jmp | 1 | 此指令是跳转指令 |
| InsAccessFlt | 1 | 指令访问错误 |
| InsPageFlt | 1 | 指令页面错误 |
| InsAddrMis | 1 | 指令地址不对齐错误 |
| Ill\_ins | 1 | 指令非法 |
| mret | 1 | 从M模式返回 |
| Sret | 1 | 从S模式返回 |
| Ecall | 1 | 环境调用 |
| fence | 1 | fence指令，用于在写回时候冲刷L1D |
| fencei | 1 | fencei指令。用于在写回时冲刷L1I |
| fencevma | 1 | fencevma指令，用于写回时冲刷TLB |
| ebreak | 1 | 环境退出 |
| System | 1 | 当前指令是系统类型指令，引起冲刷 |

## 添加表项

当一条指令被派遣到管线中，DITF中会加入一项。

## 移除表项

DITF出口处的指令的itag和任意一条管线中的等待写回的指令的itag相同，即该指令已经可以被写回，此时DITF的出口处的表项即被移除。

# 执行单元

## SIMD管线

敬请期待。

## 浮点管线

敬请期待。

## 数字协处理器Mcop/长整数管线

## 存储管线LSU

### 地址转换

地址转换的行为参考指令前端-地址转换。

**读内存访问发射：**

当一个访问不是写内存操作时，该操作立刻就可以被发出，同时在Access Table中记录下当前访问的信息。

**写内存访问发射：**

一个访问如果是需要更改内存的，如AMO指令、读内存操作、这些操作需要等到从Write Back级给出的Modify Permit之后，才可以被发出，否则是不会被发出的。

**数据移位（Data Shift L）：**

发射到存取队列中的数据，是需要移位到对应的偏移位置的。在处理器的寄存器中，数据都是以右对齐的方式存储，将寄存器里的数据写到总线上就需要进行移位，我们在这里采用128位的Cache访问数据位宽：

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 127:0 | | | | | | | | | | | | | | | |
| B15 | B14 | B13 | B12 | B11 | B10 | B9 | B8 | B7 | B6 | B5 | B4 | B3 | B2 | B1 | B0 |
| +15 | +14 | +13 | +12 | +11 | +10 | +9 | +8 | +7 | +6 | +5 | +4 | +3 | +2 | +1 | +0 |

图：128位总线的地址偏移

128bit总线上一次可以传输16个字节，地址偏移量和字节的关系如上表。例如，要在向Base + 2的地址写16位数据，则字节应该位于B2和B2。

\*这部分的代码请读者阅读/ATU.v中的子模块：“DataShiftL”。

### 访问记录表

同上文（指令前端）中描述的访问记录表。

### 结果单元

结果单元比对从结果队列中返回的操作结果和访问记录表中记录的ID，如果ID相同，则允许当前结果进行写回，否则丢弃该结果。

与ATU中的数据左移同样原理，从访问队列中返回的数据是左移位对齐偏移量的，而处理器GPR中存放的是右对齐的数据，因此需要右移位并进行符号位拓展。

## 算数逻辑管线ALU（短整数管线）

算术逻辑单元（ALU）处理整数的运算，只需要一个Cycle就可以得到结果。下表中表述了ALU opcode对应的操作：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 操作码/Opcode | 操作信息/opinfo | Data Out1 | Data Out2 | 备注 |
| NOP | XX | DataSource1 | DataSource2 | 直通 |
| JAL | XX | DataSource1 + 4 | DatsSource2 |  |
| CSRW | XX | DataSource2 | DataSource1 |  |
| CSRC | XX | DataSource2 | DataSource1中对应DataSource2的位清零 |  |
| CSRS | XX | DataSource2 | DataSource1中对应DataSource2的位置1 |  |
| ADD | Sign32 | DataSource1 + DataSource2 | XX | 操作低32位值，计算结果进行符号位拓展到64位。 |
| ADD | Sign64 | DataSouece1 + DataSource2 | XX | 全长的64位操作 |
| SUB | Sign32 | DataSource1 - DataSource2 | XX | 操作低32位值，计算结果进行符号位拓展到64位。 |
| SUB | Sign64 | DataSource1 - DataSource2 | XX | 全长的64位操作 |
| SLT | Sign64 | 如果DataSource1 < DataSource2，则为1 | XX | 把DataSource1和2当作有符号数进行比较 |
| SLT | Unsign64 | 如果DataSource1 < DataSource2，则为1 | XX | 把DataSource1和2当作无符号数进行比较 |
| XOR | XX | DataSource1 XOR DataSource2 | XX | 全长的64位操作 |
| OR | XX | DataSource1 OR DataSource2 | XX | 全长的64位操作 |
| AND | XX | DataSource1 AND DataSource2 | XX | 全长的64位操作 |
| SL | Sign32 | DataSource1左移，移位数由DataSource2指定 | XX | 对低32位进行移位操作，结果使用符号拓展到64位 |
| SL | Sign64 | DataSource1左移，移位数由DataSource2指定 | XX | 对低32位进行移位操作，结果使用符号拓展到64位 |
| SR | Sign64 | DataSource1右移，移位数由DataSource2指定 | XX | 对全长64位进行移位，右移位的产生的空位填符号位 |
| SR | Unsign64 | DataSource1右移，移位数由DataSource2指定 | XX | 对全长64位进行移位，右移位的产生的空位填0 |
| SR | Sign32 | DataSource1右移，移位数由DataSource2指定 | XX | 对32位进行移位，右移位的产生的空位填符号位 |
| SR | Unsign32 | DataSource1右移，移位数由DataSource2指定 | XX | 对全长32位进行移位，右移位的产生的空位填0 |

表：短整数操作码和操作

# 提交(Commit)/写回(Write Back)

## 写回顺序管理

从每个管线写回的数据首先送到写回仲裁单元进行仲裁，只有位于DITF出口处的指令才会被允许写回，否则是不会写回的。

## 机器状态寄存器CSR

根据RISC-V手册的划分，我们的CSR划分为三个部分：Information（机器信息）、Trap Manage（中断管理）、Trap handle（中断处理）、 Trap setup（中断配置）。每个大类下包含的CSR如下表：

|  |  |  |  |
| --- | --- | --- | --- |
| Machine & Supervisor Trap Setup | | | |
| Number | Privilege | Name | Desc |
| 0x300 | MRW | mstatus | 机器模式状态 |
| 0x100 | SRW | sstatus | Supervisor模式状态 |
| 0x301 | MRO | misa | 机器模式支持的ISA |
| 0x302 | MRW | medeleg | 机器模式异常委托 |
| 0x303 | MRW | mideleg | 机器模式中断委托 |
| 0x304 | MRW | mie | 机器模式中断使能 |
| 0x104 | SRW | sie | Supervisor模式中断使能 |
| 0x305 | MRW | mtvec | 机器模式中断向量 |
| 0x105 | SRW | stvec | Supervisor模式中断向量 |
| 0x306 | MRW | mcounteren | 机器模式计数器访问使能 |

|  |  |  |  |
| --- | --- | --- | --- |
| Machine & Supervisor Trap Handling | | | |
| Number | Privilege | Name | Desc |
| 0x340 |  | mscratch | Machine模式临时寄存器 |
| 0x140 |  | sscratch | Supervisor模式临时寄存器 |
| 0x341 |  | mepc | Machine模式异常PC寄存器 |
| 0x141 |  | sepc | Supervisor模式异常PC寄存器 |
| 0x342 |  | mecause | Machine模式异常原因寄存器 |
| 0x142 |  | scause | Supervisor模式异常原因寄存器 |
| 0x343 |  | mtval |  |
| 0x143 |  | stval |  |
| 0x344 |  | mip |  |
| 0x144 |  | sip |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Machine & Supervisor Counters | | | |
| Number | Privilege | Name | Desc |
| 0xB00 | MRW | mcycle |  |
| 0xB02 | MRW | minstret |  |
| 0x320 | MRW | mcountinhibit |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Supervisor 模式地址保护 | | | |
| Number | Privilege | Name | Desc |
| 0x180 |  | satp | Supervisor模式地址转换和保护 |
| Debug模式寄存器 | | | |
| Number | Privilege | Name | Desc |
| 0x7b0 | MRW | dcsr | Debug模式状态寄存器 |
| 0x7b1 | MRW | dpc | Debug模式PC |
| 0x7b2 | MRW | dscratch0 |  |
| 0x7b3 | MRW | dscratch1 |  |

