ICS
CCS

团　体　标　准

T/CESA XXXX—202X

第五代精简指令集架构（RISC-V）通用图形处理器扩展指令集

RISC-V GPGPU Instruction Set Architecture Extension

草案

在提交反馈意见时，请将您知道的相关专利连同支持性文件一并附上。

已授权的专利证明材料为专利证书复印件或扉页，已公开但尚未授权的专利申请证明材料为专利公开通知书复印件或扉页，未公开的专利申请的证明材料为专利申请号和申请日期。

202X-XX-XX 发布　　202X-XX-XX 实施

中国电子工业标准化技术协会　　发　布

目　　次

前　　言

本文件按照 GB/T 1.1—2020《标准化工作导则第 1 部分：标准化文件的结构和起草规则》的规定起草。

本文件由 XXXX 提出。

本文件由 XXXX、中国电子工业标准化技术协会归口。

本文件起草单位：清华大学。

第五代精简指令集架构（RISC-V）通用图形处理器扩展指令集

1 范围

本文件规定一套面向通用图形处理器（GPGPU）的 RISC-V 扩展指令集架构。其覆盖内容包括：

- **RV32I 基础整数标量指令兼容**：提供第 4 章规定的标量整数指令，并支持 Zifencei 扩展定义的 FENCE.I。
- **GPGPU 扩展指令**：包括浮点、整数、类型转换、访存、原子读改写、控制流、跨 lane 协作、同步、状态访问、矩阵乘累加、TMA、TMEM 分配管理及大块异步搬运等通用图形处理器关键功能。
- **执行模型与编程模型**：定义 thread、lane、warp、CTA、cluster、grid 与 SM 的架构关系，以及逐线程和统一标量两套执行通路，为编译器、驱动程序、运行时和工具链提供统一依据。
- **寄存器、存储与地址转换**：定义 vreg、sreg、谓词和依赖计数器，规定 global、shared、local、constant 地址空间的访问规则、可见范围和同步关系，并按第 5 章规定本架构的 Sv39 地址转换要求。
- **指令结构与编码**：定义操作数、修饰符、谓词门控、公共控制字段、64 位、96 位和 128 位变长容器，以及各 GPIDL 扩展指令形态的编码图。
- **编程模型映射**：在资料性附录中给出任务粒度、地址空间、同步原语、寄存器和执行状态的概念映射。

本文件适用范围如下：

- **硬件设计**：用于 GPGPU 硬件的架构设计、功能建模和一致性验证。
- **软件开发**：用于驱动程序、编译器、汇编器、运行时、调试工具和应用程序开发。
- **教育和研究**：用于指令集、并行体系结构、编译和系统软件的教学、研究与原型验证。
- **商业应用**：适用于高性能计算、人工智能、机器学习、科学计算、嵌入式和边缘计算等大规模并行场景。

本文件规定架构可见行为，不规定具体微架构实现方式，也不规定驱动程序和运行时的内部实现、对象文件格式或完整软件 ABI。

本文件给出全部 GPIDL 扩展指令的容器长度、前缀、操作码和有效载荷字段位置。第 4 章规定的 RV32I 和 Zifencei 指令应采用相应 RISC-V 规范定义的编码，不使用 GPIDL 扩展指令编码表。第 5 章规定 Sv39 地址转换的架构要求；根页表配置、地址转换缓存失效、页表遍历和页故障交付的具体实现应在系统设计文件中另行规定。本文件不规定专用队列布局、描述符私有二进制布局、微架构状态机或驱动私有接口。

2 规范性引用文件

下列文件中的内容通过文中的规范性引用而构成本文件必不可少的条款。其中，注日期的引用文件，仅该日期对应的版本适用于本文件；不注日期的引用文件，其最新版本（包括所有修改单）适用于本文件。

- GB/T 1.1—2020 标准化工作导则第 1 部分：标准化文件的结构和起草规则
- IEEE 754-2019 IEEE Standard for Floating-Point Arithmetic
- RISC-V Instruction Set Manual, Volume I: Unprivileged Architecture：RV32I Version 2.1 与 Zifencei Version 2.0
- RISC-V Instruction Set Manual, Volume II: Privileged Architecture Version 1.13：Sv39 page-based virtual-memory system

3 术语和定义

3.1 术语表

本文件保留部分英文缩写、寄存器名、字段名和指令助记符，作为架构标识使用。除指令名和字段名本身外，相关概念在首次出现时给出中文含义；后文相同缩写按本术语表或上下文中的中文解释理解。

- RV32I：RISC-V 32 位基础整数指令集。
- Zifencei：RISC-V 指令取指栅栏扩展；FENCE.I 属于该扩展，不属于 RV32I。
- Sv39：RISC-V 特权架构定义的 39 位虚拟地址三级页表转换模式。
- XLEN：RISC-V 基础整数寄存器宽度；本架构的 XLEN 为 32 位。
- kernel：设备侧执行函数，由运行时提交到通用图形处理器上执行的并行计算入口。
- ABI：application binary interface，应用二进制接口。
- thread：线程，本文档中表示单个 lane 承载的逻辑执行上下文。
- warp：硬件锁步执行的 32 个线程组成的执行单元，本架构基本执行粒度。
- lane：warp 中的单个执行车道。
- CTA：cooperative thread array，协作线程数组，由若干 warp 组成的共享存储和协作单元。
- cluster：协作线程数组组（可选硬件特性），用于跨 CTA 数据交换和同步。
- grid：kernel 的整体执行范围，由若干 CTA 组成。
- SM：streaming multiprocessor，流多处理器单元。
- SIMT：single instruction, multiple threads，单指令多线程执行模型。
- uniform：统一标量语义。表示 warp 共享一份标量计算结果，不按 lane 分别形成不同结果。
- vreg：向量通用寄存器（per-thread）。
- sreg：标量通用寄存器（per-warp uniform）。
- pred：per-thread 谓词。
- upred：per-warp uniform 谓词。
- sbreg：依赖计数器寄存器（dependence counter）。
- EXEC：warp 级执行掩码，每位对应一个 lane，控制其是否参与当前指令的架构可见执行。
- predicate gate：指令级谓词门控字段，控制本指令是否产生架构可见效果。
- lane mask：按 lane 位置组成的位掩码。
- CSR / special register：控制 / 状态寄存器与特殊寄存器，提供线程标识、计时、内存大小等架构状态。
- TMA：tensor memory accelerator，张量内存加速器，用于异步多维张量搬运、预取与归约。
- mbarrier：memory barrier object，内存屏障对象，用于异步搬运完成通知与等待。
- async group：异步搬运分组，用于把若干异步搬运操作归为一组并通过提交 / 等待指令确认完成。
- address space：地址空间，本架构区分 generic / global / shared / local / constant 五类语义范围。
- scope：作用域，表示同步、缓存一致性或可见性影响的范围（CTA / SM / cluster / GPU / SYS）。

3.2 寄存器命名约定

除非具体指令章节另有说明，本文采用以下操作数名约定（在 §14 各 leaf 指令形态的 Format 行中体现）：

- dst：目标寄存器。
- src0 / src1 / src2：源操作数，按编号顺序排列。

- pin0 / pin1：input predicate；pout0 / pout1：output predicate。
- 寄存器对（64-bit 数据）：{R, R+1} 偶对齐对，低 32 位存储于编号较小的寄存器。

4 RV32I 兼容性

本架构应提供与 RV32I 兼容的标量整数路径，并应支持 Zifencei 扩展定义的 FENCE.I。RV32I 和 Zifencei 指令采用 RISC-V 规定的 32 位定长编码；第 14 章定义的 GPIDL 扩展指令采用本架构的变长编码。两类指令的编码空间相互独立。

除 ECALL 和 EBREAK 外，标量整数路径应支持下列指令：

- 整数算术运算：add、addi、sub、lui、auipc；
- 整数逻辑运算：and、andi、or、ori、xor、xori；
- 整数移位运算：sll、slli、srl、srli、sra、srai；
- 整数比较运算：slt、slti、sltu、sltiu；
- 控制流：jal、jalr、beq、bne、blt、bge、bltu、bgeu；
- 标量访存：lb、lh、lw、lbu、lhu、sb、sh、sw；
- 内存栅栏：fence；
- 指令取指栅栏：fence.i。

上述指令的位字段编码、立即数编码、寄存器编号、访存对齐和异常行为应符合相应的 RISC-V 规范。其中，FENCE.I 应按 Zifencei 扩展的规定执行，不应视为 RV32I 基础指令。

本架构不实现 ECALL 和 EBREAK。系统调用和调试断点应由第 14 章控制流分类中的 bpt 指令或平台运行时陷阱机制处理。

RV32I 标量整数数据由本架构的 sreg 寄存器堆承载，本文件不另设独立的整数寄存器命名空间。逐通道整数计算由第 14 章 “整数计算（Integer compute）” 分类中的 GPIDL 扩展指令完成。

5 地址转换 — RISC-V Sv39

本架构的虚拟地址转换应采用 RISC-V 特权架构规定的 Sv39 模式。内存管理单元、根页表配置、页表遍历和页故障处理属于系统架构功能，不编码为第 14 章中的 GPIDL 扩展指令。

- 虚拟地址 VA[38:0] 应划分为 VPN[2]、VPN[1]、VPN[0] 和页内偏移，其中各级虚拟页号为 9 位，页内偏移为 12 位；
- 根页表基址应通过特权寄存器 satp 配置；页表项格式应符合 RISC-V 特权架构的规定，并应支持 4 KiB、2 MiB 和 1 GiB 页面；
- 页级权限位 R/W/X/U/G/A/D 应按 RISC-V 规范解释；
- 物理地址宽度不应小于 56 位；64 位地址值应按 8.3 的规定由偶对齐的寄存器对承载。

启用地址转换时，第 14 章中访存、原子操作和 TMA 相关指令形成的有效地址均应按照 Sv39 进行转换。global、shared、local 和 constant 等地址空间属性仅影响缓存访问路径和可见范围，不应改变 Sv39 页表格式、地址转换步骤或权限检查规则。

对于 generic 地址，addr[63:62] 为地址空间标识，仅用于选择目标地址空间。地址转换单元应先根据该标识完成地址空间路由，再移除该标识，并按照 VA[38] 对 VA[63:39] 进行符号扩展。完成上述处理后，方可进行规范地址检查和页表遍历。地址空间标识不应参与 Sv39 地址转换。

6 概述与设计取向

6.1 GPIDL 是什么

GPIDL（GPU Processor Instruction Definition Language，GPU 处理器指令定义语言）是本项目自研的一套图形处理器（GPU，Graphics Processing Unit）指令集架构（ISA，Instruction Set Architecture）。指令集架构指的是一颗处理器对软件暴露出来的全部内容：有哪些指令、有哪些寄存器、每条指令读写什么、各种操作受什么约束。它是软件唯一能看见、并且可以依赖的硬件界面；软件不需要知道硬件内部怎么实现，只要按这套界面写程序，硬件就保证按界面约定的行为执行。

GPIDL 面向两类负载：一类是通用 GPU 计算（把 GPU 当作大规模并行的算力来用，不限于图形），另一类是深度学习（神经网络的训练与推理，主体是大规模矩阵运算）。这两类负载的共同特点是有海量、彼此独立的计算可以同时做，GPIDL 的全部设计都是为了把这种并行算力尽可能高效地交付出去。

整套 GPIDL 的定义集中在一个文件里：spec.jsonc。它是一个带注释的 JSON 文件（JSONC，JSON with Comments），用结构化的方式逐条写清楚每一条指令的全部信息——这条指令有哪些操作数、操作数从哪一类寄存器取值或者是多少位的立即数、支持哪些行为变体。这个文件是整个项目的唯一真实来源（single source of truth）：寄存器有几类、每类多少个，内存分哪几种空间，一条指令带哪些公共控制字段，全都由它定义。本文档讲到的每一个寄存器名、字段名、指令名和取值，都来自 spec.jsonc 里真实存在的条目，不是举例杜撰的。

就规模而言，当前的 spec.jsonc 定义了 5 类寄存器、4 种内存空间、97 个行为修饰符，以及 170 条指令（按指令的文字名字即助记符计；同一条指令的不同操作数组合分开存放时共 439 项）。后面各章会逐块讲清这些数字背后的内容。

6.2 吞吐优先、硬件从简、软件多管事

要理解 GPIDL 为什么长这样，先要理解它和中央处理器（CPU，Central Processing Unit）在设计目标上的根本分歧。

CPU 的目标是让单个指令流尽可能快地跑完。为此 CPU 在硬件里堆了大量复杂机制：乱序执行（硬件自己分析指令间的依赖、谁不依赖谁就提前算）、分支预测（猜测条件分支往哪走、猜中就不必等条件算出来）、寄存器重命名（用更多物理寄存器消除假依赖）等等。这些机制都很费硬件面积和功耗，但它们服务的是同一件事：加速一条指令流。

GPU 的目标不是让单个线程快，而是让大量线程的总吞吐高。深度学习里一次矩阵乘法、图形里一帧画面，背后是成千上万个彼此独立、做着几乎一样计算的任务。对这种负载，与其用复杂硬件把一条指令流催快，不如把每个执行单元做得尽量简单，省下来的硬件面积用来同时容纳并推进成千上万个线程。某个线程因为等数据（比如等一次访存）而卡住时，硬件不去想办法让这个线程更快，而是直接切去推进别的已经就绪的线程，用线程数量把等待的时间填满。这种 “用并行掩盖延迟” 的做法，是 GPU 吞吐优先取向的直接结果。

把执行单元做简单，代价是很多原本由 CPU 硬件自动完成的事，在 GPU 上没人替软件做了，只能由软件在编译期事先安排好。这是贯穿 GPIDL 的一条主线，最典型的一例是指令之间的依赖与等待：CPU 的乱序硬件会自动发现 “这条乘法要用上一条加法的结果，得等加法写回再发”，GPU 的发射逻辑不做这种分析，它只会 “挑一个就绪的线程束、按程序顺序发它的下一条指令”。于是 “这条指令发射前要等多久、要等哪条指令完成” 必须由编译器算清楚，写进每条指令自带的控制字段里（GPIDL 里就是 stall、req_sb、rd_sb、wr_sb 这组字段，详见依赖与静态调度）。算少了会静默算错，算多了会白白变慢——这类原本属于硬件的责任被移交给软件，是 GPIDL 最需要使用者理解清楚的地方。

同样的取向还会反复出现在后面：寄存器为什么这么多（要同时供养大量驻留线程，见寄存器与内存）、

为什么有一条“整束只算一份”的统一通路（省掉重复计算，见执行模型）、为什么专门为矩阵计算配一条张量通路（普通逐元素运算喂不饱吞吐，见矩阵计算与大块数据搬运）。读到那些设计时，都可以回到这条总取向来理解：吞吐优先、硬件从简、软件多管事。

6.3 架构、微架构与编码

一颗处理器从规范到实现，可以分成三个相互独立的层次。分清这三层，有助于准确理解后续各章描述的是哪一层。

架构是软件可见的硬件行为：有哪些指令、有哪些寄存器和内存空间、每条指令读写什么、产生什么结果、指令之间和线程之间要遵守哪些依赖与同步规则。这是软件能依赖的契约——只要两台硬件实现同一套架构，同一段程序在它们上面就得到相同的可见结果，哪怕内部实现完全不同。后续各章描述的都是这一层。

微架构是某一颗具体芯片为实现上述架构行为而采用的硬件办法：流水线分几级、缓存多大、用什么替换策略、电路怎么排布、面积和功耗怎么权衡。同一套架构可以有多种微架构实现，它们对软件呈现的可见行为一致，但快慢、省电程度各不相同。本文偶尔提到“这类操作延迟可变”“这条通路为吞吐设计”，是在描述软件需要据此安排代码的架构可见性质，而不是规定某种微架构如何实现。

编码是把一条指令落成一串二进制位时的具体位布局：助记符占哪几位、操作数的寄存器编号放在哪、各个修饰符的取值怎么编进去。GPIDL 在设计上把指令的逻辑结构和二进制位布局分开：spec.jsonc 定义的是逻辑结构（有哪些操作数、取值来自哪类寄存器、立即数多少位），位布局则在此之上单独综合。这种正交让同一套架构可以配不同的编码方案，也让“理解一条指令做什么、怎么和别的指令配合”不必先知道它的二进制形态。

逐条指令的完整字段清单——每条指令支持哪些操作数组合、每个修饰符有哪些取值——由 spec.jsonc 自动生成的指令定义章给出。本文各章讲的是模型与设计意图：一条指令由什么组成、有哪些类指令、它们怎么控制流程、怎么保证依赖与同步正确。两者配合使用：先建立整体理解，再到指令定义章查任意一条指令的精确细节。

7 执行模型：warp、SIMT 与两套数据通路

7.1 线程、通道与线程束

GPIDL 程序里最小的执行单位是线程（thread）。程序员写代码时面对的就是单个线程：它有自己的一份寄存器、读写自己的数据、按自己看到的控制流往下走，看起来和写一段普通的标量程序没有区别。

但硬件并不是一个线程一个线程地单独调度。硬件把 32 个线程绑成一组，叫一个线程束（warp）。一个 warp 里的 32 个线程从头到尾绑在一起被调度，是硬件实际发射和执行指令的基本粒度。warp 里的每个线程占据一个固定的位置，这个位置叫通道（lane），编号 0 到 31。“第 5 个通道”和“warp 里编号为 5 的那个线程”是一回事。

为什么是 32 而不是别的数字，属于微架构权衡，这里只把它当作架构给定的事实：GPIDL 的 warp 宽度是 32，本文后面凡是讲到“整个 warp”“32 个通道”“活跃掩码 32 位”，都基于这个宽度。

7.2 锁步执行与活跃掩码

一个 warp 的 32 个线程共用同一个程序计数器（PC，Program Counter，指向当前要执行的指令的位置）。这意味着在任一时刻，这 32 个线程执行的是同一条指令，只是各自作用在自己的数据上。这种“32 个线程踏着同一个节拍、同步执行同一条指令”的方式叫锁步执行（lockstep）。

但程序里有分支，不同线程可能因为数据不同而本该走不同的路。既然 32 个线程被迫执行同一条指令，硬件就需要一个办法表达 “这条指令对哪些通道真正生效”。这个办法是活跃掩码（active mask，本文按其架构名记作 EXEC）：一个 32 位的位掩码，每一位对应一个通道，为 1 表示该通道参与当前指令的执行、其结果被保留，为 0 表示该通道这条指令空转、不产生任何可见效果。被条件关掉的通道、走了另一条分支的通道，都在 EXEC 里置 0。硬件照样让它们跟着走完这条指令的节拍，只是把它们的结果丢弃。

EXEC 是控制流的核心状态。后面讲控制流与分歧时会看到，warp 内的分支本质上就是在不同时间段用不同的 EXEC 值让不同的通道生效；像 mset（设置并保存 EXEC）、mbnz（在还有通道活跃时跳转）、kill（让某个通道把自己从 EXEC 中清掉）这些指令，操作的都是这个掩码。当前的 EXEC 值本身也可以通过读特殊寄存器（sr_id 取值 exec_mask）拿到普通寄存器里。

7.3 单指令多线程与它和 SIMD 的区别

上面这种 “一条指令同时让一束线程各自作用在自己的数据上” 的执行方式，叫单指令多线程（SIMT，Single Instruction, Multiple Threads）。

它和另一个常见概念单指令多数据（SIMD，Single Instruction, Multiple Data）很接近。SIMD 是用一条指令对一个宽向量里的多个数据元素同时做同样的运算，程序员看到的是 “向量” 这个对象，要自己把数据打包进向量、自己处理边角不齐的部分。SIMT 的不同之处在于：程序员看到的不是向量，而是一个个独立的标量线程，每个线程有自己的寄存器、自己的地址、自己的控制流。硬件在背后把 32 个这样的标量线程凑成一个 warp 一起执行，但这件事对程序员是半透明的——可以当 32 个独立线程来写，也可以在需要性能时意识到它们其实在锁步。

最关键的区别是控制流。纯 SIMD 里，一条向量指令对所有元素一视同仁，没有 “某些元素走这条分支、另一些走那条分支” 的概念。SIMT 借助 EXEC 掩码，允许 warp 里的不同通道表现得像是各走各的分支：硬件先让满足条件的通道生效跑一段，再让其余通道生效跑另一段。从单个线程的视角看，它就像独立地选择了自己的路径。这是一种方便的假象，代价是两条路径被串行地分别执行了一遍（详见控制流与分歧）。

7.4 两套数据通路：逐线程计算与整束统一计算

一个 warp 里，很多值在 32 个通道上其实是完全一样的。最常见的是循环计数器、循环上界、所有线程共用的基地址：这些量按 warp 来看只有一份，让 32 个通道各算一遍、各存一份，既浪费寄存器又浪费算力。

GPIDL 因此提供两套并存的数据通路：

- **逐线程的向量通路**：每个线程各算各的、各存各的。它用逐线程的向量通用寄存器 vreg（写作 R0–R255）和逐线程的谓词 pred（写作 P0–P7）。绝大多数计算走这条通路，因为 GPU 的本职就是让每个线程处理自己那份数据。
- **整束统一的标量通路**：整个 warp 只算一份、只存一份。它用整 warp 共享的标量寄存器 sreg（写作 UR0–UR63）和整 warp 统一的谓词 upred（写作 UP0–UP7）。凡是 warp 内所有通道一致的标量计算和地址计算，都应该走这条通路，省掉 31 份重复。

指令集为这两套通路各配了一份指令。逐线程的指令直接写助记符（如加法 iadd3、搬运 mov、读常量 ldc）；与之对应的统一版本在助记符前加 u 前缀（统一加法 uiadd3、统一搬运 umov、统一读常量 uldc），它们作用在 UR/UP 上，承担 warp 内一致的标量与地址计算。当前 GPIDL 的 170 个助记符展开成 439 条指令形态，其中 308 条走逐线程模式、129 条走统一模式（其余 2 条不接受条件，见指令的组成）。

这条 “两套通路” 的分界贯穿全文：寄存器分两套（vreg/pred 对 sreg/upred），谓词分两套，控制流也分逐线程分支与统一分支两种处理。识别一个值到底是逐线程不同、还是整 warp 一致，并把一致的部分下放到统一通路，是面向 GPIDL 写高效代码的基本功。

7.5 线程的分组层级：从 warp 到网格

warp 之上还有几层分组，每一层对应一种协作或调度的范围。从小到大：

- **线程（thread）**：最小执行单位，占一个通道。
- **线程束（warp）**：32 个线程，硬件调度与指令发射的基本粒度。
- **线程块（CTA，Cooperative Thread Array，协作线程数组）**：一组能在一起协作的线程，由若干 warp 组成。同一个 CTA 内的线程可以通过片上的共享内存（见寄存器与内存）交换数据，可以用屏障（见线程间同步与内存一致性）互相等齐。CTA 是 “能紧密协作的最大一组线程”。
- **线程块簇（cluster，线程块簇）**：若干 CTA 组成一簇，簇内的 CTA 可以互相访问对方的共享内存、做跨 CTA 的同步。这是比单个 CTA 更大、但仍受限的一层协作范围。
- **网格（grid）**：一次内核（kernel）启动所产生的全部线程，由若干 CTA（或若干簇）组成。网格内不同 CTA 之间没有片上的协作手段，只能通过全局内存交换数据。

每个线程都能知道自己在这套层级里的坐标，办法是读特殊寄存器（见寄存器与内存）：通过 sr_id 取值 laneid 拿到自己在 warp 内的通道号，tid_x/tid_y/tid_z 拿到自己在 CTA 内的三维坐标，ntid_x/ntid_y/ntid_z 拿到 CTA 的三维尺寸，ctaid_x/ctaid_y/ctaid_z 拿到自己所在 CTA 在网格内的坐标，nctaid_* 拿到网格的尺寸。簇相关的坐标（clusterid_*、cluster_ctaid_* 等）同理。线程靠这些坐标算出自己该处理整份数据里的哪一块。

7.6 流式多处理器：一个 CTA 的归宿

承载执行的硬件单元叫流式多处理器（SM，Streaming Multiprocessor）。一颗 GPU 里有很多个 SM，它们并行地各跑各的。

一条关键规则是：一个 CTA 整体被放到某一个 SM 上执行，从头到尾不跨 SM。这条事实直接决定了前面提到的协作范围。CTA 内的共享内存之所以只在 CTA 内可见，正是因为它就是这个 SM 上的一块片上存储，别的 SM 上的 CTA 物理上够不着它；CTA 内的屏障之所以只能让本 CTA 的线程等齐，也是同样的原因。换句话说，“共享内存和屏障的协作范围到 CTA 为止” 不是一条人为约定，而是 “一个 CTA 不跨 SM” 这条硬件事实的自然结果。线程块簇是对这条边界的有限放宽：簇内的 CTA 被放在物理上相邻、能互访片上存储的一组 SM 上，于是允许跨 CTA 访问共享内存。

7.7 驻留、延迟隐藏与占用率

一个 SM 上可以同时驻留（resident）多个 warp。驻留的意思是：这些 warp 的执行状态（各自的寄存器、PC、EXEC 等）都实打实地占着 SM 上的资源，硬件可以在它们之间随时切换，切换不需要把状态换出换入，因此几乎没有代价。

这正是 GPU 隐藏延迟的办法。当某个 warp 卡在一个慢操作上——最典型的是访问全局内存，要等几百个周期数据才回来——硬件不会干等，而是立刻切去执行同一个 SM 上另一个已经就绪的 warp。只要驻留的 warp 足够多，总有就绪的 warp 可发，慢操作的等待时间就被其他 warp 的有效工作填满了。这种 “用大量线程的并行来盖住单个操作的延迟”，是 GPU 吞吐优先取向（见概述与设计取向）在执行层面的具体体现，也是为什么 GPU 不像 CPU 那样依赖庞大的缓存和乱序硬件来降低单个操作的延迟。

衡量 “一个 SM 上同时驻留了多少 warp” 的指标叫占用率（occupancy）：实际驻留的 warp 数与该 SM 能驻留的上限之比。占用率不是越高越好，但太低时就没有足够的 warp 来盖住延迟。占用率受两项资源约束：每个线程用了多少向量寄存器（用得多，能同时驻留的线程就少），以及每个 CTA 用了多少共享内存（用得多，能同时驻留的 CTA 就少）。这两项是有限资源，warp 之间是在分一块固定的蛋糕。这笔账后面分配寄存器、决定共享内存用量时要反复用到——多用寄存器能减少重算和访存，但会压低占用率、削弱延迟隐藏，需要权衡。GPIDL 还提供了在运行时调整这笔账的手段：usetmaxreg 能让同一个 CTA 内不同的 warp 组配

置不同的物理寄存器数量（搬运用的 warp 组让出寄存器、计算用的 warp 组多拿），usetshmsz 能调整 CTA 的共享内存容量。

8 寄存器与内存

8.1 寄存器分五类

GPIDL 把寄存器分成五类，由 spec.jsonc 的 operand_kinds 定义。区分它们的两条线索，一是归属（一个值是每个线程各有一份，还是整个 warp 共用一份），二是用途（存普通数据、存条件位、还是存依赖计数）。五类列举如下：

类别	数量	编号位宽	归属	写法	用途
vreg	256	8 位	每线程一套	R0–R255	逐线程的向量通用寄存器，逐线程计算的主力
sreg	64	8 位	整 warp 一套	UR0–UR63	整 warp 共享的标量寄存器，承载统一通路的标量与地址计算
pred	8	3 位	每线程一套	P0–P7	逐线程谓词，每个存 1 位真/假，用于条件执行
upred	8	3 位	整 warp 一套	UP0–UP7	整 warp 统一谓词，统一通路的条件位
sbreg	8	3 位	整 warp 一套	SB0–SB7	依赖计数器，专门用于软件管理指令间依赖

“编号位宽” 指的是在指令里指定用哪个寄存器时，这个编号字段占多少位：vreg 使用 8 位；sreg 虽然只有 64 个合法槽位，但编码字段同样预留 8 位，只有编号 0–63 合法；三类只有 8 个的（pred、upred、sbreg）各用 3 位。

这五类正好对应执行模型讲的两套数据通路：vreg、pred 是逐线程的，每个线程各有独立的一套，是向量通路的状态；sreg、upred 是整 warp 一套的，是统一通路的状态。sbreg（依赖计数器）也是整 warp 一套，但它不存数据、不存条件，而是专门给软件管理指令间依赖用的一种计数器，完整机制见依赖与静态调度，本章只把它作为一类寄存器列出。

向量寄存器有 256 个之多，是 GPU 吞吐优先取向（见概述与设计取向）的直接结果：一个 SM 要同时驻留很多 warp、每个 warp 32 个线程，每个线程都要从这 256 个里分到够用的一份，才能维持高占用率而不必频繁往内存里溢出（spill）。寄存器多，是为了供养大量并行线程。

8.2 专用编号与保留编号

每一类寄存器里，都有个别编号不是普通的、可由编译器随意分配的寄存器，而是被固定赋予了特殊含义。这些专用编号在 operand_kinds 里以 special（专用槽）和 reserved（保留槽）标出：

- vreg 的 255 号是零寄存器，专名 RZ：读它恒为 0，写它则被丢弃。需要一个常数 0、或者想把某个结果扔掉时，直接用 RZ，省掉一次置零或一个临时寄存器。
- sreg 的 63 号同理是统一通路的零寄存器，专名 URZ。
- pred 的 7 号是恒真谓词，专名 PT；upred 的 7 号同理，专名 UPT。读它恒为真。一条指令想 “无条件执行” 时，就把它的谓词选成 PT（详见指令的组成）；某个需要一个谓词输入但本次不想设条件的地方，也用 PT 顶上。
- sbreg 的 7 号是 SB_NONE，表示 “不选任何依赖计数器”：一条不需要挂依赖的指令，把它的计数器字段填成 SB_NONE，就表示 “我不占用任何计数器”。sbreg 的 6 号是保留槽，不作普通分配。于是真正可供软件分配的依赖计数器只有 SB0–SB5 共 6 个——这个 “只有 6 个” 的事实在依赖调度里是个硬约束（见依赖与静态调度）。

这些专用编号的共同作用，是把几个高频需求（要一个 0、要表达无条件、要表达 “不挂依赖”）直接编进寄存器编号里，不必为它们另设指令或字段。

8.3 64 位数据用寄存器对

上面五类寄存器里，向量寄存器和标量寄存器本身是 32 位的。要存放一个 64 位的值（比如一个 64 位地址、一个双精度浮点数），用两个编号相邻、且起始编号为偶数的寄存器拼成一对：低 32 位放编号较小的那个，高 32 位放编号较大的那个。写法上记作 {R, R+1}，其中 R 必须是偶数号。

这条约定在访存里很常见。比如一次加载的访问宽度（ldst_length 修饰符）取 b64（8 字节）时，结果就写进一个偶对齐的寄存器对 {Rd, Rd+1}；取 b128（16 字节）时写进 4 个连续寄存器 {Rd..Rd+3}，且起始编号要 4 对齐；b256（32 字节）则写进 8 个连续寄存器、8 对齐。全局地址是 64 位的，所以以 64 位地址为基址的访存指令，其基址操作数就是一个 sreg 对或 vreg 对。

8.4 四种内存空间

寄存器之外，软件能读写的是内存。GPIDL 按可见范围和读写权限把内存分成四种空间，由 spec.jsonc 的 memory_spaces 定义：

空间	可见范围	读写权限	典型用途与位置
global	全设备可见	可读写	主数据区，物理上在片外显存（DRAM），容量最大、延迟最高
shared	仅 CTA 内可见	可读写	片上快速暂存，供一个 CTA 内的各 warp 互相交换数据
local	单线程私有	可读写	线程私有数据，用于寄存器放不下时的溢出和线程私有的大数组

空间	可见范围	读写权限	典型用途与位置
constant	全设备可见	只读	常量区，放编译期常量、内核参数、各种描述符（如张量搬运用的 tensormap）

可见范围决定了谁能看到谁写的数据，这一点直接和执行模型的分组层级对应。最值得强调的是共享内存（shared）：它只在一个 CTA 内可见，正是因为一个 CTA 不跨 SM、共享内存就是这个 SM 上的一块片上存储，别的 SM 上的 CTA 物理上够不着。全局内存（global）全设备可见，是不同 CTA 之间唯一的数据交换手段。局部内存（local）名字里有 “局部”，但它指的是 “单线程私有”，物理上仍在片外，主要用来兜寄存器放不下的情况。常量内存（constant）只读，硬件为它配了专门的缓存，适合所有线程都来读同一份数据（如内核参数）的场景。

GPIDL 还提供一种把上述空间统一编址的访问方式：一条通用加载 ld 或存储 st 不在指令里写死要访问哪个空间，而是看地址的最高 2 位来路由——地址 [63:62] 为 00 走全局、01 走局部、10 走共享、11 走常量，剩下的位是该空间内的字节偏移。这样同一段代码用同一条指令就能访问不同空间的数据，代价是多一步地址判断。如果编译期已经确定了空间，则可以用直接点名空间的指令（如全局加载 ldg、共享加载 lds、常量加载 ldc），跳过这步判断、走更快的路径。这一类指令的全貌见指令集纵览。

把常量直接编进指令（作为立即数，见指令的组成）和从常量内存加载，是供给常量的两条路：小而固定的值适合编成立即数省一次访存，成批的或运行时才确定的常量适合放常量内存。

8.5 特殊只读状态：线程编号、计时器、资源量

除了上面这些可读写的寄存器和内存，硬件还维护一批只读的状态，供线程查询自己的身份和环境。它们不是普通寄存器，要通过专门的读特殊寄存器指令——逐线程的 s2r（special register to register）或统一通路的 s2ur——取到普通寄存器里再用。具体取哪一个由 sr_id 修饰符选定，可取的值包括：

- **线程身份与网格坐标**：laneid（自己在 warp 内的通道号）、tid_x/tid_y/tid_z（自己在 CTA 内的三维坐标）、ntid_x/ntid_y/ntid_z（CTA 的三维尺寸）、ctaid_x/ctaid_y/ctaid_z（所在 CTA 在网格内的坐标）、nctaid_*（网格尺寸）、gridid（网格编号），以及线程块簇相关的 clusterid_*、cluster_ctaid_* 等。线程靠这些算出自己该处理整份数据里的哪一块。
- **硬件位置**：warpid（自己是 SM 上的第几个 warp）、smid（自己在哪个 SM 上）、nsmid（SM 总数）等。
- **计时**：clocklo/clockhi 或合起来的 64 位 clock64（时钟周期计数），globaltimerlo/globaltimerhi 或 globaltimer64（全局纳秒级计时器）。用于自己测量代码段耗时。
- **资源量与配置**：total_smem_size、dynamic_smem_size（共享内存容量）、regalloc（寄存器分配量）、barrieralloc（屏障分配量）等，供运行时查询当前可用资源。
- **执行状态**：exec_mask（当前的 EXEC 活跃掩码，见执行模型）、pc_lo/pc_hi 或 pc64（当前程序计数器）。

这些只读状态把 “线程是谁、跑在哪、用了多少资源、现在几点” 暴露给软件，是内核代码定位自身、做计时和自适应调度的依据。

9 指令的组成

9.1 一条指令的整体组成

一条 GPIDL 指令由五部分拼成：

- **助记符（mnemonic）**：指令的文字名字，比如浮点加法 fadd、整数乘加 imad、全局加载 ldg。它确定这条指令做什么。
- **操作数（operand）**：几个槽位，给出这条指令读写的数据来自哪、去往哪。
- **修饰符（modifier）**：若干带枚举取值的命名字段，选取这条指令在语义上的变体（比如用哪种舍入方式、源是否取负）。
- **条件（谓词门控）**：一个隐含的谓词位选择，决定这条指令这一次到底生不生效。
- **公共控制位**：每条指令都带的一组与 “等多久、依赖谁” 相关的字段。

下面逐部分展开。需要先记住一条总原则：本章讲的是这套组成模型，而每条具体指令到底带哪几个操作数、支持哪些修饰符取值，由 spec.jsonc 自动生成的指令定义章逐条给出，本章不重复那张字典。

9.2 操作数：一个位置加一个取值来源

每个操作数由两件事确定：它在指令里的位置，以及它的取值来源。

位置用一个稳定的名字表示，按角色和编号排列：目的操作数叫 dst，源操作数按顺序叫 src0、src1、src2。这些名字是位置标签，不随指令变化；看到 src1 就知道它是这条指令的第二个源。

取值来源有两种。一种是某一类寄存器，由操作数上的 kind 字段指明是 vreg、sreg、pred、upred 中的哪一类——比如一个 kind 为 vreg 的操作数，它的值就来自某个 R 寄存器。天生就是 64 位的操作数（地址、程序计数器的值、同步令牌）用对应的成对类别 paired_vreg、paired_sreg：它持有一个起始编号为偶数的寄存器对 {R, R+1}（低 32 位在偶号、高 32 位在奇号），编码里只存偶数号，所以字段宽度与普通类别相同。另一种是立即数（immediate，直接写在指令里的常数），由操作数上的 bits 字段说明它占多少位（见下一节）。一个操作数要么有 kind（来自寄存器），要么有 bits（是立即数），二者择一。

有些操作数的寄存器占用会随已编码的 modifier 改变，例如 load 的 b32/b64/b128 结果分别占 1/2/4 个寄存器。flat-2.2 把这类约束放在操作数的结构化 notes.reg_span/reg_align 中；它们供汇编器校验寄存器组，但不分配任何指令位。PC-relative 符号换算同样写在 notes.relocation，立即数字段本身仍只由 bits、signed、scale 决定。

值得说明的是：操作数上没有单独的字段标注它是输入还是输出、是读还是写。读写方向由助记符和位置共同确定——对 fadd 来说，dst 必然是写、src0/src1 必然是读，这是 fadd 的语义决定的，不需要再为每个操作数挂一个 “方向” 标记。把方向交给 “助记符 + 位置” 而不是显式字段，是 GPIDL 操作数模型的一个有意选择：它避免了同一件事用两处信息表达、可能互相矛盾。

9.3 立即数

立即数不是一类预先定义的寄存器，而是在操作数上直接声明位宽。比如 fadd 有一个写法，它的 src1 不是寄存器而是一个 32 位立即数（在 spec.jsonc 里就是该操作数带 bits: 32、不带 kind）。指令执行时，这个常数直接来自指令本身，不需要先放进寄存器、也不需要访存去取。

把常量交给指令有两条路，各有取舍。一条是编成立即数，直接嵌在指令里，适合小而固定、编译期就知道的值，省掉一次加载。另一条是放进常量内存、用常量加载指令取出来（见寄存器与内存），适合成批的、或者运行时才确定的常量（如内核参数）。

9.4 修饰符

修饰符是带枚举取值、通常带默认值的命名字段，挂在一条指令上、或挂在某个操作数上。它的作用是把大量语义变体压进同一个助记符，而不必为每个变体单设一条指令。GPIDL 当前定义了 97 个修饰符（spec.jsonc 的 modifier_defs）。举几个真实的例子：

- **浮点舍入方式** rnd_mode_fp：取 rne（就近舍入到偶数，默认）、rtz（向零）、rup（向上）、rdn（向下）之一。
- **非规格数刷零** ftz：是否把非规格数（subnormal，绝对值极小、低于正常浮点表示下界的数）在参与运算前刷成 0、把这样的结果也刷成 0。机器学习训练常开它走更快的路径。
- **饱和** saturate：把结果钳到一个区间（如 [0.0, 1.0]），常用于激活函数。
- **源操作数取负/取绝对值** src_fp_modi：挂在某个源上，取 id（原样，默认）、neg（取负）、abs（取绝对值）、abs_then_neg（先取绝对值再取负）之一。它让 -a + b、|a| + b 这类常见变体在一条指令里表达，省掉一条单独的取负或取绝对值指令。
- **操作数复用** reuse：挂在某个源上的提示位，表示下一条指令同一位置的源会从操作数缓冲里复用这个值、不必再读一次寄存器堆。
- **数据格式与矩阵形状**：如类型转换的源和目标格式 cvt_ifmt/cvt_ofmt（e5m2/e4m3 等不同指数尾数配比的 8 位浮点都在其中）、访存宽度 ldst_length（b32/b64/b128 等）、矩阵乘累加的块形状 mma_shape（m16n8k8/m16n8k16 等）。

这里要区分修饰符挂在哪一层。**挂在整条指令上的修饰符**影响整条指令的语义，列在指令的 modifiers 里；**挂在某个操作数上的修饰符**只作用于那个操作数，列在该操作数的 modifiers 里。以 fadd 为例：rnd_mode_fp、ftz、saturate 是挂在指令上的（影响整条加法的舍入、刷零、饱和），而 src_fp_modi、reuse 是挂在源操作数 src0 上的（只决定这个源怎么被预处理、是否被复用）。一个源取负、另一个源不取负，正是靠“修饰符挂在操作数上”来分别表达的。

9.5 条件执行：谓词门控

每条指令都隐含一个谓词位选择，叫谓词门控（predicate gate）：它从谓词寄存器里选一个位，用这个位的真假决定本条指令这一次是否产生效果。位为真则正常执行，为假则这条指令对所有相关通道都不生效（相当于一条空操作）。门控还能取反，即“当这个谓词为假时才执行”。

谓词门控有三种解读方式，由指令自己的 predicate_mode 字段选定（spec.jsonc 的 control_defs.predicate_gate）：

- **逐线程（simt）**：写作 @P{N}，从逐线程谓词 P0–P6 加上恒真谓词 PT 中选一个；多出的一位用于取反，写作 @!P{N}。这种模式下，32 个通道各自看自己那一份 P 寄存器，可能有的通道生效、有的不生效——这正是执行模型里逐线程通路的条件执行。选 PT（恒真）就等于无条件执行。
- **整 warp 统一（uniform）**：写作 @UP{N}，从统一谓词 UP0–UP6 加 UPT 中选一个，同样可取反。这种模式下整个 warp 看同一份 UP，要么整束生效、要么整束不生效，对应统一通路。
- **不接受条件（none）**：这条指令没有谓词门控，永远无条件执行。少数指令属于此类，比如线程束退出 exit、空操作 nop——它们要么是终结指令、不该被条件跳过，要么本身就没有“执行与否”的意义。

当前 GPIDL 的指令形态里，308 条走 simt、129 条走 uniform、2 条是 none。这三种模式与两套数据通路严格对应：逐线程的指令用逐线程谓词门控，统一通路的指令用统一谓词门控。

9.6 同一个助记符的多种写法

同一个助记符常常有多种操作数组合，每一种叫一个写法（form）。最常见的差别是某个源到底是寄存器还是立即数。

以 fadd 为例，它至少有两种写法：一种 src1 是立即数（在 spec.jsonc 里记作 fadd__v_vi，形态后缀 v_vi 表示“目的是 vreg、src0 是 vreg、src1 是立即数”），另一种 src1 是寄存器（fadd__v_vv，三个都是 vreg）。两者做的运算一样（都是浮点加法），只是第二个源的来源不同。在 spec.jsonc 里这些写法分开存放，所以按写法计，170 个助记符一共展开成 439 个指令形态。

理解 “一名多形” 的存在很重要：查指令时，先按助记符找到这条指令，再在它的若干写法里挑出操作数组合与你的需求相符的那一个。指令定义章就是按 “助记符聚合、写法并列” 的方式组织的。

9.7 每条指令都带的公共控制位

除了上面的谓词门控, 每条指令还隐含携带一组与依赖和延迟有关的公共控制位（spec.jsonc 的 control_defs）：

- stall：发射这条指令后，本 warp 要等多少个周期再发下一条。
- yield：一个提示位，让发射器这一周期偏向去发别的 warp。
- req_sb：发射这条指令前，要等哪些依赖计数器归零。
- rd_sb、wr_sb：这条指令在读完源、写回结果时，各自去给哪个依赖计数器减一。

这些字段不在助记符里、也不算操作数，而是每条指令都自带的隐含控制信息。它们正是概述与设计取向里说的 “很多原本由 CPU 硬件自动完成的事，在 GPIDL 上改由软件事先安排好” 中最典型的一项：指令之间等多久、谁等谁，全靠这组位由编译器算好填进去。这里只点名，它们的完整含义、怎么用、用错的后果，留到依赖与静态调度详讲。

9.8 逐条细节去哪查

本章给的是组成模型：一条指令由助记符、操作数、修饰符、谓词门控、公共控制位拼成，操作数是 “位置加取值来源”，修饰符把变体压进同一助记符，一个助记符可有多种写法。至于某条具体指令到底支持哪几个操作数、每个操作数是什么 kind、挂哪些修饰符、每个修饰符有哪些取值，这些逐条的精确字段由 spec.jsonc 自动生成的指令定义章给出。两者配合：先用本章建立 “指令长什么样” 的整体认识，再到指令定义章查任意一条的细节。

10 控制流与分歧

10.1 分支、跳转、调用与退出

控制流的基本动作是改变程序计数器（PC，见执行模型）。GPIDL 把它们分成几组。

分支与跳转。相对分支 bra 把 PC 改成 “当前位置加一个相对偏移”（具体是 PC = next_pc + (偏移 << 2), 其中 next_pc 是本条指令的地址加本条指令的长度——指令编码是 8/12/16 字节变长的，硬件解码时知道自己多长；偏移是一个有符号 24 位数、以 4 字节为单位，4 是三种指令长度的最大公约数，任何合法指令起点都能表达，跳转范围约 ±32 MB），整个 warp 一致地跳到目标。它的间接版本 brx 把偏移放在标量寄存器 sreg 里（而不是写死的立即数），用于跳转表、switch 派发这类目标在运行时才算出来的场景。绝对跳转 jmp 直接把 PC 设成一个 64 位绝对地址（来自一个 sreg 寄存器对），与当前 PC 无关。注意 bra、brx、jmp 都是统一通路指令（predicate_mode 为 uniform）：它们改变的是整个 warp 共用的 PC，本身是 “整束一起跳” 的动作。warp 内通道各走各路的效果不是靠它们直接实现的，而是靠活跃掩码，见后文。