## 异常管理

在指令被提交写回时，所有的中断和异常都被交给CSR模块中的Trap Manage模块进行处理，并生成Trap Target、Trap cause（标明当前异常应当交给哪个权限进行处理）。

## 返回

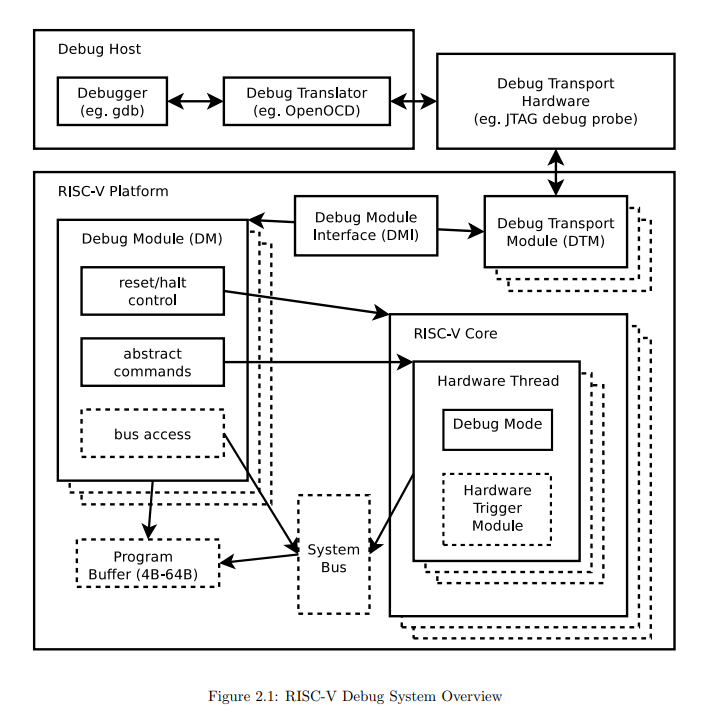
DITF中的MRET和SRET位标识了当前指令是否是返回指令，如果是返回指令，相关的CSR会执行返回动作。

# Debug机制

## 概览

RISC-V的debug系统由几个模块构成：Debug Host，如个人计算机，运行调试器，如gdb、debugger和Debug Translator，如OpenOCD进行通讯、OpenOCD和Debug Transport如JTAG适配器连接，并使用JTAG电缆连接到平台上。

平台内部由DTM、DMI、DM三大部分组成，整体模型如下图（源于riscv-debug-release.pdf）：



如果在”PRV564\_config.v”中define了DEBUG\_MODULE\_ENABLE，即开启了核心的debug单元。PRV564的Debug单元主要由三部分构成：

1. Debug Transport Module（DTM）：提供JTAG-Debug Module的协议转换。
2. Debug Module（DM），提供对核心内部的Hart Debug Unit的访问寄存器界面。
3. Hart Debug Unit，嵌入到流水线内部的Debug单元，提供数个程序可见的CSR寄存器界面和单步调试机制。

需要注意的是，PRV564并不支持program buffer。

## 调试转换模块（Debug Transport Module）

### JTAG接口

### JTAG-Kernel时钟域同步

### TAP寄存器

## 调试模块（Debug Module）

### 概览

调试模块是连接流水线中内嵌的HDU（Hart Debug Unit）和DTM的桥梁，将处理器内部的一系列调试行为封装为抽象命令方便调试器访问。

DM提供下列功能：

1. 提供必要的信息，例如调试器版本号。（支持）
2. 任意线程均可被停止或恢复（支持）
3. 提供每个线程的状态。（支持）
4. 提供读写GPRs的指令。（支持）
5. 在任意指令被执行前即可开始进行debug。（支持）
6. 提供无复位状态下的debug机制，即不管核心在任意状态下，均可debug（可选，支持，但无验证）
7. 提供访问非GPR寄存器的指令（可选，支持）
8. 提供Program Buffer来让核心支持额外的代码（可选，不支持）
9. 支持多条线程同时被停止，恢复。（可选，不支持）
10. 能够从内核端访问到内存（可选，不支持）
11. 能够直接访问系统内存。（可选，不支持）

### 调试模块接口（Debug Module Interface）

### 抽象命令（Abstract Command）

支持两种抽象命令：Access Memory和Access Registers。

#### 访问寄存器（Access Registers）

通过将此命令写入command寄存器以进行寄存器访问，命令格式如下：

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31:24 | 23 | 22:20 | 19 | 18 | 17 | 16 | 15:0 |
| cmdtype | 0 | size | aarpostincrement | postexec | transfer | write | regno |

|  |  |
| --- | --- |
| 域 | 功能 |
| cmdtype | Access Register时填0。 |
| size | 应当填3，其他的访问不支持。 |
| aarpostincrement | 0：执行完成后regno不变。  1：执行完成后regno自增1。 |
| postexec | 应当填0。 |
| transfer | 0：不执行write定义的操作。  1：执行write定义的操作。 |
| write | 0：将regno段指定的寄存器中的数据复制到data0，data1寄存器。  1：将data0，data1寄存器指定的数据复制到regno段指定的寄存器。 |
| regno | 访问的寄存器 |

|  |  |
| --- | --- |
| regno | 名称 |
| 0x0000~0x0FFF | CSR |
| 0x1000~0x101F | GPR |
| 0x1020~0xFFFF | reserved |

只有在核心被Halt之后才能执行这条命令，会进行如下操作：

1. 如果write位置0，transfer位置1，则将regno指定的寄存器拷贝到arg0中，arg0即data0和data1寄存器拼成的64位寄存器。
2. 如果write位和transfer位均置1，则将arg0中的数据拷贝到regno指定的寄存器中。
3. 如果postexec被置1，应当被忽略。

#### 访问内存（Access Memory）

调试器可以通过Access Memory命令来访问内存，命令格式如：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31：24 | 23 | 22：20 | 19 | 18：17 | 16 | 15：0 |
| cmdtype | aamvirtual | aamsize | aampostincrement | 0 | write | 0 |

|  |  |
| --- | --- |
| **域** | **描述** |
| **cmdtype** | 命令类型，应当为0 |
| **aamvirtual** | 0：使用物理地址模式进行访问。  1：使用虚拟地址模式进行访问，若不支持此模式，将会产生cmderr。 |
| **aamsize** | 0：8位  1：访问16位  2：访问32位  3：访问64位  4：访问128位 |
| **aampostincrement** | 0：无作用  1：执行完成后regno自增，自增大小等于aamsize中定义的大小。 |
| **write** | 0：将arg1定义的地址处的数据搬移到arg0寄存器。  1：将arg0的数据写到arg1定义的地址处。 |

### 调试模块寄存器

调试模块寄存器可以在Debug Module Interface上被直接访问，按照每32bit的粒度进行字编址。

|  |  |  |
| --- | --- | --- |
| 地址 | 名称 | 定义 |
| 0x00~0x03 | - | 保留 |
| 0x04~0x0f | data0~data11 | 命令参数寄存器0，低32位 |
| 0x10 | DMCONTROL | Debug Module控制寄存器 |
| 0x11 | DMSTATUS | Debug Module状态 |
| 0x12 | hartinfo |  |
| 0x13 | haltsum1 |  |
| 0x14 | hawindowsel |  |
| 0x15 | hawindow |  |
| 0x16 | ABSTRACTS | 抽象命令执行状态 |
| 0x17 | COMMAND | 抽象命令 |
| 0x18~0x39 | - |  |
| 0x40 | HALTSUM0 | 线程情况 |
| 0x41~0x7F | - |  |

表：DMI registers map

#### DMCONTROL

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 31 | 30 | 29:26 | 25:16 | 15:2 | 1 | 0 |
| haltreq | resumereq | 0 | hartsel | 0 | ndmreset | dmactive |

表：DMCONTROL寄存器

|  |  |  |  |
| --- | --- | --- | --- |
| 域 | 功能 | 访问 | 复位值 |
| haltreq | 当设置为1时，如果当前未停止，每个选定的hart将停止。  写入1或0对已停止的hart没有影响，但在hart恢复之前，应将位清除为0。同时设置haltreq和resumereq会导致未定义的行为。 | R/W | 0 |
| resumereq | 设置为1时候，线程将恢复运行，如果一个线程已经恢复则无任何影响。 | R/W | 0 |
| hartsel | 选中的线程，从0~1023 |  |  |
| ndmreset | 对除DM之外的模块进行复位，写1有效。之后写0以结束复位。 | R/W | 0 |
| dmactive |  |  |  |

#### DMSTATUS

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 31:18 | 17 | 16 | 15 | 14 | 13 |
| 0 | allresumeack | anyresumeack | allnonexistent | anynonexistent | allunavail |

|  |  |  |  |
| --- | --- | --- | --- |
| 12 | 11 | 10 | 9 |
| anyunavail | allrunning | anyrunning | allhalted |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 8 | 7 | 6 | 5 | 4 | 3:0 |
| anyhalted | authenticated | authbusy | 0 | cfgstrvalid | version |

|  |  |  |  |
| --- | --- | --- | --- |
| 域 | 功能 | 访问 | 复位值 |
| allresumeack | 1：当前所有线程已经恢复运行。 | R | 0 |
| anyresumeack | 0：当前选中的线程没有一个恢复运行。  1：当然选中的任意一个线程已经恢复运行。 | R | 0 |
| allnonexistent | 常数0，当前选中的线程总是存在的。 | R | 0 |
| anynonexistent | 常数0，当前选中的线程总是存在的。 | R | 0 |
| allunavail | 常数0，当前选中的线程总是有效的。 | R | 0 |
| anyunavail | 常数0，当前选中的线程总是有效的。 | R | 0 |
| allrunning | 0：当前选择的线程没有全部在运行。  1：当前选择的线程全部在运行。 | R | 0 |
| anyrunning | 0：当前选择的线程没有一个是运行的状态。  1：当前选择的线程有任意一个是运行的状态。 | R | 0 |
| allhalted | 0：当前选择的线程没有全部Halt。  1：当前选择的线程全部Halt。 | R | 0 |
| anyhalted | 0：当前选择的线程没有一个是Halt的状态。  1：当前选择的线程有任意一个是Halt的状态。 | R | 0 |
| authenticated | 已经被验证过，常数1 | R | 1 |
| authbusy | 验证模块没有忙，常数0 | R | 0 |
| cfgstrvalid | 0 | R | 0 |
| version | 0：如果没有配置打开debug模块。  2：如果配置打开debug模块，表示调试器版本为V0.13。 | R | 2 |

#### ABSTRACTCS

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 31:29 | 28:24 | 23:13 | 12 | 11 | 10:8 | 7:5 | 4:0 |
| 0 | progsize | 0 | busy | 0 | cmderr | 0 | datacount |

|  |  |  |  |
| --- | --- | --- | --- |
| 域 | 功能 | 访问 | 复位值 |
| progsize | 常数0，程序段大小，PRV564没有实现program buffer。 | R | 0 |
| busy | 1：当前的命令正在执行。 | R | 0 |
| cmderr | 0：no error  1：忙，当前操作正在进行。  2：当前命令不支持。  3：在执行当前命令时发生错误。  4：当前线程没有在Halt状态，命令无法执行。  7：未知错误  向此寄存器写1以清除位。 | R/W1C | 0 |
| datacount | 常数4，本机支持4个32bit的参数寄存器。 | R | 4 |

#### COMMAND

32位指令寄存器，将抽象命令写入此寄存器以使DM执行。

|  |
| --- |
| 31:0 |
| command |

## 线程Debug单元（Hart Debug Unit）

线程Debug单元，主要有以下功能：

1. 提供程序可见的CSR寄存器：dcsr、dpc、dscratch0、dscratch1.
2. 通过设置dcsr中的step位来进行单步运行。
3. 通过设置dcsr中ebreakm/s/u提供不同模式下的断点。
4. 管理调试器传来的Halt请求。

### 新增的Debug CSR

#### Debug模式下的csr：dcsr

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 31:28 | 27:16 | 15 | 14 | 13 | 12 | 11:9 | 8:6 | 5:3 | 2 | 1:0 |
| xdebugver | 0 | ebreakm | 0 | ebreaks | ebreaku | 0 | cause | 0 | step | priv |

dcsr寄存器

|  |  |  |  |
| --- | --- | --- | --- |
| 域 | 功能 | 访问 | 复位值 |
| xdebuger | 支持的外部调试器版本，如果定义了DEBUG\_MODULE\_EN，此域读出来的值为4；否则为0。 | R | 4/0 |
| ebreakm | Machine模式下的ebreak指令将会引起处理器进Debug模式，并Halt。如果不设置，则按正常流程执行Machine模式下的ebreak指令。 | R/W | 0 |
| ebreaks | Supervisior模式下的ebreak指令将会引起处理器进Debug模式，并Halt。如果不设置，则按正常流程执行S模式下的ebreak指令。 | R/W | 0 |
| ebreaku | User模式下的ebreak指令将会引起处理器进Debug模式，并Halt。如果不设置，则按正常流程执行U模式下的ebreak指令。 | R/W | 0 |
| cause | 造成进入debug模式的原因。  1：ebreak指令。  3：Debug Module请求进入Debug模式。  4：单步运行模式。 | R/W | 0 |
| step | 1：单步运行模式，每次运行完一条指令后都会进debug模式。  0：关闭单步运行模式。 | R/W | 0 |
| priv | 处理器权限，调试器可以写此寄存器来更改处理器权限。 | R/W | 2’b11（M模式） |

dcsr位定义

#### dpc：

Debug PC寄存器记录了当前进入Debug模式的指令的虚拟地址，在不同情况下的dpc寄存器值为：

|  |  |
| --- | --- |
| 情况 | 值 |
| 单步模式/调试器Halt请求 | 更新为下一条还未执行的指令的虚拟地址。 |
| ebreak指令，且在dcsr中设置了对应的位。 | 更新为ebreak指令的地址。 |

|  |
| --- |
| 63:0 |
| dpc |

如果在单步模式下执行ebreak指令，但是没有设置dcsr中对应的位，dpc会更新为下一条要执行的指令，即进入异常。这样的机制是为了方便在单步模式下跟踪处理器进入异常-退出异常的过程。

当resume之后，处理器跳转到dpc处继续运行，因此调试器可以在核心halt之后更改dpc寄存器，而后resume来更改处理器的pc。

#### dscratch0/dscratch1：

为正常的临时值寄存器。

### 调试器Halt请求

调试器通过拉高DM\_haltreq信号来使核心停止并进入debug模式，当核心停止进入debug模式后将halted信号拉高，此时调试器可以访问核心内部的寄存器。