调用与返回。相对调用 call 在跳转的同时，把返回地址（下一条指令的地址）写进一个偶对齐的标量寄存器对 {UR, UR+1}；被调函数最后用返回 ret 经这个寄存器对间接跳回。callx 是间接版本：目标是另一个寄存器对里的 64 位绝对地址，用于虚函数、函数指针这类运行时才知道入口的调用。返回地址就是一个普通的 64 位值——可以搬移、可以随普通数据一起保存到栈内存，所以调用深度只受软件栈限制；指令集里没有任何隐藏的硬件返回栈，深递归、协程都不需要特殊路径，上下文切换也没有额外状态要保存。哪个寄存器对充当 “链接寄存器” 由编译器的调用约定决定，指令集不指定。

退出与终止。线程束退出 exit 让整个 warp 的所有通道退出、不再执行后续指令，放在内核末尾。它的

predicate_mode 是 none（无条件），所以要做 “满足条件就提前退出” 时，不能在循环中条件地 exit，而要用分支 bra 跳过尾部、把 exit 放在尾部。逐通道的自我终止 kill 则相反：它让满足条件的通道把自己从活跃掩码 EXEC 中清除（EXEC &= ~ 本通道位），该通道在内核退出前不再执行任何指令，而其余通道继续——这是逐线程的（predicate_mode 为 simt），常用于图形里的丢弃像素。软件断点 bpt 让执行停在某处、交给调试器处理，不影响内核正确性。

10.2 程序计数器相关指令

有时程序需要拿到 “自己现在在哪”。读 PC 指令 getpc 把 “当前 PC 加一个相对偏移” 写进向量寄存器（dst = pc + (偏移 << 2)，pc 是这条 getpc 自身的地址，偏移以 4 字节为单位）。它的用途是位置无关代码（PIC，Position-Independent Code）：代码不知道自己被加载到哪个绝对地址，就用 getpc 取到自身的位置，再相对地算出要访问的表或数据的地址。跳转表的基址、指针自身的定位（如跳板、闭包）都靠它。call/callx 写出的返回地址也是同类的值：它就是下一条指令的 64 位地址。

10.3 分歧：一个 warp 内通道走向不同分支

现在到 SIMT 最棘手的地方。一个 warp 的 32 个通道锁步执行、共用一个 PC（见执行模型），可是程序里的条件分支，其条件在 32 个通道上算出来的结果可能不同：有的通道该走 if 分支，有的该走 else 分支。这种 “一个 warp 内通道走向不同路径” 的情况叫分歧（divergence）。

硬件处理分歧的办法是把不同路径串行地走一遍。先把该走 if 的通道在活跃掩码 EXEC 里保留、其余通道置 0，让 warp 走一遍 if 路径（被置 0 的通道跟着空转、不留下效果）；再反过来，把该走 else 的通道置 1、其余置 0，让 warp 走一遍 else 路径。两条路径都执行过之后，再让所有相关通道一起继续。

分歧的代价就在这 “串行走两遍”：if 和 else 两段代码的执行时间是相加而不是相叠的，分歧越多、路径越长，浪费越大。所以面向 GPIDL 写代码时，要尽量减少 warp 内分歧，对短的条件还有专门的优化手段（把短分支做成条件执行，见后文）。

10.4 EXEC 掩码的管理与重聚

实现分歧和重聚（reconvergence，分歧的通道在汇合点重新一起执行），靠的是一组直接操作活跃掩码 EXEC 的指令。

保存与设置 EXEC：mset。进入一段分歧前，需要先记下当前哪些通道活跃、再把 EXEC 缩小到只剩该走这条路径的通道。mset 一条指令同时做这两件事：把旧的 EXEC 存进一个寄存器（以便之后恢复），并按一个掩码运算更新 EXEC。它的运算方式由 mask_op 修饰符选定，包括 nop（直接赋新值）、and（与旧掩码相交，缩小活跃集）、or（求并，扩大）、andn2（取旧掩码里不在新掩码中的部分，这是走 else 路径的主用法）等。走完一条路径后，把之前存下的旧 EXEC 恢复回来，就能让通道在汇合点重聚。

按活跃情况分支：mbnz、mbz。循环和分歧的收尾需要判断 “还有没有通道要继续”。mbnz（branch if EXEC nonzero）在还有任何通道活跃时跳转，是逐通道条件循环回边的主用法：只要还有通道满足继续条件，就跳回循环体。mbz（branch if EXEC zero）相反，在一个通道都不活跃时跳转，用于跳过一段所有通道都被关掉的死路径。

查询活跃情况：many、mall、mpopc。这组指令读 EXEC 但不改它：many 查 “是否还有任一通道活跃”（结果写统一谓词 upred），mall 查 “是否所有通道都活跃”，mpopc 数出 “有多少个通道活跃”（精确计数 0–32，写进标量寄存器 sreg）。它们供软件在分歧逻辑里做判断。

选举领导通道：elect。有些工作整个 warp 只需要一个通道去做（比如代表全 warp 去申请一块内存、写一个共享的标志）。elect 从活跃通道里选出编号最小的一个当领导：它给所有通道返回领导的通道号，并在领导通道上把一个输出谓词置真、其余置假。后续用这个谓词门控，就能让 “只有领导通道执行” 的代码成

立。

重聚必须标对位置。如果软件把恢复 EXEC、判断收尾的逻辑放错了地方，本该一起继续的通道没有重新激活，轻则该执行的通道一直不活跃、算出错误结果，重则循环的收尾条件永远不满足、warp 卡死。所以这组掩码管理指令的配对（哪里 mset 缩小、哪里恢复、哪里 mbnz 判断回边）必须由编译器严格成对地安排好。

10.5 把短分支做成条件执行，避免分歧

不是所有条件都值得动用 EXEC 和 bra 去做成真正的分支。对很短的 if（条件成立时只做一两条指令），更划算的办法是条件执行：不跳转，而是用指令的组成讲的谓词门控，给那一两条指令挂上谓词 @P{N}，条件不成立的通道就让这几条指令对自己不生效。

这样做没有跳转、没有 EXEC 的保存恢复，warp 始终走同一条指令流，只是某些通道的某几条指令空转。它避免了分歧 “串行走两遍” 的开销，代价是条件不成立的通道也跟着走了这几条指令的节拍。所以这是个权衡：分支体很短时，条件执行更划算；分支体很长时，让不该执行的通道一直空转就不值了，这时用真正的分支（mset + bra）只让该走的通道执行才更好。判断哪条路径更优、并生成相应代码，是编译器的事。

10.6 统一控制流：让 warp 一致的条件从根上不产生分歧

分歧的前提是条件在 warp 内通道之间不一致。但很多条件其实整个 warp 是一致的：循环跑多少次（循环边界）、一个内核级的开关（所有线程都看同一个标志）等等，这些条件 32 个通道算出来的结果必然相同。

对这种 warp 一致的条件，根本不该让它产生分歧。正确的做法是走统一通路（见执行模型）：用统一比较指令在统一谓词 upred 上生成条件，再用统一谓词门控 @UP{N} 或统一分支让整个 warp 一致地跳。既然整束的判断结果一致，要么全跳、要么全不跳，压根不会出现 “一部分通道走这边、一部分走那边”，分歧从根上就不存在。把 warp 一致的条件识别出来、下放到统一通路，是消除不必要分歧的关键，也是执行模型的两套通路、指令的组成的两套谓词门控在控制流上的落点。

11 依赖与静态调度

11.1 为什么依赖要软件来管

回到概述与设计取向立下的取向：GPU 靠在大量 warp 之间快速切换来隐藏延迟，所以它的发射逻辑可以做得极简——每个周期挑一个就绪的 warp，按程序顺序发它的下一条指令，仅此而已。它不像 CPU 的乱序硬件那样去分析 “这条指令要用上一条的结果、得等它算完”。

代价是这件分析工作没人做了，只能由编译器在编译期静态完成。编译器要算清楚：每条指令发射前，它依赖的哪些前序指令必须已经完成；然后把这个信息编码进每条指令自带的公共控制位里。硬件只是忠实地按这些位去等待和发射，正确性完全压在编译期算得对不对上。

11.2 两类延迟，两套机制

依赖之所以需要 “等”，是因为前序指令的结果不是发射后立刻就绪的，要过若干周期才出来。GPIDL 按 “这个延迟编译期知不知道” 把指令分成两类，用两套不同的机制处理。

- **固定延迟**：算术和逻辑运算，从发射到结果就绪的周期数是确定的、编译期已知。这类用 stall 字段处理。
- **可变延迟**：访存、特殊函数（如三角函数、倒数）、矩阵运算、大块数据搬运，它们要多久才完成取决于运行期情况（缓存命中与否、内存拥塞程度等），编译期无法预知。这类用依赖计数器（scoreboard）

机制处理。

11.3 固定延迟：stall 与 yield

对固定延迟的指令，编译期既然知道它要几个周期出结果，就可以直接告诉硬件 “发了它之后等几个周期再发下一条”。这就是 stall 字段：它是每条指令都带的 4 位字段，取值 0 到 15，表示发射这条指令后，本 warp 要强制等待这么多个周期才能发射同一个 warp 的下一条指令。默认值 0 表示不等待。硬件为每个 warp 维护一个 stall 计数器，每周期减一，减到零之前不考虑发射这个 warp 的指令。

举例：如果一条加法的结果要 6 个周期才能被下一条指令读到，而紧接着的下一条指令正要用这个结果，编译器就把这条加法的 stall 设成 6，硬件就会在发完加法后让这个 warp 停 6 个周期再发下一条。注意 stall 只挡住本 warp，这 6 个周期里硬件照样去发别的就绪 warp，所以这段等待通常不是浪费。

另有一个 yield 字段，是每条指令都带的 1 位提示位。它置 1 时，提示发射器下一个周期不要发本 warp、让给别的 warp 跑一个周期，用于改善 warp 之间的公平、避免一个 warp 长期霸占发射。它只是提示、不强制，也不影响正确性——影响的只是发射顺序，等多久仍由 stall 和下面的计数器机制决定。

11.4 可变延迟：依赖计数器

可变延迟的指令编译期不知道它何时完成，没法用一个固定的 stall 周期数去等。GPIDL 为它配了依赖计数器机制，用的是寄存器与内存里那类整 warp 一套的依赖计数器 sbreg（即 SB0–SB7，硬件每个 warp 实际可用的是 SB0–SB5 共 6 个）。

机制的核心是 “发射时加一、完成时减一、消费者等归零”：

1. 一条可变延迟指令在发射时，给它选定的某个计数器 SBx 加一。“给哪个计数器” 由这条指令的 rd_sb、wr_sb 字段指定（见下一节）。
2. 当这条指令的关键事件完成时，硬件给同一个 SBx 减一。
3. 想依赖这条指令的后续指令，在自己发射前先等这个 SBx 归零——归零意味着挂在它上面的在途操作都完成了。“等哪些计数器” 由后续指令的 req_sb 字段指定。

req_sb 是每条指令都带的 6 位等待掩码：第 i 位置 1，表示这条指令发射前必须等 SB[i] 计数归零。6 位正好对应 6 个可用计数器 SB0–SB5。默认全 0，即不等任何计数器。

11.5 读释放与写释放分开

上面说的 “关键事件完成”，其实有两个不同的时刻，GPIDL 用两个字段分别管理，这是这套机制的关键优化。

考虑一条访存指令：它先读自己的源寄存器（拿到地址、要写出的数据），过一会儿才把加载结果写回目的寄存器。读源发生得早，写回发生得晚，两者可能差很多周期。后续指令对它的依赖也分两种：

- **wr_sb（写释放）**：这条指令把结果写回目的寄存器时，给 wr_sb 指定的计数器减一。它保护的是 “后续指令要用这条指令的结果”——后续指令要读这个目的寄存器（写后读，即 RAW，read-after-write），或者要再写同一个目的寄存器（写后写，即 WAW，write-after-write）。这两种都必须等到写回真正完成，所以挂在写释放上。
- **rd_sb（读释放）**：这条指令读完源寄存器时（远早于写回），给 rd_sb 指定的计数器减一。它保护的是 “后续指令要覆盖这条指令还没读完的源”——后续指令要写这条指令的某个源寄存器（读后写，即 WAR，write-after-read）。这种依赖只需要等源读完，不必等写回。

把读释放和写释放分开的价值就在这里。一条访存的源读得早、写回得晚：如果只有一个 “等到写回” 的释放点，那么 “后面想复用这条访存的源寄存器” 的指令就得一直等到写回，白等很多周期；有了独立的

读释放，它只要等到源被读完就能放行，省下大量等待。

rd_sb 和 wr_sb 这两个字段的默认值都是 SB_NONE（即 sbreg 的 7 号，表示 “不选任何计数器”，见寄存器与内存）。固定延迟的、或者不需要保护的指令，把它们填 SB_NONE，就不占用任何计数器。

11.6 计数器是稀缺资源

每个 warp 真正可供软件分配的依赖计数器只有 SB0–SB5 共 6 个（SB7 是 SB_NONE、SB6 保留，见寄存器与内存）。6 个不多，多笔在途的可变延迟操作要共用这 6 个，于是产生一个权衡：

- **多笔操作共用一个计数器**：省计数器，但消费者等这个计数器归零时，要等到挂在它上面的所有操作都完成——也就是被最慢的那笔拖住。
- **每笔操作各用一个计数器**：消费者可以只等自己真正依赖的那一笔，更精细，但 6 个计数器很快就分光了。

一个典型场景是双缓冲（double buffering）的软件流水线：一边在计算当前这块数据，一边在预取下一块数据，让搬运和计算重叠起来。这时 “当前块” 和 “下一块” 的在途操作必须挂在不同的计数器上——否则 “等当前块算完” 和 “等下一块取到” 会纠缠在同一个计数器里，没法分别等待，流水线就重叠不起来。所以计数器的分配是编译器排软件流水线时的一项核心资源决策。

还有一条显式的依赖屏障指令 depbar，它可以等某个计数器降到某个阈值（小于等于该阈值）而不必等到归零。这在 “发了好几笔、只想等前面几笔完成、后面几笔还可以继续在途” 时有用：用 depbar 等计数器降到剩余 N 笔，就放行，比死等归零更灵活。

11.7 一个 “加载后使用” 的完整走查

把上面的机制落到一段具体代码上。设想要从全局内存加载一个值、然后用它做一次加法：

1. **加载**（一条可变延迟指令）。给它的 wr_sb 选一个计数器，比如 SB0：发射时 SB0 加一，表示 “有一笔写回还没完成”。它读源（算地址）很快，如果没人要覆盖它的源，rd_sb 可以填 SB_NONE 不占计数器。
2. **塞入无关指令**。在加载和使用之间，编译器尽量安排一些不依赖这个加载结果的指令（别的计算、别的访存）。这些指令照常发射、照常执行，把等待加载的时间填满。这正是软件调度要做的事：不是干等，而是用无关工作盖住延迟。
3. **使用**（那条加法）。它要读加载的目的寄存器，构成写后读依赖。于是把它的 req_sb 第 0 位置 1，表示 “我发射前要等 SB0 归零”。硬件发射这条加法前会检查 SB0：只要加载还没写回（SB0 还没减回零），就不发这条加法，转去发别的 warp；一旦加载写回、SB0 归零，这条加法才放行，此时它读到的就是加载好的值。

整个过程里，加载和使用之间隔了多少无关指令、加载到底花了多少周期，都不影响正确性——req_sb 等的是 “计数器归零” 这个事件，而不是一个固定的周期数。这正是计数器机制相对 stall 的好处：它能等一个编译期不知道时长的事件。

11.8 等多了与等少了的后果

软件管依赖，算对算错的后果不对称，必须分清。

等少了（漏挂依赖、req_sb 没等该等的计数器）：消费者在生产者还没完成时就发射了，读到的是旧值或半成品，这是数据竞争。它的结果是静默地算错——没有任何报错，程序照常跑完，只是答案不对，而且因为依赖时序和运行期延迟有关，往往难以复现、极难调试。

等多了（挂了不必要的依赖、stall 设得过大、计数器共用导致被最慢的拖住）：消费者等了本不必等的时间，结果仍然正确，只是慢了。

两者一对照，软件的目标就很清楚：**绝不能少等**（少等就是错），在此前提下**尽量不多等**（少等不得、多等只是慢）。编译器排依赖时永远偏向安全：宁可多挂一点依赖保证正确，再在确有把握时把多余的等待去掉换性能。

12 线程间同步与内存一致性

12.1 三类协调，不能互相替代

正确地运行一段多线程程序，需要三类不同范围的协调，各管各的，缺一不可：

- **warp 内**：同一个 warp 里指令之间的依赖与延迟，用 stall 和依赖计数器处理（见依赖与静态调度）。
- **warp 间**：同一个 CTA（或簇）里多个 warp 之间的集合等齐，用屏障处理。
- **内存可见性**：多个线程读写同一块内存时，谁的写在谁的读之前对对方可见，用栅栏处理。

这三类的关键区别在于：warp 内的计数器只能保证本 warp 自己的指令时序，管不了别的 warp 有没有跑到某处；屏障能让多个 warp 集合等齐，但不直接保证它们写出的数据已经对彼此可见；栅栏保证内存写的可见顺序，但不让任何线程停下来等别人。把它们搞混（比如以为过了屏障就一定能读到别的 warp 写的数据）是多线程出错的常见来源，后面每一节都会点出各自的边界。

12.2 warp 间屏障

一个 CTA 内的多个 warp 要在某个点集合等齐时，用屏障。

集合等齐：bar_sync。所有参与的线程都必须执行到这条指令、彼此等齐之后，才能一起继续。到达按线程逐个计数：一个 warp 执行一次 bar_sync，向屏障贡献的到达数等于谓词门控之后仍活跃的通道数量；被永久剔除（kill）的通道不在活跃掩码里，自然不计入。屏障放行后计数自动清零、进入下一代。一个 CTA 最多有 32 个具名屏障（barrier，由一个 5 位立即数选 0–31 号），可以同时用不同的屏障号协调不同的线程子集。bar_sync 默认是整个 CTA 都参与；也可以带一个线程数参数，让只有指定数量的线程参与（counted barrier，计数屏障），用于生产者—消费者这类只有一部分 warp 参与某次等齐的模型。

先到先声明、不阻塞自己：bar_arrive。普通 bar_sync 到达后会就地阻塞、等其他线程。有时希望 “我先声明我到了，但不停下来等，继续干别的，过会儿再来等齐”。bar_arrive 就做这件事：它向屏障报告 “我已到达” 但不阻塞，本线程接着往下执行。配合后续的 bar_sync，就能让一部分计算和 “等其他线程” 这件事重叠起来，而不是干等。

等齐顺带规约：bar_red。这是在集合等齐的同时，对每个线程带来的一个 1 位谓词做一次跨线程规约，结果广播给所有线程。规约方式由 bar_red_op 选定：popc（数有多少个线程的谓词为真）、and（是否所有线程都为真）、or（是否至少一个为真）。它等价于 “CTA 级的一次投票”，把 “等齐” 和 “统计全 CTA 的某个布尔条件” 一条指令完成。

12.3 内存一致性与可见范围

多个线程读写同一块内存，就有 “谁的写何时对谁可见” 的问题。GPIDL 把可见性的范围分成几层，由 consistency_scope 修饰符选定，从小到大：

- cta：同一个 CTA（线程块）内可见。
- sm：同一个 SM 内的 warp 之间可见。
- cluster：同一个线程块簇范围内可见。
- gpu：整个设备可见（跨 CTA、跨内核启动）。
- sys：系统级可见，连主机 CPU 和其他 GPU（通过统一内存）都能观察到。

范围越大，硬件为保证一致性要做的事越多、代价越高，所以应该按实际需要选最小的够用范围：CTA 内 warp 间通信用 cta 就够，跨设备或与主机通信才用 gpu/sys。这一层范围划分和执行模型的分组层级是对应的。

12.4 栅栏与 “完成 vs 可见”

保证内存写的可见顺序，用栅栏。

栅栏 fence。它在栅栏处对内存操作施加一道顺序约束，方向和强度由 fence_kind 选定：acq（acquire，获取）让本线程后续的读都能看到别的线程在此之前已 “释放” 的写；rel（release，释放）让本线程在此之前的所有写在此之后对别的线程可见；sc（sequentially consistent，顺序一致）是双向全序——栅栏之前的所有内存操作都在指定范围上变得可见之后，才放出后续操作，对所有类型的内存操作都生效，是最强的一档，需要强顺序保证的场景用它；async 是跨异步代理的栅栏，配合张量搬运使用（见矩阵计算与大块数据搬运），它管的是 “哪个代理看得见”、不是 “多远可见”。acq 和 rel 通常成对使用：生产者写完数据后做一次 release、消费者读数据前做一次 acquire，二者配合才能保证消费者读到的是生产者写好的数据。覆盖多远的观察者范围由 consistency_scope（cta/sm/cluster/gpu/sys）选定。

这里要分清两个容易混淆的概念：**完成**（一笔内存操作本身做完了）和**可见**（这笔操作的效果何时对别的线程可见）是两回事。一笔写 “完成” 了（本 warp 的依赖计数器已减一，见依赖与静态调度），不等于别的线程立刻就能读到它——要让别的线程按预期看到，还得靠栅栏在中间排好可见顺序。warp 内的计数器只回答 “我这笔做完没”，回答不了 “别人看见没”。

12.5 原子操作与内存语义修饰符

多个线程要同时更新同一个内存位置（比如累加一个全局计数器），普通的 “读—改—写” 会互相覆盖。原子读改写（atomic read-modify-write）把 “读旧值、算新值、写回” 做成一个不可分割的整体，保证并发更新不丢失。

GPIDL 提供通用原子 atom（按地址高位自动路由空间，见寄存器与内存）、全局原子 atomg、共享原子 atoms。要做哪种运算由 atom_op 选定，共 14 种：加 add、有符号/无符号的最小最大 imin_s/imin_u/imax_s/imax_u、自增自减 inc/dec、按位 and/or/xor、交换 exch、比较交换 cas/cast、浮点加 fadd。原子指令把旧值返回到目的寄存器；如果把目的设成零寄存器 RZ（见寄存器与内存），就退化成只写回、不返回旧值的规约语义。

原子和普通访存都带两个控制一致性的修饰符。mem_sem（内存语义）选 weak（弱，默认，只保证本指令自己的数据正确，不主动提供跨线程顺序，性能最好，是普通访存主路径）、strong（强，配合 consistency_scope 提供跨线程的顺序与一致性，用于线程间通信）、constant（常量语义，可无限缓存）、mmio（映射 IO，有外部副作用、禁止缓存优化）。跨线程通信要可靠，往往是 strong 加上合适的 consistency_scope，普通的本线程数据访问用默认的 weak 即可。

12.6 异步完成追踪：提交—等待

本架构提供异步操作的完成追踪能力，并提供可由多个执行单元共同使用的内存屏障对象（mbarrier）能力。软件可以建立同步对象、报告参与方或异步事务的进展、等待或查询完成状态，并在生命周期结束后释放相关同步状态，从而支持生产者—消费者协作和缓冲区复用。

实现应保证同步对象在重复使用时能够区分不同轮次，并保证消费者只有在相关异步数据可安全使用后才观察到完成。同步对象的内部表示、状态转换方式、存储尺寸与对齐、参数含义、操作划分、操作数接口和指令编码均属于实现或配套一致性文件的范围，本文件不作规定。

13 矩阵计算与大块数据搬运

13.1 为什么要专门的张量通路

深度学习的主体计算是矩阵乘累加：D = A × B + C，两个矩阵相乘、再加上一个累加矩阵。如果用执行模型讲的普通逐元素运算来做，吞吐太低，而且为了算一个输出元素要反复从寄存器取很多输入，取数本身就成了瓶颈。

GPIDL 因此配了一个专门的张量计算单元（tensor core，张量核心）：一条矩阵乘累加指令就让它算完一整块小矩阵的乘累加，内部把取数和乘加高度复用，吞吐远高于逐元素运算。这条专门通路是 GPIDL 面向现代深度学习负载最有价值的部分。

13.2 矩阵乘累加指令族

张量计算的核心是一族矩阵乘累加（MMA，Matrix Multiply-Accumulate）指令，做的都是 D = A × B + C，由整个 warp 的 32 个通道协作算一个矩阵块。它们按数据精度分成几条：

- **半精度** hmma：A、B 是半精度矩阵，由 hmma_fmt 选 bf16（脑浮点 16 位，1 符号 + 8 指数 + 7 尾数，动态范围与单精度相当但精度低，训练主流）、f16（IEEE 754 半精度，1 + 5 + 10，精度高、范围小，推理常用）或 tf32（TensorFloat-32，一种为张量核心设计的窄精度）。32 个通道协作算一个 16×8 的输出块，一条指令沿累加方向累加 K 个元素。
- **整数** imma：A、B 是 8 位或 4 位整数，累加器固定为 32 位整数。由 imma_fmt 选 A、B 各自的有无符号组合（如 u8u8、s8s8、u8s8，对应量化里 “无符号激活 × 有符号权重” 等搭配）。
- **双精度** dmma：A、B、累加器都是 64 位浮点，算一个 8×8 的块（双精度元素占寄存器对、每行装得下的元素少，所以块比半精度的小）。它属于可选的 fp64 扩展，只在配备 FP64 执行单元的实现上提供。
- **量化浮点** qmma：A、B 是 8 位/6 位/4 位的微缩浮点（mxfp 标准的低精度训练/推理格式），由 qmma_fmt 选具体配比组合。

累加长度 K 与精度自然耦合：一个寄存器槽装 32 位，高精度元素装得少、低精度装得多，所以低精度自然支持更大的 K（8 位主路径 K=32、4 位 K=64），用同样的寄存器一次累加更多元素。块形状由 mma_shape（m16n8k8/m16n8k16/m16n8k32/m16n8k64）等修饰符选取。这正是指令的组成说的 “单个助记符加结构化修饰符覆盖大量变体”：不为每种精度、每种形状单设助记符，而是用 hmma/imma 等少数助记符配上格式和形状修饰符，组合出全部形态。

上面这些是 warp 内 32 通道同步协作、操作数从向量寄存器取的形态。还有一种异步的整 warp 组协作形态 utcmma（asynchronous warpgroup-level MMA）：单个线程发射一条就能启动整个张量核心的运算，操作数通过统一寄存器传一个描述符或张量内存地址，不占用逐线程的向量寄存器；运算是解耦异步的，发起后线程不必停下等它算完。它适合更大规模、需要把计算和搬运彻底重叠的流水线，精度由 utc_precision（f16/int8/fp8/fp4）选取。utcmma 还支持 2:4 结构化稀疏（utc_sparse 修饰符选 sp24）：A 矩阵沿累加方向每 4 个元素只保留 2 个非零并压缩存放，每组用 2 个 2 位索引记录非零位置，这份元数据放在张量内存里、由一个专门的操作数给出地址；硬件按元数据从 B 侧挑出与非零元素配对的数据参与乘加，B 保持稠密。压缩让同样的硬件一次累加原来两倍长度的 K，把权重剪枝出的 50% 稀疏度变成实打实的吞吐收益。

13.3 低精度数据格式

深度学习能用很低的精度算而不明显损失效果，于是 GPIDL 支持一批低于半精度的数据格式，让同样的硬件和带宽算更多。

8 位浮点（FP8）有多种指数尾数配比，由格式修饰符（如类型转换指令的 cvt_ifmt/cvt_ofmt、量化矩阵

乘累加的 qmma_fmt）选取：e5m2（5 指数 + 2 尾数，动态范围大、精度低）、e4m3（4 指数 + 3 尾数，精度高、范围小）等等。指数位多则能表示的数值范围大、但有效数字少，尾数位多则反之，这是一对此消彼长的取舍。整数侧有 8 位和 4 位。更低的位宽（如 FP4，可表示的数只有 ±{0, 0.5, 1, 1.5, 2, 3, 4, 6} 十六个编码）必须配合块缩放（block scaling）使用：把张量切成小块，每块共享一个缩放因子，块内元素只表达相对大小，跨块的数量级差异由缩放因子吸收，从而在极低位宽下仍保住动态范围。utcmma 的缩放形态用 utc_scale_vec 修饰符一次选定块粒度和因子格式，三种合法组合：block32_ue8m0（每 32 个元素共享一个纯指数因子，OCP 微缩放标准 MXFP8/6/4 的方案）、block16_ue8m0（块减半、适配更细）、block16_ue4m3（因子换成带尾数的 8 位无符号浮点，不再被迫取 2 的整数次幂，缩放本身的量化误差更小——这就是 FP4 推理的主流方案 NVFP4）。

选哪种格式是软件在精度和数值范围之间的权衡，训练和推理需求还不同：训练的反向传播梯度量级跨度大，偏向动态范围大的格式（如 e5m2）；推理对范围要求低、更看重精度，偏向 e4m3 这类。GPIDL 把这些格式作为修饰符暴露出来，由上层按数值需求选择。

13.4 张量内存与就近供数

张量核心吞吐极高，如果它的输入输出都要经过逐线程的向量寄存器堆来回搬，寄存器堆的带宽会跟不上。GPIDL 因此提供一块专供张量核心的近端存储，叫张量内存（tensor memory）：它是 SM 内部的一块片上暂存，由张量核心直接读写，不经过向量寄存器堆。

围绕张量内存有一组指令：utcld（从张量内存加载到向量寄存器）、utcst（从向量寄存器写入张量内存）、utccp（从共享内存拷贝矩阵块到张量内存）、utcbar（张量核心屏障，同步张量核心异步操作的完成）。张量内存的访问有 warp 组协作的特点：一个 warp 的一次加载只能覆盖当前 CTA 张量内存的四分之一，需要整个 warp 组（4 个 warp）协作才能覆盖完整的张量内存。

张量内存的容量由软件自己管理。硬件为每个 CTA 维护一张 32 位的分配位图：把张量内存的列空间等分成 32 块，每一位记录一块的占用状态。utcalloc 在位图里寻找一段连续的空闲块、原子地占下并返回起始块号（找不到就通过谓词输出报告失败，还可以用 alloc_align 修饰符要求基址按 2 的幂对齐）；utcfree 按掩码把用完的块原子地清回空闲。更复杂的策略——比如生产者 warp 组分配、消费者 warp 组释放的移交，或软件调度器保存并恢复整张位图——用 utcamap 的读、写、置位、比较交换四个原始操作自行搭建，其中比较交换是多方并发修改位图时的无锁仲裁手段。

另有一对共享内存矩阵搬运指令 ldsm（shared matrix load）和 stsm（shared matrix store），把共享内存里的矩阵块按矩阵乘累加需要的布局一次摆进 32 个通道的向量寄存器、或反向写回。它们是给 hmma/imma 这类从向量寄存器取操作数的形态供数用的，按 matrix_shape 指定块的布局。

13.5 多维异步搬运引擎

本架构提供张量及大块数据的异步搬运能力（TMA）。该能力可在不同存储层次之间完成多维或连续数据块的搬运，并可支持预取、数据布局变换以及实现所支持的归约操作。搬运可与计算并行执行，以减少数据准备对计算通路的阻塞。

软件通过实现定义的控制信息描述搬运需求，硬件负责地址生成、数据传输和完成通知。描述信息的组织形式与大小、支持的维数和模式、缓存与多播策略、操作数接口、指令划分、完成协议和指令编码均不在本文件中规定。

13.6 异步拷贝与软件流水线

软件可以使用多个缓冲区，把异步搬运与矩阵或向量计算重叠执行。程序在使用目标数据或复用源、目标缓冲区之前，应通过本架构提供的完成追踪或 mbarrier 能力确认相关操作已经达到所需完成点，并按内存

一致性要求建立必要的可见顺序。

具体的提交粒度、队列组织、计数方式、等待阈值和指令序列由实现及配套软件接口规定，本文件仅规定具备上述功能能力。

14 架构定义

本章定义本架构的寄存器堆、内存空间、公共控制字段、操作数类型、Modifier 总览以及指令集。完整架构共 17 个文档分类、172 个指令助记符和 441 个 leaf 指令形态。对于 mbarrier 与 TMA，本文件仅声明其功能族和软件可观察效果，不规定具体指令清单、指令格式、编码、字段布局、对象或描述符内部格式及实现协议。

14.1 寄存器堆

寄存器堆由 operand_kinds 定义。表中的编号字段为该类操作数在指令中的编码位宽；槽位数为架构定义的合法命名槽位数量。立即数不在本表中定义，而由具体操作数的 bits 字段内联声明。

寄存器堆	槽位数	编号字段	命名	scope	特殊值	保留槽
pred	8	3-bit	P	per_thread	7=PT	—
sbreg	8	3-bit	SB	per_warp	7=SB_NONE	6
sreg	64	8-bit	UR	per_warp	63=URZ	—
upred	8	3-bit	UP	per_warp	7=UPT	—
vreg	256	8-bit	R	per_thread	255=RZ	—
paired_vreg	256	8-bit	R	per_thread	255=RZ	—
paired_sreg	64	8-bit	UR	per_warp	63=URZ	—

14.2 内存空间

内存空间	作用域	访问属性
global	device	read_write
shared	threadblock	read_write
local	thread	read_write
constant	device	read_only

14.3 公共控制字段

每条指令都隐含以下公共控制字段，asm 写法形如 &<field>=<value>，{} 表示可选（有 default）。下文每条指令的 Format 行**不重复列出**这些字段。

字段	位宽	默认值	说明
predicate_gate	—	—	整条指令的 4-bit predicate gate 控制字段。

字段	位宽	默认值	说明
stall	4	0	整条指令的 4-bit fixed-latency stall counter。
yield	1	0	整条指令的 1-bit yield hint。
req_sb	6	0	整条指令的 6-bit dependence-counter wait mask。
rd_sb	—	SB_NONE	整条指令的 dependence counter (SBx) read-release 选择字段。
wr_sb	—	SB_NONE	整条指令的 dependence counter (SBx) write-release 选择字段。

Predicate gate：simt 指令前缀 @{!}Pg（per-thread predicate），uniform 指令前缀 @{!}UPg（per-warp uniform predicate），predicate_mode == none 的指令无 predicate gate。下文 Format 行**不重复列出**。

14.4 操作数类型

kind	描述
pred	8 个槽位；3-bit 编号字段；scope per_thread；普通命名前缀 P；特殊值 7=PT
sbreg	8 个槽位；3-bit 编号字段；scope per_warp；普通命名前缀 SB；特殊值 7=SB_NONE；保留槽 6
sreg	64 个槽位；8-bit 编号字段；scope per_warp；普通命名前缀 UR；特殊值 63=URZ
upred	8 个槽位；3-bit 编号字段；scope per_warp；普通命名前缀 UP；特殊值 7=UPT
vreg	256 个槽位；8-bit 编号字段；scope per_thread；普通命名前缀 R；特殊值 255=RZ
paired_vreg	256 个槽位；8-bit 编号字段；scope per_thread；普通命名前缀 R；特殊值 255=RZ
paired_sreg	64 个槽位；8-bit 编号字段；scope per_warp；普通命名前缀 UR；特殊值 63=URZ

立即数操作数不作为全局 kind 预定义；指令 leaf 以内联 bits 字段声明立即数字段宽度。

14.5 Modifier 总览

所有 modifier 共用单一 modifier_defs 全局池，下方按字段名字典序列出。有 default 的引用在 Format 中写作 {.<modifier>}，没有 default 的引用写作 .<modifier> 并且实例必须赋值；引用作用域见 §7 指令章节。

modifier	enum 取值	default	field_class
accum_type	f32 / f16	f32	—
alloc_align	noalign / align	noalign	—
atom_op	add / imin_s / imin_u / imax_s / imax_u / inc / dec / and / or / xor / exch /…	—	—
bar_red_op	popc / and / or	popc	—
bool_format	bm / bf	—	—
bound_mode	clamp / wrap	clamp	—
byte_select	b0 / b1 / b2 / b3	b0	—
cache_evict	ef / en / el / lu / eu / na	—	—
cctl_level	l1 / l2 / const / tex	—	—
cctl_op	pf1 / pf2 / wb / iv / rs / ivall / wball	—	—
consistency_scope	cta / sm / cluster / gpu / sys	—	—
cvt_ifmt	f16 / bf16 / f32 / f64 / tf32 / u8 / s8 / u16 / s16 / u32 / s32 / u64 / s64 /…	—	—
cvt_ofmt	f16 / bf16 / f32 / f64 / tf32 / u8 / s8 / u16 / s16 / u32 / s32 / u64 / s64 /…	—	—
denorm_handling	keep / ftz / fmz	keep	—
dp_mode	4a / 2a_lo / 2a_hi	—	—
dst_byte_oft	b0 / b1 / b2 / b3	b0	—
dst_ofmt	lo / hi	lo	—
fence_kind	acq / rel / sc / async	sc	—
fp_cmp_type	f / lt / eq / le / gt / ne / ge / num / nan / ltu / equ / leu / gtu / neu / g…	—	—
ftz	disable / ftz	—	—
h_and	disable / h_and	disable	—
hmma_fmt	bf16 / f16 / tf32	bf16	—
imma_fmt	u8u8 / s8s8 / u8s8 / s8u8 / u4u4 / s4s4 / u4s4 / s4u4	s8s8	—
int_cmp_type	f / lt / eq / le / gt / ne / ge / t	—	—
is_signed	unsigned / signed	—	—
lane_reduce	broadcast / fill / or / or_fill	broadcast	—
ldst_length	u8 / s8 / u16 / s16 / b32 / b64 / b128 / b256	—	—
mask_negate	off / on	off	—
mask_op	nop / and / or / xor / andn2	nop	—
match_size	b32 / b64	b32	—
matrix_num	x1 / x2 / x4	—	—

modifier	enum 取值	default	field_class
matrix_shape	m88 / mt88 / m816 / m832 / mt1616	—	—
matrix_size	b16 / u4to8 / s4to8 / u2to4 / s2to4 / u4x16p64to8 / u6x16p32to8 / b8	—	—
maxreg_op	try_alloc / dealloc	try_alloc	—
mem_sem	constant / weak / strong / mmio	weak	—
merge_mode	normal / pack_hi / merge / pack_merge / unpack / unpack_merge	normal	—
mma_shape	m16n8k8 / m16n8k16 / m16n8k32 / m16n8k64	—	—
mufu_func	cos / sin / ex2 / lg2 / rcp / rsq / sqrt / tanh / fp64_rcp_seed / fp64_rsq_se…	—	—
nan	disable / nan	disable	—
nan_handling	nonan / nan	nonan	—
output_modifier	d2 / noscale / m2 / m4	—	—
pred_comb	and / or / xor	—	—
prmt_mode	idx / f4e / b4e / rc8 / ecl / ecr / rc16	idx	—
qmma_fmt	e4m3_e4m3 / e5m2_e5m2 / e4m3_e5m2 / e5m2_e4m3 / e2m3_e2m3 / e3m2_e3m2 / e2m3_…	e4m3_e4m3	—
redux_dtype	u32 / s32 / f32	—	—
redux_op	add / min / max / and / or / xor / maxabs / minabs	—	—
relu	disable / relu	disable	—
reuse	no_reuse / reuse	no_reuse	—
rnd_mode_fp	rne / rtz / rup / rdn / rs	—	—
rnd_mode_int	rni / rzi / rupi / rdni	—	—
sat	off / on	off	—
satfinite	disable / satfinite	disable	—
saturate	disable / saturate	disable	—
shf_dir	l / r	—	—
shf_type	lo / hi / u64 / s64	—	—
shfl_mode	idx / up / down / bfly	idx	—
shiftamt	no_shift / shiftamt	no_shift	—

modifier	enum 取值	default	field_class
sr_id	laneid / tid_x / tid_y / tid_z / ntid_x / ntid_y / ntid_z / ctaid_x / ctaid_y…	—	—
sr_sz	b32 / b64	b32	—
src0_int_fmt	u8 / s8 / u16 / s16	—	—
src1_int_fmt	u8 / s8	—	—
src_bit_modi	id / inv	id	—
src_byte_oft	b0 / b1 / b2 / b3	b0	—
src_extract	lo / hi	—	—
src_fp_modi	id / neg / abs / abs_then_neg	id	—
src_int_modi	id / neg	id	—
src_iswz	hilo / lo / hi / b32	hilo	—
src_sign_extract	off / on	off	—
type_fp16_bf16	fp16 / bf16	—	—
utc_buffer	buf0 / buf1 / buf2 / buf3	buf0	—
utc_keep_a	no_keep / keep	no_keep	—
utc_keep_b	no_keep / keep	no_keep	—
utc_precision	f16 / int8 / fp8 / fp4	—	—
utc_reuse_a	no_reuse / reuse	no_reuse	—
utc_reuse_b	no_reuse / reuse	no_reuse	—
utc_scale_vec	block32_ue8m0 / block16_ue8m0 / block16_ue4m3	block32_ue8m0	—
utc_sparse	dense / sp24	dense	—
utc_warp_special	off / on	off	—
utcbar_type	arrive_one_thread_zero / flush	arrive_one_thread_zero	—
utccp_shape	s128x256b / s128x128b / s64x128b / s32x128b	s128x256b	—
utcld_num	x1 / x2 / x4 / x8 / x16 / x32 / x64 / x128	x1	—
utcld_shape	s16x64b / s16x128b / s16x256b / s32x32b / s16x32bx2	s16x64b	—
vote_type	all / any / eq	all	—
xorsign	disable / xorsign	disable	—

14.6 指令编码公共区

本文件提供每个 leaf 指令形态的二进制编码。指令采用 64 bit、96 bit、128 bit 三类变长容器；字段 bit 编号以容器最低位为 bit 0。

所有容器的公共控制区位于 bit [22:2]，共 21 bit；payload 字段从 bit 23 开始。

容器	prefix	prefix 位	opcode 位	payload 区间
64b	0	bit 0	opcode 低位放 bit 1，其余 opcode 位放容器最高位连续字段	bit 23 至 opcode 高位字段之前
96b	10	bit [1:0]	opcode 位放容器最高位连续字段	bit 23 至 opcode 字段之前
128b	11	bit [1:0]	opcode 位放容器最高位连续字段	bit 23 至 opcode 字段之前

14.7 指令集

14.7.1 FP32 compute

fadd **category**: FP32 compute **predicate_mode**: simt

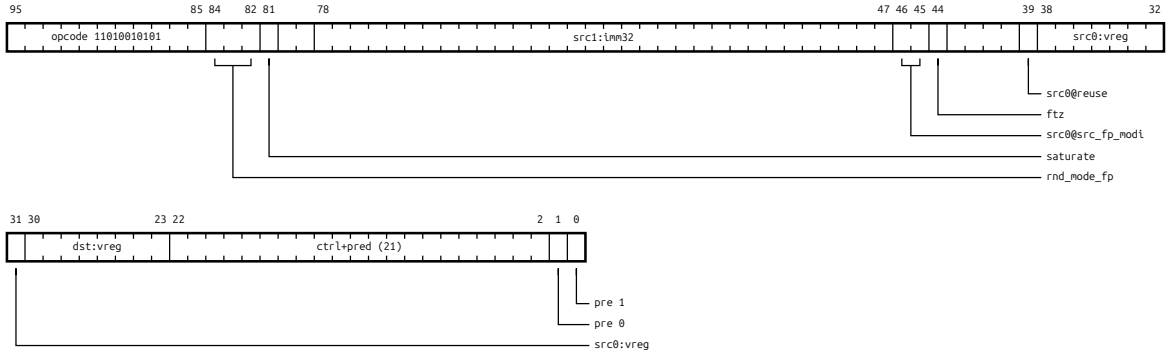
Leaf: fadd__v_vi

Format:

```
fadd.rnd_mode_fp.ftz{.saturate}  dst, src0{.src_fp_modi}{.reuse}, src1
```

Operands: dst: vreg; src0: vreg; src1: imm32u

Encoding Diagram: 96b, prefix 10, opcode 11010010101



Notes:

FP32 单精度浮点加法: dst = src0 + src1, 整 warp 32 个 active thread 各自独立计算, 结果按 IEEE 754 binary32 格式。

src0 / src1 各可挂 src_fp_modi: neg (取负) / abs (取绝对值) / abs_then_neg (先绝对值再取负), 一条指令内表达 -a+b、|a|+b 这种常见变体, 省一条 mov / abs。

ftz=disable (默认): 走 IEEE 754 行为, denorm (subnormal) 输入按原值参运算, denorm 输出按原值写 dst。

ftz=ftz: input denorm 进入运算前 flush 为 sign-preserving 0, output denorm 也 flush 为 sign-preserving 0 (ML 训练常用, 走更快路径)。

saturate=saturate: dst clamp 到 [+0.0, 1.0], NaN → +0.0 (ML activation 路径如 sigmoid / clamp 用)。

rnd_mode_fp: 4 种 IEEE 754 标准舍入 (rne 默认 round-to-nearest-even / rtz / rup / rdn)。

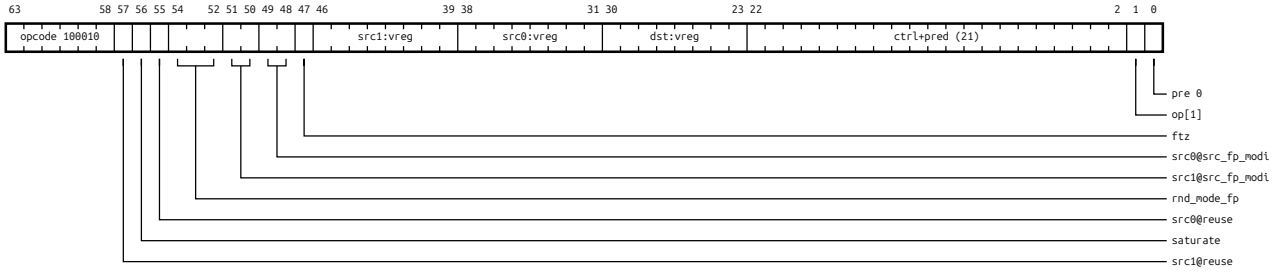
Leaf: fadd__v_vv

Format:

```
fadd.rnd_mode_fp.ftz{.saturate}  dst, src0{.src_fp_modi}{.reuse},
↪  src1{.src_fp_modi}{.reuse}
```

Operands: dst: vreg; src0: vreg; src1: vreg

Encoding Diagram: 64b, prefix 0, opcode 1100010



Notes:

FP32 单精度浮点加法: dst = src0 + src1, 整 warp 32 个 active thread 各自独立计算, 结果按 IEEE 754 binary32 格式。

src0 / src1 各可挂 src_fp_modi: neg (取负) / abs (取绝对值) / abs_then_neg (先绝对值再取负), 一条指令内表达 -a+b、|a|+b 这种常见变体, 省一条 mov / abs。

ftz=disable (默认): 走 IEEE 754 行为, denorm (subnormal) 输入按原值参运算, denorm 输出按原值写 dst。

ftz=ftz: input denorm 进入运算前 flush 为 sign-preserving 0, output denorm 也 flush 为 sign-preserving 0 (ML 训练常用, 走更快路径)。

saturate=saturate: dst clamp 到 [+0.0, 1.0], NaN → +0.0 (ML activation 路径如 sigmoid / clamp 用)。

rnd_mode_fp: 4 种 IEEE 754 标准舍入 (rne 默认 round-to-nearest-even / rtz / rup / rdn)。

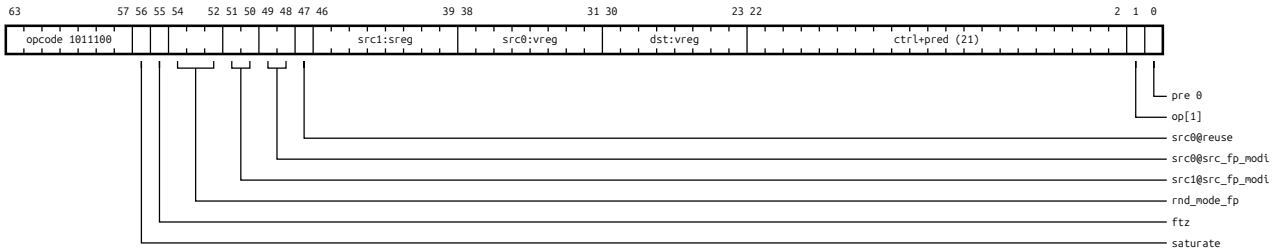
Leaf: fadd__v_vx

Format:

```
fadd.rnd_mode_fp.ftz{.saturate}  dst, src0{.src_fp_modi}{.reuse}, src1{.src_fp_modi}
```

Operands: dst: vreg; src0: vreg; src1: sreg

Encoding Diagram: 64b, prefix 0, opcode 11011100



Notes:

FP32 单精度浮点加法: dst = src0 + src1, 整 warp 32 个 active thread 各自独立计算, 结果按 IEEE 754 binary32 格式。

src0 / src1 各可挂 src_fp_modi: neg (取负) / abs (取绝对值) / abs_then_neg (先绝对值再取负), 一条指令内表达 -a+b、|a|+b 这种常见变体, 省一条 mov / abs。

ftz=disable (默认): 走 IEEE 754 行为, denorm (subnormal) 输入按原值参运算, denorm 输出按原值写 dst。

ftz=ftz: input denorm 进入运算前 flush 为 sign-preserving 0, output denorm 也 flush 为 sign-preserving 0 (ML 训练常用, 走更快路径)。

saturate=saturate: dst clamp 到 [+0.0, 1.0], NaN → +0.0 (ML activation 路径如 sigmoid / clamp 用)。

rnd_mode_fp: 4 种 IEEE 754 标准舍入 (rne 默认 round-to-nearest-even / rtz / rup / rdn)。

fchk **category**: FP32 compute **predicate_mode**: simt

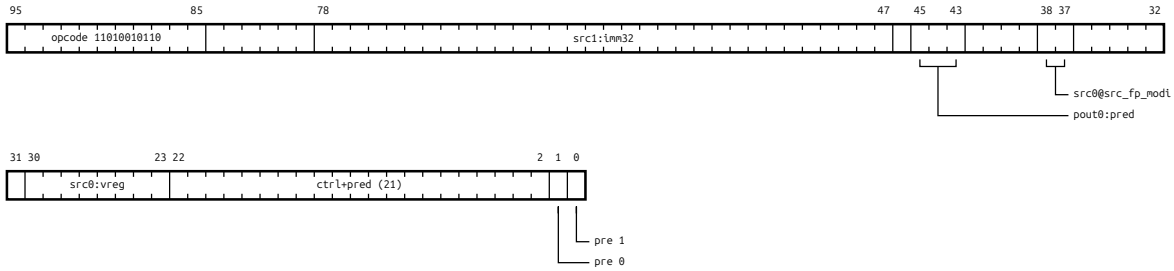
Leaf: fchk__p_vi

Format:

```
fchk  pout0, src0{.src_fp_modi}, src1
```

Operands: pout0: pred; src0: vreg; src1: imm32u

Encoding Diagram: 96b, prefix 10, opcode 11010010110



Notes:

FP divide range check: 检测 src0 (dividend) / src1 (divisor) 的 IEEE 754 unbiased exponent 是否会让软件 fdiv 序列 (典型 Newton-Raphson 迭代) 产生溢出 / 下溢 / 失精。pout0=1 表示输入处于风险范围, 需要走 fallback (精确除法 / 软件特殊处理); pout0=0 表示可走 NR 迭代 fast path。

用法: 放在软件 fdiv 实现序列的开头, 编译器按 pout0 分支选 fast / slow path。

记号: eS0 = unbiased exponent of src0 (dividend), eS1 = unbiased exponent of src1 (divisor); emin = -126, emax = +127, N = 24 (FP32 mantissa + 1)。

命中以下 6 个边界条件之一即 pout0=1:

- 1) eS0 ≤ emin + N - 1 (dividend 接近 subnormal 下界, 可能下溢)
- 2) eS0 ≥ emax + 1 (dividend 是 ±Inf 或 NaN)
- 3) eS1 ≤ emin (divisor 是 subnormal 或更小)
- 4) eS1 ≥ emax - 2 (divisor 是非常大的 normal 或 ±Inf)
- 5) eS0 - eS1 ≤ emin + 1 (商可能下溢)
- 6) eS0 - eS1 ≥ emax (商可能溢出)

src0 / src1 都可挂 src_fp_modi (neg / abs / abs_then_neg), 方便对 |src| 做范围检查。

FP 值分类 (单 src 类别检测) 见 fclass — fclass 跟 fchk 是不同概念。

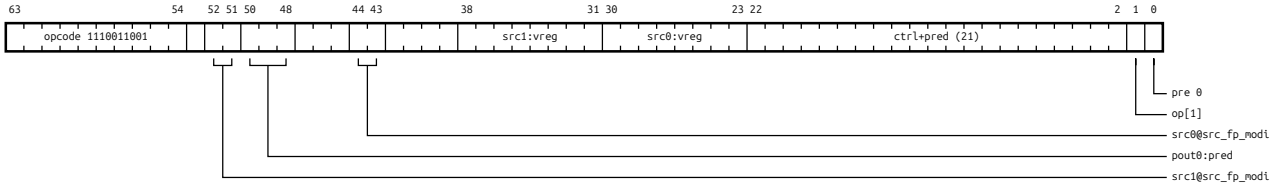
Leaf: fchk__p_vv

Format:

```
fchk  pout0, src0{.src_fp_modi}, src1{.src_fp_modi}
```

Operands: pout0: pred; src0: vreg; src1: vreg

Encoding Diagram: 64b, prefix 0, opcode 11110011001



Notes:

FP divide range check: 检测 src0 (dividend) / src1 (divisor) 的 IEEE 754 unbiased exponent 是否会让软件 fdiv 序列 (典型 Newton-Raphson 迭代) 产生溢出 / 下溢 / 失精。pout0=1 表示输入处于风险范围, 需要走 fallback (精确除法 / 软件特殊处理); pout0=0 表示可走 NR 迭代 fast path。

用法: 放在软件 fdiv 实现序列的开头, 编译器按 pout0 分支选 fast / slow path。

记号: eS0 = unbiased exponent of src0 (dividend), eS1 = unbiased exponent of src1 (divisor); emin = -126, emax = +127, N = 24 (FP32 mantissa + 1)。

命中以下 6 个边界条件之一即 pout0=1:

- 1) eS0 ≤ emin + N - 1 (dividend 接近 subnormal 下界, 可能下溢)
- 2) eS0 ≥ emax + 1 (dividend 是 ±Inf 或 NaN)
- 3) eS1 ≤ emin (divisor 是 subnormal 或更小)
- 4) eS1 ≥ emax - 2 (divisor 是非常大的 normal 或 ±Inf)
- 5) eS0 - eS1 ≤ emin + 1 (商可能下溢)
- 6) eS0 - eS1 ≥ emax (商可能溢出)

src0 / src1 都可挂 src_fp_modi (neg / abs / abs_then_neg), 方便对 |src| 做范围检查。

FP 值分类 (单 src 类别检测) 见 fclass — fclass 跟 fchk 是不同概念。

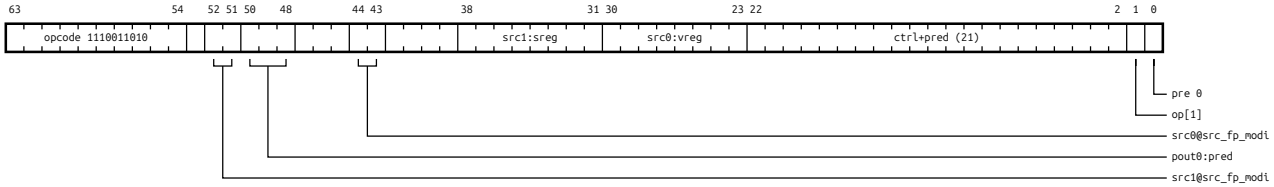
Leaf: fchk__p_vx

Format:

```
fchk  pout0, src0{.src_fp_modi}, src1{.src_fp_modi}
```

Operands: pout0: pred; src0: vreg; src1: sreg

Encoding Diagram: 64b, prefix 0, opcode 11110011010



Notes:

FP divide range check: 检测 src0 (dividend) / src1 (divisor) 的 IEEE 754 unbiased exponent 是否会让软件 fdiv 序列 (典型 Newton-Raphson 迭代) 产生溢出 / 下溢 / 失精。pout0=1 表示输入处于风险范围, 需要走 fallback (精确除法 / 软件特殊处理); pout0=0 表示可走 NR 迭代 fast path。

用法: 放在软件 fdiv 实现序列的开头, 编译器按 pout0 分支选 fast / slow path。

记号: eS0 = unbiased exponent of src0 (dividend), eS1 = unbiased exponent of src1 (divisor); emin = -126, emax = +127, N = 24 (FP32 mantissa + 1)。

命中以下 6 个边界条件之一即 pout0=1:

- 1) eS0 ≤ emin + N - 1 (dividend 接近 subnormal 下界, 可能下溢)
- 2) eS0 ≥ emax + 1 (dividend 是 ±Inf 或 NaN)

- 3) eS1 ≤ emin (divisor 是 subnormal 或更小)
- 4) eS1 ≥ emax - 2 (divisor 是非常大的 normal 或 ±Inf)
- 5) eS0 - eS1 ≤ emin + 1 (商可能下溢)
- 6) eS0 - eS1 ≥ emax (商可能溢出)

src0 / src1 都可挂 src_fp_modi (neg / abs / abs_then_neg), 方便对 |src| 做范围检查。

FP 值分类 (单 src 类别检测) 见 fclass — fclass 跟 fchk 是不同概念。

fclass **category**: FP32 compute **predicate_mode**: simt

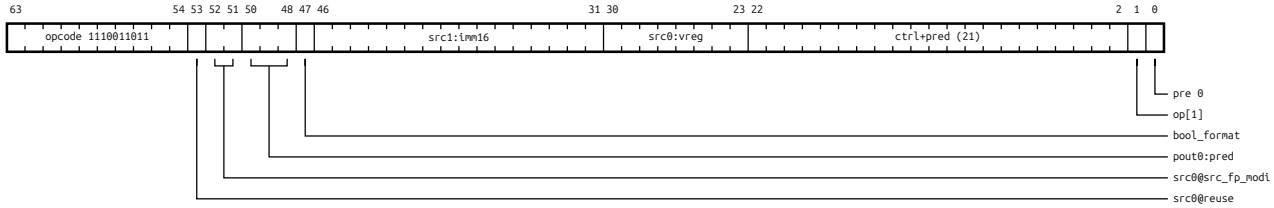
Leaf: fclass__p_vi

Format:

```
fclass.bool_format  pout0, src0{.src_fp_modi}{.reuse}, src1
```

Operands: pout0: pred; src0: vreg; src1: imm16u

Encoding Diagram: 64b, prefix 0, opcode 11110011011



Notes:

FP 值分类: 从 src0 (fp32) 推断它属于 IEEE 754 哪一种类别 (共 10 类, 每类对应 1 个 bit), 再跟 src1 编码的 10-bit or-mask 做按位 and, 任一 bit 命中即 pout0=1。整体语义 pout0 = (class_of(src0) ∈ src1.mask)。

10 类 IEEE 754 浮点值 (按符号对称, 从 SNaN 单调排到 +Inf):

- bit 0: SNaN (Signaling NaN, exponent=0xFF, mantissa MSB=0, 至少 1 个其他 mantissa bit=1)
- bit 1: QNaN (Quiet NaN, exponent=0xFF, mantissa MSB=1)
- bit 2: -Inf (negative infinity)
- bit 3: -Norm (negative normal)
- bit 4: -Subnormal (negative denormal)
- bit 5: -Zero
- bit 6: +Zero
- bit 7: +Subnormal (positive denormal)
- bit 8: +Norm (positive normal)
- bit 9: +Inf (positive infinity)

常用 mask 示例:

- 0x003 = SNaN | QNaN → '是任何 NaN'
- 0x204 = -Inf | +Inf → '是 ±Inf'
- 0x060 = -Zero | +Zero → '是 ±Zero'
- 0x108 = -Norm | +Norm → '是 ±Norm'
- 0x3F8 = 全部非 NaN, 非 Inf → '是 finite'
- 0x3FC = 全部非 NaN → '是 number'

src1 (10-bit mask) 三种来源: imm.16 (静态 mask, 最常见, 编译期固定) / reg.vreg (per-lane 动态 mask, 少见但可做 conditional class test) / reg.sreg (per-warp 动态 mask)。寄存器形式只读低 10 bit, 高 22 bit 忽略。

vp_* 指令形态同时输出 dst (vreg) + pout0 (pred), bool_format 选 dst 写入格式: bm (boolean mask, true→0xFFFFFFFF / false→0) / bf (boolean float, true→1.0f / false→0.0f)。p_* 指令形态仅 pout0 不写 dst, 节省 vreg 槽位。

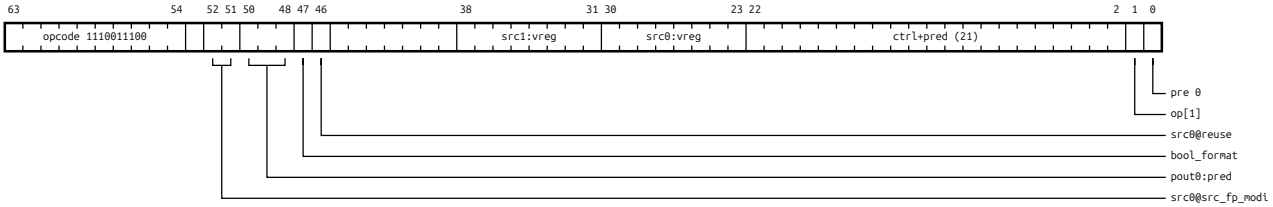
Leaf: fclass__p_vv

Format:

```
fclass.bool_format  pout0, src0{.src_fp_modi}{.reuse}, src1
```

Operands: pout0: pred; src0: vreg; src1: vreg

Encoding Diagram: 64b, prefix 0, opcode 11110011100



Notes:

FP 值分类: 从 src0 (fp32) 推断它属于 IEEE 754 哪一种类别 (共 10 类, 每类对应 1 个 bit), 再跟 src1 编码的 10-bit or-mask 做按位 and, 任一 bit 命中即 pout0=1。整体语义 pout0 = (class_of(src0) ∈ src1.mask)。

10 类 IEEE 754 浮点值 (按符号对称, 从 SNaN 单调排到 +Inf):

- bit 0: SNaN (Signaling NaN, exponent=0xFF, mantissa MSB=0, 至少 1 个其他 mantissa bit=1)
- bit 1: QNaN (Quiet NaN, exponent=0xFF, mantissa MSB=1)
- bit 2: -Inf (negative infinity)
- bit 3: -Norm (negative normal)
- bit 4: -Subnormal (negative denormal)
- bit 5: -Zero
- bit 6: +Zero
- bit 7: +Subnormal (positive denormal)
- bit 8: +Norm (positive normal)
- bit 9: +Inf (positive infinity)

常用 mask 示例:

- 0x003 = SNaN | QNaN → '是任何 NaN'
- 0x204 = -Inf | +Inf → '是 ±Inf'
- 0x060 = -Zero | +Zero → '是 ±Zero'
- 0x108 = -Norm | +Norm → '是 ±Norm'
- 0x3F8 = 全部非 NaN, 非 Inf → '是 finite'
- 0x3FC = 全部非 NaN → '是 number'

src1 (10-bit mask) 三种来源: imm.16 (静态 mask, 最常见, 编译期固定) / reg.vreg (per-lane 动态 mask, 少见但可做 conditional class test) / reg.sreg (per-warp 动态 mask)。寄存器形式只读低 10 bit, 高 22 bit 忽略。

vp_* 指令形态同时输出 dst (vreg) + pout0 (pred), bool_format 选 dst 写入格式: bm (boolean mask, true→0xFFFFFFFF / false→0) / bf (boolean float, true→1.0f / false→0.0f)。p_* 指令形态仅 pout0 不写 dst, 节省

vreg 槽位。

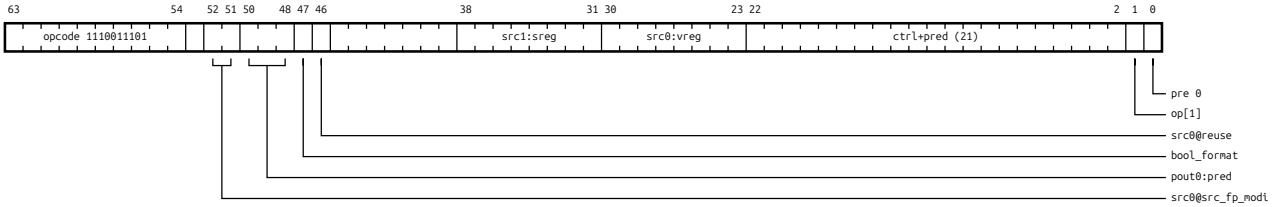
Leaf: fclass__p_vx

Format:

```
fclass.bool_format  pout0, src0{.src_fp_modi}{.reuse}, src1
```

Operands: pout0: pred; src0: vreg; src1: sreg

Encoding Diagram: 64b, prefix 0, opcode 11110011101



Notes:

FP 值分类: 从 src0 (fp32) 推断它属于 IEEE 754 哪一种类别 (共 10 类, 每类对应 1 个 bit), 再跟 src1 编码的 10-bit or-mask 做按位 and, 任一 bit 命中即 pout0=1。整体语义 pout0 = (class_of(src0) ∈ src1.mask)。

10 类 IEEE 754 浮点值 (按符号对称, 从 SNaN 单调排到 +Inf):

- bit 0: SNaN (Signaling NaN, exponent=0xFF, mantissa MSB=0, 至少 1 个其他 mantissa bit=1)
- bit 1: QNaN (Quiet NaN, exponent=0xFF, mantissa MSB=1)
- bit 2: -Inf (negative infinity)
- bit 3: -Norm (negative normal)
- bit 4: -Subnormal (negative denormal)
- bit 5: -Zero
- bit 6: +Zero
- bit 7: +Subnormal (positive denormal)
- bit 8: +Norm (positive normal)
- bit 9: +Inf (positive infinity)

常用 mask 示例:

- 0x003 = SNaN | QNaN → '是任何 NaN'
- 0x204 = -Inf | +Inf → '是 ±Inf'
- 0x060 = -Zero | +Zero → '是 ±Zero'
- 0x108 = -Norm | +Norm → '是 ±Norm'
- 0x3F8 = 全部非 NaN, 非 Inf → '是 finite'
- 0x3FC = 全部非 NaN → '是 number'

src1 (10-bit mask) 三种来源: imm.16 (静态 mask, 最常见, 编译期固定) / reg.vreg (per-lane 动态 mask, 少见但可做 conditional class test) / reg.sreg (per-warp 动态 mask)。寄存器形式只读低 10 bit, 高 22 bit 忽略。

vp_* 指令形态同时输出 dst (vreg) + pout0 (pred), bool_format 选 dst 写入格式: bm (boolean mask, true→0xFFFFFFFF / false→0) / bf (boolean float, true→1.0f / false→0.0f)。p_* 指令形态仅 pout0 不写 dst, 节省 vreg 槽位。

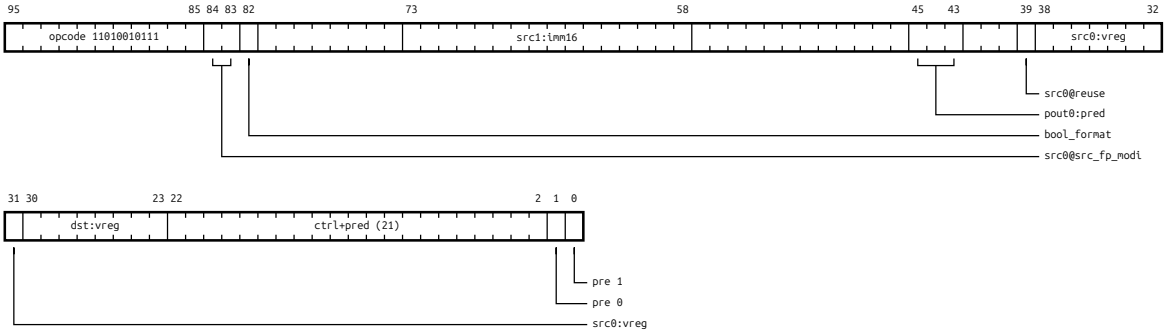
Leaf: fclass__vp_vi

Format:

```
fclass.bool_format  dst, pout0, src0{.src_fp_modi}{.reuse}, src1
```

Operands: dst: vreg; pout0: pred; src0: vreg; src1: imm16u

Encoding Diagram: 96b, prefix 10, opcode 11010010111



Notes:

FP 值分类: 从 src0 (fp32) 推断它属于 IEEE 754 哪一种类别 (共 10 类, 每类对应 1 个 bit), 再跟 src1 编码的 10-bit or-mask 做按位 and, 任一 bit 命中即 pout0=1。整体语义 pout0 = (class_of(src0) ∈ src1.mask)。

10 类 IEEE 754 浮点值 (按符号对称, 从 SNaN 单调排到 +Inf):

- bit 0: SNaN (Signaling NaN, exponent=0xFF, mantissa MSB=0, 至少 1 个其他 mantissa bit=1)
- bit 1: QNaN (Quiet NaN, exponent=0xFF, mantissa MSB=1)
- bit 2: -Inf (negative infinity)
- bit 3: -Norm (negative normal)
- bit 4: -Subnormal (negative denormal)
- bit 5: -Zero
- bit 6: +Zero
- bit 7: +Subnormal (positive denormal)
- bit 8: +Norm (positive normal)
- bit 9: +Inf (positive infinity)

常用 mask 示例:

- 0x003 = SNaN | QNaN → '是任何 NaN'
- 0x204 = -Inf | +Inf → '是 ±Inf'
- 0x060 = -Zero | +Zero → '是 ±Zero'
- 0x108 = -Norm | +Norm → '是 ±Norm'
- 0x3F8 = 全部非 NaN, 非 Inf → '是 finite'
- 0x3FC = 全部非 NaN → '是 number'

src1 (10-bit mask) 三种来源: imm.16 (静态 mask, 最常见, 编译期固定) / reg.vreg (per-lane 动态 mask, 少见但可做 conditional class test) / reg.sreg (per-warp 动态 mask)。寄存器形式只读低 10 bit, 高 22 bit 忽略。

vp_* 指令形态同时输出 dst (vreg) + pout0 (pred), bool_format 选 dst 写入格式: bm (boolean mask, true→0xFFFFFFFF / false→0) / bf (boolean float, true→1.0f / false→0.0f)。p_* 指令形态仅 pout0 不写 dst, 节省 vreg 槽位。

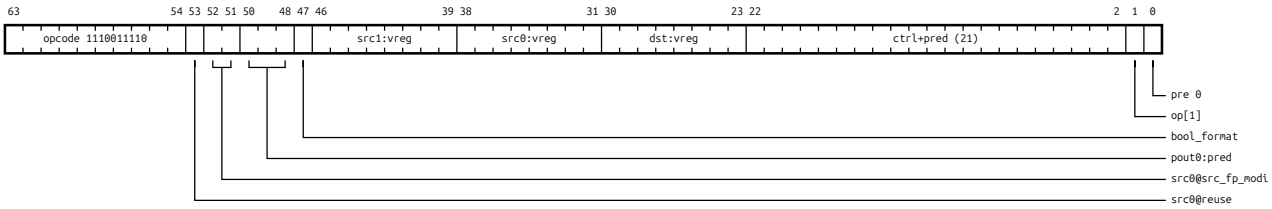
Leaf: fclass__vp_vv

Format:

```
fclass.bool_format  dst, pout0, src0{.src_fp_modi}{.reuse}, src1
```

Operands: dst: vreg; pout0: pred; src0: vreg; src1: vreg

Encoding Diagram: 64b, prefix 0, opcode 11110011110



Notes:

FP 值分类: 从 src0 (fp32) 推断它属于 IEEE 754 哪一种类别 (共 10 类, 每类对应 1 个 bit), 再跟 src1 编码的 10-bit or-mask 做按位 and, 任一 bit 命中即 pout0=1。整体语义 pout0 = (class_of(src0) ∈ src1.mask)。

10 类 IEEE 754 浮点值 (按符号对称, 从 SNaN 单调排到 +Inf):

- bit 0: SNaN (Signaling NaN, exponent=0xFF, mantissa MSB=0, 至少 1 个其他 mantissa bit=1)
- bit 1: QNaN (Quiet NaN, exponent=0xFF, mantissa MSB=1)
- bit 2: -Inf (negative infinity)
- bit 3: -Norm (negative normal)
- bit 4: -Subnormal (negative denormal)
- bit 5: -Zero
- bit 6: +Zero
- bit 7: +Subnormal (positive denormal)
- bit 8: +Norm (positive normal)
- bit 9: +Inf (positive infinity)

常用 mask 示例:

- 0x003 = SNaN | QNaN → '是任何 NaN'
- 0x204 = -Inf | +Inf → '是 ±Inf'
- 0x060 = -Zero | +Zero → '是 ±Zero'
- 0x108 = -Norm | +Norm → '是 ±Norm'
- 0x3F8 = 全部非 NaN, 非 Inf → '是 finite'
- 0x3FC = 全部非 NaN → '是 number'

src1 (10-bit mask) 三种来源: imm.16 (静态 mask, 最常见, 编译期固定) / reg.vreg (per-lane 动态 mask, 少见但可做 conditional class test) / reg.sreg (per-warp 动态 mask)。寄存器形式只读低 10 bit, 高 22 bit 忽略。

vp_* 指令形态同时输出 dst (vreg) + pout0 (pred), bool_format 选 dst 写入格式: bm (boolean mask, true→0xFFFFFFFF / false→0) / bf (boolean float, true→1.0f / false→0.0f)。p_* 指令形态仅 pout0 不写 dst, 节省 vreg 槽位。

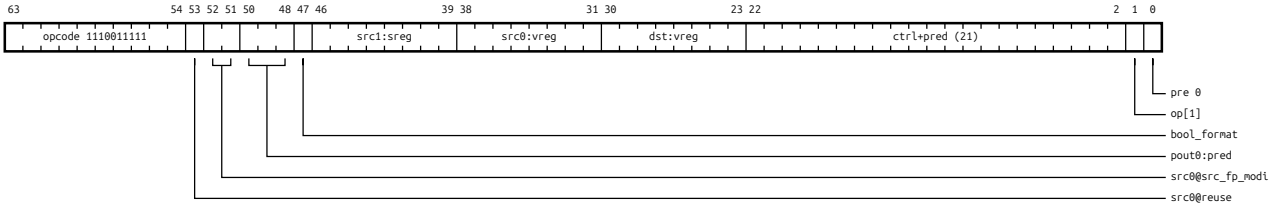
Leaf: fclass__vp_vx

Format:

```
fclass.bool_format  dst, pout0, src0{.src_fp_modi}{.reuse}, src1
```

Operands: dst: vreg; pout0: pred; src0: vreg; src1: sreg

Encoding Diagram: 64b, prefix 0, opcode 11110011111



Notes:

FP 值分类: 从 src0 (fp32) 推断它属于 IEEE 754 哪一种类别 (共 10 类, 每类对应 1 个 bit), 再跟 src1 编码的 10-bit or-mask 做按位 and, 任一 bit 命中即 pout0=1。整体语义 pout0 = (class_of(src0) ∈ src1.mask)。

10 类 IEEE 754 浮点值 (按符号对称, 从 SNaN 单调排到 +Inf):

- bit 0: SNaN (Signaling NaN, exponent=0xFF, mantissa MSB=0, 至少 1 个其他 mantissa bit=1)
- bit 1: QNaN (Quiet NaN, exponent=0xFF, mantissa MSB=1)
- bit 2: -Inf (negative infinity)
- bit 3: -Norm (negative normal)
- bit 4: -Subnormal (negative denormal)
- bit 5: -Zero
- bit 6: +Zero
- bit 7: +Subnormal (positive denormal)
- bit 8: +Norm (positive normal)
- bit 9: +Inf (positive infinity)

常用 mask 示例:

- 0x003 = SNaN | QNaN → '是任何 NaN'
- 0x204 = -Inf | +Inf → '是 ±Inf'
- 0x060 = -Zero | +Zero → '是 ±Zero'
- 0x108 = -Norm | +Norm → '是 ±Norm'
- 0x3F8 = 全部非 NaN, 非 Inf → '是 finite'
- 0x3FC = 全部非 NaN → '是 number'

src1 (10-bit mask) 三种来源: imm.16 (静态 mask, 最常见, 编译期固定) / reg.vreg (per-lane 动态 mask, 少见但可做 conditional class test) / reg.sreg (per-warp 动态 mask)。寄存器形式只读低 10 bit, 高 22 bit 忽略。

vp_* 指令形态同时输出 dst (vreg) + pout0 (pred), bool_format 选 dst 写入格式: bm (boolean mask, true→0xFFFFFFFF / false→0) / bf (boolean float, true→1.0f / false→0.0f)。p_* 指令形态仅 pout0 不写 dst, 节省 vreg 槽位。

ffma **category**: FP32 compute **predicate_mode**: simt

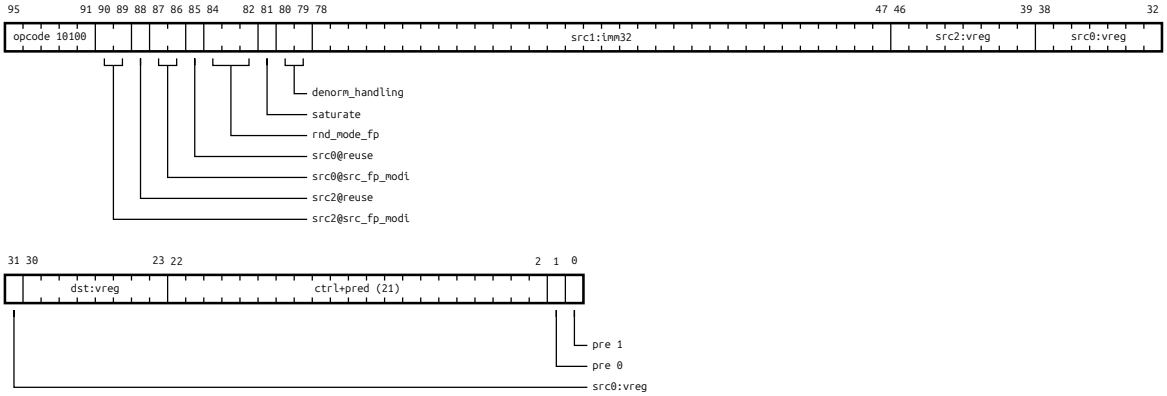
Leaf: ffma__v_viv

Format:

```
ffma.rnd_mode_fp{.denorm_handling}{.saturate}  dst, src0{.src_fp_modi}{.reuse}, src1,
↪  src2{.src_fp_modi}{.reuse}
```

Operands: dst: vreg; src0: vreg; src1: imm32u; src2: vreg

Encoding Diagram: 96b, prefix 10, opcode 10100



Notes:

FP32 fused multiply-add: dst = src0 × src1 + src2, 整个表达式按 IEEE 754 binary32 算到无限精度后只按 rnd_mode_fp 舍入一次, 比 fmul + fadd 序列精度高一位。

src0 / src1 / src2 各可挂 src_fp_modi (neg / abs / abs_then_neg), 一条 ffma 表达 -a×b + c、|a|×b - c 这种 ML 常见变体, 省一条 mov / abs。

denorm_handling=keep (默认): IEEE 754 行为, denorm 输入 / 输出 / 中间子积全部按原值保留。

denorm_handling=ftz: input / 中间子积 / output denorm 全部 flush 为 sign-preserving 0。

denorm_handling=fmz: 等同 ftz (输入/输出 denorm 全 flush 为保号 0), 并且任一乘法源 (src0 / src1, 0 测试在输入 flush 之后做) 为 0 时强制中间子积 src0 × src1 = +0 (无视 inf / NaN / sign, 压掉 inf×0=NaN 路径); 加数 src2 不参与 0 测试, 照常相加。fmz 严格强于 ftz, 适合 ML packed fp16 主路径。

saturate=saturate: dst clamp 到 [+0.0, 1.0], NaN → +0.0。

rnd_mode_fp: 4 种 IEEE 754 舍入 (rne 默认 / rtz / rup / rdn)。

fp16 single-half FMA 见 fhfma; packed fp16 / bf16 FMA 见 hma_pk。

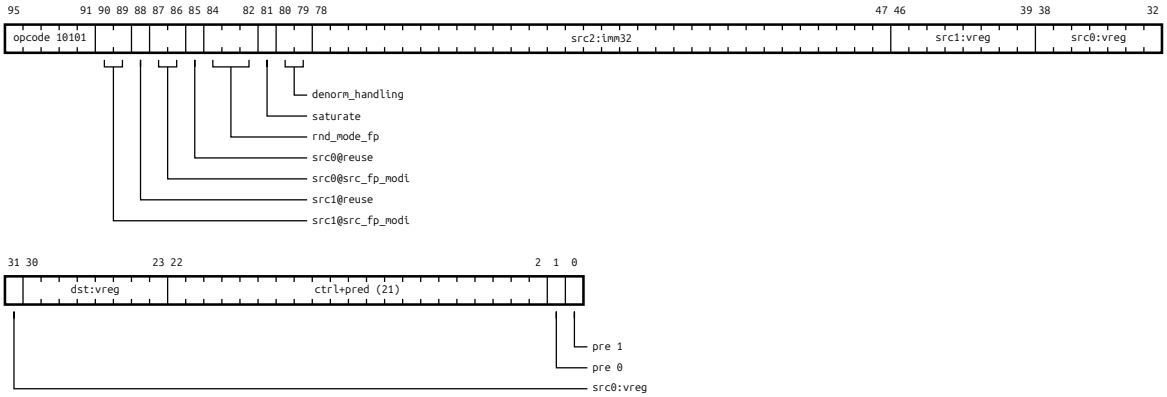
Leaf: ffma__v_vvi

Format:

```
ffma.rnd_mode_fp{.denorm_handling}{.saturate}  dst, src0{.src_fp_modi}{.reuse},
↪  src1{.src_fp_modi}{.reuse}, src2
```

Operands: dst: vreg; src0: vreg; src1: vreg; src2: imm32u

Encoding Diagram: 96b, prefix 10, opcode 10101



Notes:

FP32 fused multiply-add: dst = src0 × src1 + src2, 整个表达式按 IEEE 754 binary32 算到无限精度后只按 rnd_mode_fp 舍入一次, 比 fmul + fadd 序列精度高一位。

src0 / src1 / src2 各可挂 src_fp_modi (neg / abs / abs_then_neg), 一条 ffma 表达 -a×b + c、|a|×b - c 这种 ML

常见变体, 省一条 mov / abs。

denorm_handling=keep (默认): IEEE 754 行为, denorm 输入 / 输出 / 中间子积全部按原值保留。

denorm_handling=ftz: input / 中间子积 / output denorm 全部 flush 为 sign-preserving 0。

denorm_handling=fmz: 等同 ftz (输入/输出 denorm 全 flush 为保号 0), 并且任一乘法源 (src0 / src1, 0 测试在输入 flush 之后做) 为 0 时强制中间子积 src0 × src1 = +0 (无视 inf / NaN / sign, 压掉 inf×0=NaN 路径); 加数 src2 不参与 0 测试, 照常相加。fmz 严格强于 ftz, 适合 ML packed fp16 主路径。

saturate=saturate: dst clamp 到 [+0.0, 1.0], NaN → +0.0。

rnd_mode_fp: 4 种 IEEE 754 舍入 (rne 默认 / rtz / rup / rdn)。

fp16 single-half FMA 见 fhfma; packed fp16 / bf16 FMA 见 hma_pk。

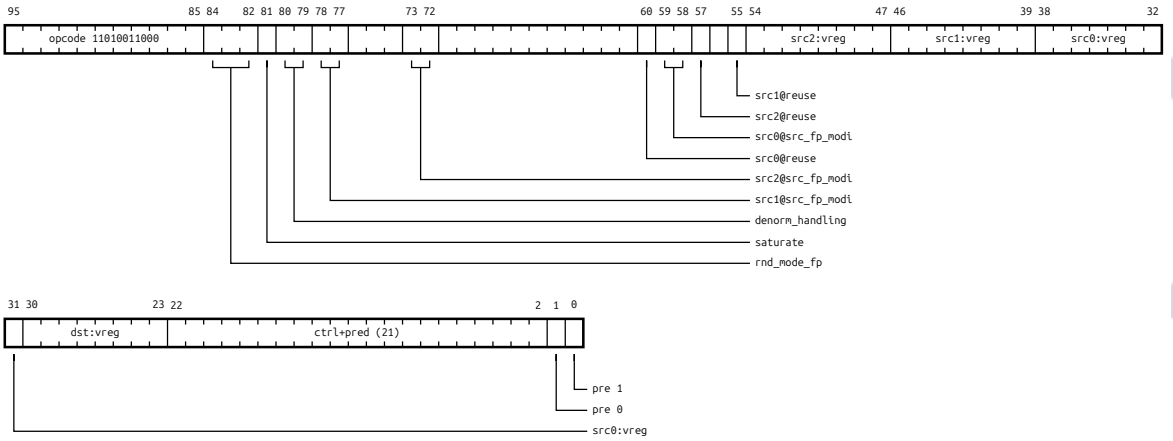
Leaf: ffma__v_vvv

Format:

```
ffma.rnd_mode_fp{.denorm_handling}{.saturate}  dst, src0{.src_fp_modi}{.reuse},
↪  src1{.src_fp_modi}{.reuse}, src2{.src_fp_modi}{.reuse}
```

Operands: dst: vreg; src0: vreg; src1: vreg; src2: vreg

Encoding Diagram: 96b, prefix 10, opcode 11010011000



Notes:

FP32 fused multiply-add: dst = src0 × src1 + src2, 整个表达式按 IEEE 754 binary32 算到无限精度后只按 rnd_mode_fp 舍入一次, 比 fmul + fadd 序列精度高一位。

src0 / src1 / src2 各可挂 src_fp_modi (neg / abs / abs_then_neg), 一条 ffma 表达 -a×b + c、|a|×b - c 这种 ML 常见变体, 省一条 mov / abs。

denorm_handling=keep (默认): IEEE 754 行为, denorm 输入 / 输出 / 中间子积全部按原值保留。

denorm_handling=ftz: input / 中间子积 / output denorm 全部 flush 为 sign-preserving 0。

denorm_handling=fmz: 等同 ftz (输入/输出 denorm 全 flush 为保号 0), 并且任一乘法源 (src0 / src1, 0 测试在输入 flush 之后做) 为 0 时强制中间子积 src0 × src1 = +0 (无视 inf / NaN / sign, 压掉 inf×0=NaN 路径); 加数 src2 不参与 0 测试, 照常相加。fmz 严格强于 ftz, 适合 ML packed fp16 主路径。

saturate=saturate: dst clamp 到 [+0.0, 1.0], NaN → +0.0。

rnd_mode_fp: 4 种 IEEE 754 舍入 (rne 默认 / rtz / rup / rdn)。

fp16 single-half FMA 见 fhfma; packed fp16 / bf16 FMA 见 hma_pk。

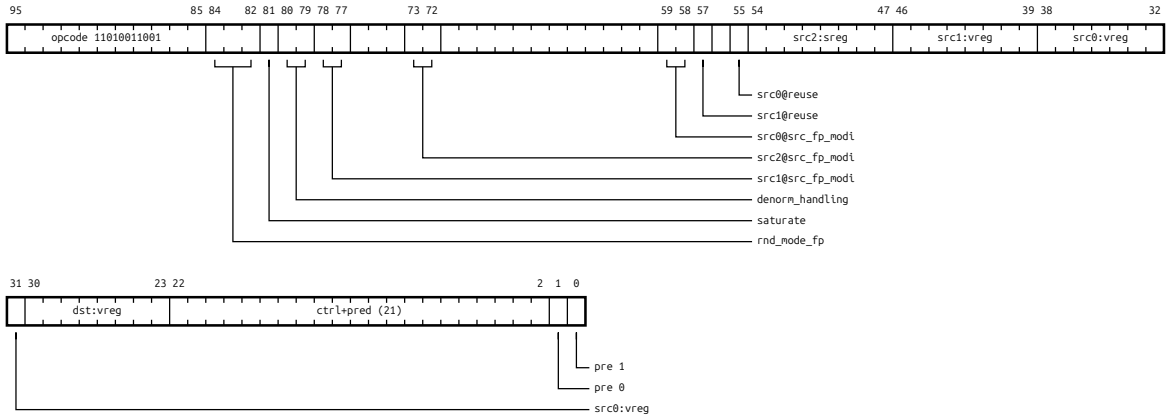
Leaf: ffma__v_vvx

Format:

```
ffma.rnd_mode_fp{.denorm_handling}{.saturate}  dst, src0{.src_fp_modi}{.reuse},
↪  src1{.src_fp_modi}{.reuse}, src2{.src_fp_modi}
```

Operands: dst: vreg; src0: vreg; src1: vreg; src2: sreg

Encoding Diagram: 96b, prefix 10, opcode 11010011001



Notes:

FP32 fused multiply-add: dst = src0 × src1 + src2, 整个表达式按 IEEE 754 binary32 算到无限精度后只按 rnd_mode_fp 舍入一次, 比 fmul + fadd 序列精度高一位。

src0 / src1 / src2 各可挂 src_fp_modi (neg / abs / abs_then_neg), 一条 ffma 表达 -a×b + c、|a|×b - c 这种 ML 常见变体, 省一条 mov / abs。

denorm_handling=keep (默认): IEEE 754 行为, denorm 输入 / 输出 / 中间子积全部按原值保留。

denorm_handling=ftz: input / 中间子积 / output denorm 全部 flush 为 sign-preserving 0。

denorm_handling=fmz: 等同 ftz (输入/输出 denorm 全 flush 为保号 0), 并且任一乘法源 (src0 / src1, 0 测试在输入 flush 之后做) 为 0 时强制中间子积 src0 × src1 = +0 (无视 inf / NaN / sign, 压掉 inf×0=NaN 路径); 加数 src2 不参与 0 测试, 照常相加。fmz 严格强于 ftz, 适合 ML packed fp16 主路径。

saturate=saturate: dst clamp 到 [+0.0, 1.0], NaN → +0.0。

rnd_mode_fp: 4 种 IEEE 754 舍入 (rne 默认 / rtz / rup / rdn)。

fp16 single-half FMA 见 fhfma; packed fp16 / bf16 FMA 见 hma_pk。

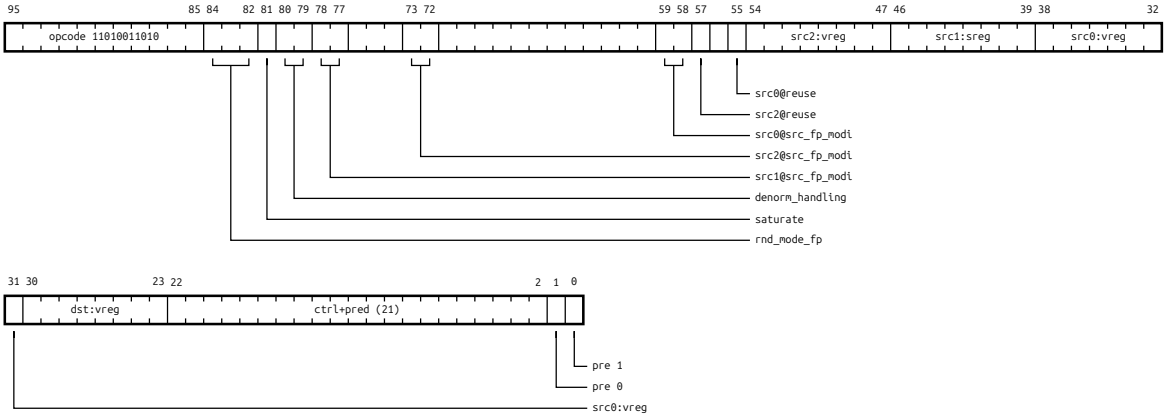
Leaf: ffma__v_vxv

Format:

```
ffma.rnd_mode_fp{.denorm_handling}{.saturate}  dst, src0{.src_fp_modi}{.reuse},
↪  src1{.src_fp_modi}, src2{.src_fp_modi}{.reuse}
```

Operands: dst: vreg; src0: vreg; src1: sreg; src2: vreg

Encoding Diagram: 96b, prefix 10, opcode 11010011010



Notes:

FP32 fused multiply-add: dst = src0 × src1 + src2, 整个表达式按 IEEE 754 binary32 算到无限精度后只按 rnd_mode_fp 舍入一次, 比 fmul + fadd 序列精度高一位。

src0 / src1 / src2 各可挂 src_fp_modi (neg / abs / abs_then_neg), 一条 ffma 表达 -a×b + c、|a|×b - c 这种 ML 常见变体, 省一条 mov / abs。

denorm_handling=keep (默认): IEEE 754 行为, denorm 输入 / 输出 / 中间子积全部按原值保留。

denorm_handling=ftz: input / 中间子积 / output denorm 全部 flush 为 sign-preserving 0。

denorm_handling=fmz: 等同 ftz (输入/输出 denorm 全 flush 为保号 0), 并且任一乘法源 (src0 / src1, 0 测试在输入 flush 之后做) 为 0 时强制中间子积 src0 × src1 = +0 (无视 inf / NaN / sign, 压掉 inf×0=NaN 路径); 加数 src2 不参与 0 测试, 照常相加。fmz 严格强于 ftz, 适合 ML packed fp16 主路径。

saturate=saturate: dst clamp 到 [+0.0, 1.0], NaN → +0.0。

rnd_mode_fp: 4 种 IEEE 754 舍入 (rne 默认 / rtz / rup / rdn)。

fp16 single-half FMA 见 fhfma; packed fp16 / bf16 FMA 见 hma_pk。

fhadd **category**: FP32 compute **predicate_mode**: simt

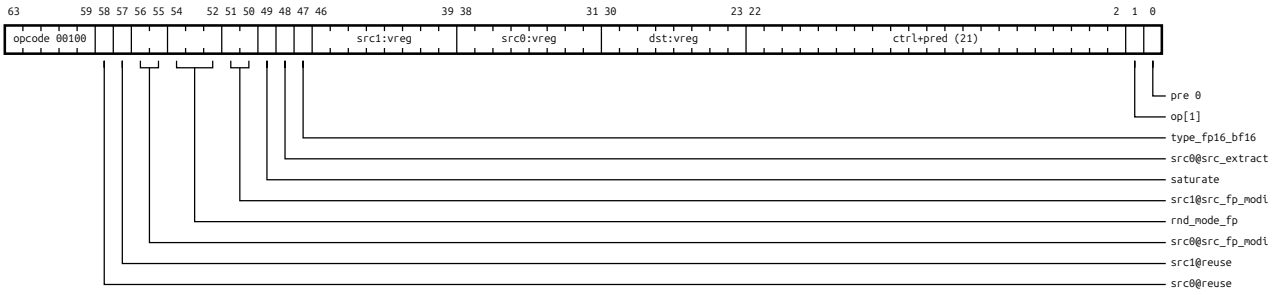
Leaf: fhadd__v_vv

Format:

```
fhadd.rnd_mode_fp{.saturate}.type_fp16_bf16  dst,
↪  src0{.src_fp_modi}{.src_extract}{.reuse}, src1{.src_fp_modi}{.reuse}
```

Operands: dst: vreg; src0: vreg; src1: vreg

Encoding Diagram: 64b, prefix 0, opcode 100100



Notes:

Single-half fp16 / bf16 加法: 从 src0 抽 16-bit half (H0 低半 / H1 高半) 当 fp16/bf16 标量, 加 src1 (32-bit reg 当 fp16/bf16 标量, 默认低 16 位), 按 IEEE 754 单次舍入后写 dst 低 16 位。

src_extract 仅挂 src0 (operand-level modifier): H0 / H1 选半。src1 没有 EXTRACT (默认整 32-bit reg 当低

16 位 fp16/bf16 标量), 这是 fhadd 跟 fhfma (src0 / src1 都有 EXTRACT) 的不对称设计。

type_fp16_bf16 (instruction-level modifier): fp16 (IEEE 754 binary16) / bf16 (Brain Float, 8-bit exp + 7-bit mantissa)。所有 src + dst 共用同一格式。

saturate=saturate: dst clamp 到 [+0.0, 1.0], NaN → +0.0。

rnd_mode_fp: 4 种 IEEE 754 舍入 (rne 默认 / rtz / rup / rdn)。

Packed 路径 (一条指令同时算 H0 + H1 两 half) 见 hadd_pk; FP32 加法见 fadd。

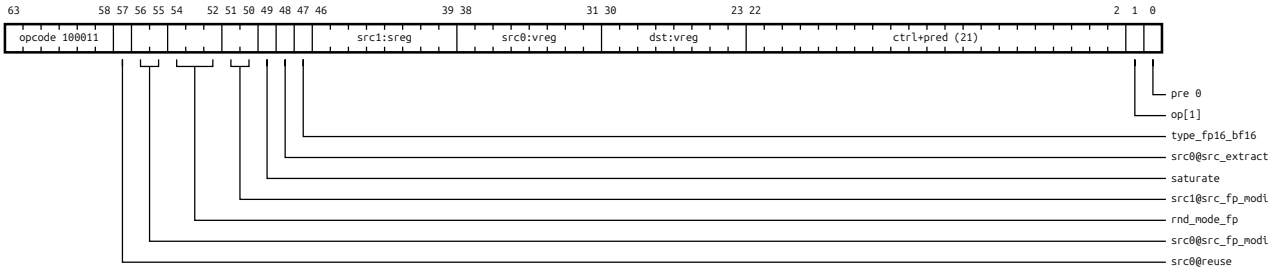
Leaf: fhadd__v_vx

Format:

```
fhadd.rnd_mode_fp{.saturate}.type_fp16_bf16  dst,
↪  src0{.src_fp_modi}{.src_extract}{.reuse}, src1{.src_fp_modi}
```

Operands: dst: vreg; src0: vreg; src1: sreg

Encoding Diagram: 64b, prefix 0, opcode 1100011



Notes:

Single-half fp16 / bf16 加法: 从 src0 抽 16-bit half (H0 低半 / H1 高半) 当 fp16/bf16 标量, 加 src1 (32-bit reg 当 fp16/bf16 标量, 默认低 16 位), 按 IEEE 754 单次舍入后写 dst 低 16 位。

src_extract 仅挂 src0 (operand-level modifier): H0 / H1 选半。src1 没有 EXTRACT (默认整 32-bit reg 当低 16 位 fp16/bf16 标量), 这是 fhadd 跟 fhfma (src0 / src1 都有 EXTRACT) 的不对称设计。

type_fp16_bf16 (instruction-level modifier): fp16 (IEEE 754 binary16) / bf16 (Brain Float, 8-bit exp + 7-bit mantissa)。所有 src + dst 共用同一格式。

saturate=saturate: dst clamp 到 [+0.0, 1.0], NaN → +0.0。

rnd_mode_fp: 4 种 IEEE 754 舍入 (rne 默认 / rtz / rup / rdn)。

Packed 路径 (一条指令同时算 H0 + H1 两 half) 见 hadd_pk; FP32 加法见 fadd。

fhfma **category**: FP32 compute **predicate_mode**: simt

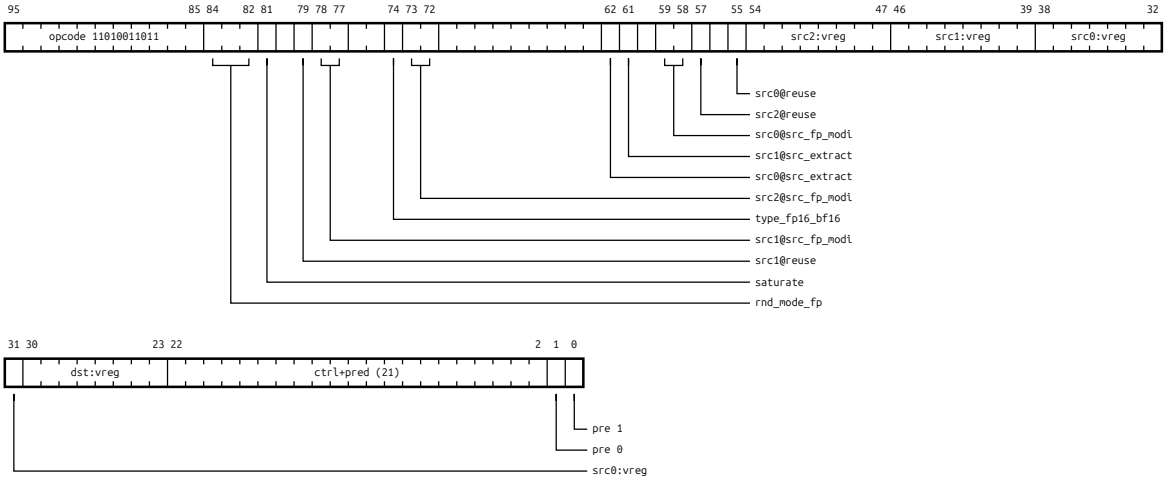
Leaf: fhfma__v_vvv

Format:

```
fhfma.rnd_mode_fp{.saturate}.type_fp16_bf16  dst,
↪  src0{.src_fp_modi}{.src_extract}{.reuse}, src1{.src_fp_modi}{.src_extract}{.reuse},
↪  src2{.src_fp_modi}{.reuse}
```

Operands: dst: vreg; src0: vreg; src1: vreg; src2: vreg

Encoding Diagram: 96b, prefix 10, opcode 11010011011



Notes:

Single-half fp16 / bf16 fused multiply-add: 从 src0 / src1 各抽 1 个 16-bit half (H0 低半 / H1 高半) 当 fp16/bf16 标量, 相乘后加 src2 (32-bit reg 当 fp16/bf16 标量, 默认低 16 位), 按 IEEE 754 单次舍入后写 dst 低 16 位 (高 16 位行为见硬件实现)。

src_extract 必填于 src0 / src1 (operand-level modifier): H0 选低半 / H1 选高半, 用来从一个 32-bit reg 中抽 fp16/bf16 标量参运算。

src2 不挂 src_extract: 整个 32-bit reg 当 fp16/bf16 标量 (默认低 16 位)。这是 fhfma 跟 fhadd (仅 src0 有 EXTRACT) 的不对称设计。

type_fp16_bf16 (instruction-level modifier): fp16 (IEEE 754 binary16) / bf16 (Brain Float, 8-bit exp + 7-bit mantissa)。所有 src + dst 共用同一格式。

saturate=saturate: dst clamp 到 [+0.0, 1.0], NaN → +0.0。

rnd_mode_fp: 4 种 IEEE 754 舍入 (rne 默认 / rtz / rup / rdn)。

Packed 路径 (一条指令同时算 H0 + H1 两 half) 见 hma_pk; FP32 FMA 见 ffma。

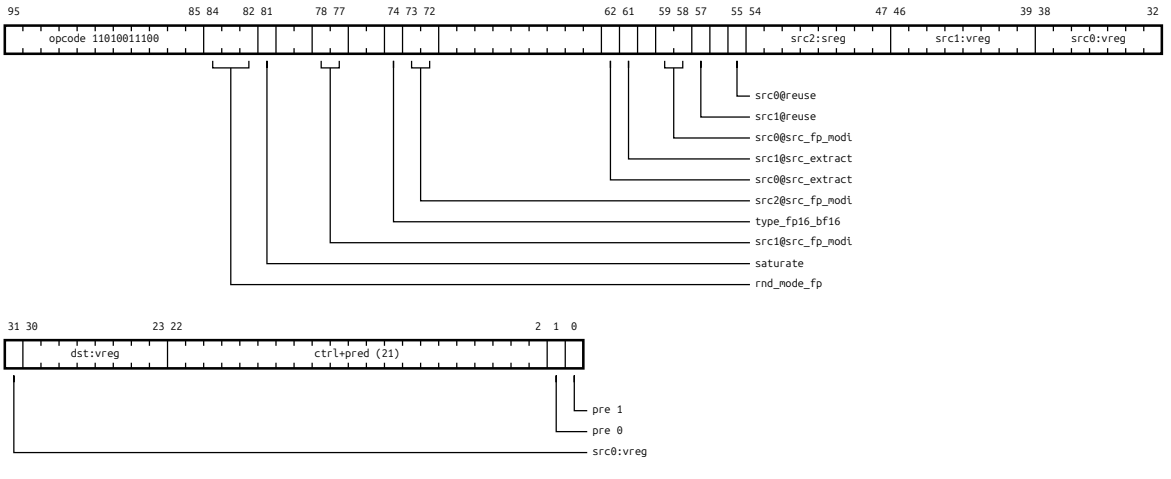
Leaf: fhfma__v_vvx

Format:

```
fhfma.rnd_mode_fp{.saturate}.type_fp16_bf16  dst,
↪  src0{.src_fp_modi}{.src_extract}{.reuse}, src1{.src_fp_modi}{.src_extract}{.reuse},
↪  src2{.src_fp_modi}
```

Operands: dst: vreg; src0: vreg; src1: vreg; src2: sreg

Encoding Diagram: 96b, prefix 10, opcode 11010011100



Notes:

Single-half fp16 / bf16 fused multiply-add: 从 src0 / src1 各抽 1 个 16-bit half (H0 低半 / H1 高半) 当 fp16/bf16 标量, 相乘后加 src2 (32-bit reg 当 fp16/bf16 标量, 默认低 16 位), 按 IEEE 754 单次舍入后写 dst 低 16 位 (高 16 位行为见硬件实现)。

src_extract 必填于 src0 / src1 (operand-level modifier): H0 选低半 / H1 选高半, 用来从一个 32-bit reg 中抽 fp16/bf16 标量参运算。

src2 不挂 src_extract: 整个 32-bit reg 当 fp16/bf16 标量 (默认低 16 位)。这是 fhfma 跟 fhadd (仅 src0 有 EXTRACT) 的不对称设计。

type_fp16_bf16 (instruction-level modifier): fp16 (IEEE 754 binary16) / bf16 (Brain Float, 8-bit exp + 7-bit mantissa)。所有 src + dst 共用同一格式。

saturate=saturate: dst clamp 到 [+0.0, 1.0], NaN → +0.0。

rnd_mode_fp: 4 种 IEEE 754 舍入 (rne 默认 / rtz / rup / rdn)。

Packed 路径 (一条指令同时算 H0 + H1 两 half) 见 hma_pk; FP32 FMA 见 ffma。

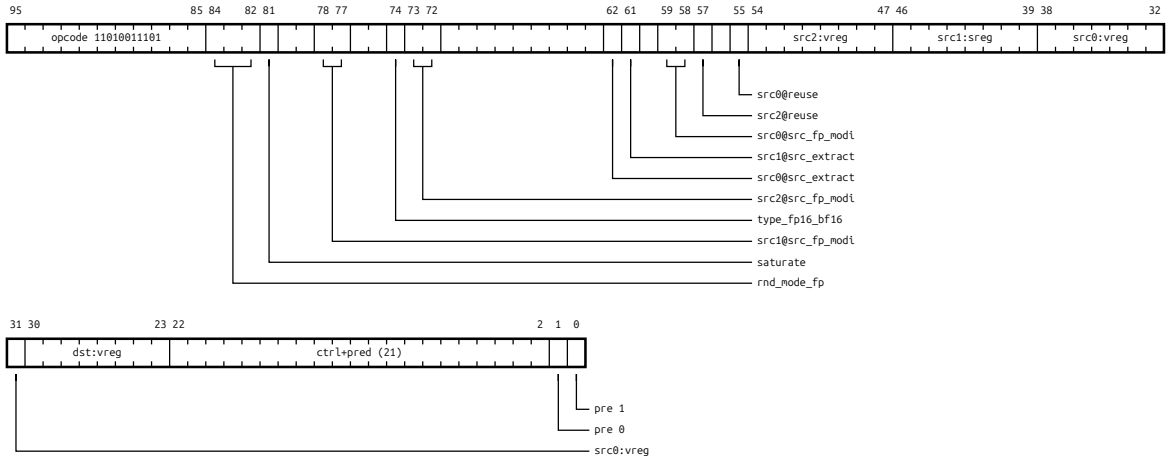
Leaf: fhfma__v_vxv

Format:

```
fhfma.rnd_mode_fp{.saturate}.type_fp16_bf16  dst,
↪  src0{.src_fp_modi}{.src_extract}{.reuse}, src1{.src_fp_modi}{.src_extract},
↪  src2{.src_fp_modi}{.reuse}
```

Operands: dst: vreg; src0: vreg; src1: sreg; src2: vreg

Encoding Diagram: 96b, prefix 10, opcode 11010011101



Notes:

Single-half fp16 / bf16 fused multiply-add: 从 src0 / src1 各抽 1 个 16-bit half (H0 低半 / H1 高半) 当 fp16/bf16 标量, 相乘后加 src2 (32-bit reg 当 fp16/bf16 标量, 默认低 16 位), 按 IEEE 754 单次舍入后写 dst 低 16 位 (高 16 位行为见硬件实现)。

src_extract 必填于 src0 / src1 (operand-level modifier): H0 选低半 / H1 选高半, 用来从一个 32-bit reg 中抽 fp16/bf16 标量参运算。

src2 不挂 src_extract: 整个 32-bit reg 当 fp16/bf16 标量 (默认低 16 位)。这是 fhfma 跟 fhadd (仅 src0 有 EXTRACT) 的不对称设计。

type_fp16_bf16 (instruction-level modifier): fp16 (IEEE 754 binary16) / bf16 (Brain Float, 8-bit exp + 7-bit mantissa)。所有 src + dst 共用同一格式。

saturate=saturate: dst clamp 到 [+0.0, 1.0], NaN → +0.0。

rnd_mode_fp: 4 种 IEEE 754 舍入 (rne 默认 / rtz / rup / rdn)。

Packed 路径 (一条指令同时算 H0 + H1 两 half) 见 hma_pk; FP32 FMA 见 ffma。

fmnmx **category**: FP32 compute **predicate_mode**: simt

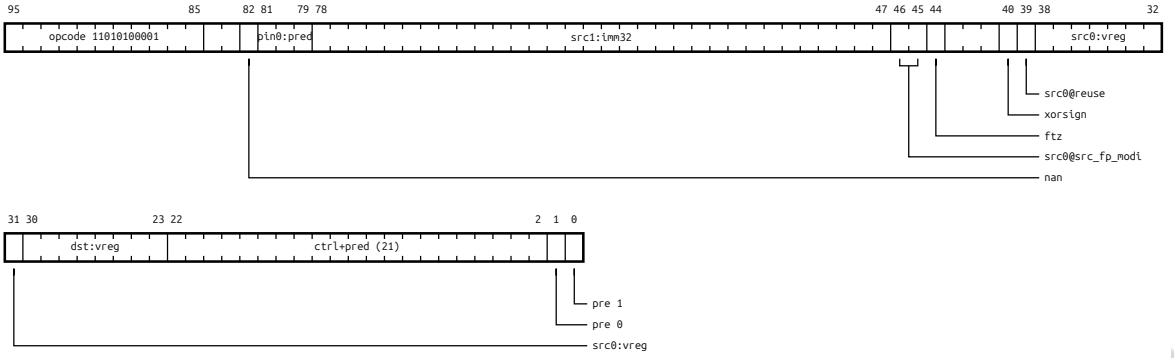
Leaf: fmnmx__v_vi

Format:

```
fmnmx{.nan}.ftz{.xorsign}  dst, src0{.src_fp_modi}{.reuse}, src1, pin0
```

Operands: dst: vreg; src0: vreg; src1: imm32u; pin0: pred

Encoding Diagram: 96b, prefix 10, opcode 11010100001



Notes:

FP32 per-lane min / max: pin0 决定取 min 还是 max — pin0=true 时 dst = min(src0, src1), pin0=false 时 dst = max(src0, src1)。pin0=PT 退化为永远 min, pin0=!PT 退化为永远 max。

src0 / src1 各可挂 src_fp_modi (neg / abs / abs_then_neg), 在比较前先做符号 / 绝对值变换, 例如 |a| vs |b| 取最值。

nan=disable (默认): 任一 src 是 NaN 时按 IEEE 754 默认行为返 NaN (NaN 传染)。

nan=nan: 跳过 NaN 输入, 仅在两 src 都 NaN 时 dst = canonical NaN; 一边是 NaN 时 dst 取另一边的 finite 值 (ML 训练常用, 抑制 NaN 传染)。

ftz=ftz: input / output denorm flush 为 sign-preserving 0。

xorsign=xorsign: dst 的符号位 = sign(src0) xor sign(src1) (而不是 min/max 选中的那一边的符号), 用于 fast |a| × sign(b) 这种符号控制路径。

+0 vs -0 边界: 按 IEEE 754, +0 与 -0 数值相等; 默认 dst 取选中 src 的符号, xorsign=xorsign 时按上一条覆盖。

仅 FP32。Packed fp16 / bf16 min / max 见 hmnmx_pk; 不暴露 single-half fp16 / bf16 比较 (硬件 FP 比较单元共用 32-bit 比较器, 没有专门 single-half 指令)。

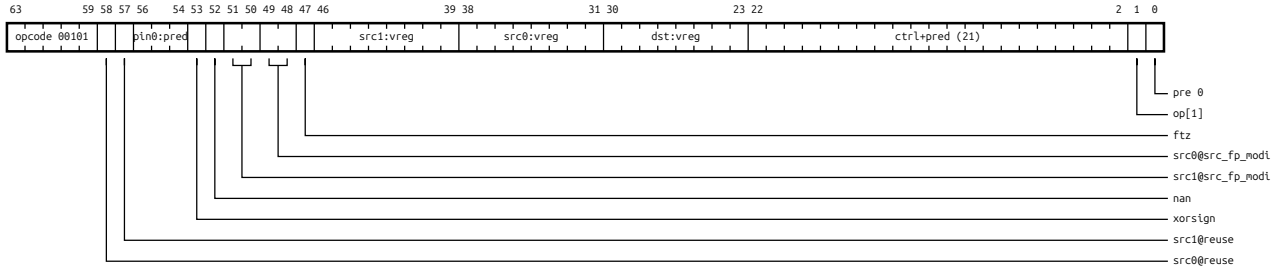
Leaf: fmnmx__v_vv

Format:

```
fmnmx{.nan}.ftz{.xorsign}  dst, src0{.src_fp_modi}{.reuse}, src1{.src_fp_modi}{.reuse},
↪  pin0
```

Operands: dst: vreg; src0: vreg; src1: vreg; pin0: pred

Encoding Diagram: 64b, prefix 0, opcode 100101



Notes:

FP32 per-lane min / max: pin0 决定取 min 还是 max — pin0=true 时 dst = min(src0, src1), pin0=false 时 dst = max(src0, src1)。pin0=PT 退化为永远 min, pin0=!PT 退化为永远 max。

src0 / src1 各可挂 src_fp_modi (neg / abs / abs_then_neg), 在比较前先做符号 / 绝对值变换, 例如 |a| vs |b| 取最值。

nan=disable (默认): 任一 src 是 NaN 时按 IEEE 754 默认行为返 NaN (NaN 传染)。

nan=nan: 跳过 NaN 输入, 仅在两 src 都 NaN 时 dst = canonical NaN; 一边是 NaN 时 dst 取另一边的 finite 值 (ML 训练常用, 抑制 NaN 传染)。

ftz=ftz: input / output denorm flush 为 sign-preserving 0。

xorsign=xorsign: dst 的符号位 = sign(src0) xor sign(src1) (而不是 min/max 选中的那一边的符号), 用于 fast |a| × sign(b) 这种符号控制路径。

+0 vs -0 边界: 按 IEEE 754, +0 与 -0 数值相等; 默认 dst 取选中 src 的符号, xorsign=xorsign 时按上一条覆盖。

仅 FP32。Packed fp16 / bf16 min / max 见 hmnmx_pk; 不暴露 single-half fp16 / bf16 比较 (硬件 FP 比较单元共用 32-bit 比较器, 没有专门 single-half 指令)。

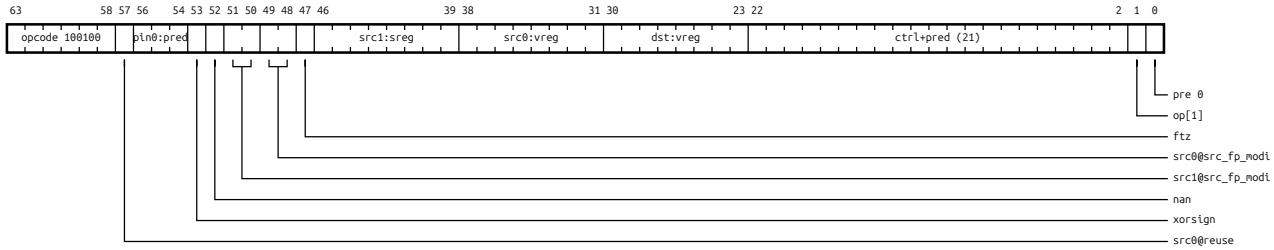
Leaf: fmnmx__v_vx

Format:

```
fmnmx{.nan}.ftz{.xorsign}  dst, src0{.src_fp_modi}{.reuse}, src1{.src_fp_modi}, pin0
```

Operands: dst: vreg; src0: vreg; src1: sreg; pin0: pred

Encoding Diagram: 64b, prefix 0, opcode 1100100



Notes:

FP32 per-lane min / max: pin0 决定取 min 还是 max — pin0=true 时 dst = min(src0, src1), pin0=false 时 dst = max(src0, src1)。pin0=PT 退化为永远 min, pin0=!PT 退化为永远 max。

src0 / src1 各可挂 src_fp_modi (neg / abs / abs_then_neg), 在比较前先做符号 / 绝对值变换, 例如 |a| vs |b| 取最值。

nan=disable (默认): 任一 src 是 NaN 时按 IEEE 754 默认行为返 NaN (NaN 传染)。

nan=nan: 跳过 NaN 输入, 仅在两 src 都 NaN 时 dst = canonical NaN; 一边是 NaN 时 dst 取另一边的 finite 值 (ML 训练常用, 抑制 NaN 传染)。

ftz=ftz: input / output denorm flush 为 sign-preserving 0。

xorsign=xorsign: dst 的符号位 = sign(src0) xor sign(src1) (而不是 min/max 选中的那一边的符号), 用于 fast |a| × sign(b) 这种符号控制路径。

+0 vs -0 边界: 按 IEEE 754, +0 与 -0 数值相等; 默认 dst 取选中 src 的符号, xorsign=xorsign 时按上一条覆盖。

仅 FP32。Packed fp16 / bf16 min / max 见 hmnmx_pk; 不暴露 single-half fp16 / bf16 比较 (硬件 FP 比较单元共用 32-bit 比较器, 没有专门 single-half 指令)。

fmnmx3 **category**: FP32 compute **predicate_mode**: simt

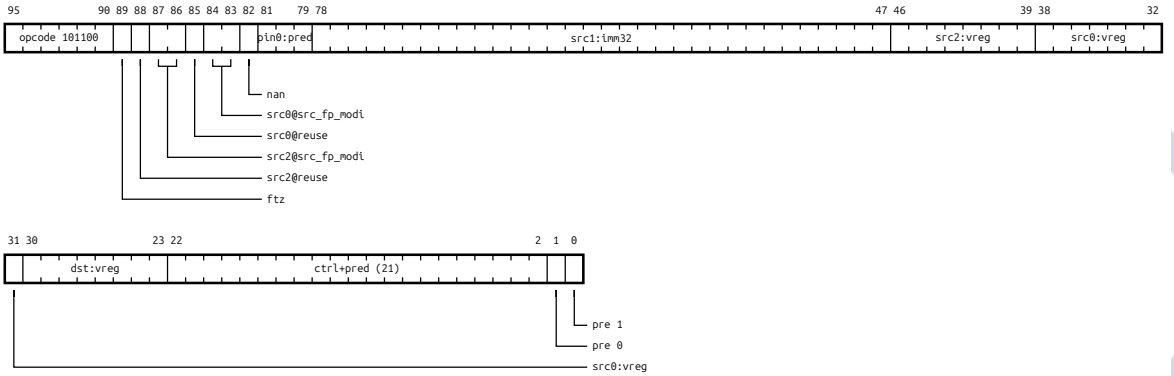
Leaf: fmnmx3__v_viv

Format:

```
fmnmx3{.nan}.ftz  dst, src0{.src_fp_modi}{.reuse}, src1, src2{.src_fp_modi}{.reuse}, pin0
```

Operands: dst: vreg; src0: vreg; src1: imm32u; src2: vreg; pin0: pred

Encoding Diagram: 96b, prefix 10, opcode 101100



Notes:

三入 FP per-lane min / max: pin0 决定取 min 还是 max — pin0=true 时 dst = min(src0, src1, src2), pin0=false 时 dst = max(src0, src1, src2)。pin0=PT 退化为永远 min, pin0=!PT 退化为永远 max。

src0 / src1 / src2 各可挂 src_fp_modi (neg / abs / abs_then_neg), 在比较前先做符号 / 绝对值变换。

nan=disable (默认): 任一 src 是 NaN 时按 IEEE 754 默认返 NaN。

nan=nan: 跳过 NaN 输入, 仅在所有 src 都是 NaN 时 dst = canonical NaN; 其余情况取非 NaN src 中的最值。

ftz=ftz: input / output denorm flush 为 sign-preserving 0。

典型用例: triple-clamp (一条指令实现 clamp(x, lo, hi))、sorting network、ML 三目激活函数 lowering。

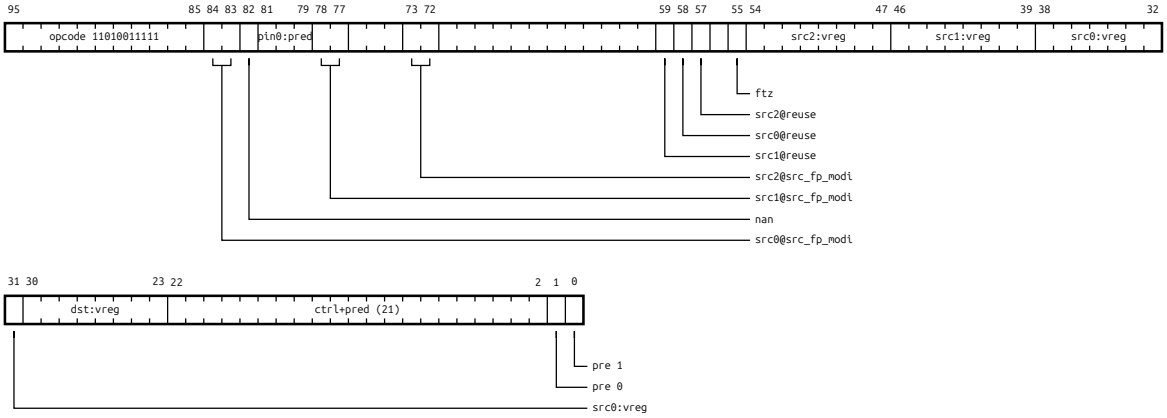
Leaf: fmnmx3__v_vvv

Format:

```
fmnmx3{.nan}.ftz  dst, src0{.src_fp_modi}{.reuse}, src1{.src_fp_modi}{.reuse},
↪  src2{.src_fp_modi}{.reuse}, pin0
```

Operands: dst: vreg; src0: vreg; src1: vreg; src2: vreg; pin0: pred

Encoding Diagram: 96b, prefix 10, opcode 11010011111



Notes:

三入 FP per-lane min / max: pin0 决定取 min 还是 max — pin0=true 时 dst = min(src0, src1, src2), pin0=false 时 dst = max(src0, src1, src2)。pin0=PT 退化为永远 min, pin0=!PT 退化为永远 max。

src0 / src1 / src2 各可挂 src_fp_modi (neg / abs / abs_then_neg), 在比较前先做符号 / 绝对值变换。

nan=disable (默认): 任一 src 是 NaN 时按 IEEE 754 默认返 NaN。

nan=nan: 跳过 NaN 输入, 仅在所有 src 都是 NaN 时 dst = canonical NaN; 其余情况取非 NaN src 中的最值。

ftz=ftz: input / output denorm flush 为 sign-preserving 0。

典型用例: triple-clamp (一条指令实现 clamp(x, lo, hi))、sorting network、ML 三目激活函数 lowering。

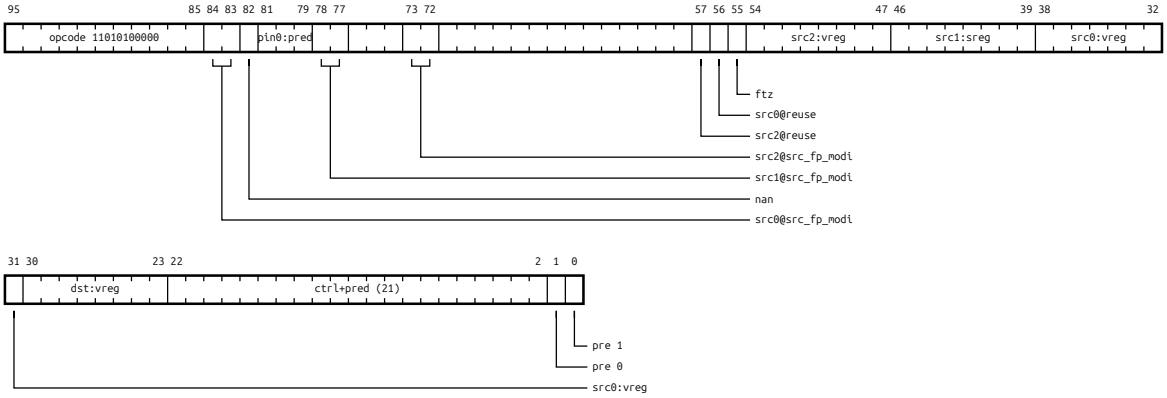
Leaf: fmnmx3__v_vxv

Format:

```
fmnmx3{.nan}.ftz  dst, src0{.src_fp_modi}{.reuse}, src1{.src_fp_modi},
↪  src2{.src_fp_modi}{.reuse}, pin0
```

Operands: dst: vreg; src0: vreg; src1: sreg; src2: vreg; pin0: pred

Encoding Diagram: 96b, prefix 10, opcode 11010100000



Notes:

三入 FP per-lane min / max: pin0 决定取 min 还是 max — pin0=true 时 dst = min(src0, src1, src2), pin0=false 时 dst = max(src0, src1, src2)。pin0=PT 退化为永远 min, pin0=!PT 退化为永远 max。

src0 / src1 / src2 各可挂 src_fp_modi (neg / abs / abs_then_neg), 在比较前先做符号 / 绝对值变换。

nan=disable (默认): 任一 src 是 NaN 时按 IEEE 754 默认返 NaN。

nan=nan: 跳过 NaN 输入, 仅在所有 src 都是 NaN 时 dst = canonical NaN; 其余情况取非 NaN src 中的最值。

ftz=ftz: input / output denorm flush 为 sign-preserving 0。

典型用例: triple-clamp (一条指令实现 clamp(x, lo, hi))、sorting network、ML 三目激活函数 lowering。

fmul **category**: FP32 compute **predicate_mode**: simt

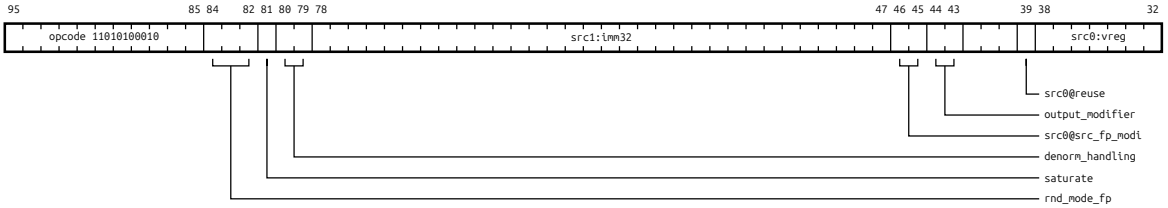
Leaf: fmul__v_vi

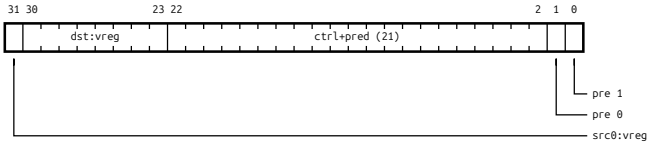
Format:

```
fmul.rnd_mode_fp{.denorm_handling}{.saturate}.output_modifier  dst,
↪  src0{.src_fp_modi}{.reuse}, src1
```

Operands: dst: vreg; src0: vreg; src1: imm32u

Encoding Diagram: 96b, prefix 10, opcode 11010100010





Notes:

FP32 单精度浮点乘法: dst = src0 × src1, 整 warp 32 个 active thread 各自独立计算, 结果按 IEEE 754 binary32 格式。

src0 / src1 各可挂 src_fp_modi (neg / abs / abs_then_neg), 实现 -a × b、|a| × b 这种符号 / 绝对值控制变体。

denorm_handling=keep (默认): IEEE 754 行为, denorm 输入 / 输出按原值。

denorm_handling=ftz: input / output denorm flush 为 sign-preserving 0。

denorm_handling=fmz: 等同 ftz (denorm flush 为保号 0), 并且任一 src (0 测试在输入 flush 之后做) 为 0 时强制乘积为 +0 (无视 inf / NaN / sign, 压掉 inf × 0 = NaN 路径)。fmz 严格强于 ftz。

saturate=saturate: dst clamp 到 [+0.0, 1.0], NaN → +0.0。

rnd_mode_fp: 4 种 IEEE 754 舍入 (rne 默认 / rtz / rup / rdn)。

output_modifier: dst 写出前预乘 scale, 4 valid = d2 (÷2) / noscale (×1, 默认) / m2 (×2) / m4 (×4), 见 modifier_defs.output_modifier (NV 同类字段是 7 值 D8/D4/D2/noscale/M2/M4/M8, gpidl 收敛为围绕 ×1 对称的 4 值)。

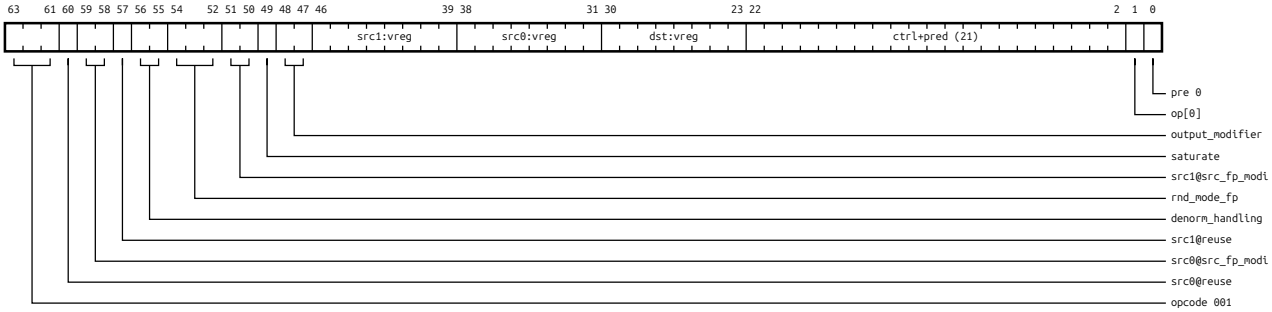
Leaf: fmul__v_vv

Format:

```
fmul.rnd_mode_fp{.denorm_handling}{.saturate}.output_modifier  dst,
↪  src0{.src_fp_modi}{.reuse}, src1{.src_fp_modi}{.reuse}
```

Operands: dst: vreg; src0: vreg; src1: vreg

Encoding Diagram: 64b, prefix 0, opcode 0001



Notes:

FP32 单精度浮点乘法: dst = src0 × src1, 整 warp 32 个 active thread 各自独立计算, 结果按 IEEE 754 binary32 格式。

src0 / src1 各可挂 src_fp_modi (neg / abs / abs_then_neg), 实现 -a × b、|a| × b 这种符号 / 绝对值控制变体。

denorm_handling=keep (默认): IEEE 754 行为, denorm 输入 / 输出按原值。

denorm_handling=ftz: input / output denorm flush 为 sign-preserving 0。

denorm_handling=fmz: 等同 ftz (denorm flush 为保号 0), 并且任一 src (0 测试在输入 flush 之后做) 为 0 时强制乘积为 +0 (无视 inf / NaN / sign, 压掉 inf × 0 = NaN 路径)。fmz 严格强于 ftz。

saturate=saturate: dst clamp 到 [+0.0, 1.0], NaN → +0.0。

rnd_mode_fp: 4 种 IEEE 754 舍入 (rne 默认 / rtz / rup / rdn)。

output_modifier: dst 写出前预乘 scale, 4 valid = d2 (÷2) / noscale (×1, 默认) / m2 (×2) / m4 (×4), 见 modifier_defs.output_modifier (NV 同类字段是 7 值 D8/D4/D2/noscale/M2/M4/M8, gpidl 收敛为围绕 ×1 对称的 4 值)。

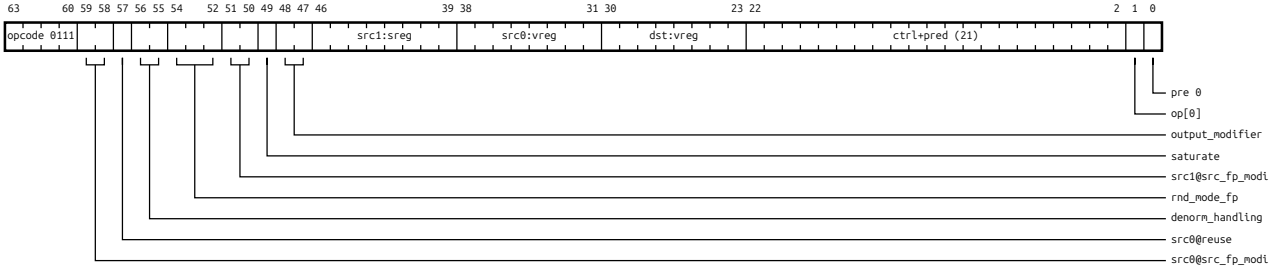
Leaf: fmul__v_vx

Format:

```
fmul.rnd_mode_fp{.denorm_handling}{.saturate}.output_modifier  dst,
↪  src0{.src_fp_modi}{.reuse}, src1{.src_fp_modi}
```

Operands: dst: vreg; src0: vreg; src1: sreg

Encoding Diagram: 64b, prefix 0, opcode 00111



Notes:

FP32 单精度浮点乘法: dst = src0 × src1, 整 warp 32 个 active thread 各自独立计算, 结果按 IEEE 754 binary32 格式。

src0 / src1 各可挂 src_fp_modi (neg / abs / abs_then_neg), 实现 -a × b、|a| × b 这种符号 / 绝对值控制变体。

denorm_handling=keep (默认): IEEE 754 行为, denorm 输入 / 输出按原值。

denorm_handling=ftz: input / output denorm flush 为 sign-preserving 0。

denorm_handling=fmz: 等同 ftz (denorm flush 为保号 0), 并且任一 src (0 测试在输入 flush 之后做) 为 0 时强制乘积为 +0 (无视 inf / NaN / sign, 压掉 inf × 0 = NaN 路径)。fmz 严格强于 ftz。

saturate=saturate: dst clamp 到 [+0.0, 1.0], NaN → +0.0。

rnd_mode_fp: 4 种 IEEE 754 舍入 (rne 默认 / rtz / rup / rdn)。

output_modifier: dst 写出前预乘 scale, 4 valid = d2 (÷2) / noscale (×1, 默认) / m2 (×2) / m4 (×4), 见 modifier_defs.output_modifier (NV 同类字段是 7 值 D8/D4/D2/noscale/M2/M4/M8, gpidl 收敛为围绕 ×1 对称的 4 值)。

fsel **category**: FP32 compute **predicate_mode**: simt

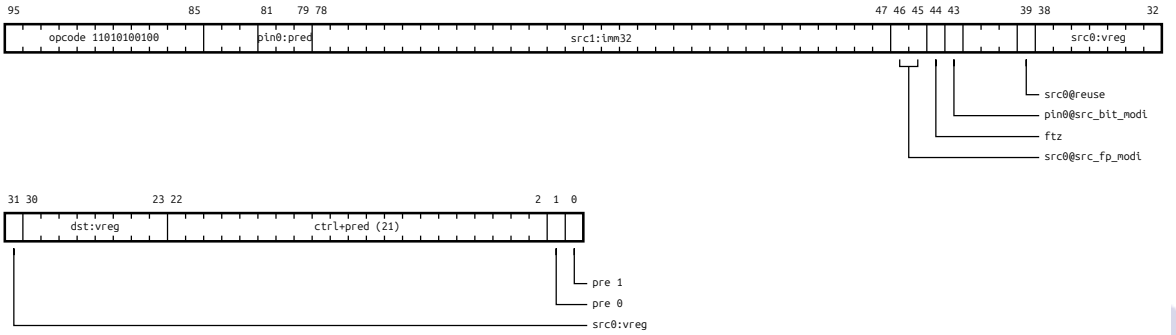
Leaf: fsel__v_vip

Format:

```
fsel.ftz  dst, src0{.src_fp_modi}{.reuse}, src1, pin0{.src_bit_modi}
```

Operands: dst: vreg; src0: vreg; src1: imm32u; pin0: pred

Encoding Diagram: 96b, prefix 10, opcode 11010100100



Notes:

FP per-lane conditional select: pin0=true 时 dst = src0, pin0=false 时 dst = src1。pin0 是 per-thread predicate, 每 lane 独立选。

等价于 fsetp + 整数 sel 的 2 指令序列, 但 fsel 单条 issue, 用在 conditional move 模式 (例如 abs / clamp / 三目运算 lowering)。

src0 / src1 各可挂 src_fp_modi (neg / abs / abs_then_neg); pin0 可挂 src_bit_modi (id 不取反 / inv 取反), 一条 fsel 同时支持 dst = (!P) ? a : b 的 inverted 模式而无需独立的 not 指令。

ftz=ftz: dst 写出前 denorm flush 为 sign-preserving 0 (输入已是合法 fp 值, 无需 flush input)。

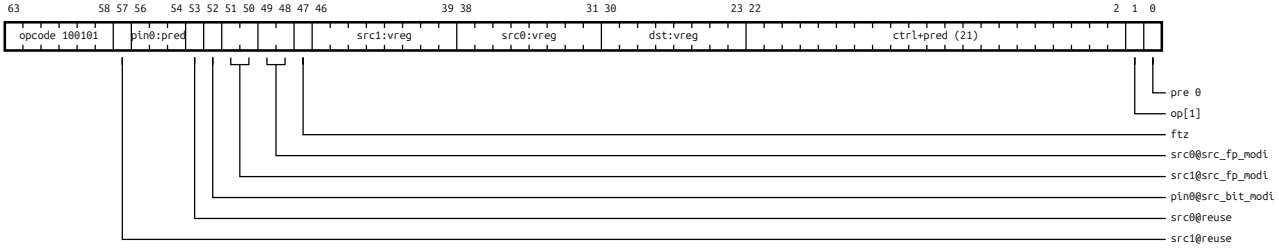
Leaf: fsel__v_vvp

Format:

```
fsel.ftz  dst, src0{.src_fp_modi}{.reuse}, src1{.src_fp_modi}{.reuse},
↪  pin0{.src_bit_modi}
```

Operands: dst: vreg; src0: vreg; src1: vreg; pin0: pred

Encoding Diagram: 64b, prefix 0, opcode 1100101



Notes:

FP per-lane conditional select: pin0=true 时 dst = src0, pin0=false 时 dst = src1。pin0 是 per-thread predicate, 每 lane 独立选。

等价于 fsetp + 整数 sel 的 2 指令序列, 但 fsel 单条 issue, 用在 conditional move 模式 (例如 abs / clamp / 三目运算 lowering)。

src0 / src1 各可挂 src_fp_modi (neg / abs / abs_then_neg); pin0 可挂 src_bit_modi (id 不取反 / inv 取反), 一条 fsel 同时支持 dst = (!P) ? a : b 的 inverted 模式而无需独立的 not 指令。

ftz=ftz: dst 写出前 denorm flush 为 sign-preserving 0 (输入已是合法 fp 值, 无需 flush input)。

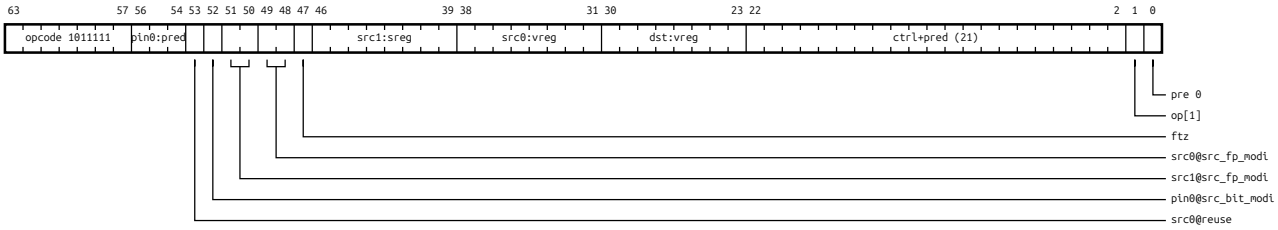
Leaf: fsel__v_vxp

Format:

```
fsel.ftz  dst, src0{.src_fp_modi}{.reuse}, src1{.src_fp_modi}, pin0{.src_bit_modi}
```

Operands: dst: vreg; src0: vreg; src1: sreg; pin0: pred

Encoding Diagram: 64b, prefix 0, opcode 11011111



Notes:

FP per-lane conditional select: pin0=true 时 dst = src0, pin0=false 时 dst = src1。pin0 是 per-thread predicate, 每 lane 独立选。

等价于 fsetp + 整数 sel 的 2 指令序列, 但 fsel 单条 issue, 用在 conditional move 模式 (例如 abs / clamp / 三目运算 lowering)。

src0 / src1 各可挂 src_fp_modi (neg / abs / abs_then_neg); pin0 可挂 src_bit_modi (id 不取反 / inv 取反), 一条 fsel 同时支持 dst = (!P) ? a : b 的 inverted 模式而无需独立的 not 指令。

ftz=ftz: dst 写出前 denorm flush 为 sign-preserving 0 (输入已是合法 fp 值, 无需 flush input)。

fset **category**: FP32 compute **predicate_mode**: simt

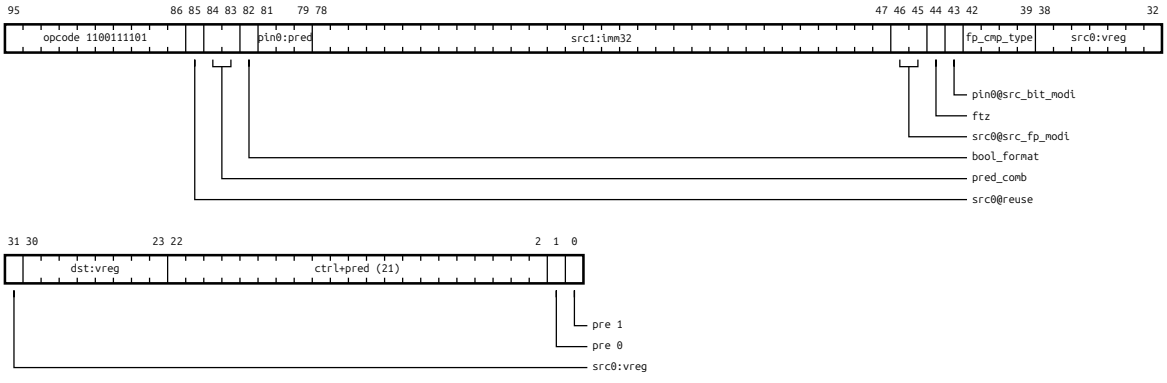
Leaf: fset__v_vip

Format:

```
fset.fp_cmp_type.pred_comb.bool_format.ftz  dst, src0{.src_fp_modi}{.reuse}, src1,
↪  pin0{.src_bit_modi}
```

Operands: dst: vreg; src0: vreg; src1: imm32u; pin0: pred

Encoding Diagram: 96b, prefix 10, opcode 1100111101



Notes:

FP 比较 → vreg: 把 (src0 cmp src1) 的 boolean 结果按 pred_comb 跟 pin0 组合, 再按 bool_format 写入 vreg。整体语义 dst = ((src0 cmp src1) pred_comb pin0) ? 真值: 0。

fp_cmp_type 选 FP 比较算子, 共 16 个, 按 numeric / unordered 两组 (跟 IEEE 754 严格对应):

- numeric (NaN 输入返 false): f (always false) / lt / eq / le / gt / ne / ge / num (任一 NaN→false, 否则 true 即 '两 src 都是数值')。
- unordered (NaN 输入返 true): nan (任一 NaN→true) / ltu / equ / leu / gtu / neu / geu / t (always true)。

pred_comb 把 (src0 cmp src1) 跟 pin0 按布尔运算组合: and / or / xor。pin0=PT + and 退化为 cmp 结果直传 (即不组合)。pin0 通常带 src_bit_modi (id / inv) 用于嵌套谓词 (外层 pin0 控内层 cmp)。

bool_format 控 dst 写入格式:

- bm (boolean mask): true → 0xFFFFFFFF, false → 0x00000000。可直接当 mask 跟其他 vreg 做 and / or / xor。
- bf (boolean float): true → 1.0f (0x3F800000), false → 0.0f (0x00000000)。可直接参 FP 算术 (例如 boolean × value 实现 conditional-zero)。

ftz=ftz: input denorm flush 为 sign-preserving 0 进比较, 避免 denorm 比较语义边界 case。

FP 比较 → predicate (写 P 寄存器) 见 fsetp。

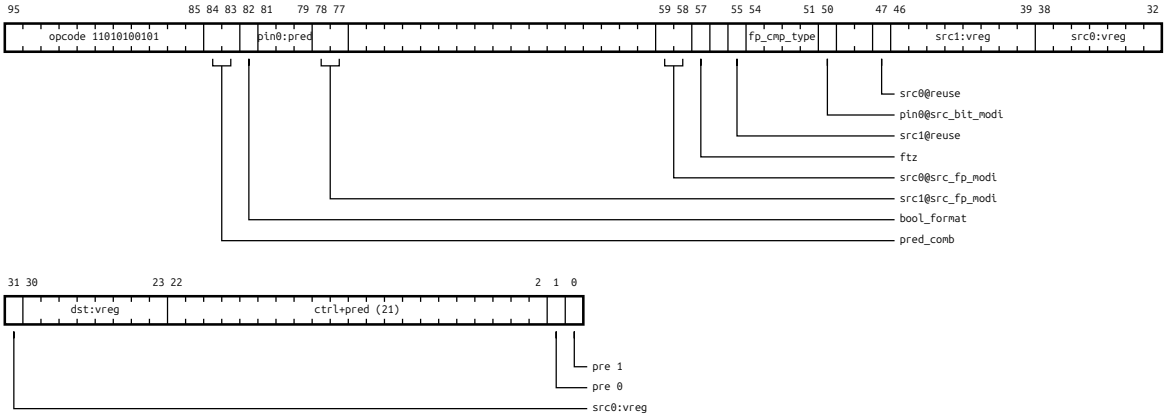
Leaf: fset__v_vvp

Format:

```
fset.fp_cmp_type.pred_comb.bool_format.ftz  dst, src0{.src_fp_modi}{.reuse},
↪  src1{.src_fp_modi}{.reuse}, pin0{.src_bit_modi}
```

Operands: dst: vreg; src0: vreg; src1: vreg; pin0: pred

Encoding Diagram: 96b, prefix 10, opcode 11010100101



Notes:

FP 比较 → vreg: 把 (src0 cmp src1) 的 boolean 结果按 pred_comb 跟 pin0 组合, 再按 bool_format 写入 vreg。整体语义 dst = ((src0 cmp src1) pred_comb pin0) ? 真值: 0。

fp_cmp_type 选 FP 比较算子, 共 16 个, 按 numeric / unordered 两组 (跟 IEEE 754 严格对应):

- numeric (NaN 输入返 false): f (always false) / lt / eq / le / gt / ne / ge / num (任一 NaN→false, 否则 true 即 '两 src 都是数值')。
- unordered (NaN 输入返 true): nan (任一 NaN→true) / ltu / equ / leu / gtu / neu / geu / t (always true)。

pred_comb 把 (src0 cmp src1) 跟 pin0 按布尔运算组合: and / or / xor。pin0=PT + and 退化为 cmp 结果直传 (即不组合)。pin0 通常带 src_bit_modi (id / inv) 用于嵌套谓词 (外层 pin0 控内层 cmp)。

bool_format 控 dst 写入格式:

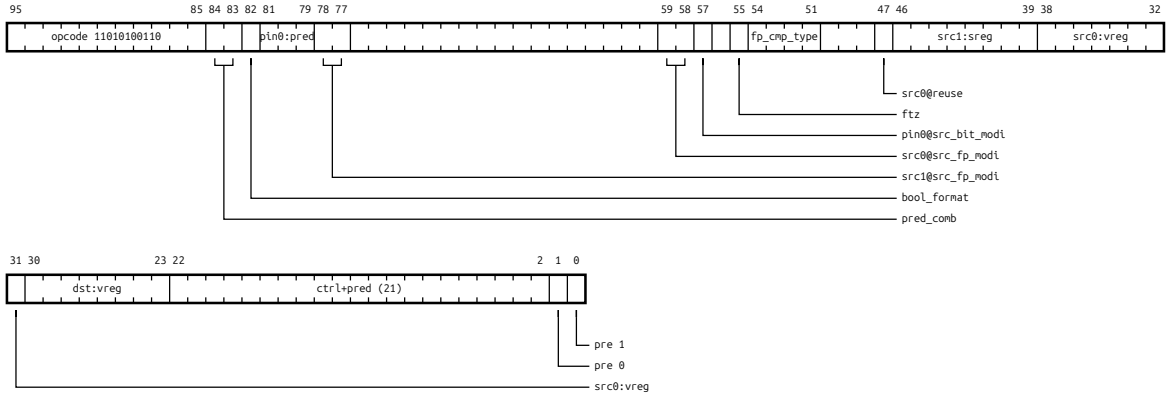
- bm (boolean mask): true → 0xFFFFFFFF, false → 0x00000000。可直接当 mask 跟其他 vreg 做 and / or / xor。
- bf (boolean float): true → 1.0f (0x3F800000), false → 0.0f (0x00000000)。可直接参 FP 算术 (例如 boolean × value 实现 conditional-zero)。

ftz=ftz: input denorm flush 为 sign-preserving 0 进比较, 避免 denorm 比较语义边界 case。
FP 比较 → predicate (写 P 寄存器) 见 fsetp。

Leaf: fset__v_vxp
Format:

```
fset.fp_cmp_type.pred_comb.bool_format.ftz  dst, src0{.src_fp_modi}{.reuse},
↪  src1{.src_fp_modi}, pin0{.src_bit_modi}
```

Operands: dst: vreg; src0: vreg; src1: sreg; pin0: pred
Encoding Diagram: 96b, prefix 10, opcode 11010100110



Notes:
FP 比较 → vreg: 把 (src0 cmp src1) 的 boolean 结果按 pred_comb 跟 pin0 组合, 再按 bool_format 写入 vreg。整体语义 dst = ((src0 cmp src1) pred_comb pin0) ? 真值: 0。

fp_cmp_type 选 FP 比较算子, 共 16 个, 按 numeric / unordered 两组 (跟 IEEE 754 严格对应):

- numeric (NaN 输入返 false): f (always false) / lt / eq / le / gt / ne / ge / num (任一 NaN→false, 否则 true 即 ‘两 src 都是数值’)。
- unordered (NaN 输入返 true): nan (任一 NaN→true) / ltu / equ / leu / gtu / neu / geu / t (always true)。

pred_comb 把 (src0 cmp src1) 跟 pin0 按布尔运算组合: and / or / xor。pin0=PT + and 退化为 cmp 结果直传 (即不组合)。pin0 通常带 src_bit_modi (id / inv) 用于嵌套谓词 (外层 pin0 控内层 cmp)。

bool_format 控 dst 写入格式:

- bm (boolean mask): true → 0xFFFFFFFF, false → 0x00000000。可直接当 mask 跟其他 vreg 做 and / or / xor。
- bf (boolean float): true → 1.0f (0x3F800000), false → 0.0f (0x00000000)。可直接参 FP 算术 (例如 boolean × value 实现 conditional-zero)。

ftz=ftz: input denorm flush 为 sign-preserving 0 进比较, 避免 denorm 比较语义边界 case。
FP 比较 → predicate (写 P 寄存器) 见 fsetp。

fsetp **category**: FP32 compute **predicate_mode**: simt

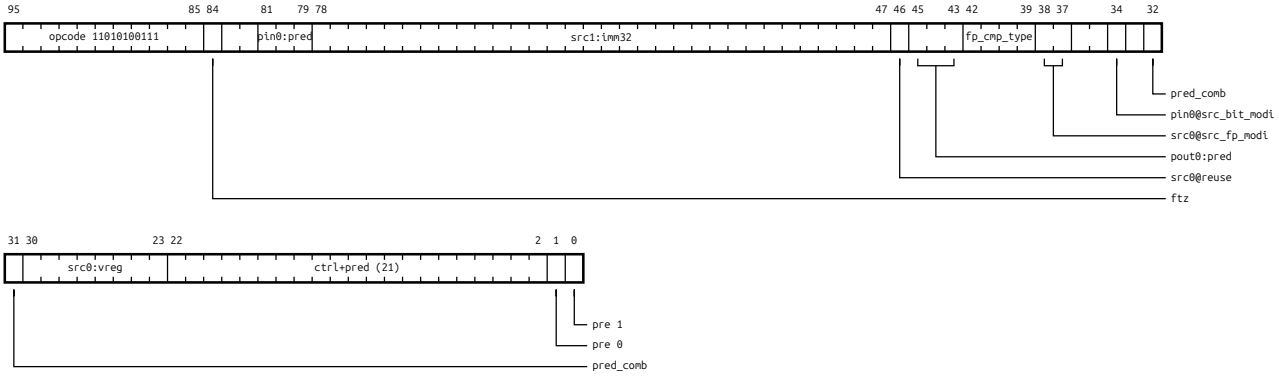
Leaf: fsetp__p_vip

Format:

```
fsetp.fp_cmp_type.pred_comb.ftz  pout0, src0{.src_fp_modi}{.reuse}, src1,
↪  pin0{.src_bit_modi}
```

Operands: pout0: pred; src0: vreg; src1: imm32u; pin0: pred

Encoding Diagram: 96b, prefix 10, opcode 11010100111



Notes:

FP 比较 → predicate: 把 (src0 cmp src1) 的 boolean 结果按 pred_comb 跟 pin0 组合, 写入 1 或 2 个 predicate (pout0 / pout1 可选)。

整体语义: cmp = (src0 cmp src1); pout0 = pred_comb(cmp, pin0); pout1 = pred_comb(!cmp, pin0) (仅 pp_* 指令形态输出)。注意 pout1 不是!pout0 — 是用同样的 pred_comb 跟 pin0 组合 cmp 的取反: pout0 = pin0 ∧ cmp 是 ‘in true’ 路径, pout1 = pin0 ∧ !cmp 是 ‘in false’ 路径, 用于嵌套谓词 (外层 pin0 控内层 cmp 的真假分支)。

fp_cmp_type 同 fset, 16 个 numeric / unordered 比较算子 (f / lt / eq / le / gt / ne / ge / num / nan / ltu / equ / leu / gtu / neu / geu / t)。

pred_comb (and / or / xor) 把 cmp 跟 pin0 组合。pin0=PT + and 退化为 cmp 结果直传。pin0 可挂 src_bit_modi (id / inv) 实现 cmp ∧ !pin0 等组合而无需额外 not 指令。

ftz=ftz: input denorm flush 为 sign-preserving 0 进比较。

p_* 指令形态单输出 pout0; pp_* 指令形态双输出 pout0 + pout1, 一条 fsetp 同时拿到 ‘in true’ 跟 ‘in false’ 两个分支谓词, 后续 @pout0 / @pout1 控两条分支。

FP 比较 → vreg (写 boolean mask / boolean float 到 vreg) 见 fset。

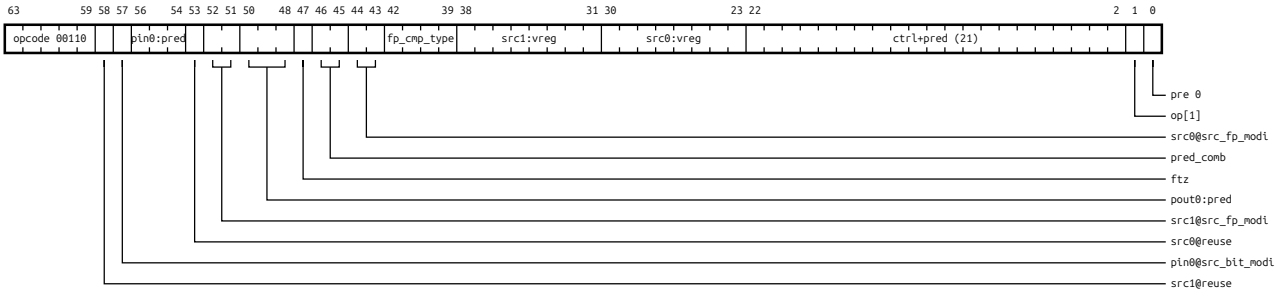
Leaf: fsetp__p_vvp

Format:

```
fsetp.fp_cmp_type.pred_comb.ftz  pout0, src0{.src_fp_modi}{.reuse},
↪  src1{.src_fp_modi}{.reuse}, pin0{.src_bit_modi}
```

Operands: pout0: pred; src0: vreg; src1: vreg; pin0: pred

Encoding Diagram: 64b, prefix 0, opcode 100110



Notes:

FP 比较 → predicate: 把 (src0 cmp src1) 的 boolean 结果按 pred_comb 跟 pin0 组合, 写入 1 或 2 个 predicate (pout0 / pout1 可选)。

整体语义: cmp = (src0 cmp src1); pout0 = pred_comb(cmp, pin0); pout1 = pred_comb(!cmp, pin0) (仅 pp_* 指令形态输出)。注意 pout1 不是!pout0 — 是用同样的 pred_comb 跟 pin0 组合 cmp 的取反: pout0 = pin0 ∧ cmp 是 'in true' 路径, pout1 = pin0 ∧ !cmp 是 'in false' 路径, 用于嵌套谓词 (外层 pin0 控内层 cmp 的真假分支)。

fp_cmp_type 同 fset, 16 个 numeric / unordered 比较算子 (f / lt / eq / le / gt / ne / ge / num / nan / ltu / equ / leu / gtu / neu / geu / t)。

pred_comb (and / or / xor) 把 cmp 跟 pin0 组合。pin0=PT + and 退化为 cmp 结果直传。pin0 可挂 src_bit_modi (id / inv) 实现 cmp ∧ !pin0 等组合而无需额外 not 指令。

ftz=ftz: input denorm flush 为 sign-preserving 0 进比较。

p_* 指令形态单输出 pout0; pp_* 指令形态双输出 pout0 + pout1, 一条 fsetp 同时拿到 'in true' 跟 'in false' 两个分支谓词, 后续 @pout0 / @pout1 控两条分支。

FP 比较 → vreg (写 boolean mask / boolean float 到 vreg) 见 fset。

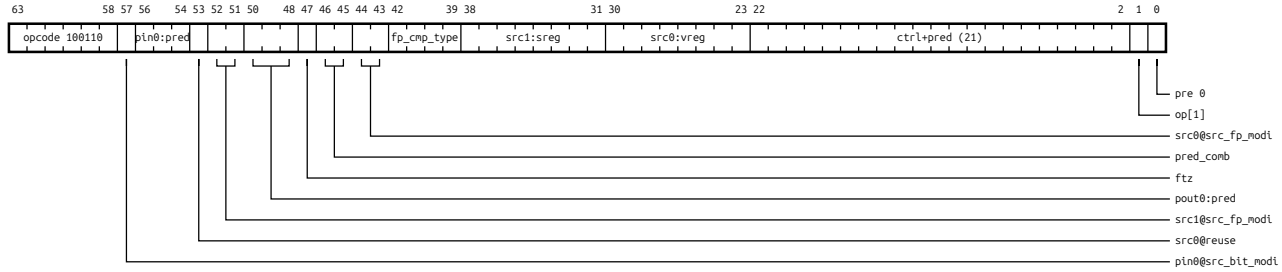
Leaf: fsetp__p_vxp

Format:

```
fsetp.fp_cmp_type.pred_comb.ftz  pout0, src0{.src_fp_modi}{.reuse}, src1{.src_fp_modi},
↪  pin0{.src_bit_modi}
```

Operands: pout0: pred; src0: vreg; src1: sreg; pin0: pred

Encoding Diagram: 64b, prefix 0, opcode 1100110



Notes:

FP 比较 → predicate: 把 (src0 cmp src1) 的 boolean 结果按 pred_comb 跟 pin0 组合, 写入 1 或 2 个 predicate (pout0 / pout1 可选)。

整体语义: cmp = (src0 cmp src1); pout0 = pred_comb(cmp, pin0); pout1 = pred_comb(!cmp, pin0) (仅 pp_* 指令形态输出)。注意 pout1 不是!pout0 — 是用同样的 pred_comb 跟 pin0 组合 cmp 的取反: pout0 = pin0 ∧ cmp 是 'in true' 路径, pout1 = pin0 ∧ !cmp 是 'in false' 路径, 用于嵌套谓词 (外层 pin0 控内层 cmp 的真假分支)。

fp_cmp_type 同 fset, 16 个 numeric / unordered 比较算子 (f / lt / eq / le / gt / ne / ge / num / nan / ltu / equ / leu / gtu / neu / geu / t)。

pred_comb (and / or / xor) 把 cmp 跟 pin0 组合。pin0=PT + and 退化为 cmp 结果直传。pin0 可挂 src_bit_modi (id / inv) 实现 cmp ∧ !pin0 等组合而无需额外 not 指令。

ftz=ftz: input denorm flush 为 sign-preserving 0 进比较。

p_* 指令形态单输出 pout0; pp_* 指令形态双输出 pout0 + pout1, 一条 fsetp 同时拿到 'in true' 跟 'in false' 两个分支谓词, 后续 @pout0 / @pout1 控两条分支。

FP 比较 → vreg (写 boolean mask / boolean float 到 vreg) 见 fset。

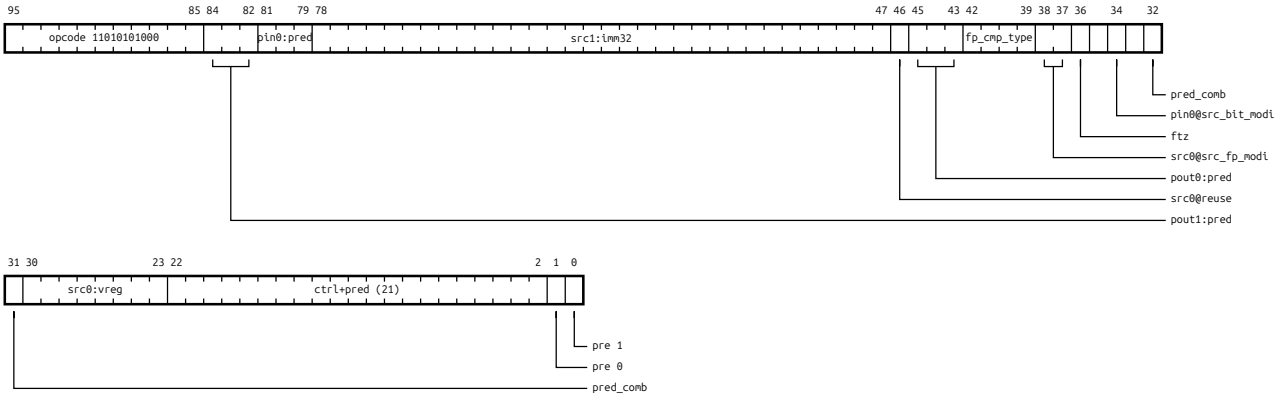
Leaf: fsetp__pp_vip

Format:

```
fsetp.fp_cmp_type.pred_comb.ftz  pout0, pout1, src0{.src_fp_modi}{.reuse}, src1,
↪  pin0{.src_bit_modi}
```

Operands: pout0: pred; pout1: pred; src0: vreg; src1: imm32u; pin0: pred

Encoding Diagram: 96b, prefix 10, opcode 11010101000



Notes:

FP 比较 → predicate: 把 (src0 cmp src1) 的 boolean 结果按 pred_comb 跟 pin0 组合, 写入 1 或 2 个 predicate (pout0 / pout1 可选)。

整体语义: cmp = (src0 cmp src1); pout0 = pred_comb(cmp, pin0); pout1 = pred_comb(!cmp, pin0) (仅 pp_* 指令形态输出)。注意 pout1 不是!pout0 — 是用同样的 pred_comb 跟 pin0 组合 cmp 的取反: pout0 = pin0 ∧ cmp 是 ‘in true’ 路径, pout1 = pin0 ∧ !cmp 是 ‘in false’ 路径, 用于嵌套谓词 (外层 pin0 控内层 cmp 的真假分支)。

fp_cmp_type 同 fset, 16 个 numeric / unordered 比较算子 (f / lt / eq / le / gt / ne / ge / num / nan / ltu / equ / leu / gtu / neu / geu / t)。

pred_comb (and / or / xor) 把 cmp 跟 pin0 组合。pin0=PT + and 退化为 cmp 结果直传。pin0 可挂 src_bit_modi (id / inv) 实现 cmp ∧ !pin0 等组合而无需额外 not 指令。

ftz=ftz: input denorm flush 为 sign-preserving 0 进比较。

p_* 指令形态单输出 pout0; pp_* 指令形态双输出 pout0 + pout1, 一条 fsetp 同时拿到 ‘in true’ 跟 ‘in false’ 两个分支谓词, 后续 @pout0 / @pout1 控两条分支。

FP 比较 → vreg (写 boolean mask / boolean float 到 vreg) 见 fset。

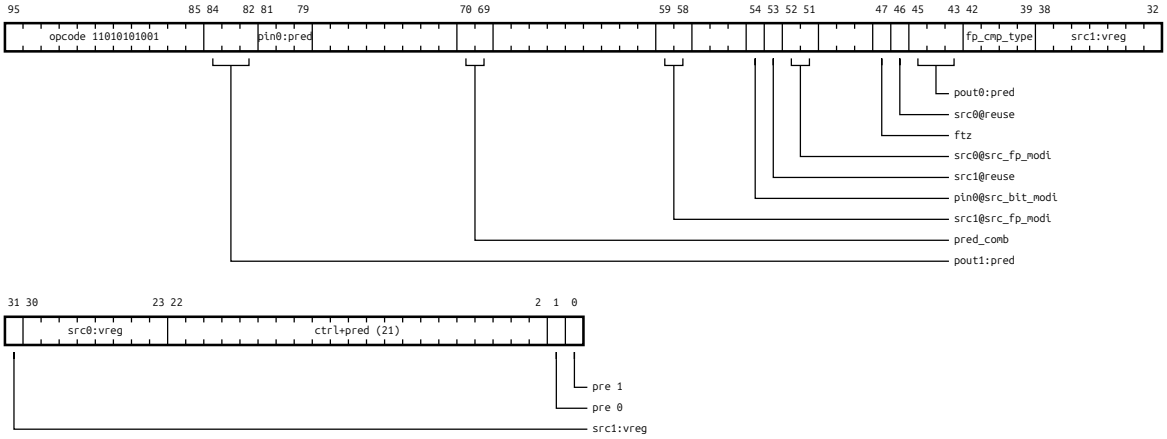
Leaf: fsetp__pp_vvp

Format:

```
fsetp.fp_cmp_type.pred_comb.ftz  pout0, pout1, src0{.src_fp_modi}{.reuse},
↪  src1{.src_fp_modi}{.reuse}, pin0{.src_bit_modi}
```

Operands: pout0: pred; pout1: pred; src0: vreg; src1: vreg; pin0: pred

Encoding Diagram: 96b, prefix 10, opcode 11010101001



Notes:

FP 比较 → predicate: 把 (src0 cmp src1) 的 boolean 结果按 pred_comb 跟 pin0 组合, 写入 1 或 2 个 predicate (pout0 / pout1 可选)。

整体语义: cmp = (src0 cmp src1); pout0 = pred_comb(cmp, pin0); pout1 = pred_comb(!cmp, pin0) (仅 pp_* 指令形态输出)。注意 pout1 不是!pout0 — 是用同样的 pred_comb 跟 pin0 组合 cmp 的取反: pout0 = pin0 ∧ cmp 是 'in true' 路径, pout1 = pin0 ∧ !cmp 是 'in false' 路径, 用于嵌套谓词 (外层 pin0 控内层 cmp 的真假分支)。

fp_cmp_type 同 fset, 16 个 numeric / unordered 比较算子 (f / lt / eq / le / gt / ne / ge / num / nan / ltu / equ / leu / gtu / neu / geu / t)。

pred_comb (and / or / xor) 把 cmp 跟 pin0 组合。pin0=PT + and 退化为 cmp 结果直传。pin0 可挂 src_bit_modi (id / inv) 实现 cmp ∧ !pin0 等组合而无需额外 not 指令。

ftz=ftz: input denorm flush 为 sign-preserving 0 进比较。

p_* 指令形态单输出 pout0; pp_* 指令形态双输出 pout0 + pout1, 一条 fsetp 同时拿到 'in true' 跟 'in false' 两个分支谓词, 后续 @pout0 / @pout1 控两条分支。

FP 比较 → vreg (写 boolean mask / boolean float 到 vreg) 见 fset。

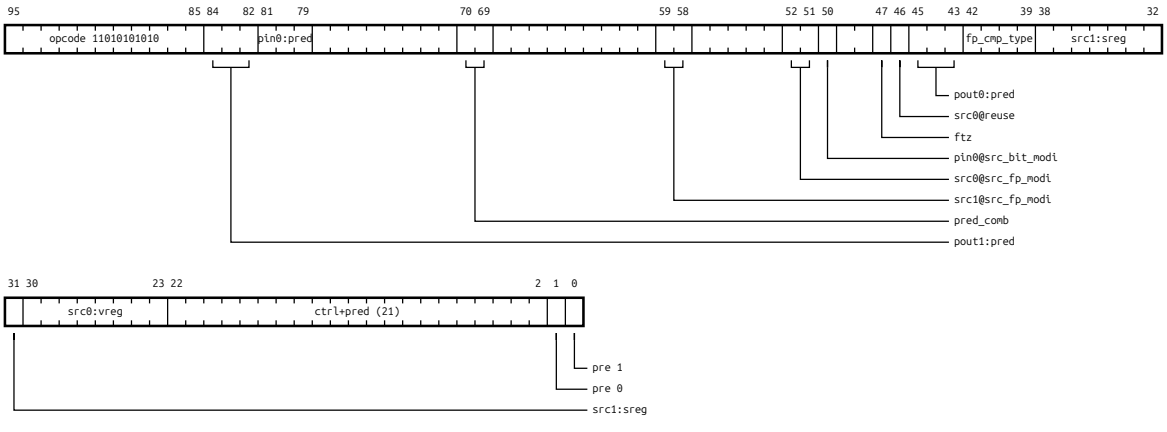
Leaf: fsetp__pp_vxp

Format:

```
fsetp.fp_cmp_type.pred_comb.ftz  pout0, pout1, src0{.src_fp_modi}{.reuse},
↪  src1{.src_fp_modi}, pin0{.src_bit_modi}
```

Operands: pout0: pred; pout1: pred; src0: vreg; src1: sreg; pin0: pred

Encoding Diagram: 96b, prefix 10, opcode 11010101010



Notes:

FP 比较 → predicate: 把 (src0 cmp src1) 的 boolean 结果按 pred_comb 跟 pin0 组合, 写入 1 或 2 个 predicate

(pout0 / pout1 可选)。

整体语义: cmp = (src0 cmp src1); pout0 = pred_comb(cmp, pin0); pout1 = pred_comb(!cmp, pin0) (仅 pp_* 指令形态输出)。注意 pout1 不是!pout0 — 是用同样的 pred_comb 跟 pin0 组合 cmp 的取反: pout0 = pin0 ∧ cmp 是 'in true' 路径, pout1 = pin0 ∧ !cmp 是 'in false' 路径, 用于嵌套谓词 (外层 pin0 控内层 cmp 的真假分支)。

fp_cmp_type 同 fset, 16 个 numeric / unordered 比较算子 (f / lt / eq / le / gt / ne / ge / num / nan / ltu / equ / leu / gtu / neu / geu / t)。

pred_comb (and / or / xor) 把 cmp 跟 pin0 组合。pin0=PT + and 退化为 cmp 结果直传。pin0 可挂 src_bit_modi (id / inv) 实现 cmp ∧ !pin0 等组合而无需额外 not 指令。

ftz=ftz: input denorm flush 为 sign-preserving 0 进比较。

p_* 指令形态单输出 pout0; pp_* 指令形态双输出 pout0 + pout1, 一条 fsetp 同时拿到 'in true' 跟 'in false' 两个分支谓词, 后续 @pout0 / @pout1 控两条分支。

FP 比较 → vreg (写 boolean mask / boolean float 到 vreg) 见 fset。

mufu **category**: FP32 compute **predicate_mode**: simt

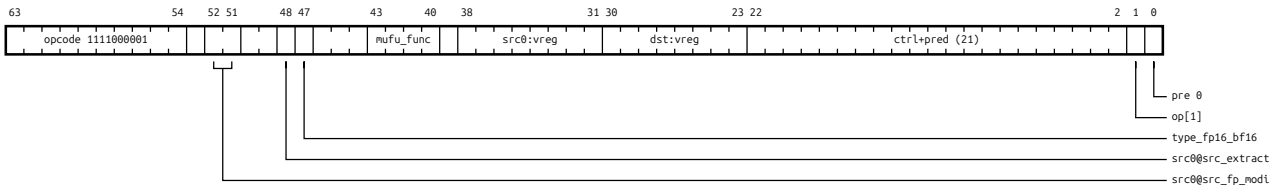
Leaf: mufu__fp16.v_v

Format:

```
mufu.mufu_func.type_fp16_bf16  dst, src0{.src_fp_modi}{.src_extract}
```

Operands: dst: vreg; src0: vreg

Encoding Diagram: 64b, prefix 0, opcode 11111000001



Notes:

Multi-Function Unit: 硬件 transcendental / special function 单元, 单 src 计算 cos / sin / ex2 (2^x) / lg2 (log2) / rcp (1/x) / rsq (1/√x) / sqrt (√x) / tanh / gelu / silu 等。具体函数由 mufu_func modifier 选, 一条 mnemonic 收敛多函数。

FP32 路径 (fp32.* 指令形态): src / dst 都是 32-bit FP32, mufu_func 全集 (含 fp64_rcp_seed / fp64_rsq_seed — FP64 倒数 / 倒数平方根的 seed 值高位部分, 后接 Newton-Raphson 软件迭代得完整 FP64 结果; 这两个取值属于 fp64 扩展)。

fp16 / bf16 路径 (fp16.* 指令形态): src 是 32-bit reg + src_extract operand-modifier 选 H0 / H1 抽 fp16/bf16 单值, dst 写 32-bit reg 低 16 位; mufu_func 可用 cos / sin / ex2 / lg2 / rcp / rsq / sqrt / tanh / gelu / silu (fp64_rcp_seed / fp64_rsq_seed seed 路径不暴露)。

type_fp16_bf16 是指令形态局部 modifier (仅 fp16.* 指令形态在该指令形态的 modifiers 列表引用, fp32.* 指令形态不读这一位 bit), 取值 fp16 (IEEE 754 binary16) / bf16 (Brain Float)。

精度契约与特殊值行为见 mufu_func modifier 的 meaning: mufu 是近似单元, 不承诺 correctly-rounded; 每个函数带最大误差界 (数值待硬件定案), 特殊值 (±0 / ±Inf / NaN / 定义域外输入) 按对应精确函数的 IEEE 754 规则、所有实现逐 bit 一致。

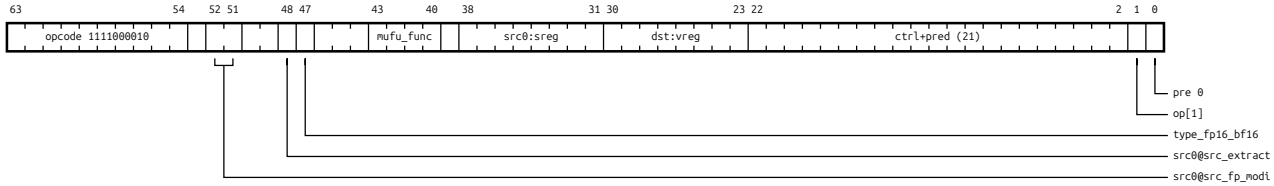
Leaf: mufu__fp16.v_x

Format:

```
mufu.mufu_func.type_fp16_bf16  dst, src0{.src_fp_modi}{.src_extract}
```

Operands: dst: vreg; src0: sreg

Encoding Diagram: 64b, prefix 0, opcode 11111000010



Notes:

Multi-Function Unit: 硬件 transcendental / special function 单元, 单 src 计算 cos / sin / ex2 (2^x) / lg2 (log2) / rcp (1/x) / rsq (1/√x) / sqrt (√x) / tanh / gelu / silu 等。具体函数由 mufu_func modifier 选, 一条 mnemonic 收敛多函数。

FP32 路径 (fp32.* 指令形态): src / dst 都是 32-bit FP32, mufu_func 全集 (含 fp64_rcp_seed / fp64_rsq_seed — FP64 倒数 / 倒数平方根的 seed 值高位部分, 后接 Newton-Raphson 软件迭代得完整 FP64 结果; 这两个取值属于 fp64 扩展)。

fp16 / bf16 路径 (fp16.* 指令形态): src 是 32-bit reg + src_extract operand-modifier 选 H0 / H1 抽 fp16/bf16 单值, dst 写 32-bit reg 低 16 位; mufu_func 可用 cos / sin / ex2 / lg2 / rcp / rsq / sqrt / tanh / gelu / silu (fp64_rcp_seed / fp64_rsq_seed seed 路径不暴露)。

type_fp16_bf16 是指令形态局部 modifier (仅 fp16.* 指令形态在该指令形态的 modifiers 列表引用, fp32.* 指令形态不读这一位 bit), 取值 fp16 (IEEE 754 binary16) / bf16 (Brain Float)。

精度契约与特殊值行为见 mufu_func modifier 的 meaning: mufu 是近似单元, 不承诺 correctly-rounded; 每个函数带最大误差界 (数值待硬件定案), 特殊值 (±0 / ±Inf / NaN / 定义域外输入) 按对应精确函数的 IEEE 754 规则、所有实现逐 bit 一致。

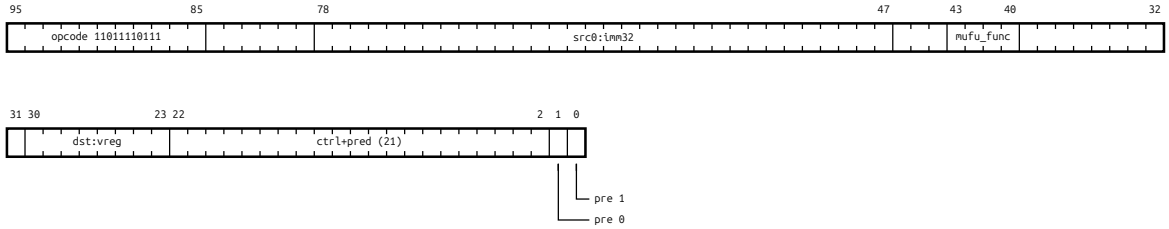
Leaf: mufu__fp32.v_i

Format:

```
mufu.mufu_func  dst, src0
```

Operands: dst: vreg; src0: imm32u

Encoding Diagram: 96b, prefix 10, opcode 11011110111



Notes:

Multi-Function Unit: 硬件 transcendental / special function 单元, 单 src 计算 cos / sin / ex2 (2^x) / lg2 (log2) / rcp (1/x) / rsq (1/√x) / sqrt (√x) / tanh / gelu / silu 等。具体函数由 mufu_func modifier 选, 一条 mnemonic 收敛多函数。