### 单步模式（Step Mode）

调试器或者程序都可以设置dcsr寄存器的step位来使核心单步运行，在单步运行模式下，每执行一条指令都会使核心进入debug模式。除非调试模块（Debug Module）使能DM\_resume信号，否则核心会一直等待。

### 断点模式（Break Point）

在设置了dcsr中的ebreakm/s/u位后，在程序中使用ebreak指令可以让处理器在对应位置停下进入debug模式。

如果没有设置dcsr的ebreakm/s/u位，处理器将不会在这个ebreak指令处进入debug模式，而是按照正常流程运行，即进入异常。

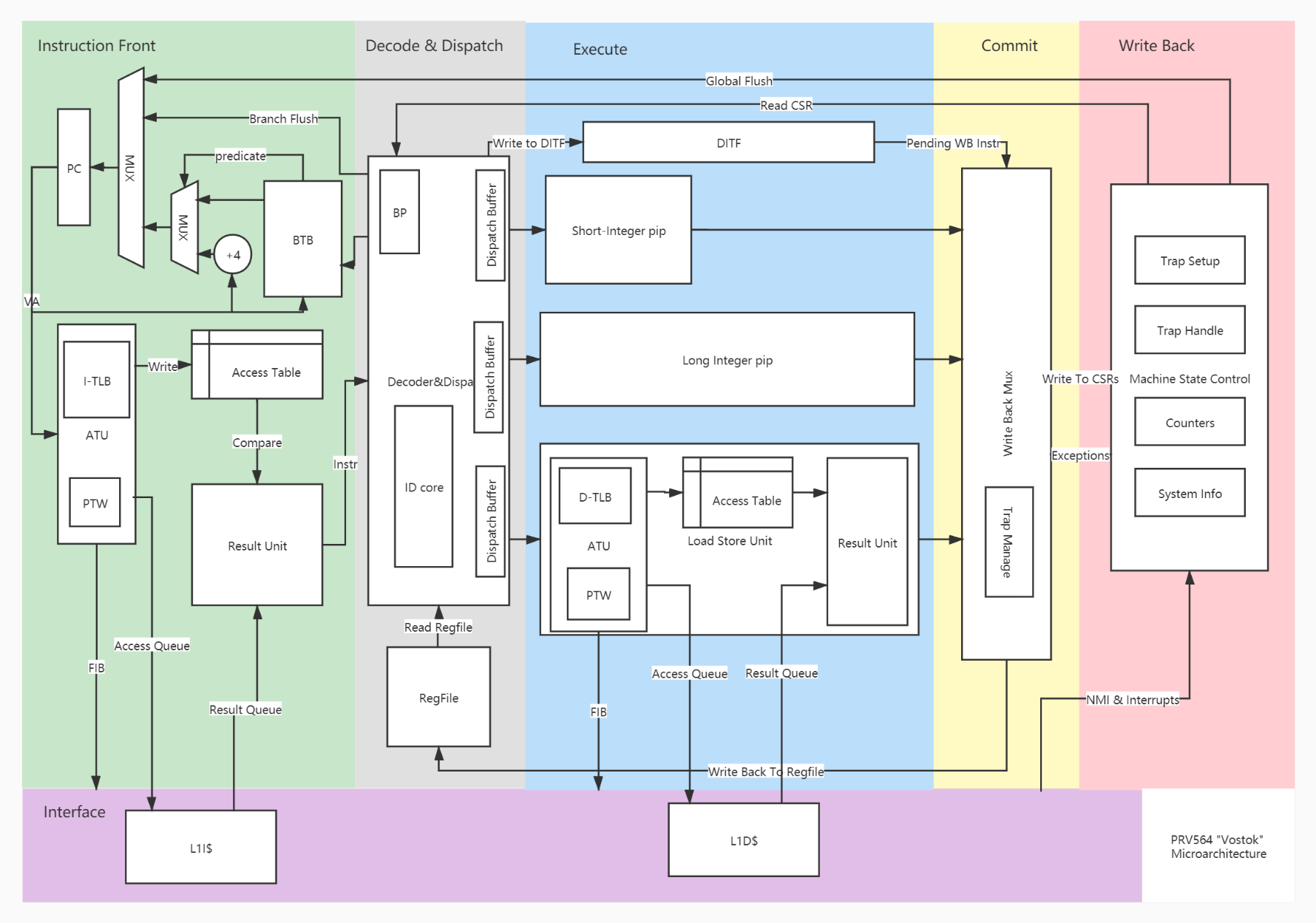
## 安全模式（Security）

有两种方案实现安全模式：

1. 在DMI总线上映射额外的寄存器，完成内部测试后写eFUSE锁定DM。
2. 使用RISC-V标准定义的安全机制：向Authentication Data寄存器中填密钥，读取dmstatus寄存器中authenticated位，确定是否已经被验证，仅当被验证之后才能访问到DM中的寄存器。

|  |
| --- |
| NOTE：   1. 在每次复位之后，都需要重新填写密钥。 2. 564不支持安全方案，在每次复位后authenticated位都会被置1表示已经通过验证。 |

# 附录：核心架构图



# L1缓存

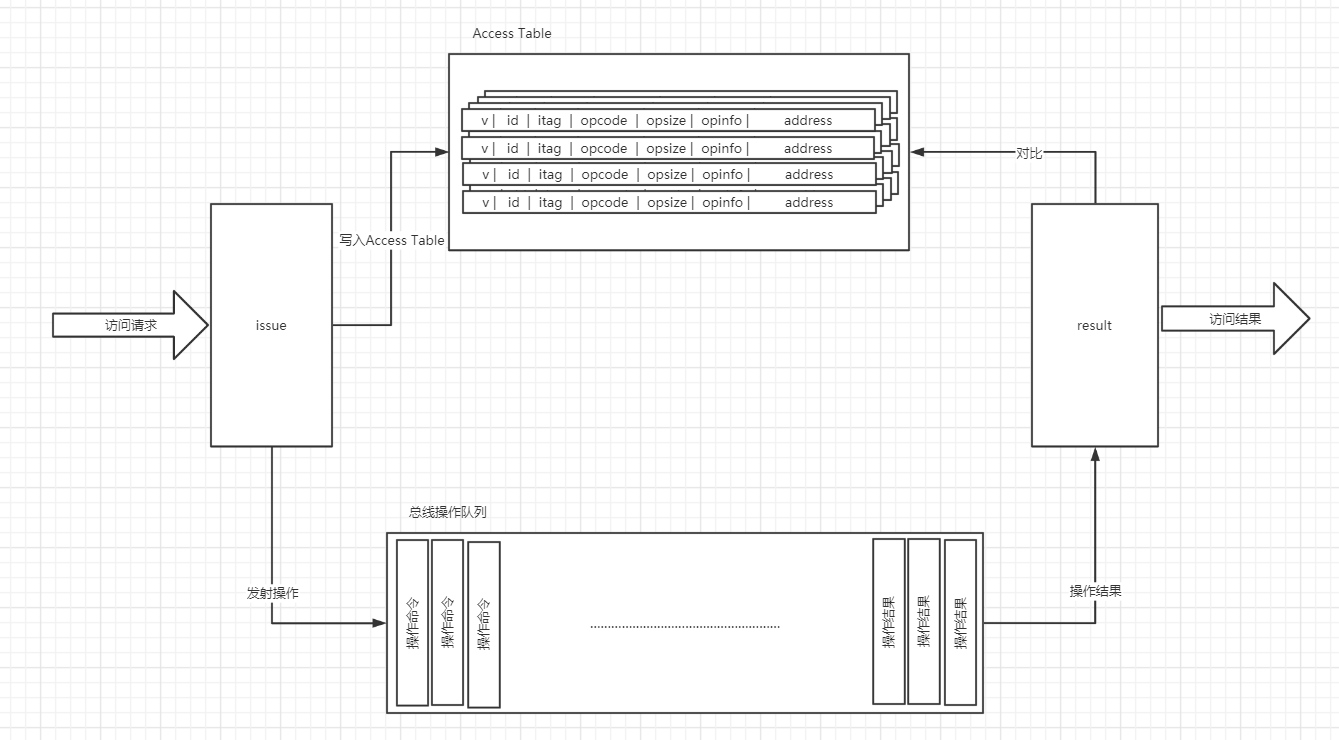
## 组织方式

# 附录：TLB间互联

TLB间通过ID+传输命令的方式进行互联，发起传输的主机端会给每个传输分配一个独一无二的ID，并且在主机端内部记录此ID。

# 附录：队列化的总线访问接口

基本架构由Access Table，发射单元，结果单元，总线操作队列构成，模块之间的连接情况如下图：



图：基本架构

## 发射单元（Issue）

发射单元将总线操作命令发射到操作队列中，并给这个操作命令分配一个独一无二的ID，在发射成功的同时，将当前访问的信息（指令itag、指令地址等信息）写入到Access Table中。