FP32 路径 (fp32.* 指令形态): src / dst 都是 32-bit FP32, mufu_func 全集 (含 fp64_rcp_seed / fp64_rsq_seed — FP64 倒数 / 倒数平方根的 seed 值高位部分, 后接 Newton-Raphson 软件迭代得完整 FP64 结果; 这两个取值属于 fp64 扩展)。

fp16 / bf16 路径 (fp16.* 指令形态): src 是 32-bit reg + src_extract operand-modifier 选 H0 / H1 抽 fp16/bf16

单值, dst 写 32-bit reg 低 16 位; mufu_func 可用 cos / sin / ex2 / lg2 / rcp / rsq / sqrt / tanh / gelu / silu (fp64_rcp_seed / fp64_rsq_seed seed 路径不暴露)。

type_fp16_bf16 是指令形态局部 modifier (仅 fp16.* 指令形态在该指令形态的 modifiers 列表引用, fp32.* 指令形态不读这一位 bit), 取值 fp16 (IEEE 754 binary16) / bf16 (Brain Float)。

精度契约与特殊值行为见 mufu_func modifier 的 meaning: mufu 是近似单元, 不承诺 correctly-rounded; 每个函数带最大误差界 (数值待硬件定案), 特殊值 (±0 / ±Inf / NaN / 定义域外输入) 按对应精确函数的 IEEE 754 规则、所有实现逐 bit 一致。

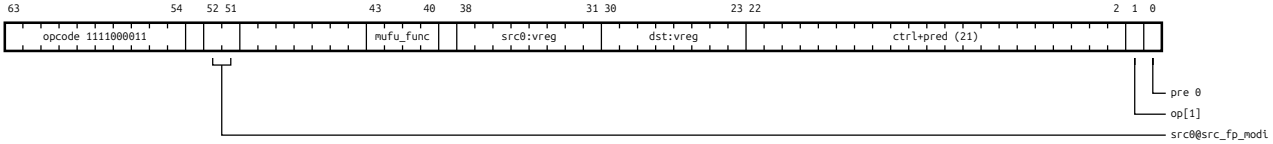
Leaf: mufu__fp32.v_v

Format:

```
mufu.mufu_func  dst, src0{.src_fp_modi}
```

Operands: dst: vreg; src0: vreg

Encoding Diagram: 64b, prefix 0, opcode 11111000011



Notes:

Multi-Function Unit: 硬件 transcendental / special function 单元, 单 src 计算 cos / sin / ex2 (2^x) / lg2 (log2) / rcp (1/x) / rsq (1/√x) / sqrt (√x) / tanh / gelu / silu 等。具体函数由 mufu_func modifier 选, 一条 mnemonic 收敛多函数。

FP32 路径 (fp32.* 指令形态): src / dst 都是 32-bit FP32, mufu_func 全集 (含 fp64_rcp_seed / fp64_rsq_seed — FP64 倒数 / 倒数平方根的 seed 值高位部分, 后接 Newton-Raphson 软件迭代得完整 FP64 结果; 这两个取值属于 fp64 扩展)。

fp16 / bf16 路径 (fp16.* 指令形态): src 是 32-bit reg + src_extract operand-modifier 选 H0 / H1 抽 fp16/bf16 单值, dst 写 32-bit reg 低 16 位; mufu_func 可用 cos / sin / ex2 / lg2 / rcp / rsq / sqrt / tanh / gelu / silu (fp64_rcp_seed / fp64_rsq_seed seed 路径不暴露)。

type_fp16_bf16 是指令形态局部 modifier (仅 fp16.* 指令形态在该指令形态的 modifiers 列表引用, fp32.* 指令形态不读这一位 bit), 取值 fp16 (IEEE 754 binary16) / bf16 (Brain Float)。

精度契约与特殊值行为见 mufu_func modifier 的 meaning: mufu 是近似单元, 不承诺 correctly-rounded; 每个函数带最大误差界 (数值待硬件定案), 特殊值 (±0 / ±Inf / NaN / 定义域外输入) 按对应精确函数的 IEEE 754 规则、所有实现逐 bit 一致。

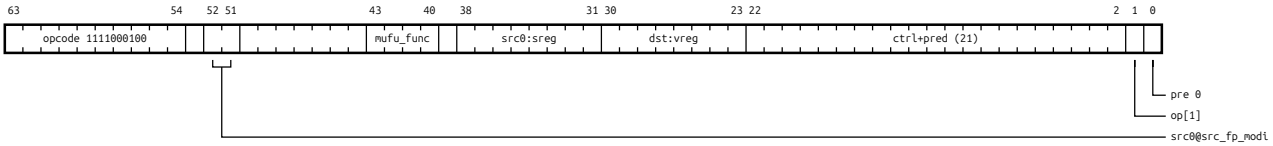
Leaf: mufu__fp32.v_x

Format:

```
mufu.mufu_func  dst, src0{.src_fp_modi}
```

Operands: dst: vreg; src0: sreg

Encoding Diagram: 64b, prefix 0, opcode 11111000100



Notes:

Multi-Function Unit: 硬件 transcendental / special function 单元, 单 src 计算 cos / sin / ex2 (2^x) / lg2 (log2) / rcp (1/x) / rsq (1/√x) / sqrt (√x) / tanh / gelu / silu 等。具体函数由 mufu_func modifier 选, 一条 mnemonic 收敛多函数。

FP32 路径 (fp32.* 指令形态): src / dst 都是 32-bit FP32, mufu_func 全集 (含 fp64_rcp_seed / fp64_rsq_seed — FP64 倒数 / 倒数平方根的 seed 值高位部分, 后接 Newton-Raphson 软件迭代得完整 FP64 结果; 这两个取值属于 fp64 扩展)。

fp16 / bf16 路径 (fp16.* 指令形态): src 是 32-bit reg + src_extract operand-modifier 选 H0 / H1 抽 fp16/bf16 单值, dst 写 32-bit reg 低 16 位; mufu_func 可用 cos / sin / ex2 / lg2 / rcp / rsq / sqrt / tanh / gelu / silu (fp64_rcp_seed / fp64_rsq_seed seed 路径不暴露)。

type_fp16_bf16 是指令形态局部 modifier (仅 fp16.* 指令形态在该指令形态的 modifiers 列表引用, fp32.* 指令形态不读这一位 bit), 取值 fp16 (IEEE 754 binary16) / bf16 (Brain Float)。

精度契约与特殊值行为见 mufu_func modifier 的 meaning: mufu 是近似单元, 不承诺 correctly-rounded; 每个函数带最大误差界 (数值待硬件定案), 特殊值 (±0 / ±Inf / NaN / 定义域外输入) 按对应精确函数的 IEEE 754 规则、所有实现逐 bit 一致。

14.7.2 FP64 compute

dadd **category**: FP64 compute **predicate_mode**: simt

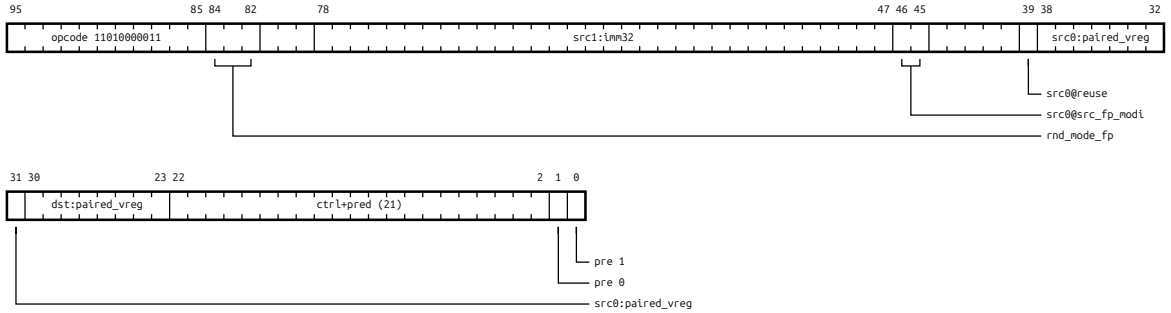
Leaf: dadd__v_vi

Format:

```
dadd.rnd_mode_fp  dst, src0{.src_fp_modi}{.reuse}, src1
```

Operands: dst: paired_vreg; src0: paired_vreg; src1: imm32u

Encoding Diagram: 96b, prefix 10, opcode 11010000011



Notes:

FP64 双精度浮点加法: {dst, dst+1} = {src0, src0+1} + {src1, src1+1}, 整 warp 32 个 active thread 各自独立计算, 结果按 IEEE 754 binary64 格式。

dst / src0 / src1 都是 register pair (低 32 bit 是命名 reg, 高 32 bit 隐式占 +1 reg), dst 必须 even-aligned。

src0 / src1 各可挂 src_fp_modi: neg (取负) / abs (取绝对值) / abs_then_neg, 一条指令内表达 -a+b、|a|+b 这种常见变体。

FP64 默认走 IEEE 754 行为 (denormal 保留), 没有 ftz / saturate modifier (跟 FP32 fadd 不同; FP64 路径硬件成本高, 不暴露快路径开关)。

rnd_mode_fp: 4 种 IEEE 754 标准舍入 (rne 默认 / rtz / rup / rdn)。

FP64 imm 编码: 32-bit imm 字段存 fp64 的高 32 位 (sign + exponent + 高 fraction), 低 32 位隐式补 0 (decoded_fp64 = imm « 32)。可表达常量限于低 32 位为 0 的 fp64 子集 (例: 1.0 = 0x3FF00000_00000000 编码

0x3FF00000)。

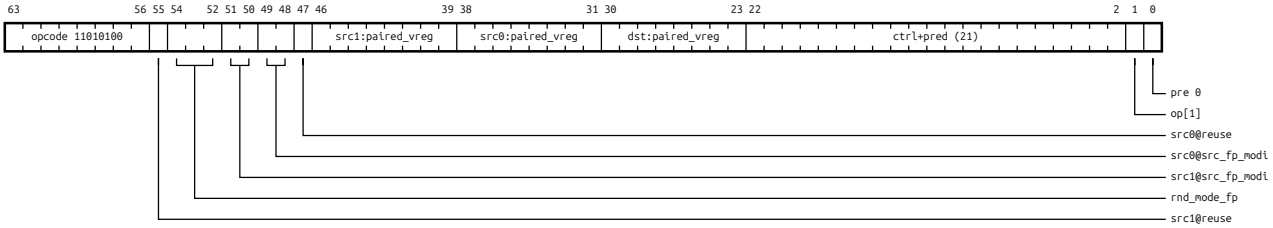
Leaf: dadd__v_vv

Format:

```
dadd.rnd_mode_fp  dst, src0{.src_fp_modi}{.reuse}, src1{.src_fp_modi}{.reuse}
```

Operands: dst: paired_vreg; src0: paired_vreg; src1: paired_vreg

Encoding Diagram: 64b, prefix 0, opcode 111010100



Notes:

FP64 双精度浮点加法: {dst, dst+1} = {src0, src0+1} + {src1, src1+1}, 整 warp 32 个 active thread 各自独立计算, 结果按 IEEE 754 binary64 格式。

dst / src0 / src1 都是 register pair (低 32 bit 是命名 reg, 高 32 bit 隐式占 +1 reg), dst 必须 even-aligned。

src0 / src1 各可挂 src_fp_modi: neg (取负) / abs (取绝对值) / abs_then_neg, 一条指令内表达 -a+b、|a|+b 这种常见变体。

FP64 默认走 IEEE 754 行为 (denormal 保留), 没有 ftz / saturate modifier (跟 FP32 fadd 不同; FP64 路径硬件成本高, 不暴露快路径开关)。

rnd_mode_fp: 4 种 IEEE 754 标准舍入 (rne 默认 / rtz / rup / rdn)。

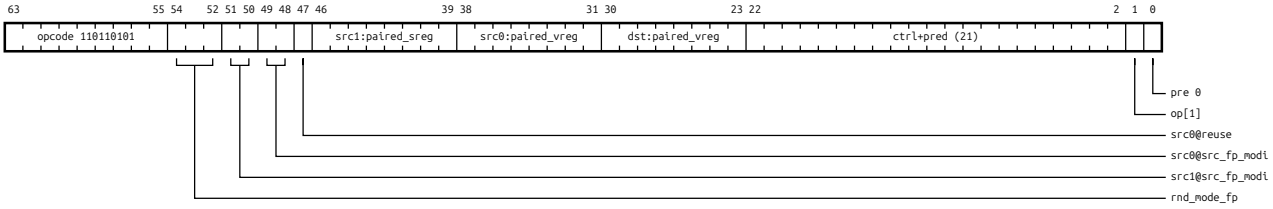
Leaf: dadd__v_vx

Format:

```
dadd.rnd_mode_fp  dst, src0{.src_fp_modi}{.reuse}, src1{.src_fp_modi}
```

Operands: dst: paired_vreg; src0: paired_vreg; src1: paired_sreg

Encoding Diagram: 64b, prefix 0, opcode 1110110101



Notes:

FP64 双精度浮点加法: {dst, dst+1} = {src0, src0+1} + {src1, src1+1}, 整 warp 32 个 active thread 各自独立计算, 结果按 IEEE 754 binary64 格式。

dst / src0 / src1 都是 register pair (低 32 bit 是命名 reg, 高 32 bit 隐式占 +1 reg), dst 必须 even-aligned。

src0 / src1 各可挂 src_fp_modi: neg (取负) / abs (取绝对值) / abs_then_neg, 一条指令内表达 -a+b、|a|+b 这种常见变体。

FP64 默认走 IEEE 754 行为 (denormal 保留), 没有 ftz / saturate modifier (跟 FP32 fadd 不同; FP64 路径硬件成本高, 不暴露快路径开关)。

rnd_mode_fp: 4 种 IEEE 754 标准舍入 (rne 默认 / rtz / rup / rdn)。

dfma **category**: FP64 compute **predicate_mode**: simt

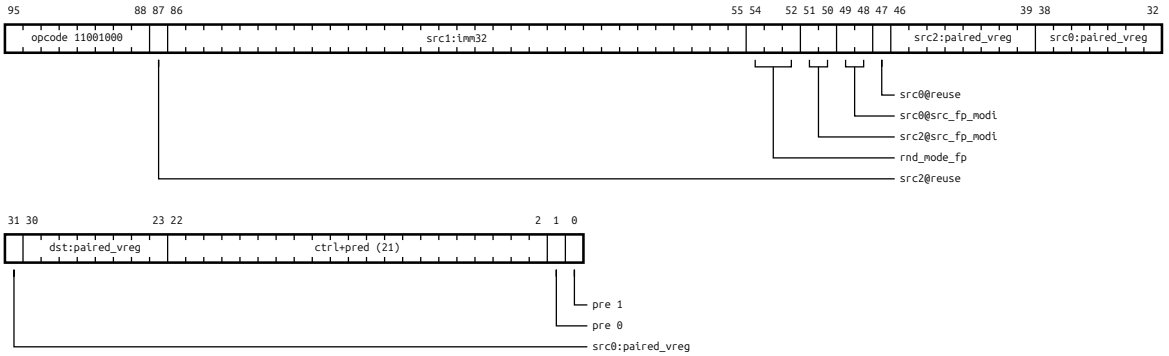
Leaf: dfma__v_viv

Format:

```
dfma.rnd_mode_fp  dst, src0{.src_fp_modi}{.reuse}, src1, src2{.src_fp_modi}{.reuse}
```

Operands: dst: paired_vreg; src0: paired_vreg; src1: imm32u; src2: paired_vreg

Encoding Diagram: 96b, prefix 10, opcode 11001000



Notes:

FP64 fused multiply-add: {dst, dst+1} = {src0, src0+1} × {src1, src1+1} + {src2, src2+1}, src0×src1 在无限精度下计算, 加上 src2 后按 rnd_mode_fp 做一次 IEEE 754 rounding。比 dmul + dadd 序列精度高 (单次舍入误差 ≤ 0.5 ulp)。

src0 / src1 / src2 都是 register pair (低 + 高两 reg), dst 必须 even-aligned。

三个 src 各可挂 src_fp_modi: neg / abs / abs_then_neg, 实现 -a*b+c、a*b-c、|a|*b+c 等变体。

FP64 默认走 IEEE 754 (denormal 保留), 没有 ftz / fmz / saturate。

rnd_mode_fp: 4 种 IEEE 754 舍入 (rne / rtz / rup / rdn)。

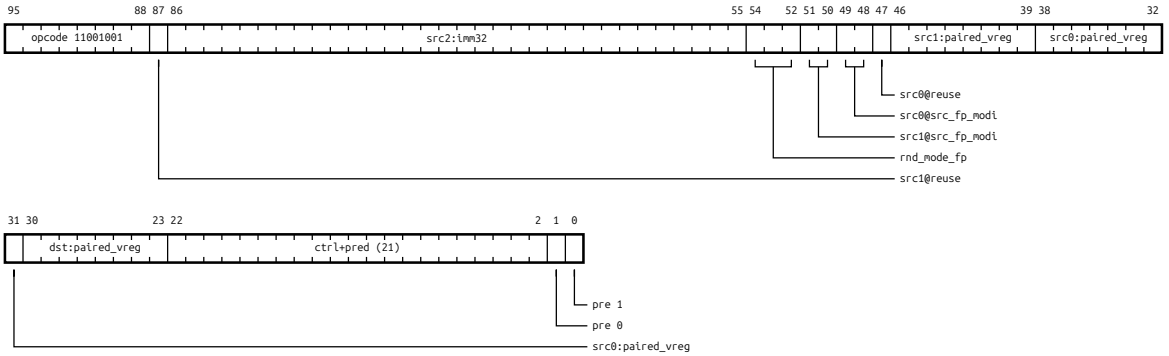
Leaf: dfma__v_vvi

Format:

```
dfma.rnd_mode_fp  dst, src0{.src_fp_modi}{.reuse}, src1{.src_fp_modi}{.reuse}, src2
```

Operands: dst: paired_vreg; src0: paired_vreg; src1: paired_vreg; src2: imm32u

Encoding Diagram: 96b, prefix 10, opcode 11001001



Notes:

FP64 fused multiply-add: {dst, dst+1} = {src0, src0+1} × {src1, src1+1} + {src2, src2+1}, src0×src1 在无限精度下计算, 加上 src2 后按 rnd_mode_fp 做一次 IEEE 754 rounding。比 dmul + dadd 序列精度高 (单次舍入误差 ≤ 0.5 ulp)。

src0 / src1 / src2 都是 register pair (低 + 高两 reg), dst 必须 even-aligned。

三个 src 各可挂 src_fp_modi: neg / abs / abs_then_neg, 实现 -a*b+c、a*b-c、|a|*b+c 等变体。

FP64 默认走 IEEE 754 (denormal 保留), 没有 ftz / fmz / saturate。

rnd_mode_fp: 4 种 IEEE 754 舍入 (rne / rtz / rup / rdn)。

FP64 imm 编码: 32-bit imm 字段存 fp64 高 32 位 (sign + exponent + 高 fraction), 低 32 位隐式补 0 (decoded_fp64 = imm « 32)。可表达常量限于低 32 位为 0 的 fp64 子集。

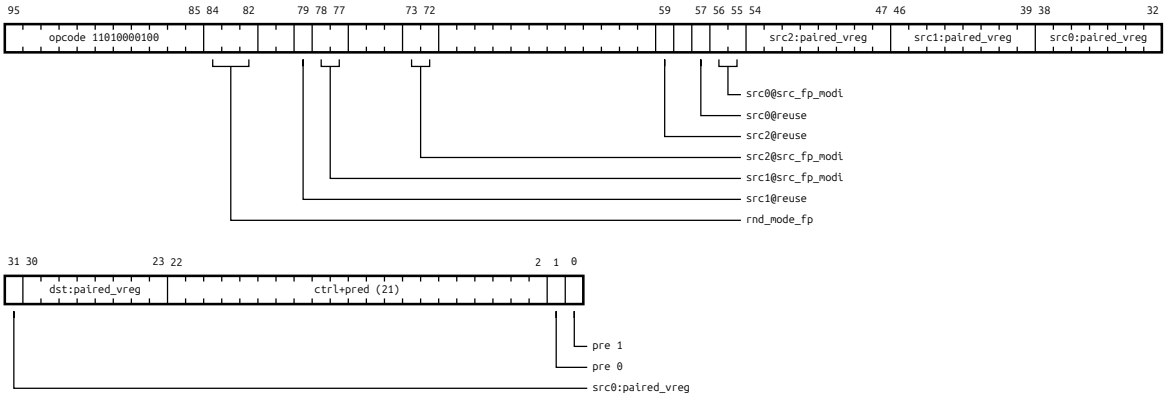
Leaf: dfma__v_vvv

Format:

```
dfma.rnd_mode_fp  dst, src0{.src_fp_modi}{.reuse}, src1{.src_fp_modi}{.reuse},
↪  src2{.src_fp_modi}{.reuse}
```

Operands: dst: paired_vreg; src0: paired_vreg; src1: paired_vreg; src2: paired_vreg

Encoding Diagram: 96b, prefix 10, opcode 11010000100



Notes:

FP64 fused multiply-add: {dst, dst+1} = {src0, src0+1} × {src1, src1+1} + {src2, src2+1}, src0×src1 在无限精度下计算, 加上 src2 后按 rnd_mode_fp 做一次 IEEE 754 rounding。比 dmul + dadd 序列精度高 (单次舍入误差 ≤ 0.5 ulp)。

src0 / src1 / src2 都是 register pair (低 + 高两 reg), dst 必须 even-aligned。

三个 src 各可挂 src_fp_modi: neg / abs / abs_then_neg, 实现 -a*b+c、a*b-c、|a|*b+c 等变体。

FP64 默认走 IEEE 754 (denormal 保留), 没有 ftz / fmz / saturate。

rnd_mode_fp: 4 种 IEEE 754 舍入 (rne / rtz / rup / rdn)。

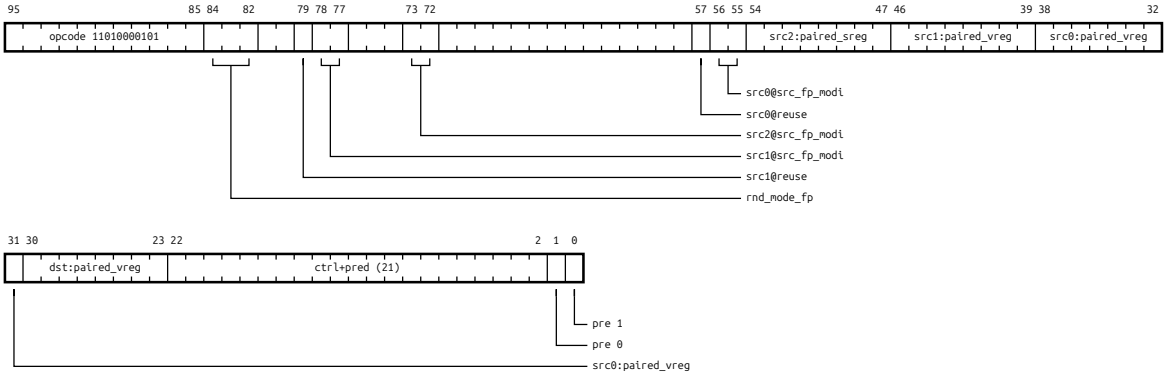
Leaf: dfma__v_vvx

Format:

```
dfma.rnd_mode_fp  dst, src0{.src_fp_modi}{.reuse}, src1{.src_fp_modi}{.reuse},
↪  src2{.src_fp_modi}
```

Operands: dst: paired_vreg; src0: paired_vreg; src1: paired_vreg; src2: paired_sreg

Encoding Diagram: 96b, prefix 10, opcode 11010000101



Notes:

FP64 fused multiply-add: {dst, dst+1} = {src0, src0+1} × {src1, src1+1} + {src2, src2+1}, src0×src1 在无限精度下计算, 加上 src2 后按 rnd_mode_fp 做一次 IEEE 754 rounding。比 dmul + dadd 序列精度高 (单次舍入误差 ≤ 0.5 ulp)。

src0 / src1 / src2 都是 register pair (低 + 高两 reg), dst 必须 even-aligned。

三个 src 各可挂 src_fp_modi: neg / abs / abs_then_neg, 实现 -a*b+c、a*b-c、|a|*b+c 等变体。

FP64 默认走 IEEE 754 (denormal 保留), 没有 ftz / fmz / saturate。

rnd_mode_fp: 4 种 IEEE 754 舍入 (rne / rtz / rup / rdn)。

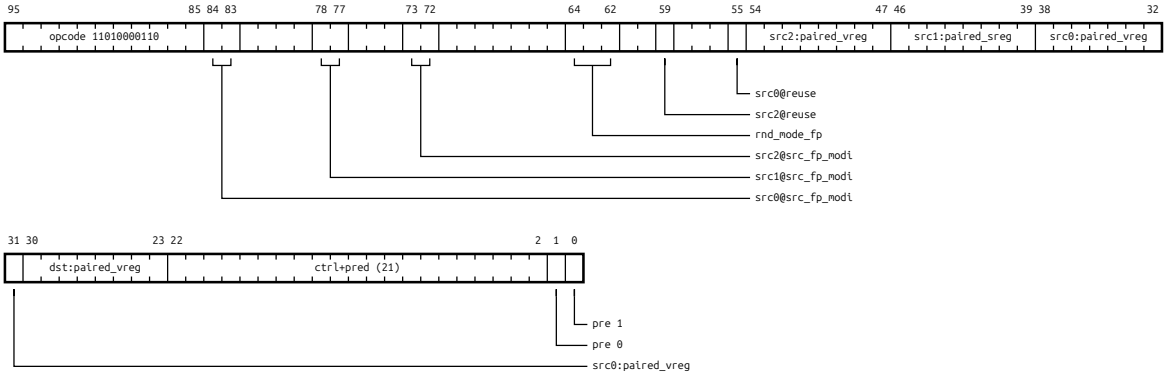
Leaf: dfma__v_vxv

Format:

```
dfma.rnd_mode_fp  dst, src0{.src_fp_modi}{.reuse}, src1{.src_fp_modi},
↪  src2{.src_fp_modi}{.reuse}
```

Operands: dst: paired_vreg; src0: paired_vreg; src1: paired_sreg; src2: paired_vreg

Encoding Diagram: 96b, prefix 10, opcode 11010000110



Notes:

FP64 fused multiply-add: {dst, dst+1} = {src0, src0+1} × {src1, src1+1} + {src2, src2+1}, src0×src1 在无限精度下计算, 加上 src2 后按 rnd_mode_fp 做一次 IEEE 754 rounding。比 dmul + dadd 序列精度高 (单次舍入误差 ≤ 0.5 ulp)。

src0 / src1 / src2 都是 register pair (低 + 高两 reg), dst 必须 even-aligned。

三个 src 各可挂 src_fp_modi: neg / abs / abs_then_neg, 实现 -a*b+c、a*b-c、|a|*b+c 等变体。

FP64 默认走 IEEE 754 (denormal 保留), 没有 ftz / fmz / saturate。

rnd_mode_fp: 4 种 IEEE 754 舍入 (rne / rtz / rup / rdn)。

dmnmx **category**: FP64 compute **predicate_mode**: simt

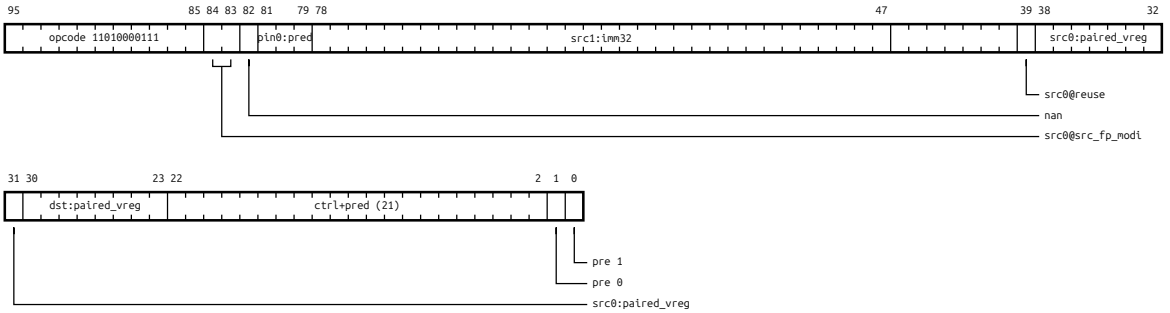
Leaf: dmnmx__v_vi

Format:

```
dmnmx{.nan}  dst, src0{.src_fp_modi}{.reuse}, src1, pin0
```

Operands: dst: paired_vreg; src0: paired_vreg; src1: imm32u; pin0: pred

Encoding Diagram: 96b, prefix 10, opcode 11010000111



Notes:

FP64 per-lane min/max: {dst, dst+1} = pin0 ? min(src0, src1) : max(src0, src1), 由 pin0 predicate 控制 min/max 选择。pin0=PT 永远 min, pin0=!PT 永远 max。

src0 / src1 都是 register pair (low+high), dst 必须 even-aligned。fmnmx 的 64-bit 镜像。

NaN modifier: 跟 fmnmx 一致 (DEFAULT / nan), 控制 NaN 输入的传播行为, 对应 IEEE 754-2008 min/max 规则。

FP64 没有 ftz, 走 IEEE 754 (denormal 保留)。

FP64 imm 编码: 32-bit imm 字段存 fp64 高 32 位 (sign + exponent + 高 fraction), 低 32 位隐式补 0 (decoded_fp64 = imm « 32)。

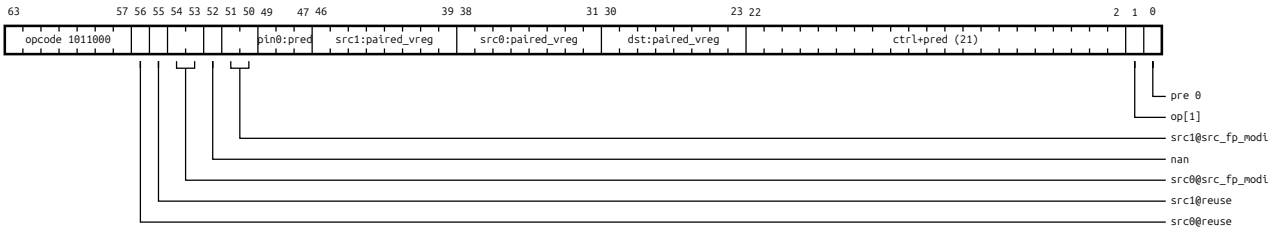
Leaf: dmnmx__v_vv

Format:

```
dmnmx{.nan}  dst, src0{.src_fp_modi}{.reuse}, src1{.src_fp_modi}{.reuse}, pin0
```

Operands: dst: paired_vreg; src0: paired_vreg; src1: paired_vreg; pin0: pred

Encoding Diagram: 64b, prefix 0, opcode 11011000



Notes:

FP64 per-lane min/max: {dst, dst+1} = pin0 ? min(src0, src1) : max(src0, src1), 由 pin0 predicate 控制 min/max 选择。pin0=PT 永远 min, pin0=!PT 永远 max。

src0 / src1 都是 register pair (low+high), dst 必须 even-aligned。fmnmx 的 64-bit 镜像。

NaN modifier: 跟 fmnmx 一致 (DEFAULT / nan), 控制 NaN 输入的传播行为, 对应 IEEE 754-2008 min/max 规则。

FP64 没有 ftz, 走 IEEE 754 (denormal 保留)。

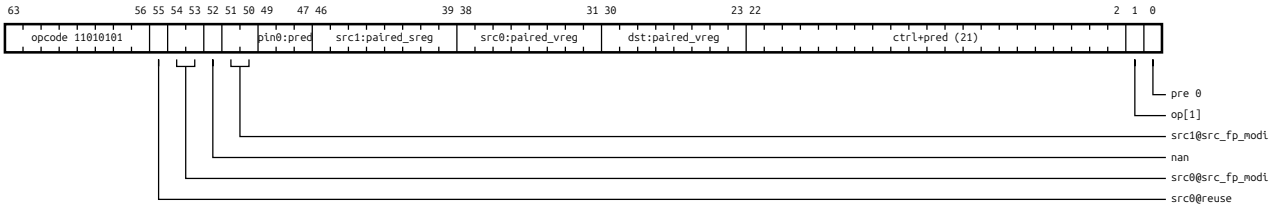
Leaf: dmnmx__v_vx

Format:

```
dmnmx{.nan}  dst, src0{.src_fp_modi}{.reuse}, src1{.src_fp_modi}, pin0
```

Operands: dst: paired_vreg; src0: paired_vreg; src1: paired_sreg; pin0: pred

Encoding Diagram: 64b, prefix 0, opcode 111010101



Notes:

FP64 per-lane min/max: {dst, dst+1} = pin0 ? min(src0, src1) : max(src0, src1), 由 pin0 predicate 控制 min/max 选择。pin0=PT 永远 min, pin0=!PT 永远 max。

src0 / src1 都是 register pair (low+high), dst 必须 even-aligned。fmnmx 的 64-bit 镜像。

NaN modifier: 跟 fmnmx 一致 (DEFAULT / nan), 控制 NaN 输入的传播行为, 对应 IEEE 754-2008 min/max 规则。

FP64 没有 ftz, 走 IEEE 754 (denormal 保留)。

dmul　**category**: FP64 compute　**predicate_mode**: simt

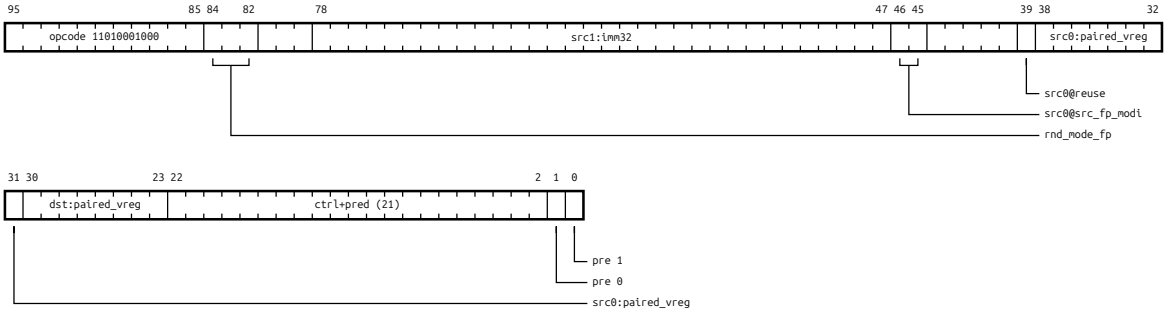
Leaf: dmul__v_vi

Format:

```
dmul.rnd_mode_fp  dst, src0{.src_fp_modi}{.reuse}, src1
```

Operands: dst: paired_vreg; src0: paired_vreg; src1: imm32u

Encoding Diagram: 96b, prefix 10, opcode 11010001000



Notes:

FP64 双精度浮点乘法: {dst, dst+1} = {src0, src0+1} × {src1, src1+1}, 整 warp 32 个 active thread 各自独立计算, 结果按 IEEE 754 binary64 格式。

dst / src0 / src1 都是 register pair, dst 必须 even-aligned。

src0 / src1 各可挂 src_fp_modi: neg / abs / abs_then_neg, 实现符号 / 绝对值控制。

FP64 默认走 IEEE 754 (denormal 保留), 没有 ftz / fmz / saturate (跟 fmul 不同)。

rnd_mode_fp: 4 种 IEEE 754 舍入 (rne / rtz / rup / rdn)。

FP64 imm 编码: 32-bit imm 字段存 fp64 高 32 位, 低 32 位隐式补 0 (decoded_fp64 = imm « 32)。

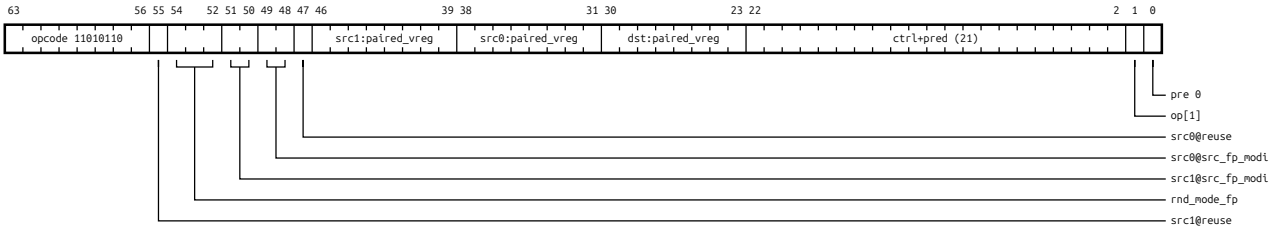
Leaf: dmul__v_vv

Format:

```
dmul.rnd_mode_fp  dst, src0{.src_fp_modi}{.reuse}, src1{.src_fp_modi}{.reuse}
```

Operands: dst: paired_vreg; src0: paired_vreg; src1: paired_vreg

Encoding Diagram: 64b, prefix 0, opcode 111010110



Notes:

FP64 双精度浮点乘法: {dst, dst+1} = {src0, src0+1} × {src1, src1+1}, 整 warp 32 个 active thread 各自独立计算, 结果按 IEEE 754 binary64 格式。

dst / src0 / src1 都是 register pair, dst 必须 even-aligned。

src0 / src1 各可挂 src_fp_modi: neg / abs / abs_then_neg, 实现符号 / 绝对值控制。

FP64 默认走 IEEE 754 (denormal 保留), 没有 ftz / fmz / saturate (跟 fmul 不同)。

rnd_mode_fp: 4 种 IEEE 754 舍入 (rne / rtz / rup / rdn)。

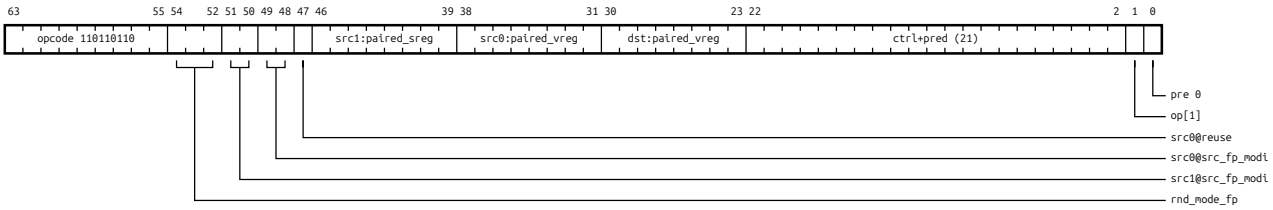
Leaf: dmul__v_vx

Format:

```
dmul.rnd_mode_fp  dst, src0{.src_fp_modi}{.reuse}, src1{.src_fp_modi}
```

Operands: dst: paired_vreg; src0: paired_vreg; src1: paired_sreg

Encoding Diagram: 64b, prefix 0, opcode 1110110110



Notes:

FP64 双精度浮点乘法: {dst, dst+1} = {src0, src0+1} × {src1, src1+1}, 整 warp 32 个 active thread 各自独立计算, 结果按 IEEE 754 binary64 格式。

dst / src0 / src1 都是 register pair, dst 必须 even-aligned。

src0 / src1 各可挂 src_fp_modi: neg / abs / abs_then_neg, 实现符号 / 绝对值控制。

FP64 默认走 IEEE 754 (denormal 保留), 没有 ftz / fmz / saturate (跟 fmul 不同)。

rnd_mode_fp: 4 种 IEEE 754 舍入 (rne / rtz / rup / rdn)。

dsetp　**category**: FP64 compute　**predicate_mode**: simt

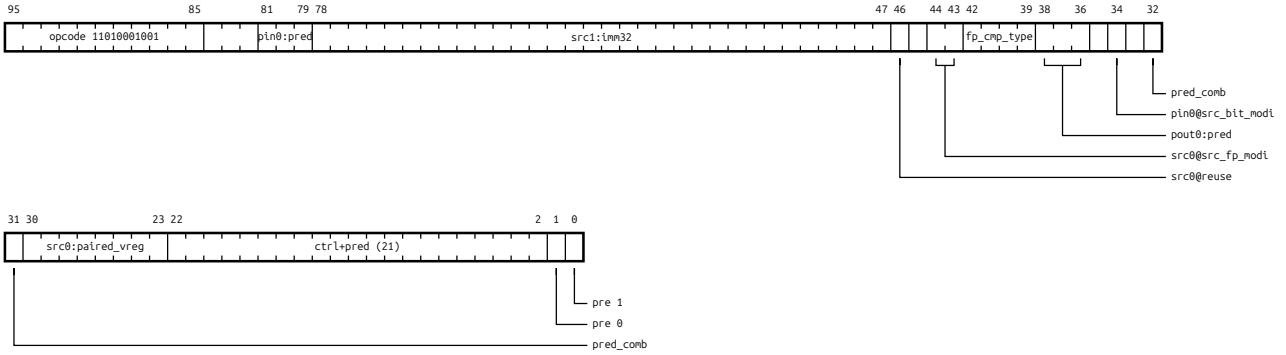
Leaf: dsetp__p_vip

Format:

```
dsetp.fp_cmp_type.pred_comb  pout0, src0{.src_fp_modi}{.reuse}, src1, pin0{.src_bit_modi}
```

Operands: pout0: pred; src0: paired_vreg; src1: imm32u; pin0: pred

Encoding Diagram: 96b, prefix 10, opcode 11010001001



Notes:

FP64 比较 → predicate: 先做 ({src0, src0+1} CMP {src1, src1+1}) 得 1-bit cmp; 然后 pout0 = pred_comb(cmp, pin0), pout1 (若声明) = pred_comb(!cmp, pin0), 整 warp per-lane 各自独立。

pout1 不是!pout0, 而是用同一个 pred_comb 跟同一个 pin0 把!cmp 组合一次 (外层 pin0 控制内层 cmp 的嵌套谓词模式)。pin0=PT + pred_comb=and 时退化为 pout0 = cmp / pout1 = !cmp。

src0 / src1 都是 register pair (low+high); fsetp 的 64-bit 镜像。

fp_cmp_type: 16 种比较 (lt / le / eq / ne / ge / gt / num 非 NaN / nan / f 恒假 / t 恒真等), 跟 fsetp 一致。

pred_comb: and / or / xor 三种 cmp 跟 pin0 的组合方式。

FP64 没有 ftz, 走 IEEE 754 (denormal 保留)。

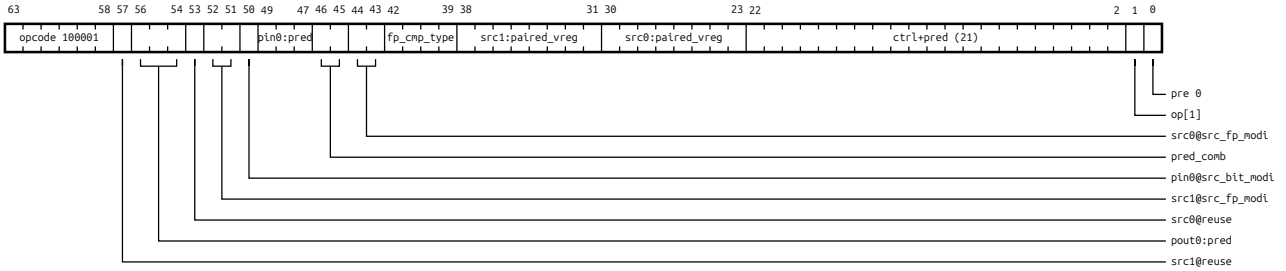
Leaf: dsetp__p_vvp

Format:

```
dsetp.fp_cmp_type.pred_comb  pout0, src0{.src_fp_modi}{.reuse},
↪  src1{.src_fp_modi}{.reuse}, pin0{.src_bit_modi}
```

Operands: pout0: pred; src0: paired_vreg; src1: paired_vreg; pin0: pred

Encoding Diagram: 64b, prefix 0, opcode 1100001



Notes:

FP64 比较 → predicate: 先做 ({src0, src0+1} CMP {src1, src1+1}) 得 1-bit cmp; 然后 pout0 = pred_comb(cmp, pin0), pout1 (若声明) = pred_comb(!cmp, pin0), 整 warp per-lane 各自独立。

pout1 不是!pout0, 而是用同一个 pred_comb 跟同一个 pin0 把!cmp 组合一次 (外层 pin0 控制内层 cmp 的嵌套谓词模式)。pin0=PT + pred_comb=and 时退化为 pout0 = cmp / pout1 = !cmp。

src0 / src1 都是 register pair (low+high); fsetp 的 64-bit 镜像。

fp_cmp_type: 16 种比较 (lt / le / eq / ne / ge / gt / num 非 NaN / nan / f 恒假 / t 恒真等), 跟 fsetp 一致。

pred_comb: and / or / xor 三种 cmp 跟 pin0 的组合方式。

FP64 没有 ftz, 走 IEEE 754 (denormal 保留)。

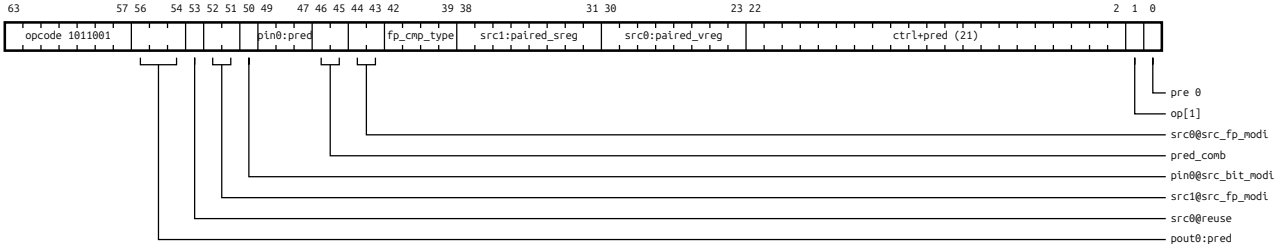
Leaf: dsetp__p_vxp

Format:

```
dsetp.fp_cmp_type.pred_comb  pout0, src0{.src_fp_modi}{.reuse}, src1{.src_fp_modi},
↪  pin0{.src_bit_modi}
```

Operands: pout0: pred; src0: paired_vreg; src1: paired_sreg; pin0: pred

Encoding Diagram: 64b, prefix 0, opcode 11011001



Notes:

FP64 比较 → predicate: 先做 ({src0, src0+1} CMP {src1, src1+1}) 得 1-bit cmp; 然后 pout0 = pred_comb(cmp, pin0), pout1 (若声明) = pred_comb(!cmp, pin0), 整 warp per-lane 各自独立。

pout1 不是!pout0, 而是用同一个 pred_comb 跟同一个 pin0 把!cmp 组合一次 (外层 pin0 控制内层 cmp 的嵌套谓词模式)。pin0=PT + pred_comb=and 时退化为 pout0 = cmp / pout1 = !cmp。

src0 / src1 都是 register pair (low+high); fsetp 的 64-bit 镜像。

fp_cmp_type: 16 种比较 (lt / le / eq / ne / ge / gt / num 非 NaN / nan / f 恒假 / t 恒真等), 跟 fsetp 一致。

pred_comb: and / or / xor 三种 cmp 跟 pin0 的组合方式。

FP64 没有 ftz, 走 IEEE 754 (denormal 保留)。

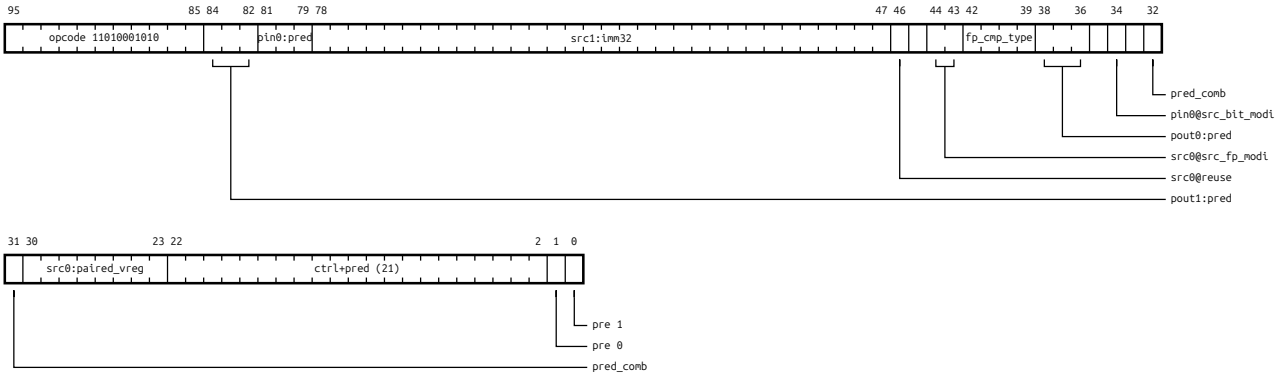
Leaf: dsetp__pp_vip

Format:

```
dsetp.fp_cmp_type.pred_comb  pout0, pout1, src0{.src_fp_modi}{.reuse}, src1,
↪  pin0{.src_bit_modi}
```

Operands: pout0: pred; pout1: pred; src0: paired_vreg; src1: imm32u; pin0: pred

Encoding Diagram: 96b, prefix 10, opcode 11010001010



Notes:

FP64 比较 → predicate: 先做 ({src0, src0+1} CMP {src1, src1+1}) 得 1-bit cmp; 然后 pout0 = pred_comb(cmp, pin0), pout1 (若声明) = pred_comb(!cmp, pin0), 整 warp per-lane 各自独立。

pout1 不是!pout0, 而是用同一个 pred_comb 跟同一个 pin0 把!cmp 组合一次 (外层 pin0 控制内层 cmp 的嵌套谓词模式)。pin0=PT + pred_comb=and 时退化为 pout0 = cmp / pout1 = !cmp。

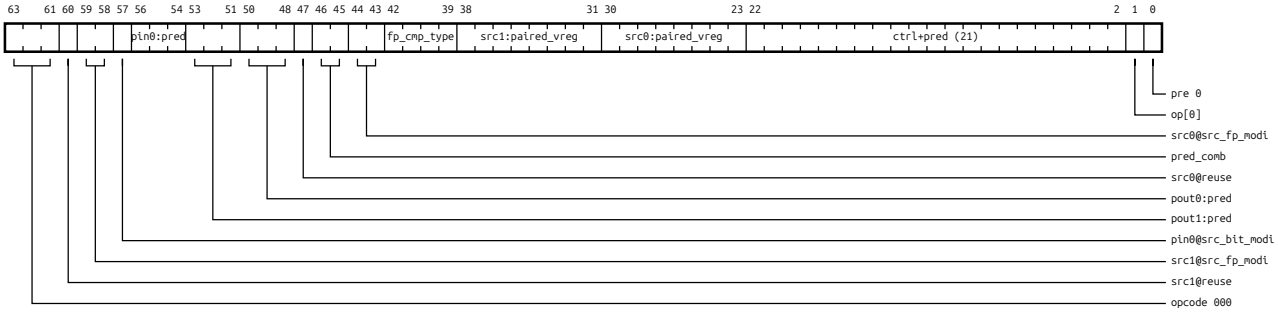
src0 / src1 都是 register pair (low+high); fsetp 的 64-bit 镜像。

fp_cmp_type: 16 种比较 (lt / le / eq / ne / ge / gt / num 非 NaN / nan / f 恒假 / t 恒真等), 跟 fsetp 一致。
pred_comb: and / or / xor 三种 cmp 跟 pin0 的组合方式。
FP64 没有 ftz, 走 IEEE 754 (denormal 保留)。

Leaf: dsetp__pp_vvp
Format:

```
dsetp.fp_cmp_type.pred_comb  pout0, pout1, src0{.src_fp_modi}{.reuse},
↪  src1{.src_fp_modi}{.reuse}, pin0{.src_bit_modi}
```

Operands: pout0: pred; pout1: pred; src0: paired_vreg; src1: paired_vreg; pin0: pred
Encoding Diagram: 64b, prefix 0, opcode 0000

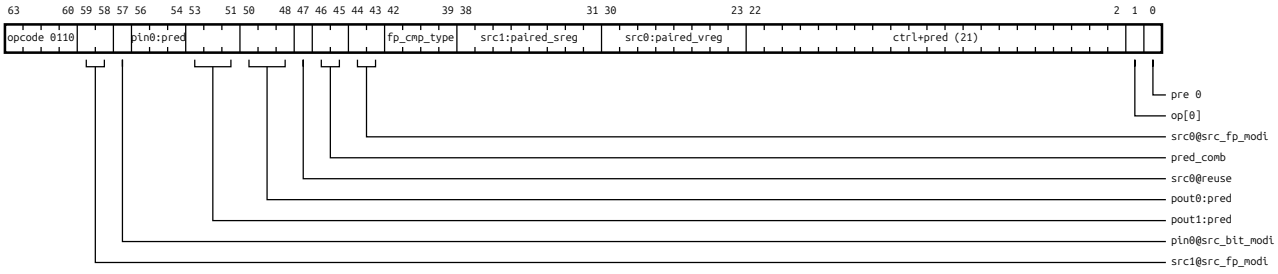


Notes:
FP64 比较→predicate: 先做 ({src0, src0+1} CMP {src1, src1+1}) 得 1-bit cmp; 然后 pout0 = pred_comb(cmp, pin0), pout1 (若声明) = pred_comb(!cmp, pin0), 整 warp per-lane 各自独立。
pout1 不是!pout0, 而是用同一个 pred_comb 跟同一个 pin0 把!cmp 组合一次 (外层 pin0 控制内层 cmp 的嵌套谓词模式)。pin0=PT + pred_comb=and 时退化为 pout0 = cmp / pout1 = !cmp。
src0 / src1 都是 register pair (low+high); fsetp 的 64-bit 镜像。
fp_cmp_type: 16 种比较 (lt / le / eq / ne / ge / gt / num 非 NaN / nan / f 恒假 / t 恒真等), 跟 fsetp 一致。
pred_comb: and / or / xor 三种 cmp 跟 pin0 的组合方式。
FP64 没有 ftz, 走 IEEE 754 (denormal 保留)。

Leaf: dsetp__pp_vxp
Format:

```
dsetp.fp_cmp_type.pred_comb  pout0, pout1, src0{.src_fp_modi}{.reuse},
↪  src1{.src_fp_modi}, pin0{.src_bit_modi}
```

Operands: pout0: pred; pout1: pred; src0: paired_vreg; src1: paired_sreg; pin0: pred
Encoding Diagram: 64b, prefix 0, opcode 00110



Notes:
FP64 比较→predicate: 先做 ({src0, src0+1} CMP {src1, src1+1}) 得 1-bit cmp; 然后 pout0 = pred_comb(cmp, pin0), pout1 (若声明) = pred_comb(!cmp, pin0), 整 warp per-lane 各自独立。

pout1 不是!pout0, 而是用同一个 pred_comb 跟同一个 pin0 把!cmp 组合一次 (外层 pin0 控制内层 cmp 的嵌套谓词模式)。pin0=PT + pred_comb=and 时退化为 pout0 = cmp / pout1 = !cmp。

src0 / src1 都是 register pair (low+high); fsetp 的 64-bit 镜像。

fp_cmp_type: 16 种比较 (lt / le / eq / ne / ge / gt / num 非 NaN / nan / f 恒假 / t 恒真等), 跟 fsetp 一致。

pred_comb: and / or / xor 三种 cmp 跟 pin0 的组合方式。

FP64 没有 ftz, 走 IEEE 754 (denormal 保留)。

14.7.3 Packed and low-precision compute

hadd_pk **category**: Packed and low-precision compute **predicate_mode**: simt

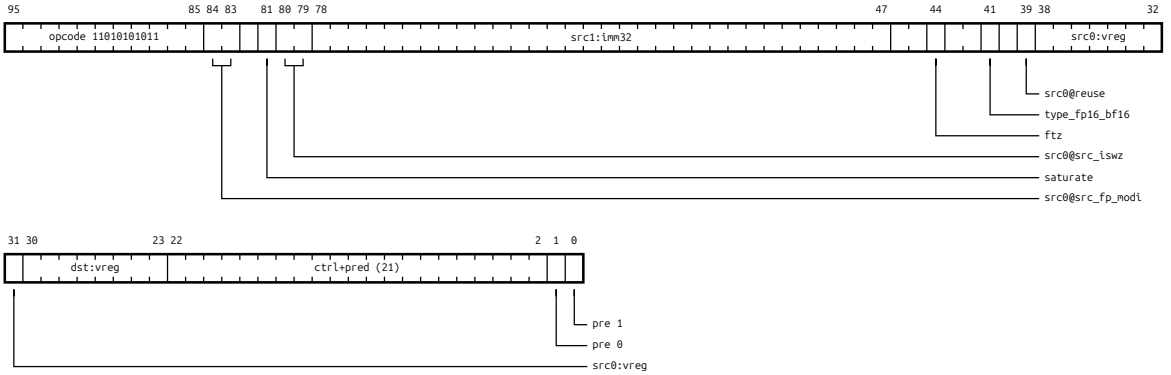
Leaf: hadd_pk__f16.v_vi

Format:

```
hadd_pk.ftz{.saturate}.type_fp16_bf16  dst, src0{.src_fp_modi}{.src_iswz}{.reuse}, src1
```

Operands: dst: vreg; src0: vreg; src1: imm32u

Encoding Diagram: 96b, prefix 10, opcode 11010101011



Notes:

Packed fp16/bf16 加法: src0 / src1 都是 32-bit reg 各装 2 个 fp16/bf16, 两 half 各自独立相加, 结果 packed 写 dst。整 warp 32 个 active thread 各自计算。

src_iswz (packed swizzle): hilo (默认两 half 各自参与) / lo (h0_h0 broadcast 低半字给两 half) / hi (h1_h1 broadcast) / b32 (src 是 FP32 → 转 fp16 truncation 复制到两 half, 计算仍走 fp16 path)。

type_fp16_bf16: 选 src / dst 浮点格式 (fp16 或 bf16)。bf16 模式下 ftz / saturate 是否生效由 implementation 决定; 跨 vendor 一致建议 bf16 下保持 default。

指令形态路径分两簇: f16.* = 双 half packed 输出 (主路径); f32.* = mixed-precision 仅算 src0.lo + src1.lo 的单个 fp16 加法, 再 widen 成 FP32 写整 32-bit dst (hi 半字不参与, dst 是 FP32 single value)。

ftz=disable: 走 IEEE 行为, denorm 保留。

ftz=ftz: input / output denorm flush 为 sign-preserving 0。

saturate=saturate: dst clamp 到 [+0.0, 1.0], NaN → +0.0 (ML activation 路径用)。

Packed imm 编码: 32-bit imm 装 2 个 fp16/bf16, [15:0]=H0, [31:16]=H1。

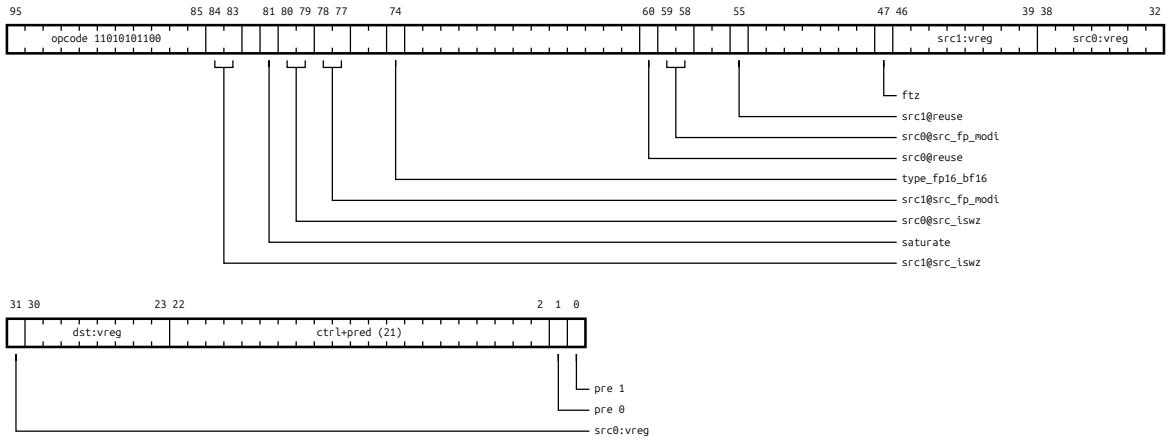
Leaf: hadd_pk__f16.v_vv

Format:

```
hadd_pk.ftz{.saturate}.type_fp16_bf16  dst, src0{.src_fp_modi}{.src_iswz}{.reuse},
↪  src1{.src_fp_modi}{.src_iswz}{.reuse}
```

Operands: dst: vreg; src0: vreg; src1: vreg

Encoding Diagram: 96b, prefix 10, opcode 11010101100



Notes:

Packed fp16/bf16 加法: src0 / src1 都是 32-bit reg 各装 2 个 fp16/bf16, 两 half 各自独立相加, 结果 packed 写 dst。整 warp 32 个 active thread 各自计算。

src_iswz (packed swizzle): hilo (默认两 half 各自参与) / lo (h0_h0 broadcast 低半字给两 half) / hi (h1_h1 broadcast) / b32 (src 是 FP32 → 转 fp16 truncation 复制到两 half, 计算仍走 fp16 path)。

type_fp16_bf16: 选 src / dst 浮点格式 (fp16 或 bf16)。bf16 模式下 ftz / saturate 是否生效由 implementation 决定; 跨 vendor 一致建议 bf16 下保持 default。

指令形态路径分两簇: f16.* = 双 half packed 输出 (主路径); f32.* = mixed-precision 仅算 src0.lo + src1.lo 的单个 fp16 加法, 再 widen 成 FP32 写整 32-bit dst (hi 半字不参与, dst 是 FP32 single value)。

ftz=disable: 走 IEEE 行为, denorm 保留。

ftz=ftz: input / output denorm flush 为 sign-preserving 0。

saturate=saturate: dst clamp 到 [+0.0, 1.0], NaN → +0.0 (ML activation 路径用)。

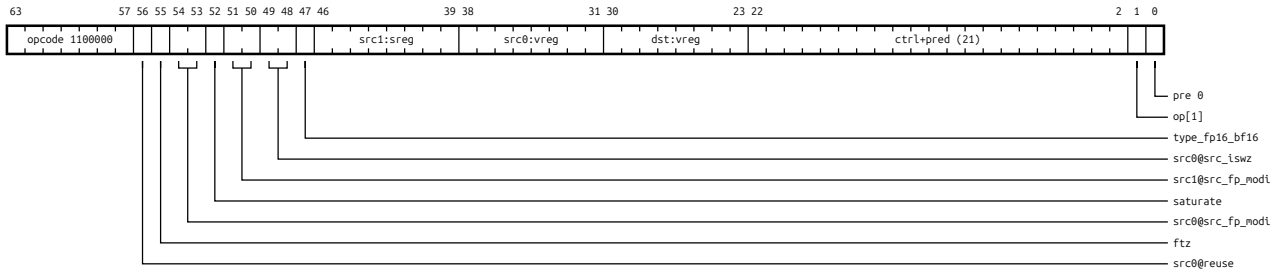
Leaf: hadd_pk__f16.v_vx

Format:

```
hadd_pk.ftz{.saturate}.type_fp16_bf16  dst, src0{.src_fp_modi}{.src_iswz}{.reuse},
↪  src1{.src_fp_modi}
```

Operands: dst: vreg; src0: vreg; src1: sreg

Encoding Diagram: 64b, prefix 0, opcode 11100000



Notes:

Packed fp16/bf16 加法: src0 / src1 都是 32-bit reg 各装 2 个 fp16/bf16, 两 half 各自独立相加, 结果 packed 写 dst。整 warp 32 个 active thread 各自计算。

src_iswz (packed swizzle): hilo (默认两 half 各自参与) / lo (h0_h0 broadcast 低半字给两 half) / hi (h1_h1 broadcast) / b32 (src 是 FP32 → 转 fp16 truncation 复制到两 half, 计算仍走 fp16 path)。

type_fp16_bf16: 选 src / dst 浮点格式 (fp16 或 bf16)。bf16 模式下 ftz / saturate 是否生效由 implementation 决定; 跨 vendor 一致建议 bf16 下保持 default。

指令形态路径分两簇: f16.* = 双 half packed 输出 (主路径); f32.* = mixed-precision 仅算 src0.lo + src1.lo 的单个 fp16 加法, 再 widen 成 FP32 写整 32-bit dst (hi 半字不参与, dst 是 FP32 single value)。

ftz=disable: 走 IEEE 行为, denorm 保留。

ftz=ftz: input / output denorm flush 为 sign-preserving 0。

saturate=saturate: dst clamp 到 [+0.0, 1.0], NaN → +0.0 (ML activation 路径用)。

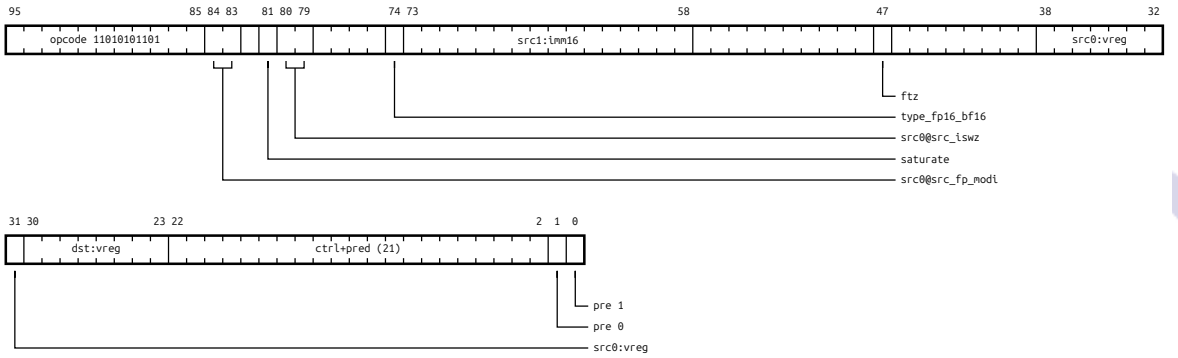
Leaf: hadd_pk__f32.v_vi

Format:

```
hadd_pk.ftz{.saturate}.type_fp16_bf16  dst, src0{.src_fp_modi}{.src_iswz}, src1
```

Operands: dst: vreg; src0: vreg; src1: imm16u

Encoding Diagram: 96b, prefix 10, opcode 11010101101



Notes:

Packed fp16/bf16 加法: src0 / src1 都是 32-bit reg 各装 2 个 fp16/bf16, 两 half 各自独立相加, 结果 packed 写 dst。整 warp 32 个 active thread 各自计算。

src_iswz (packed swizzle): hilo (默认两 half 各自参与) / lo (h0_h0 broadcast 低半字给两 half) / hi (h1_h1 broadcast) / b32 (src 是 FP32 → 转 fp16 truncation 复制到两 half, 计算仍走 fp16 path)。

type_fp16_bf16: 选 src / dst 浮点格式 (fp16 或 bf16)。bf16 模式下 ftz / saturate 是否生效由 implementation 决定; 跨 vendor 一致建议 bf16 下保持 default。

指令形态路径分两簇: f16.* = 双 half packed 输出 (主路径); f32.* = mixed-precision 仅算 src0.lo + src1.lo 的单个 fp16 加法, 再 widen 成 FP32 写整 32-bit dst (hi 半字不参与, dst 是 FP32 single value)。

ftz=disable: 走 IEEE 行为, denorm 保留。

ftz=ftz: input / output denorm flush 为 sign-preserving 0。

saturate=saturate: dst clamp 到 [+0.0, 1.0], NaN → +0.0 (ML activation 路径用)。

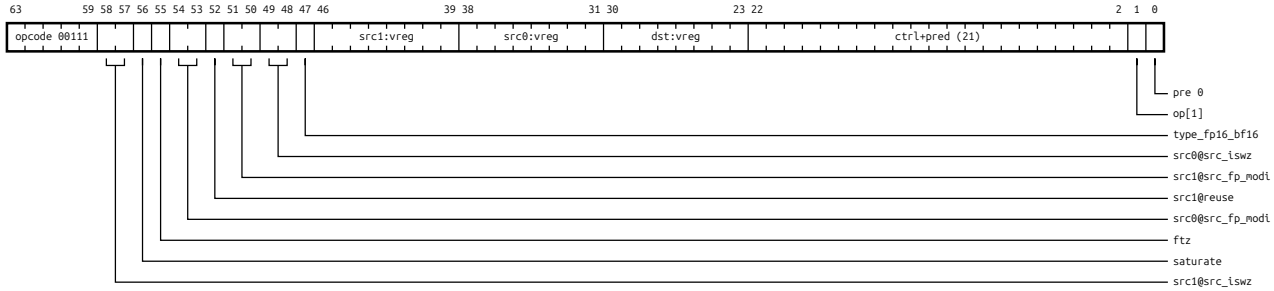
Leaf: hadd_pk__f32.v_vv

Format:

```
hadd_pk.ftz{.saturate}.type_fp16_bf16  dst, src0{.src_fp_modi}{.src_iswz},
↪  src1{.src_fp_modi}{.src_iswz}{.reuse}
```

Operands: dst: vreg; src0: vreg; src1: vreg

Encoding Diagram: 64b, prefix 0, opcode 100111



Notes:

Packed fp16/bf16 加法: src0 / src1 都是 32-bit reg 各装 2 个 fp16/bf16, 两 half 各自独立相加, 结果 packed 写 dst。整 warp 32 个 active thread 各自计算。

src_iswz (packed swizzle): hilo (默认两 half 各自参与) / lo (h0_h0 broadcast 低半字给两 half) / hi (h1_h1 broadcast) / b32 (src 是 FP32 → 转 fp16 truncation 复制到两 half, 计算仍走 fp16 path)。

type_fp16_bf16: 选 src / dst 浮点格式 (fp16 或 bf16)。bf16 模式下 ftz / saturate 是否生效由 implementation 决定; 跨 vendor 一致建议 bf16 下保持 default。

指令形态路径分两簇: f16.* = 双 half packed 输出 (主路径); f32.* = mixed-precision 仅算 src0.lo + src1.lo 的单个 fp16 加法, 再 widen 成 FP32 写整 32-bit dst (hi 半字不参与, dst 是 FP32 single value)。

ftz=disable: 走 IEEE 行为, denorm 保留。

ftz=ftz: input / output denorm flush 为 sign-preserving 0。

saturate=saturate: dst clamp 到 [+0.0, 1.0], NaN → +0.0 (ML activation 路径用)。

Mixed-precision lane 0 path: 算 src0.lo + src1.lo 的单个 fp16 add, 再 widen 成 FP32 写 dst (仅 lane 0 算, hi 半字不参与, dst 是 FP32 single value)。src_a 不挂 reuse。

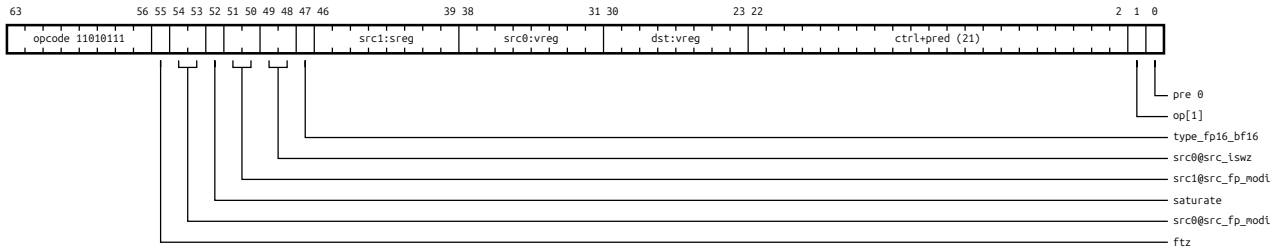
Leaf: hadd_pk__f32.v_vx

Format:

```
hadd_pk.ftz{.saturate}.type_fp16_bf16  dst, src0{.src_fp_modi}{.src_iswz},
↪  src1{.src_fp_modi}
```

Operands: dst: vreg; src0: vreg; src1: sreg

Encoding Diagram: 64b, prefix 0, opcode 111010111



Notes:

Packed fp16/bf16 加法: src0 / src1 都是 32-bit reg 各装 2 个 fp16/bf16, 两 half 各自独立相加, 结果 packed 写 dst。整 warp 32 个 active thread 各自计算。

src_iswz (packed swizzle): hilo (默认两 half 各自参与) / lo (h0_h0 broadcast 低半字给两 half) / hi (h1_h1 broadcast) / b32 (src 是 FP32 → 转 fp16 truncation 复制到两 half, 计算仍走 fp16 path)。

type_fp16_bf16: 选 src / dst 浮点格式 (fp16 或 bf16)。bf16 模式下 ftz / saturate 是否生效由 implementation 决定; 跨 vendor 一致建议 bf16 下保持 default。

指令形态路径分两簇: f16.* = 双 half packed 输出 (主路径); f32.* = mixed-precision 仅算 src0.lo + src1.lo 的单个 fp16 加法, 再 widen 成 FP32 写整 32-bit dst (hi 半字不参与, dst 是 FP32 single value)。

ftz=disable: 走 IEEE 行为, denorm 保留。

ftz=ftz: input / output denorm flush 为 sign-preserving 0。

saturate=saturate: dst clamp 到 [+0.0, 1.0], NaN → +0.0 (ML activation 路径用)。

hma_pk **category**: Packed and low-precision compute **predicate_mode**: simt

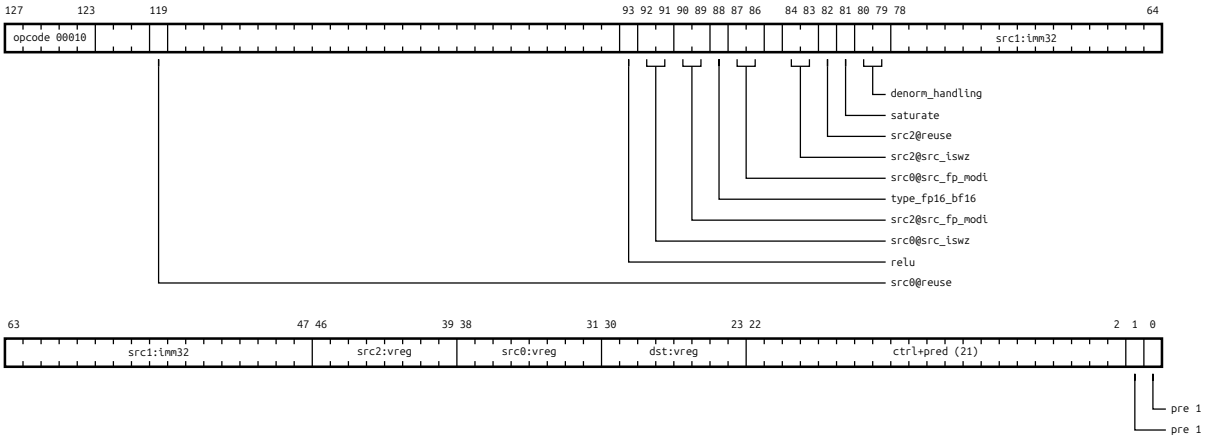
Leaf: hma_pk__f16.v_viv

Format:

```
hma_pk{.denorm_handling}{.saturate}.type_fp16_bf16{.relu}  dst,
↪  src0{.src_fp_modi}{.src_iswz}{.reuse}, src1, src2{.src_fp_modi}{.src_iswz}{.reuse}
```

Operands: dst: vreg; src0: vreg; src1: imm32u; src2: vreg

Encoding Diagram: 128b, prefix 11, opcode 00010



Notes:

Packed fp16/bf16 fused multiply-add: src0 / src1 / src2 各是 32-bit reg 装 2 个 fp16/bf16, 两 half 各自做 FMA (无限精度乘 + 加 + 单次 round), 结果 packed 写 dst。比 hmul_pk + hadd_pk 序列精度高。

src_iswz: hilo / lo / hi / b32, 跟 hadd_pk 一致。

type_fp16_bf16: fp16 或 bf16; bf16 下 fmz / saturate / relu 行为由 implementation 决定。

指令形态路径: f16.* = 双 half packed 输出 (主路径); f32.* = mixed-precision 仅算 src0.lo × src1.lo + src2.lo 的 fp16 FMA, widen 成 FP32 写 dst (hi 半字不参与)。

denorm_handling=keep: IEEE 默认, denorm 保留。ftz: input / output denorm flush 0。fmz: 任一 src=0 强制乘积 +0 (ML 量化路径常用)。

saturate=saturate: dst clamp 到 [+0.0, 1.0], NaN → +0.0。

relu=relu: dst 中负值 clamp 到 +0.0, NaN → canonical NaN (ML activation 高频)。saturate / relu 编码上共用 satrelu 字段, 互斥不能同时启用。

Packed imm 编码: 32-bit imm 装 2 个 fp16/bf16, [15:0]=H0, [31:16]=H1。

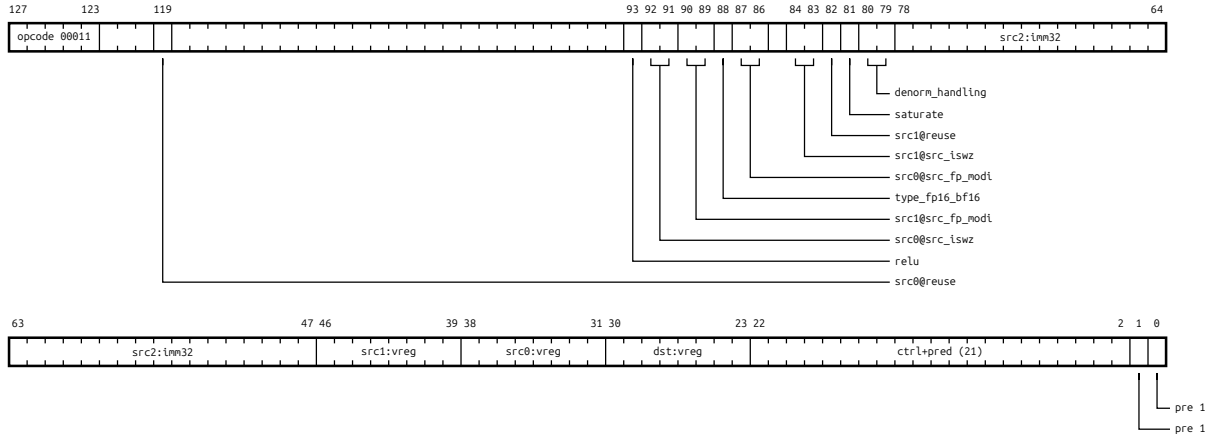
Leaf: hma_pk__f16.v_vvi

Format:

```
hma_pk{.denorm_handling}{.saturate}.type_fp16_bf16{.relu}  dst,
↪  src0{.src_fp_modi}{.src_iswz}{.reuse}, src1{.src_fp_modi}{.src_iswz}{.reuse}, src2
```

Operands: dst: vreg; src0: vreg; src1: vreg; src2: imm32u

Encoding Diagram: 128b, prefix 11, opcode 00011



Notes:

Packed fp16/bf16 fused multiply-add: src0 / src1 / src2 各是 32-bit reg 装 2 个 fp16/bf16, 两 half 各自做 FMA (无限精度乘 + 加 + 单次 round), 结果 packed 写 dst。比 hmul_pk + hadd_pk 序列精度高。

src_iswz: hilo / lo / hi / b32, 跟 hadd_pk 一致。

type_fp16_bf16: fp16 或 bf16; bf16 下 fmz / saturate / relu 行为由 implementation 决定。

指令形态路径: f16.* = 双 half packed 输出 (主路径); f32.* = mixed-precision 仅算 src0.lo × src1.lo + src2.lo 的 fp16 FMA, widen 成 FP32 写 dst (hi 半字不参与)。

denorm_handling=keep: IEEE 默认, denorm 保留。ftz: input / output denorm flush 0。fmz: 任一 src=0 强制乘积 +0 (ML 量化路径常用)。

saturate=saturate: dst clamp 到 [+0.0, 1.0], NaN → +0.0。

relu=relu: dst 中负值 clamp 到 +0.0, NaN → canonical NaN (ML activation 高频)。saturate / relu 编码上共用 satrelu 字段, 互斥不能同时启用。

Packed imm 编码: 32-bit imm 装 2 个 fp16/bf16, [15:0]=H0, [31:16]=H1。

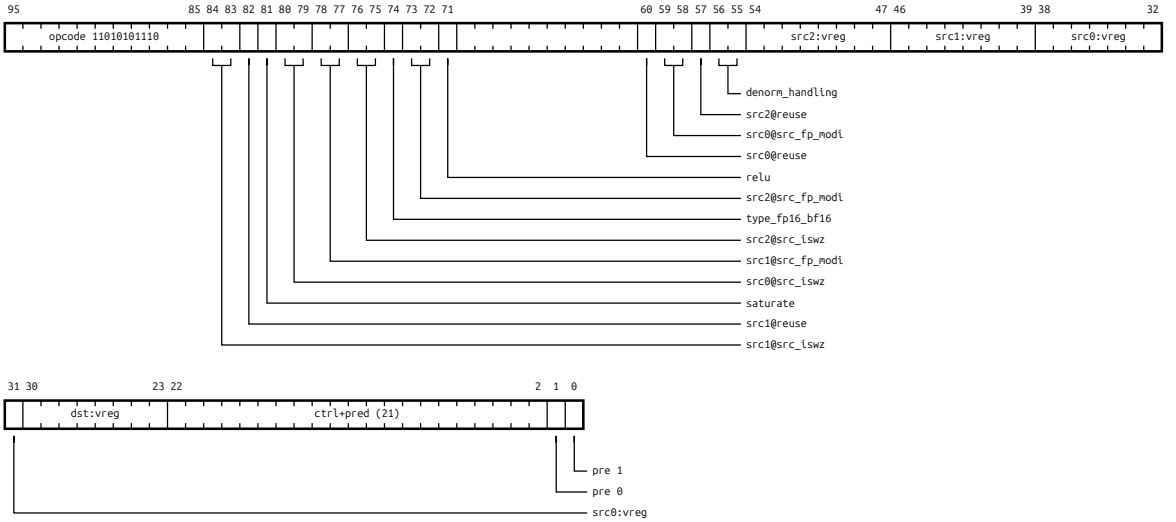
Leaf: hma_pk__f16.v_vvv

Format:

```
hma_pk{.denorm_handling}{.saturate}.type_fp16_bf16{.relu}  dst,
↪  src0{.src_fp_modi}{.src_iswz}{.reuse}, src1{.src_fp_modi}{.src_iswz}{.reuse},
↪  src2{.src_fp_modi}{.src_iswz}{.reuse}
```

Operands: dst: vreg; src0: vreg; src1: vreg; src2: vreg

Encoding Diagram: 96b, prefix 10, opcode 11010101110



Notes:

Packed fp16/bf16 fused multiply-add: src0 / src1 / src2 各是 32-bit reg 装 2 个 fp16/bf16, 两 half 各自做 FMA (无限精度乘 + 加 + 单次 round), 结果 packed 写 dst。比 hmul_pk + hadd_pk 序列精度高。

src_iswz: hilo / lo / hi / b32, 跟 hadd_pk 一致。

type_fp16_bf16: fp16 或 bf16; bf16 下 fmz / saturate / relu 行为由 implementation 决定。

指令形态路径: f16.* = 双 half packed 输出 (主路径); f32.* = mixed-precision 仅算 src0.lo × src1.lo + src2.lo 的 fp16 FMA, widen 成 FP32 写 dst (hi 半字不参与)。

denorm_handling=keep: IEEE 默认, denorm 保留。ftz: input / output denorm flush 0。fmz: 任一 src=0 强制乘积 +0 (ML 量化路径常用)。

saturate=saturate: dst clamp 到 [+0.0, 1.0], NaN → +0.0。

relu=relu: dst 中负值 clamp 到 +0.0, NaN → canonical NaN (ML activation 高频)。saturate / relu 编码上共用 satrelu 字段, 互斥不能同时启用。

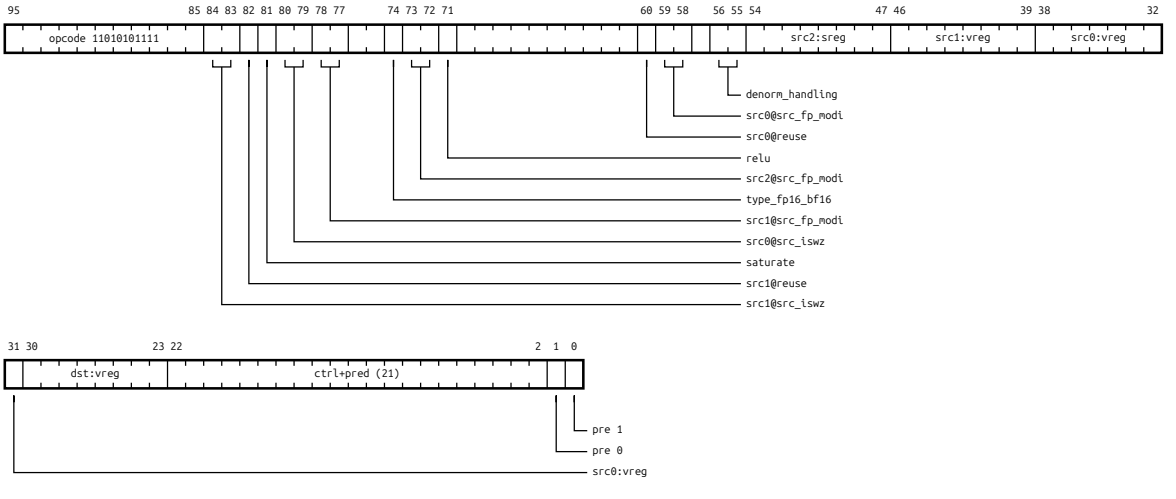
Leaf: hma_pk__f16.v_vvx

Format:

```
hma_pk{.denorm_handling}{.saturate}.type_fp16_bf16{.relu}  dst,
↪  src0{.src_fp_modi}{.src_iswz}{.reuse}, src1{.src_fp_modi}{.src_iswz}{.reuse},
↪  src2{.src_fp_modi}
```

Operands: dst: vreg; src0: vreg; src1: vreg; src2: sreg

Encoding Diagram: 96b, prefix 10, opcode 11010101111



Notes:

Packed fp16/bf16 fused multiply-add: src0 / src1 / src2 各是 32-bit reg 装 2 个 fp16/bf16, 两 half 各自做 FMA (无限精度乘 + 加 + 单次 round), 结果 packed 写 dst。比 hmul_pk + hadd_pk 序列精度高。

src_iswz: hilo / lo / hi / b32, 跟 hadd_pk 一致。

type_fp16_bf16: fp16 或 bf16; bf16 下 fmz / saturate / relu 行为由 implementation 决定。

指令形态路径: f16.* = 双 half packed 输出 (主路径); f32.* = mixed-precision 仅算 src0.lo × src1.lo + src2.lo 的 fp16 FMA, widen 成 FP32 写 dst (hi 半字不参与)。

denorm_handling=keep: IEEE 默认, denorm 保留。ftz: input / output denorm flush 0。fmz: 任一 src=0 强制乘积 +0 (ML 量化路径常用)。

saturate=saturate: dst clamp 到 [+0.0, 1.0], NaN → +0.0。

relu=relu: dst 中负值 clamp 到 +0.0, NaN → canonical NaN (ML activation 高频)。saturate / relu 编码上共用 satrelu 字段, 互斥不能同时启用。

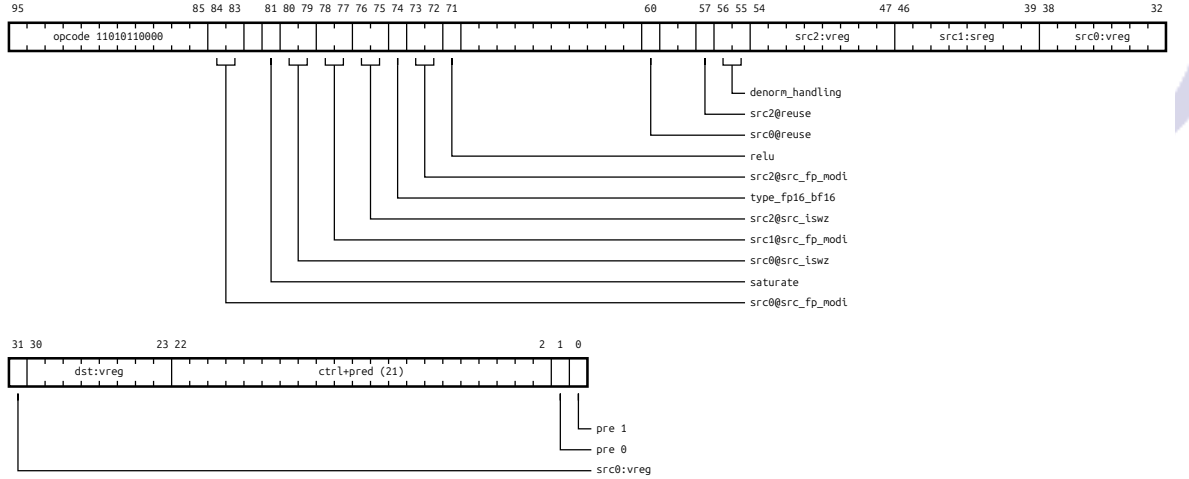
Leaf: hma_pk__f16.v_vxv

Format:

```
hma_pk{.denorm_handling}{.saturate}.type_fp16_bf16{.relu}  dst,
↪  src0{.src_fp_modi}{.src_iswz}{.reuse}, src1{.src_fp_modi},
↪  src2{.src_fp_modi}{.src_iswz}{.reuse}
```

Operands: dst: vreg; src0: vreg; src1: sreg; src2: vreg

Encoding Diagram: 96b, prefix 10, opcode 11010110000



Notes:

Packed fp16/bf16 fused multiply-add: src0 / src1 / src2 各是 32-bit reg 装 2 个 fp16/bf16, 两 half 各自做 FMA (无限精度乘 + 加 + 单次 round), 结果 packed 写 dst。比 hmul_pk + hadd_pk 序列精度高。

src_iswz: hilo / lo / hi / b32, 跟 hadd_pk 一致。

type_fp16_bf16: fp16 或 bf16; bf16 下 fmz / saturate / relu 行为由 implementation 决定。

指令形态路径: f16.* = 双 half packed 输出 (主路径); f32.* = mixed-precision 仅算 src0.lo × src1.lo + src2.lo 的 fp16 FMA, widen 成 FP32 写 dst (hi 半字不参与)。

denorm_handling=keep: IEEE 默认, denorm 保留。ftz: input / output denorm flush 0。fmz: 任一 src=0 强制乘积 +0 (ML 量化路径常用)。

saturate=saturate: dst clamp 到 [+0.0, 1.0], NaN → +0.0。

relu=relu: dst 中负值 clamp 到 +0.0, NaN → canonical NaN (ML activation 高频)。saturate / relu 编码上共用 satrelu 字段, 互斥不能同时启用。