默认情况下，我们认为被发射到操作队列里的操作是不可以被取消的，因此发射单元在不确定当前派遣的操作是否会被取消的时候，不应当派遣修改内存的指令（Write操作），但是可以发射读内存操作。

## 访问表（Access Table）

Access table记录了当前所有正在等待结果的指令，这样在操作返回结果后可以找到是哪一条指令访问的内存，并从表中提取出这条指令的信息以进行写回，并移除这个表项。

当流水线冲刷时，表项中的所有内容都将被清零，即所有等待内存访问结果的指令都被取消了。

## 结果单元（Result）

从访问队列中返回的结果会和Access Table中的表项进行比对，如果在Access Table中找到了正在等待结果的指令（即ID相同），则这条指令已经执行完成，就可以从访问表中取出。

如果在Access Table里面找不到对应的表项，说明这条操作已经被取消了，操作结果作废。

## 操作和结果队列

将总线的操作抽象成如同FIFO一样的操作队列，本文在这里提出一种接口：

|  |  |  |
| --- | --- | --- |
| 名称 | 位宽 | 功能 |
| AQ\_V | 1 | Valid，添加一个表项到操作队列 |
| AQ\_ID | 8 | 这条操作的ID |
| AQ\_CMD | 8 | 命令：8bit |
| AQ\_BSEL | 16 | 字节选择 |
| AQ\_CI | 1 | Cache Inhibit，为1时禁用缓存 |
| AQ\_WDATA | 128 | 要写的数据 |
| AQ\_ADDR | 64 | 地址 |
| AQ\_FULL | 1 | 命令队列满 |

\*AQ：Access Queue，操作队列

|  |  |  |
| --- | --- | --- |
| RQ\_V | 1 | 这条结果有效 |
| RQ\_ID | 8 | 这条结果的ID |
| RQ\_WRERR | 1 | 写错误 |
| RQ\_RDERR | 1 | 读错误 |
| RQ\_RDY | 1 | 传输完成 |
| RQ\_RDATA | 128 | 读到的数据 |
| RQ\_ACK | 1 | 从结果队列中读出一项 |

RQ： Result Queue。结果队列

每一个Cache操作都对应一个CMD，CMD考虑有：

|  |  |
| --- | --- |
| 操作助记符 | 操作 |
| NOP | No operation |
| READ | Read |
| WRITE | Write |
| SYNC | Sync memory，将缓存中的脏行写回，并无效化缓存。 |
| LR | 互斥读 |
| SC | 互斥写 |
| AMOSWAP | 原子交换 |
| AMOADD | 原子加 |
| AMOAND | 原子AND |
| AMOOR | 原子OR |
| AMOXOR | 原子XOR |
| AMOMAX | 原子MAX |
| AMOMAXU | 原子MAX（不带大小） |
| AMOMIN | 原子小 |
| AMOMINU | 原子小（不带符号） |

# 附录：灵活互联总线（FIB）

灵活互联总线，简写为FIB（Flexible Interconnection Bus）用于处理器内部和总线接口部分的互联，该总线被设计为适应多级缓存和跨时钟的结构，一个FIB连接实例如下所示：

FIB总线跨时钟桥

FIB总线接口单元

FIB主机2

FIB主机1

FIB主机3

FIB主机0

　　FIB总线的所有传输都不涉及到精确到多个时钟Cycle的同步，因此没有AHB总线般的别扭握手协议。

## 信号规定

FIB主机接口含有如下信号：

|  |  |  |
| --- | --- | --- |
| 名称 | 位宽 | 功能 |
| FIBMo\_CLK | 1 | 主机时钟输出 |
| FIBMo\_WREN | 1 | 写有效 |
| FIBMo\_REQ | 1 | 请求占用总线（给总线复用器用） |
| FIBMi\_ACK | 1 | 总线占用允许（给总线复用器用） |
| FIBMi\_FULL | 1 | 当前FIFO已被写满 |
| FIBMo\_ID | 8 | 当前传输所用ID |
| FIBMo\_CMD | 8 | 当前传输命令 |
| FIBMo\_BURST | 4 | 连续传输模式 |
| FIBMo\_SIZE | 4 | 传输大小/连续传输自增大小 |
| FIBMo\_ADDR | XLEN | 地址 |
| FIBMo\_DATA | XLEN | 数据 |
| FIBMi\_V | 1 | 接收有效 |
| FIBMi\_RPL | 8 | 应答 |
| FIBMi\_ID | 8 | 应答ＩＤ |
| FIBMi\_ADDR | XLEN | 应答地址（不需要内存一致性的模块实现可以忽略） |
| FIBMi\_DATA | XLEN | 应答数据 |

FIB从机接口含有相似的信号：

|  |  |  |
| --- | --- | --- |
| 名称 | 位宽 | 功能 |
| FIBSo\_CLK | 1 | 从机读时钟 |
| FIBSo\_RDEN | 1 | 读使能 |
| FIBSi\_V | 1 | 接收到的命令有效 |
| FIBSi\_ID | 8 | 接收到的传输ID |
| FIBSi\_CMD | 8 | 接收到的传输命令 |
| FIBSi\_BURST | 4 | 接收到的连续传输大小 |
| FIBSi\_SIZE | 4 | 接收到的传输大小 |
| FIBSi\_ADDR | XLEN |  |
| FIBSi\_DATA | XLEN |  |
| FIBSo\_WREN | 1 | 回传应答信息写使能 |
| FIBSi\_FULL | 1 | 回传通道满，请等待 |
| FIBSo\_ID | 8 | 回传应答ID |
| FIBSo\_RPL | 8 | 回传应答信息 |
| FIBSo\_ADDR | XLEN |  |
| FIBSo\_DATA | XLEN |  |

注意事项：

FIB跨时钟桥应当具备Valid位功能，用于指示跨过该时钟桥的数据是否是有效的，这个valid位直接连接到FIB主/从机端的V位上。

FIB总线中的应答地址用于传输缓存一致性信息，如果模块不需要接收一致性信息，则忽略地址。

FIB设计用于跨FIFO的数据包传输，因此信号中包含了方便连接FIFO的信号。一个FIB端到端连接的例子如下图：

FIB主机

FIFO

FIFO

FIB从机

一个多主机的FIB连接如下图：

FIB主机

FIB主机

FIB主机

复用器

FIFO

FIFO

FIB从机

## 发送数据包格式

FIB总线数据包中包含了ID、传输类型和数据。

|  |  |  |
| --- | --- | --- |
| 名称 | 位宽 | 功能 |
| ID | 8 | 传输ID |
| CMD | 8 | 传输名称 |
| BURST | 4 | 连续传输长度 |
| SIZE | 4 | 传输大小 |
| ADDR | XLEN | 传输地址 |
| DATA | XLEN | 传输数据 |

\*XLEN表示用户可以根据需求改变长度

一个64位代表宽度的FIB数据包格式如下表：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 63：0 | 63：0 | 3：0 | 3：0 | 7：0 | 7：0 |
| DATA | ADDR | SIZE | BURST | CMD | ID |

### ID

ID标明当前传输的ID，每个FIB主机拥有唯一的一个ID，在返回的数据包中，只有符合ID的数据包才会被主机识别到。

设备可用的ID范围是1-255，ID=0时表示全局广播。

例如：一FIB设备的ID号为0x05，则该设备发出的总线操作命令的ID号均为0x05，在返回的数据包中只有ID为0x05的才会被该设备接受处理，否则不处理。

### CMD

传输命令。

|  |  |  |
| --- | --- | --- |
| 值 | 助记符 | 描述 |
| 0x00 | NOOP | 空操作 |
| 0x01 | SIGR | 读取一个值 |
| 0x02 | SIGW | 写一个值 |
| 0x03 | SEQR | 连续读取值 |
| 0x04 | SEQW | 连续写值 |
| 0x05 | WAPR | 回卷读（暂不实现） |
| 0x06 | WAPW | 回卷写（暂不实现） |
| 0x07 | AMOR | 原子读，执行该命令后锁定总线 |
| 0x08 | AMOW | 原子写，执行完该命令后释放总线 |
| 0x14 | SEQE | 连续传输结束符 |

### BURST

Burst表示了当前连续传输/回卷传输的拍数。拍数大小为2^(Burst)。

|  |  |
| --- | --- |
| Burst值 | 大小 |
| 0x00 | 0 |
| 0x01 | 2 |
| 0x02 | 4 |
| 0x03 | 8 |
| 0x04 | 16 |
| …… | …… |

### SIZE

SIZE描述了传输的大小，传输大小=2^(SIZE)。

### ADDR

### DATA

## 接收数据包格式

FIB总线接受到的数据包中包含了ID、返回值和数据。

|  |  |  |
| --- | --- | --- |
| 名称 | 位宽 | 功能 |
| ID | 8 | 传输ID |
| RPL | 16 | 回复 |
| ADDR | XLEN | 传输地址 |
| DATA | XLEN | 传输数据 |

\*XLEN表示用户可以根据需求改变长度

一个64位代表宽度的FIB数据包格式如下表：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 63：0 | 63：0 | 3：0 | 7：0 | 7：0 |
| DATA | ADDR | SIZE | RPL | ID |

### RPL

RPL标明当前传输的情况。

|  |  |  |
| --- | --- | --- |
| 值 | 助记符 | 描述 |
| 0x00 | NOOP | 空 |
| 0x10 | TRDY | 传输完成（用于传输完成的应答） |
| 0x14 | SEQ | 正在连续传输（用于连续传输的应答） |
| 0x20 | MODF | 修改对应地址的数据（用于全局一致性） |
| 0xF0 | TERR | 发生错误 |
| 0xFF | IDLE | 当前所有总线命令完成，当FIB从机完成所有指令后发出，如果有数据包在传输过程中丢失了，则主机会再次发起传输。 |

## 传输

### 单次传输（SIGR & SIGW）

所有传输都是从REQ开始的，当ACK=1,即复用器通知确认可以进行传输后，开始进行传输，V=1时，即当前数据帧有效，每个数据帧只需要1个cycle即可传输完成。

**32bit单次写传输，回执为正确：**

发送的数据帧：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| DATA | ADDR | SIZE | BURST | CMD | ID | V |
| data | address | 4’b0100 | 0000 | SIGW | 主机ID | 1 |

接收到的数据帧：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| DATA | ADDR | SIZE | RPL | ID | V |
| xxxxxxx | xxxxxx | xxxx | TRDY | 主机ID | 1 |

**32bit单次读传输，回执正确：**

发送的数据帧：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| DATA | ADDR | SIZE | BURST | CMD | ID | V |
| xxxx | address | 4’b0100 | 0000 | SIGR | 主机ID | 1 |

接收的数据帧：

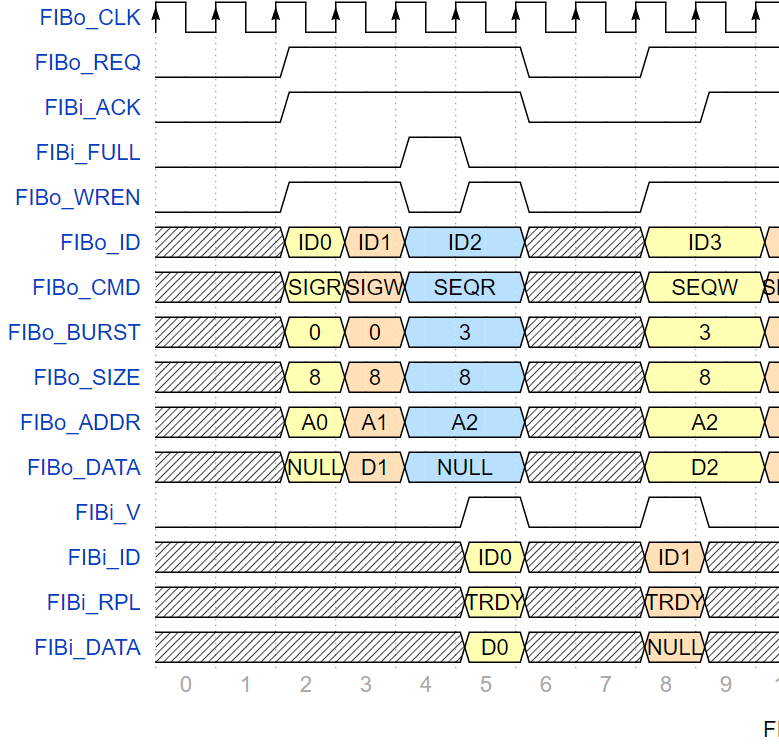
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| DATA | ADDR | SIZE | RPL | ID | V |
| read data | xxxxxx | xxxx | TRDY | 主机ID | 1 |

**传输发生错误：**

如果一个传输发生错误，则收到的回执格式如下：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| DATA | ADDR | SIZE | RPL | ID | V |
| xxxxx | xxxxxx | xxxx | TERR | 主机ID | 1 |

单次传输时序如下所示：



图：FIB单次传输

T2： 传输ID=0，传输命令SIGR（单次读），BURST大小填0，SIZE填8（8字节），传输地址A0，传输数据空。

T3： 传输ID=1，传输命令SIGW（单次写），BURST大小填0，SIZE填8（8字节），传输地址A1，传输数据D0。

T5： ID=0的传输被正确执行了，读取的数据为D0.

T8： ID=1的传输被正确完成了。

### 连续读传输（SEQR）

连续读命令以ID+SEQR开始，BURST区段填传输长度。

以一次64bit为粒度，Burst长度为8次的连续读传输为例：发送的数据帧的ADDR填连续读传输的基地址，SIZE填8，表明是8字节一次，BURST填3，表明访问的长度是2^3即8拍：

**发送的数据帧：**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| DATA | ADDR | SIZE | BURST | CMD | ID | V |
| xxxx | address | 4’b1000 | 0011 | SEQR | 主机ID | 1 |

当这个数据帧跨过重重阻碍，到达FIB桥，或者到达FIB主机后，经过处理给出回执，因为连续读写的长度是8，即有8个回执帧会送回来。所有回执帧中，只有在传输的最后一帧的回复是TRDY（已用红色部分标记），其余帧的回复均为SEQ，表明当前传输正在继续。

**回执帧0：**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| DATA | ADDR | SIZE | RPL | ID | V |
| read data0 | xxxxxx | xxxx | SEQ | 主机ID | 1 |

**回执帧1：**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| DATA | ADDR | SIZE | RPL | ID | V |
| read data1 | xxxxxx | xxxx | SEQ | 主机ID | 1 |

**回执帧2：**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| DATA | ADDR | SIZE | RPL | ID | V |
| read data2 | xxxxxx | xxxx | SEQ | 主机ID | 1 |

**回执帧3：**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| DATA | ADDR | SIZE | RPL | ID | V |
| read data3 | xxxxxx | xxxx | SEQ | 主机ID | 1 |

**回执帧4：**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| DATA | ADDR | SIZE | RPL | ID | V |
| read data4 | xxxxxx | xxxx | SEQ | 主机ID | 1 |

**回执帧5：**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| DATA | ADDR | SIZE | RPL | ID | V |
| read data5 | xxxxxx | xxxx | SEQ | 主机ID | 1 |

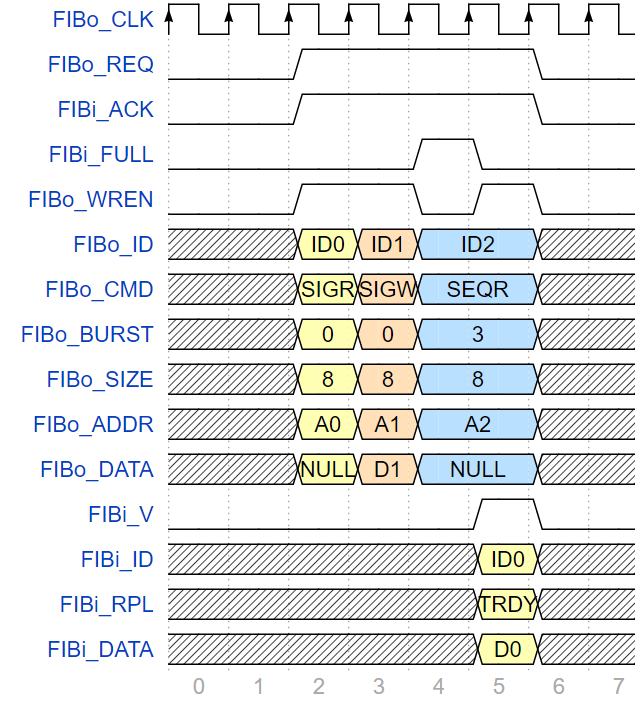
**回执帧6：**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| DATA | ADDR | SIZE | RPL | ID | V |
| read data6 | xxxxxx | xxxx | SEQ | 主机ID | 1 |

**回执帧7：**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| DATA | ADDR | SIZE | RPL | ID | V |
| read data7 | xxxxxx | xxxx | TRDY | 主机ID | 1 |

主机发起的一次连续读传输如下图所示：

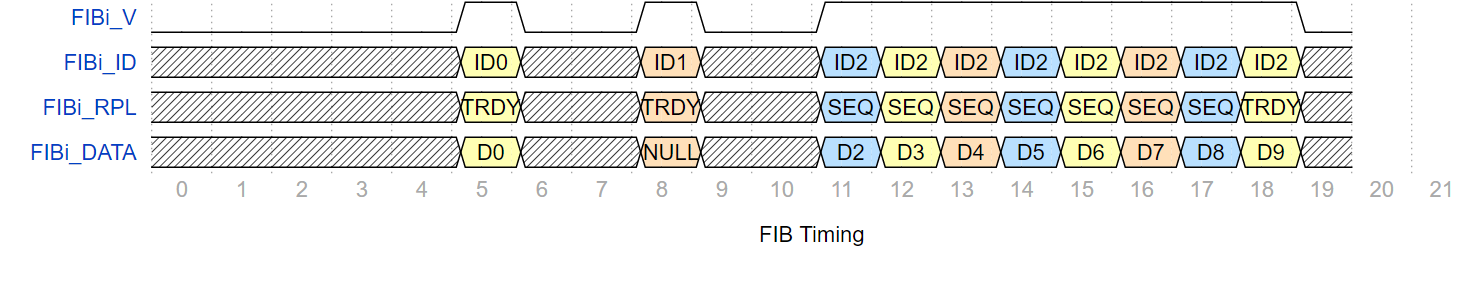


图：主机发起连续读传输

T4：主机发起连续读传输，但是FIFO满了，因此等待一个周期。

T5：发送FIFO空，主机发起连续传输有效。

等待N个周期后，数据连续读入：



T11-T18：数据被连续读取，回复内容为SEQ，在连续传输的最后一个位置回复为TRDY，标明此传输结束。

### 连续写传输

连续写传输由多个数据帧组成，这些数据帧以SEQW为开头，以SEQE为结束（红色部分标记），以一个8字节访问、8拍为长度的BURST传输为例：

**发送数据帧0：**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| DATA | ADDR | SIZE | BURST | CMD | ID | V |
| data0 | address | 4’b1000 | 0011 | SEQW | 主机ID | 1 |

**发送数据帧1：**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| DATA | ADDR | SIZE | BURST | CMD | ID | V |
| data1 | address | 4’b1000 | 0011 | SEQW | 主机ID | 1 |

**发送数据帧2：**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| DATA | ADDR | SIZE | BURST | CMD | ID | V |
| data2 | address | 4’b1000 | 0011 | SEQW | 主机ID | 1 |

**发送数据帧3：**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| DATA | ADDR | SIZE | BURST | CMD | ID | V |
| data3 | address | 4’b1000 | 0011 | SEQW | 主机ID | 1 |

**发送数据帧4：**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| DATA | ADDR | SIZE | BURST | CMD | ID | V |
| data4 | address | 4’b1000 | 0011 | SEQW | 主机ID | 1 |

**发送数据帧5：**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| DATA | ADDR | SIZE | BURST | CMD | ID | V |
| data5 | address | 4’b1000 | 0011 | SEQW | 主机ID | 1 |

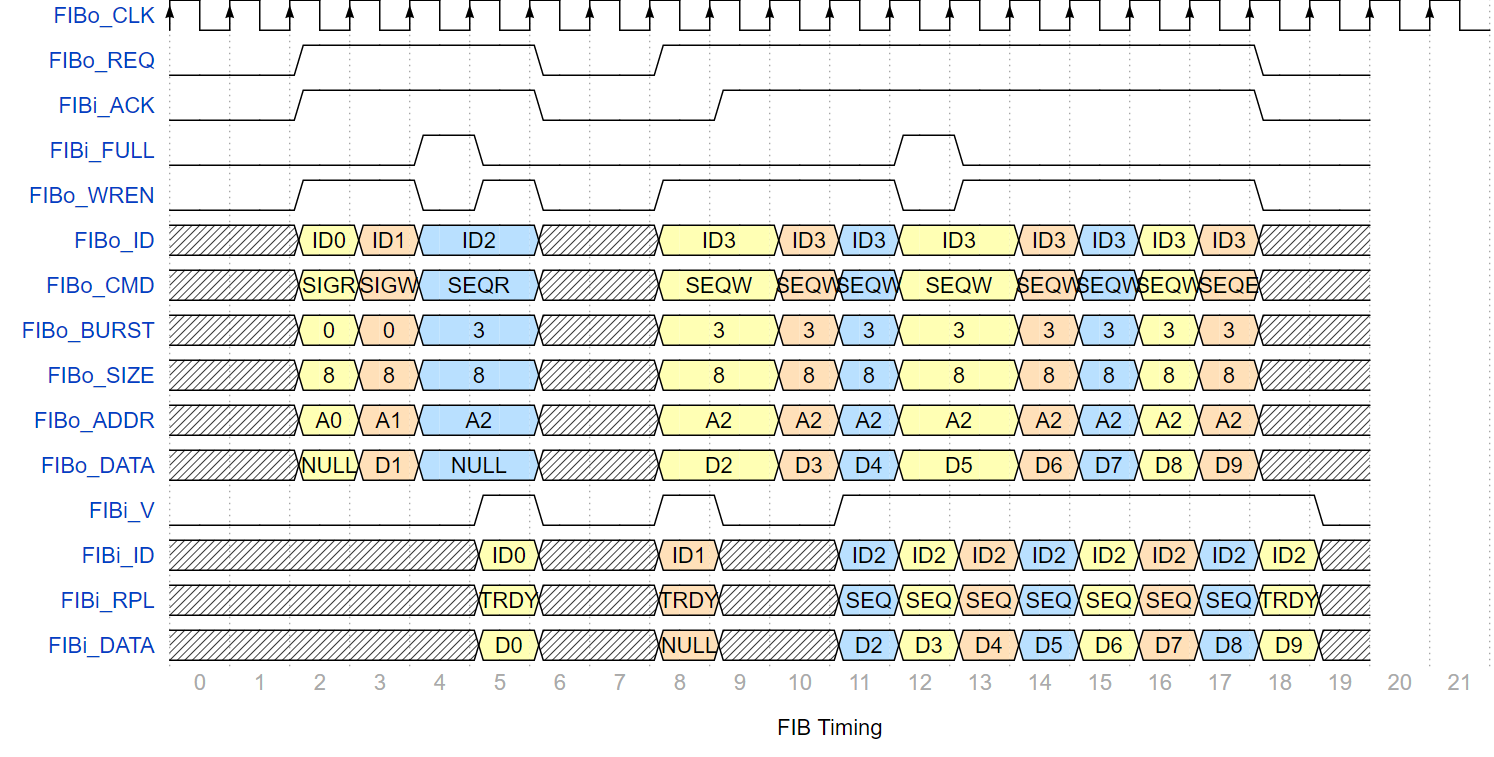
**发送数据帧6：**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| DATA | ADDR | SIZE | BURST | CMD | ID | V |
| data6 | address | 4’b1000 | 0011 | SEQW | 主机ID | 1 |

**发送数据帧7：**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| DATA | ADDR | SIZE | BURST | CMD | ID | V |
| data7 | address | 4’b1000 | 0011 | SEQE | 主机ID | 1 |

回复时，如果传输正确，即回复TRDY、传输错误，即回复TERR。



图：SEQW传输

T8：主机发起SEQ传输，但是没有ACK信号，等待一个周期。

T9：给出了ACK信号，主机可以占用FIB总线，开始传输。

T10-T16：连续进行传输

T17：传输的最后一拍，以SEQE结尾

### 锁定传输

在锁定传输中，FIB主机应当在整个锁定传输周期中保持FIBM\_REQ信号为高以确保整个FIB信令通道上只有此主机在使用。同时，为了让FIB从机能正确识别到这是一个锁定传输，应当使用特殊的FIB指令：AMOR和AMOW。

# 附录：FIB-A拓展

FIB-A拓展用于在标准FIB协议上拓展出支持原子指令的操作，从而更好地支持多核处理器。

## 新增命令

|  |  |  |
| --- | --- | --- |
| 值 | 助记符 | 操作 |
| 0x20 | LR | 从内存中读取一个值，并检测该地址是否被改动 |
| 0x21 | SC | 如果该地址在上次LR之后没有被改动，则存入。 |
| 0x22 | AMOSWAP | 对数据进行原子的交换，将读到的数据返回 |
| 0x23 | AMOADD | 对地址处的数据和命令中的数据进行ADD，结果存回地址处，并将刚才读到的数据返回。 |
| 0x24 | AMOXOR | 对地址处的数据和命令中的数据进行XOR，结果存回地址处，并将刚才读到的数据返回。 |
| 0x25 | AMOAND | 对地址处的数据和命令中的数据进行AND，结果存回地址处，并将刚才读到的数据返回。 |
| 0x26 | AMOOR | 对地址处的数据和命令中的数据进行OR，结果存回地址处，并将刚才读到的数据返回。 |
| 0x27 | AMOMIN | 对地址处的数据和命令中的数据进行比较，较小的存回地址处，并将刚才读到的数据返回。 |
| 0x28 | AMOMAX | 对地址处的数据和命令中的数据进行比较，较大的结果存回地址处，并将刚才读到的数据返回。 |
| 0x29 | AMOMINU | 对地址处的数据和命令中的数据（当作无符号数）进行比较，较小的结果存回地址处，并将刚才读到的数据返回。 |
| 0x2A | AMOMAXU | 对地址处的数据和命令中的数据（当作无符号数）进行比较，较大的结果存回地址处，并将刚才读到的数据返回。 |

## 操作格式

### 互斥读LR

以一个64位宽度的LR操作为例：

数据帧中包含了要进行AMO操作的地址

**发送的数据帧：**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| DATA | ADDR | SIZE | BURST | CMD | ID | V |
| data | address | 4’b1000 | 0000 | LR | 主机ID | 1 |

**回执帧：**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| DATA | ADDR | SIZE | RPL | ID | V |
| read data | xxxxxx | xxxx | TRDY | 主机ID | 1 |

### 条件写SC

条件写，Store Conditionally，是为了支持释放一致性模型而规定的。如果刚才执行了一条LR指令，则总线单元应当监视总线并确定刚才的地址是否被其他主机修改了，如果没有修改，则执行成功，数据被正确存放在地址处；如果被修改了，则执行SC命令时将会失败，数据不会存入地址处。

对于简单的实现，LR和SC允许处理成普通的读和写。

**数据帧：**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| DATA | ADDR | SIZE | BURST | CMD | ID | V |
| data | address | 4’b1000 | 0000 | SC | 主机ID | 1 |

**回执帧(如果执行成功)：**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| DATA | ADDR | SIZE | RPL | ID | V |
| 0x0 | xxxxxx | xxxx | TRDY | 主机ID | 1 |

**回执帧（如果执行失败）：**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| DATA | ADDR | SIZE | RPL | ID | V |
| 0x1 | xxxxxx | xxxx | TRDY | 主机ID | 1 |

### AMOSWAP等其他AMO指令

AMO指令是原子的从内存中读取一个值，并和命令中包含的数据进行运算得到结果，结果存回内存中。

以一个64位大小的AMO操作为例：数据帧包含了要操作的data和address。

**数据帧：**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| DATA | ADDR | SIZE | BURST | CMD | ID | V |
| data | address | 4’b1000 | 0000 | AMOSWAP | 主机ID | 1 |

从内存中读取的值会被返回，同时从address处取得的数据将会和数据帧里包含的数据进行运算，运算的结果存回内存中。

**回执帧：**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| DATA | ADDR | SIZE | RPL | ID | V |
| read data | xxxxxx | xxxx | TRDY | 主机ID | 1 |

# 附录：核心架构

# 附录：问题记录

2021/7/3：

标准化的缓存单元应当包含：读地址信号、对应缓存行写入、对应缓存行清除。执行AMO指令的过程中不通过缓存单元，直接操作FIB总线进行操作，操作完成后清除缓存中对应的缓存行（如果命中的话）。

2021/7/4：

冲刷应当影响流水线级中的指令，对于流水线输出寄存器应当谨慎处理，避免冲刷掉正在等待的指令。例如，假设ALU发出局部冲刷请求，此举将会清除派遣之前的所有指令，但是不能清除ID输出寄存器中的指令，因为这些指令可能正在等待后级处理。

2021/7/5

指令在写回时处理中断和异常，如果此指令是一条跳转指令，则不对该指令进行中断处理，此举是为了避免在写回阶段异常处理时，有多种PC\_next的情况。

2021/7/15

FENCE指令应当被发送到LSU进行处理，LSU中的ATU和数据单元。

2021/8/15

FENCE、FENCEI、FENCEVMA指令都应当在writeback级被处理。