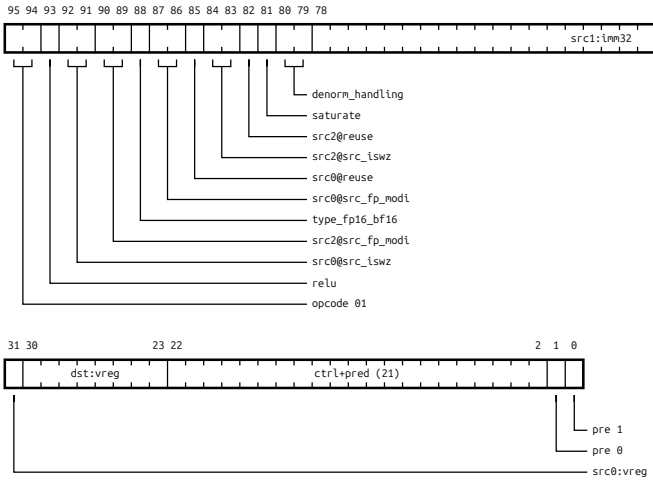
Leaf: hma_pk__f32.v_viv

Format:

```
hma_pk{.denorm_handling}{.saturate}.type_fp16_bf16{.relu}  dst,
↪  src0{.src_fp_modi}{.src_iswz}{.reuse}, src1, src2{.src_fp_modi}{.src_iswz}{.reuse}
```

Operands: dst: vreg; src0: vreg; src1: imm32u; src2: vreg

Encoding Diagram: 96b, prefix 10, opcode 01



Notes:

Packed fp16/bf16 fused multiply-add: src0 / src1 / src2 各是 32-bit reg 装 2 个 fp16/bf16, 两 half 各自做 FMA (无限精度乘 + 加 + 单次 round), 结果 packed 写 dst。比 hmul_pk + hadd_pk 序列精度高。

src_iswz: hilo / lo / hi / b32, 跟 hadd_pk 一致。

type_fp16_bf16: fp16 或 bf16; bf16 下 fmz / saturate / relu 行为由 implementation 决定。

指令形态路径: f16.* = 双 half packed 输出 (主路径); f32.* = mixed-precision 仅算 src0.lo × src1.lo + src2.lo 的 fp16 FMA, widen 成 FP32 写 dst (hi 半字不参与)。

denorm_handling=keep: IEEE 默认, denorm 保留。ftz: input / output denorm flush 0。fmz: 任一 src=0 强制乘积 +0 (ML 量化路径常用)。

saturate=saturate: dst clamp 到 [+0.0, 1.0], NaN → +0.0。

relu=relu: dst 中负值 clamp 到 +0.0, NaN → canonical NaN (ML activation 高频)。saturate / relu 编码上共用 satrelu 字段, 互斥不能同时启用。

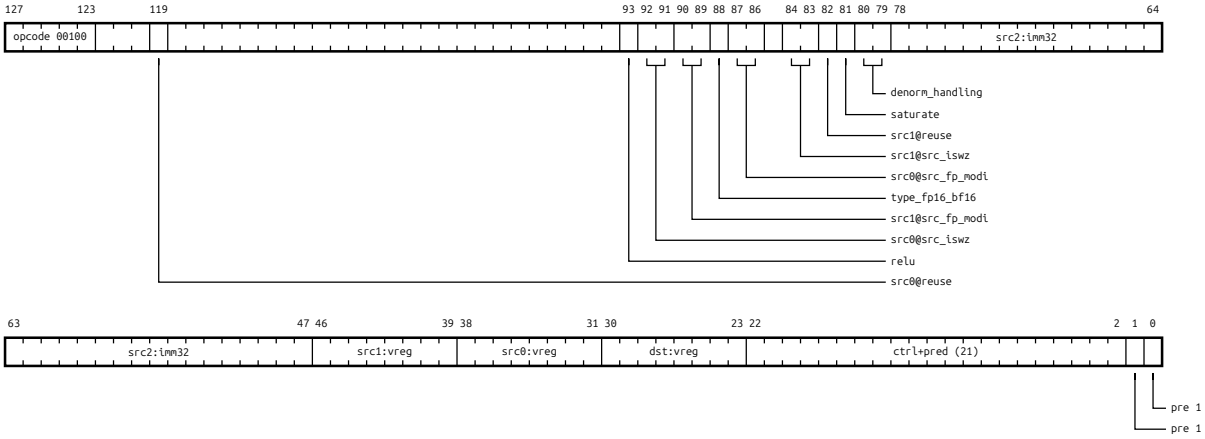
Leaf: hma_pk__f32.v_vvi

Format:

```
hma_pk{.denorm_handling}{.saturate}.type_fp16_bf16{.relu}  dst,
↪  src0{.src_fp_modi}{.src_iswz}{.reuse}, src1{.src_fp_modi}{.src_iswz}{.reuse}, src2
```

Operands: dst: vreg; src0: vreg; src1: vreg; src2: imm32u

Encoding Diagram: 128b, prefix 11, opcode 00100



Notes:

Packed fp16/bf16 fused multiply-add: src0 / src1 / src2 各是 32-bit reg 装 2 个 fp16/bf16, 两 half 各自做 FMA (无限精度乘 + 加 + 单次 round), 结果 packed 写 dst。比 hmul_pk + hadd_pk 序列精度高。

src_iswz: hilo / lo / hi / b32, 跟 hadd_pk 一致。

type_fp16_bf16: fp16 或 bf16; bf16 下 fmz / saturate / relu 行为由 implementation 决定。

指令形态路径: f16.* = 双 half packed 输出 (主路径); f32.* = mixed-precision 仅算 src0.lo × src1.lo + src2.lo 的 fp16 FMA, widen 成 FP32 写 dst (hi 半字不参与)。

denorm_handling=keep: IEEE 默认, denorm 保留。ftz: input / output denorm flush 0。fmz: 任一 src=0 强制乘积 +0 (ML 量化路径常用)。

saturate=saturate: dst clamp 到 [+0.0, 1.0], NaN → +0.0。

relu=relu: dst 中负值 clamp 到 +0.0, NaN → canonical NaN (ML activation 高频)。saturate / relu 编码上共用 satrelu 字段, 互斥不能同时启用。

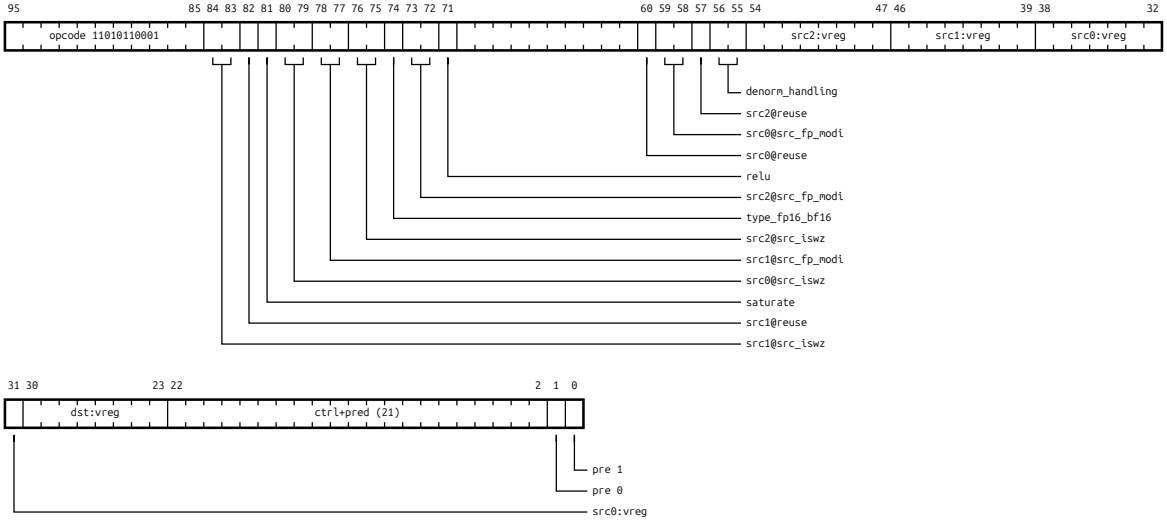
Leaf: hma_pk__f32.v_vvv

Format:

```
hma_pk{.denorm_handling}{.saturate}.type_fp16_bf16{.relu}  dst,
↪  src0{.src_fp_modi}{.src_iswz}{.reuse}, src1{.src_fp_modi}{.src_iswz}{.reuse},
↪  src2{.src_fp_modi}{.src_iswz}{.reuse}
```

Operands: dst: vreg; src0: vreg; src1: vreg; src2: vreg

Encoding Diagram: 96b, prefix 10, opcode 11010110001



Notes:

Packed fp16/bf16 fused multiply-add: src0 / src1 / src2 各是 32-bit reg 装 2 个 fp16/bf16, 两 half 各自做 FMA (无限精度乘 + 加 + 单次 round), 结果 packed 写 dst。比 hmul_pk + hadd_pk 序列精度高。

src_iswz: hilo / lo / hi / b32, 跟 hadd_pk 一致。

type_fp16_bf16: fp16 或 bf16; bf16 下 fmz / saturate / relu 行为由 implementation 决定。

指令形态路径: f16.* = 双 half packed 输出 (主路径); f32.* = mixed-precision 仅算 src0.lo × src1.lo + src2.lo 的 fp16 FMA, widen 成 FP32 写 dst (hi 半字不参与)。

denorm_handling=keep: IEEE 默认, denorm 保留。ftz: input / output denorm flush 0。fmz: 任一 src=0 强制乘积 +0 (ML 量化路径常用)。

saturate=saturate: dst clamp 到 [+0.0, 1.0], NaN → +0.0。

relu=relu: dst 中负值 clamp 到 +0.0, NaN → canonical NaN (ML activation 高频)。saturate / relu 编码上共用 satrelu 字段, 互斥不能同时启用。

Mixed-precision lane 0 path: 算 src0.lo × src1.lo + src2.lo 的单个 fp16 FMA, 再 widen 成 FP32 写 dst (.f32 ofmt 模式, 仅 lane 0 算, hi 半字不参与)。

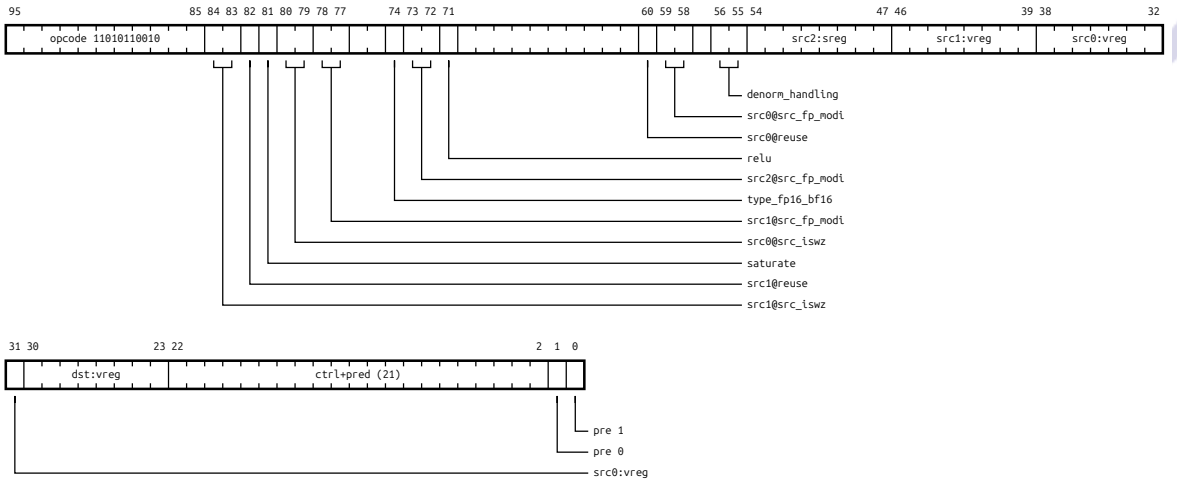
Leaf: hma_pk__f32.v_vvx

Format:

```
hma_pk{.denorm_handling}{.saturate}.type_fp16_bf16{.relu}  dst,
↪  src0{.src_fp_modi}{.src_iswz}{.reuse}, src1{.src_fp_modi}{.src_iswz}{.reuse},
↪  src2{.src_fp_modi}
```

Operands: dst: vreg; src0: vreg; src1: vreg; src2: sreg

Encoding Diagram: 96b, prefix 10, opcode 11010110010



Notes:

Packed fp16/bf16 fused multiply-add: src0 / src1 / src2 各是 32-bit reg 装 2 个 fp16/bf16, 两 half 各自做 FMA (无限精度乘 + 加 + 单次 round), 结果 packed 写 dst。比 hmul_pk + hadd_pk 序列精度高。

src_iswz: hilo / lo / hi / b32, 跟 hadd_pk 一致。

type_fp16_bf16: fp16 或 bf16; bf16 下 fmz / saturate / relu 行为由 implementation 决定。

指令形态路径: f16.* = 双 half packed 输出 (主路径); f32.* = mixed-precision 仅算 src0.lo × src1.lo + src2.lo 的 fp16 FMA, widen 成 FP32 写 dst (hi 半字不参与)。

denorm_handling=keep: IEEE 默认, denorm 保留。ftz: input / output denorm flush 0。fmz: 任一 src=0 强制乘积 +0 (ML 量化路径常用)。

saturate=saturate: dst clamp 到 [+0.0, 1.0], NaN → +0.0。

relu=relu: dst 中负值 clamp 到 +0.0, NaN → canonical NaN (ML activation 高频)。saturate / relu 编码上共用 satrelu 字段, 互斥不能同时启用。

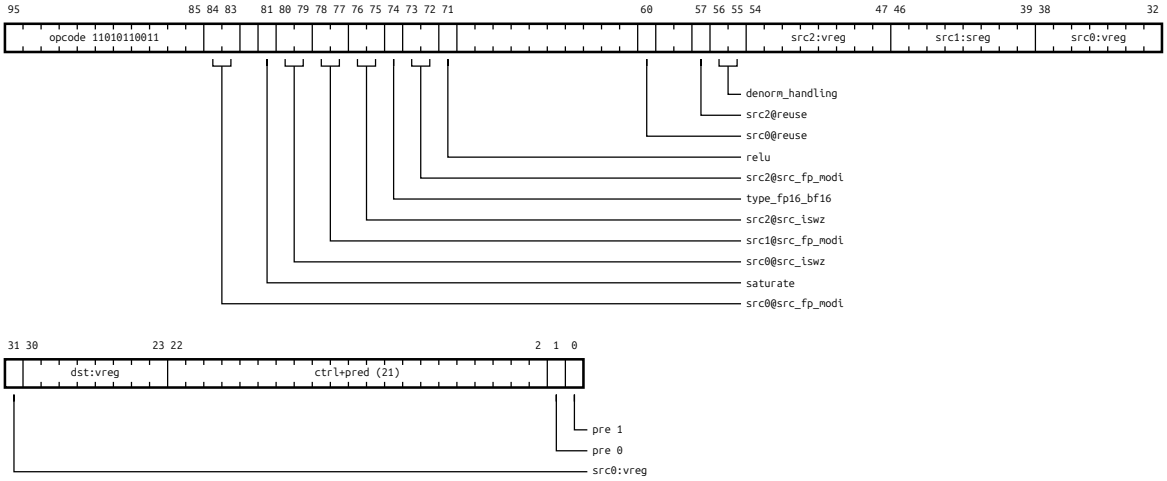
Leaf: hma_pk__f32.v_vxv

Format:

```
hma_pk{.denorm_handling}{.saturate}.type_fp16_bf16{.relu}  dst,
↪  src0{.src_fp_modi}{.src_iswz}{.reuse}, src1{.src_fp_modi},
↪  src2{.src_fp_modi}{.src_iswz}{.reuse}
```

Operands: dst: vreg; src0: vreg; src1: sreg; src2: vreg

Encoding Diagram: 96b, prefix 10, opcode 11010110011



Notes:

Packed fp16/bf16 fused multiply-add: src0 / src1 / src2 各是 32-bit reg 装 2 个 fp16/bf16, 两 half 各自做 FMA (无限精度乘 + 加 + 单次 round), 结果 packed 写 dst。比 hmul_pk + hadd_pk 序列精度高。

src_iswz: hilo / lo / hi / b32, 跟 hadd_pk 一致。

type_fp16_bf16: fp16 或 bf16; bf16 下 fmz / saturate / relu 行为由 implementation 决定。

指令形态路径: f16.* = 双 half packed 输出 (主路径); f32.* = mixed-precision 仅算 src0.lo × src1.lo + src2.lo 的 fp16 FMA, widen 成 FP32 写 dst (hi 半字不参与)。

denorm_handling=keep: IEEE 默认, denorm 保留。ftz: input / output denorm flush 0。fmz: 任一 src=0 强制乘积 +0 (ML 量化路径常用)。

saturate=saturate: dst clamp 到 [+0.0, 1.0], NaN → +0.0。

relu=relu: dst 中负值 clamp 到 +0.0, NaN → canonical NaN (ML activation 高频)。saturate / relu 编码上共用 satrelu 字段, 互斥不能同时启用。

hmnmx_pk　**category**: Packed and low-precision compute　**predicate_mode**: simt

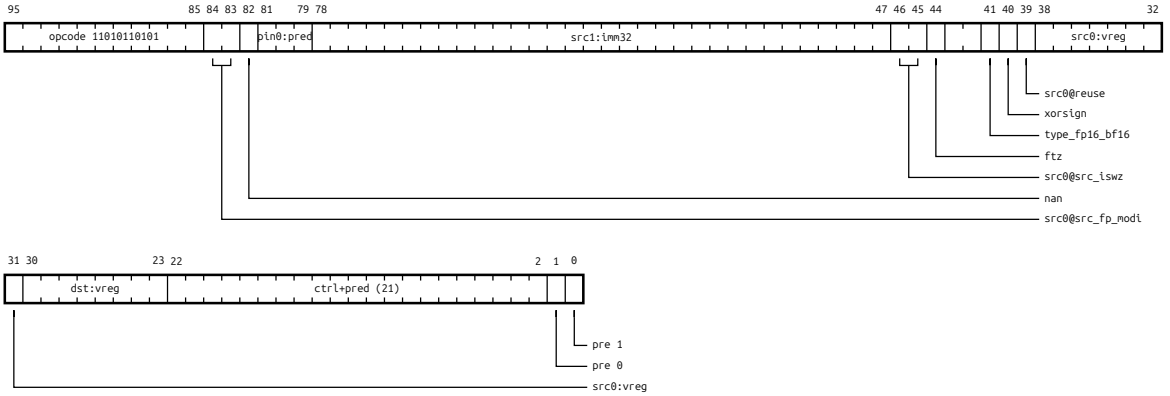
Leaf: hmnmx_pk__v_vi

Format:

```
hmnmx_pk.ftz{.nan}{.xorsign}.type_fp16_bf16  dst, src0{.src_fp_modi}{.src_iswz}{.reuse},
↪  src1, pin0
```

Operands: dst: vreg; src0: vreg; src1: imm32u; pin0: pred

Encoding Diagram: 96b, prefix 10, opcode 11010110101



Notes:

Packed fp16/bf16 per-half min/max: dst = pin0 ? pack_min(src0, src1) : pack_max(src0, src1), 每对 fp16 各自独立, pin0=true → 全 half 取 min, pin0=false → 全 half 取 max。pin0=PT 永远 min, pin0=!PT 永远 max。fmnmx 的 packed 镜像。

src_iswz: hilo / lo / hi / b32, 跟 hadd_pk 一致。

type_fp16_bf16: fp16 或 bf16; bf16 下 ftz / xorsign 行为由 implementation 决定。

ftz: ftz 时 input / output denorm flush 0。

NaN: 控制 NaN 输入传播行为 (DEFAULT / nan), 对应 IEEE 754-2008 min/max 规则。

xorsign: 跟 fmnmx 一致, 把 src1 sign-bit xor 进结果 sign-bit (ML pyramid sign tracking 用)。

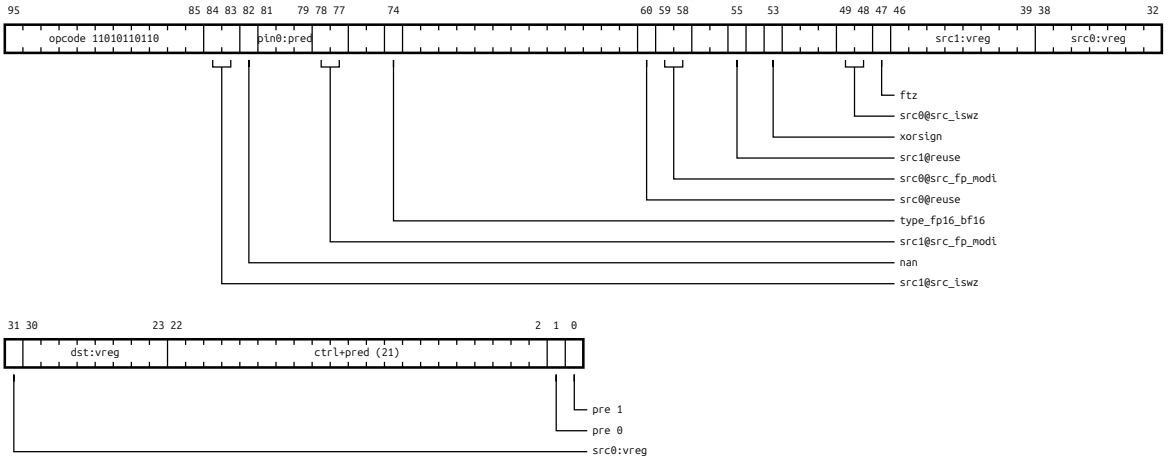
Leaf: hmnmx_pk__v_vv

Format:

```
hmnmx_pk.ftz{.nan}{.xorsign}.type_fp16_bf16  dst, src0{.src_fp_modi}{.src_iswz}{.reuse},
↪  src1{.src_fp_modi}{.src_iswz}{.reuse}, pin0
```

Operands: dst: vreg; src0: vreg; src1: vreg; pin0: pred

Encoding Diagram: 96b, prefix 10, opcode 11010110110



Notes:

Packed fp16/bf16 per-half min/max: dst = pin0 ? pack_min(src0, src1) : pack_max(src0, src1), 每对 fp16 各自独立, pin0=true → 全 half 取 min, pin0=false → 全 half 取 max。pin0=PT 永远 min, pin0=!PT 永远 max。fmnmx 的 packed 镜像。

src_iswz: hilo / lo / hi / b32, 跟 hadd_pk 一致。

type_fp16_bf16: fp16 或 bf16; bf16 下 ftz / xorsign 行为由 implementation 决定。

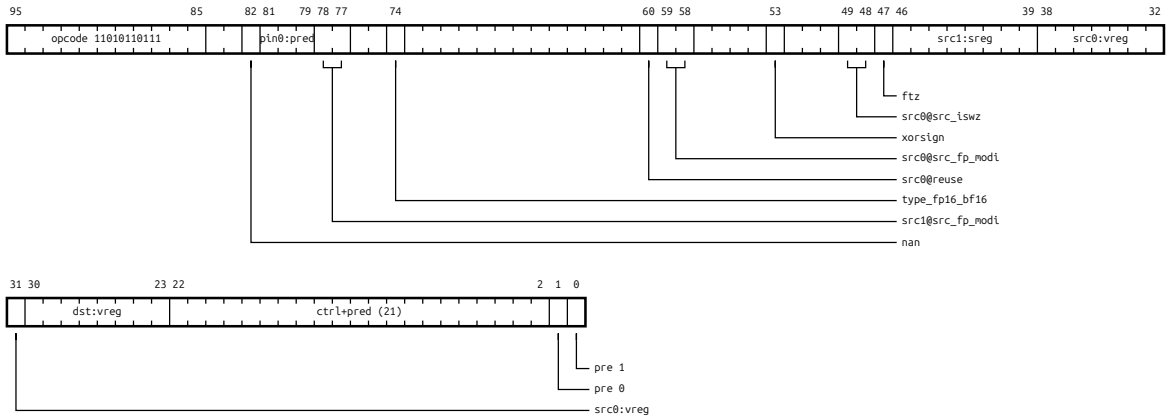
ftz: ftz 时 input / output denorm flush 0。

NaN: 控制 NaN 输入传播行为 (DEFAULT / nan), 对应 IEEE 754-2008 min/max 规则。
xorsign: 跟 fmnmx 一致, 把 src1 sign-bit xor 进结果 sign-bit (ML pyramid sign tracking 用)。

Leaf: hmnmx_pk__v_vx
Format:

```
hmnmx_pk.ftz{.nan}{.xorsign}.type_fp16_bf16  dst, src0{.src_fp_modi}{.src_iswz}{.reuse},
↪  src1{.src_fp_modi}, pin0
```

Operands: dst: vreg; src0: vreg; src1: sreg; pin0: pred
Encoding Diagram: 96b, prefix 10, opcode 11010110111



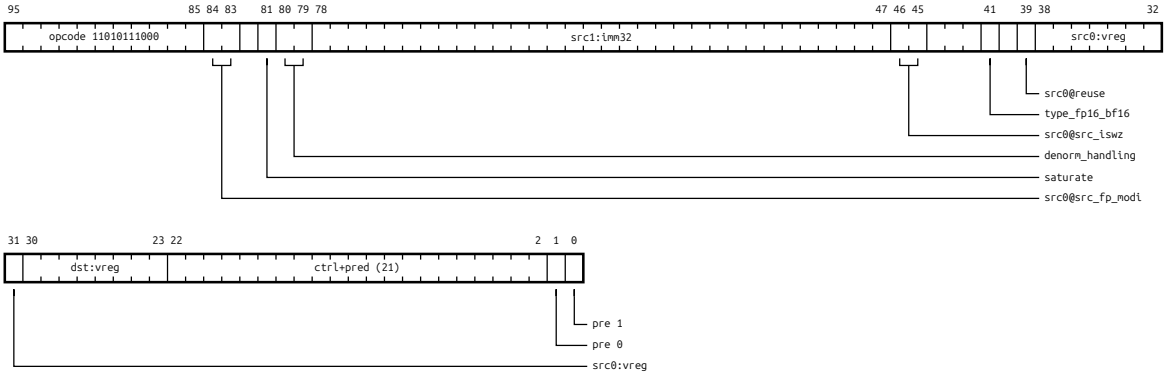
Notes:
Packed fp16/bf16 per-half min/max: dst = pin0 ? pack_min(src0, src1) : pack_max(src0, src1), 每对 fp16 各自独立, pin0=true → 全 half 取 min, pin0=false → 全 half 取 max。pin0=PT 永远 min, pin0=!PT 永远 max。fmnmx 的 packed 镜像。
src_iswz: hilo / lo / hi / b32, 跟 hadd_pk 一致。
type_fp16_bf16: fp16 或 bf16; bf16 下 ftz / xorsign 行为由 implementation 决定。
ftz: ftz 时 input / output denorm flush 0。
NaN: 控制 NaN 输入传播行为 (DEFAULT / nan), 对应 IEEE 754-2008 min/max 规则。
xorsign: 跟 fmnmx 一致, 把 src1 sign-bit xor 进结果 sign-bit (ML pyramid sign tracking 用)。

hmul_pk **category**: Packed and low-precision compute **predicate_mode**: simt

Leaf: hmul_pk__v_vi
Format:

```
hmul_pk{.denorm_handling}{.saturate}.type_fp16_bf16  dst,
↪  src0{.src_fp_modi}{.src_iswz}{.reuse}, src1
```

Operands: dst: vreg; src0: vreg; src1: imm32u
Encoding Diagram: 96b, prefix 10, opcode 11010111000



Notes:

Packed fp16/bf16 乘法: src0 / src1 各是 32-bit reg 装 2 个 fp16/bf16, 每对 fp16 分别相乘, 结果 packed 写 dst。

src_iswz: hilo / lo / hi / b32, 跟 hadd_pk 一致。

type_fp16_bf16: fp16 或 bf16; bf16 下 fmz / saturate 行为由 implementation 决定。

denorm_handling=keep: IEEE 默认。ftz: input / output denorm flush 0。fmz: 任一 src=0 强制乘积 +0 (跟 fmul 一致)。

saturate=saturate: dst clamp 到 [+0.0, 1.0], NaN → +0.0。

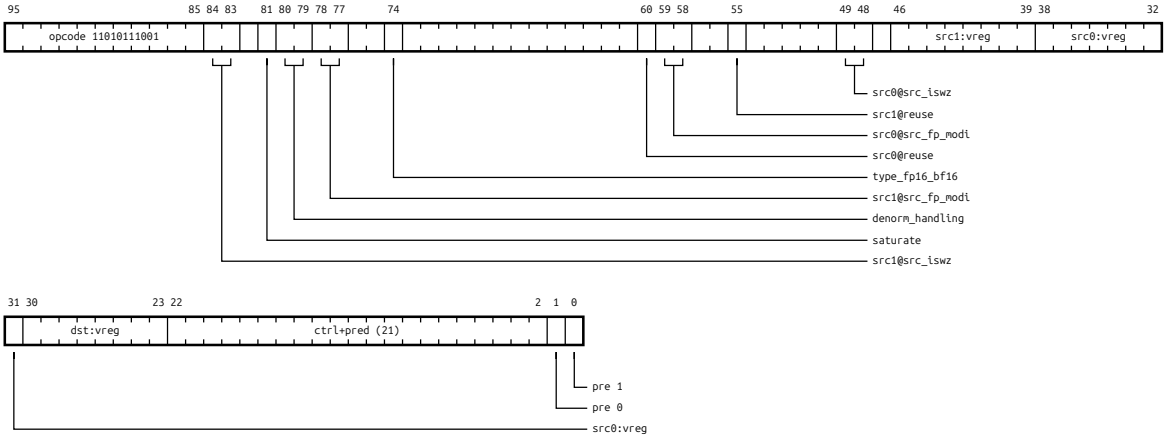
Leaf: hmul_pk__v_vv

Format:

```
hmul_pk{.denorm_handling}{.saturate}.type_fp16_bf16  dst,
↪  src0{.src_fp_modi}{.src_iswz}{.reuse}, src1{.src_fp_modi}{.src_iswz}{.reuse}
```

Operands: dst: vreg; src0: vreg; src1: vreg

Encoding Diagram: 96b, prefix 10, opcode 11010111001



Notes:

Packed fp16/bf16 乘法: src0 / src1 各是 32-bit reg 装 2 个 fp16/bf16, 每对 fp16 分别相乘, 结果 packed 写 dst。

src_iswz: hilo / lo / hi / b32, 跟 hadd_pk 一致。

type_fp16_bf16: fp16 或 bf16; bf16 下 fmz / saturate 行为由 implementation 决定。

denorm_handling=keep: IEEE 默认。ftz: input / output denorm flush 0。fmz: 任一 src=0 强制乘积 +0 (跟 fmul 一致)。

saturate=saturate: dst clamp 到 [+0.0, 1.0], NaN → +0.0。

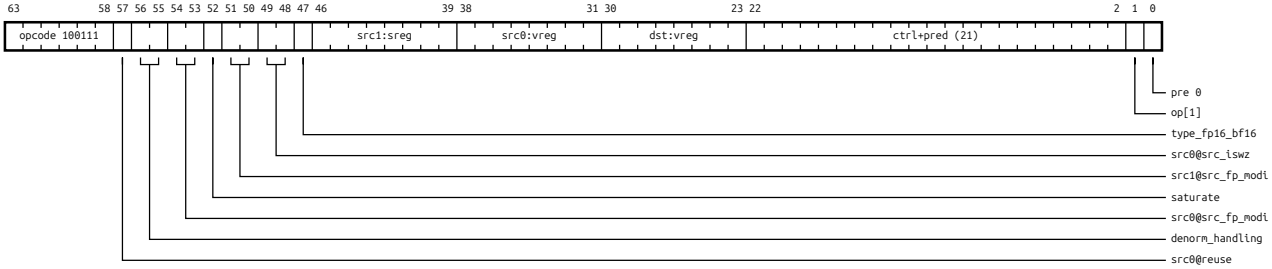
Leaf: hmul_pk__v_vx

Format:

```
hmul_pk{.denorm_handling}{.saturate}.type_fp16_bf16  dst,
↪  src0{.src_fp_modi}{.src_iswz}{.reuse}, src1{.src_fp_modi}
```

Operands: dst: vreg; src0: vreg; src1: sreg

Encoding Diagram: 64b, prefix 0, opcode 1100111



Notes:

Packed fp16/bf16 乘法: src0 / src1 各是 32-bit reg 装 2 个 fp16/bf16, 每对 fp16 分别相乘, 结果 packed 写 dst。

src_iswz: hilo / lo / hi / b32, 跟 hadd_pk 一致。

type_fp16_bf16: fp16 或 bf16; bf16 下 fmz / saturate 行为由 implementation 决定。

denorm_handling=keep: IEEE 默认。ftz: input / output denorm flush 0。fmz: 任一 src=0 强制乘积 +0 (跟 fmul 一致)。

saturate=saturate: dst clamp 到 [+0.0, 1.0], NaN → +0.0。

hset2　**category**: Packed and low-precision compute　**predicate_mode**: simt

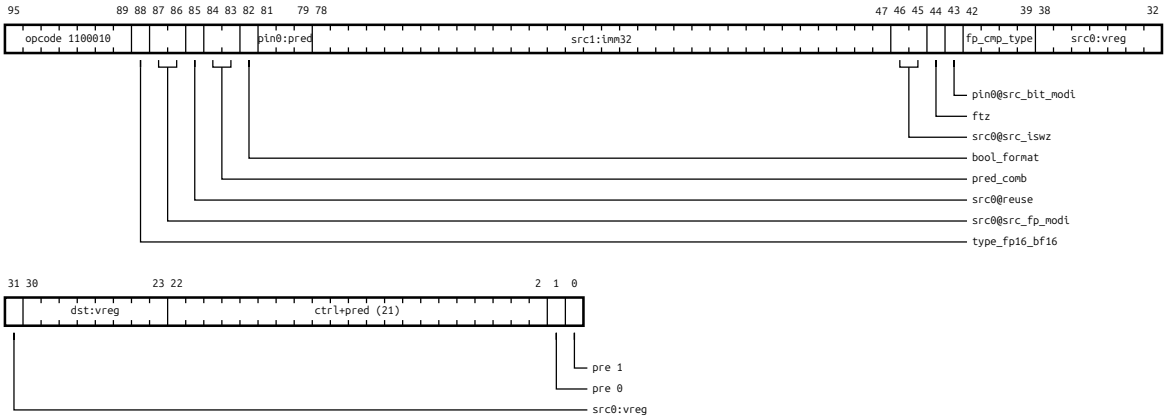
Leaf: hset2__v_vip

Format:

```
hset2.ftz.fp_cmp_type.pred_comb.type_fp16_bf16.bool_format  dst,
↪  src0{.src_fp_modi}{.src_iswz}{.reuse}, src1, pin0{.src_bit_modi}
```

Operands: dst: vreg; src0: vreg; src1: imm32u; pin0: pred

Encoding Diagram: 96b, prefix 10, opcode 1100010



Notes:

Packed fp16/bf16 比较 → vreg: 每对 fp16 各自做 (src0[half] CMP src1[half]), 跟 pin0 用 pred_comb 组合, 结果按 bool_format 写到 dst 对应的 16-bit 半。

src_iswz: hilo / lo / hi / b32, 跟 hadd_pk 一致。

type_fp16_bf16: fp16 或 bf16; bf16 下 ftz 行为由 implementation 决定。

fp_cmp_type: 16 种比较 (lt / le / eq / ne / ge / gt / num / nan / f / t 等), 跟 fset 一致。

pred_comb: and / or / xor; pin0=PT + pred_comb=and 时退化为 cmp 自身。

bool_format=bm (mask): true → 0xFFFF, false → 0 (packed 各 half 独立)。

bool_format=bf (boolean float): true → 1.0, false → 0.0 (fp 表示)。

ftz=ftz: input denorm flush 0 后再比较。

Packed imm 编码: 32-bit imm 装 2 个 fp16/bf16, [15:0]=H0, [31:16]=H1。

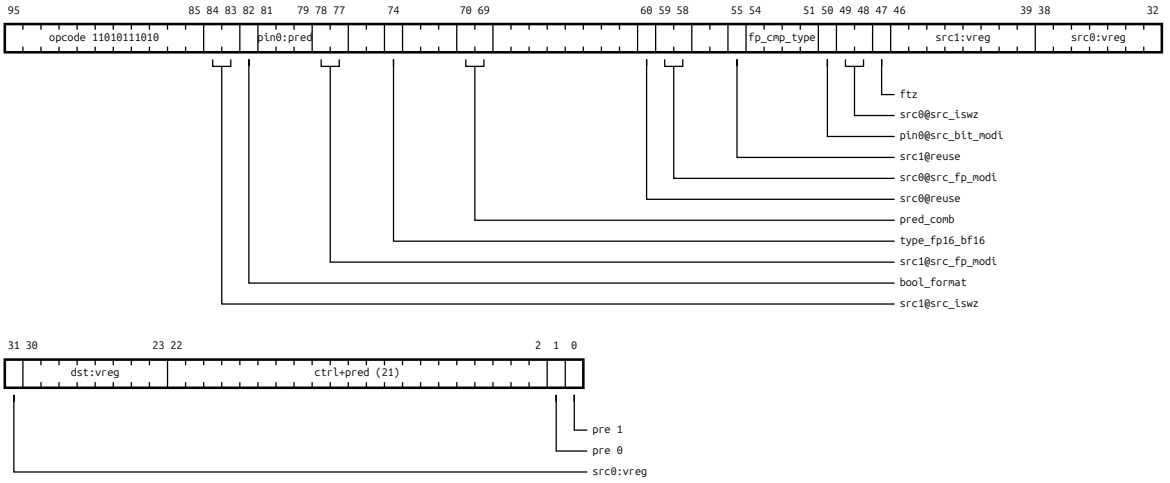
Leaf: hset2__v_vvp

Format:

```
hset2.ftz.fp_cmp_type.pred_comb.type_fp16_bf16.bool_format  dst,
↪  src0{.src_fp_modi}{.src_iswz}{.reuse}, src1{.src_fp_modi}{.src_iswz}{.reuse},
↪  pin0{.src_bit_modi}
```

Operands: dst: vreg; src0: vreg; src1: vreg; pin0: pred

Encoding Diagram: 96b, prefix 10, opcode 11010111010



Notes:

Packed fp16/bf16 比较 → vreg: 每对 fp16 各自做 (src0[half] CMP src1[half]), 跟 pin0 用 pred_comb 组合, 结果按 bool_format 写到 dst 对应的 16-bit 半。

src_iswz: hilo / lo / hi / b32, 跟 hadd_pk 一致。

type_fp16_bf16: fp16 或 bf16; bf16 下 ftz 行为由 implementation 决定。

fp_cmp_type: 16 种比较 (lt / le / eq / ne / ge / gt / num / nan / f / t 等), 跟 fset 一致。

pred_comb: and / or / xor; pin0=PT + pred_comb=and 时退化为 cmp 自身。

bool_format=bm (mask): true → 0xFFFF, false → 0 (packed 各 half 独立)。

bool_format=bf (boolean float): true → 1.0, false → 0.0 (fp 表示)。

ftz=ftz: input denorm flush 0 后再比较。

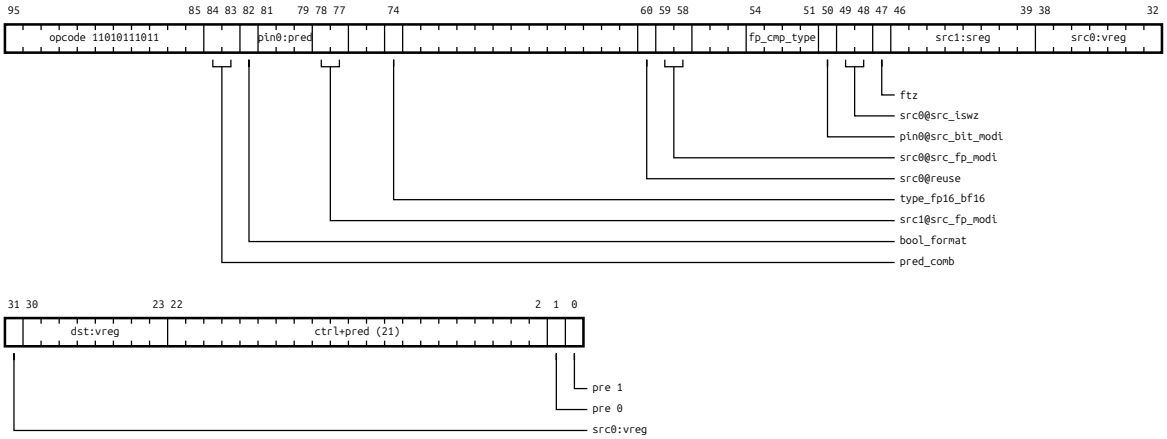
Leaf: hset2__v_vxp

Format:

```
hset2.ftz.fp_cmp_type.pred_comb.type_fp16_bf16.bool_format  dst,
↪  src0{.src_fp_modi}{.src_iswz}{.reuse}, src1{.src_fp_modi}, pin0{.src_bit_modi}
```

Operands: dst: vreg; src0: vreg; src1: sreg; pin0: pred

Encoding Diagram: 96b, prefix 10, opcode 11010111011



Notes:

Packed fp16/bf16 比较 → vreg: 每对 fp16 各自做 (src0[half] CMP src1[half]), 跟 pin0 用 pred_comb 组合, 结果按 bool_format 写到 dst 对应的 16-bit 半。

src_iswz: hilo / lo / hi / b32, 跟 hadd_pk 一致。

type_fp16_bf16: fp16 或 bf16; bf16 下 ftz 行为由 implementation 决定。

fp_cmp_type: 16 种比较 (lt / le / eq / ne / ge / gt / num / nan / f / t 等), 跟 fset 一致。

pred_comb: and / or / xor; pin0=PT + pred_comb=and 时退化为 cmp 自身。

bool_format=bm (mask): true → 0xFFFF, false → 0 (packed 各 half 独立)。

bool_format=bf (boolean float): true → 1.0, false → 0.0 (fp 表示)。

ftz=ftz: input denorm flush 0 后再比较。

hsetp2 **category**: Packed and low-precision compute **predicate_mode**: simt

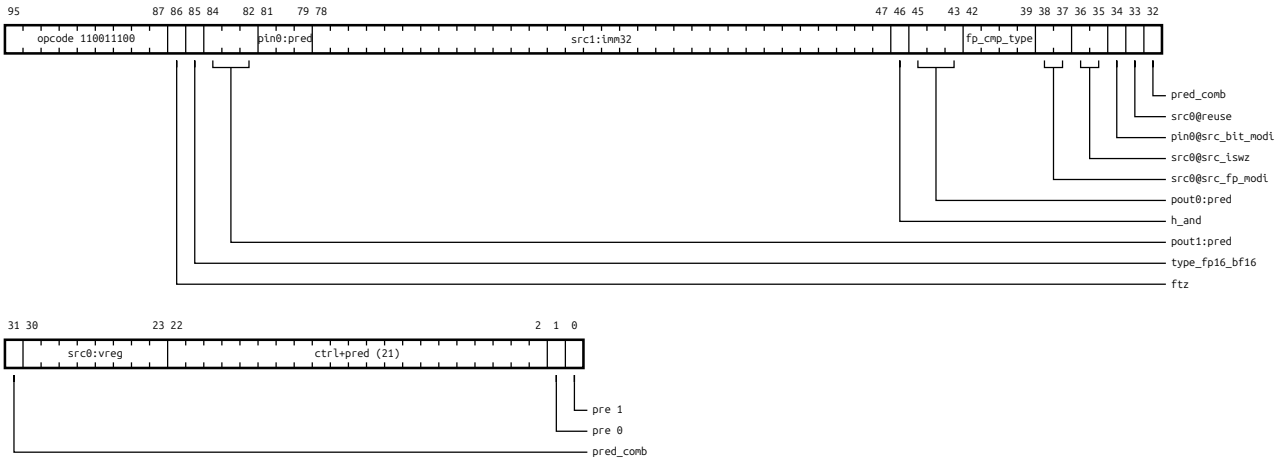
Leaf: hsetp2__pp_vip

Format:

```
hsetp2.ftz.fp_cmp_type.pred_comb.type_fp16_bf16{.h_and}  pout0, pout1,
↪  src0{.src_fp_modi}{.src_iswz}{.reuse}, src1, pin0{.src_bit_modi}
```

Operands: pout0: pred; pout1: pred; src0: vreg; src1: imm32u; pin0: pred

Encoding Diagram: 96b, prefix 10, opcode 110011100



Notes:

Packed fp16/bf16 比较 → 双 pred: 每对 fp16 各自做 (src0[half] CMP src1[half]), 得 cmp_lo / cmp_hi 两

boolean。

h_and=disable (默认): pout0 = pred_comb(cmp_lo, pin0), pout1 = pred_comb(cmp_hi, pin0), 两 half 各自独立输出 pred。

h_and=h_and: 先做 cmp_and = cmp_lo and cmp_hi 横向 reduce, 然后 pout0 = pred_comb(cmp_and, pin0), 单 reduced 输出 (用于 “两 half 都满足” 的 packed reduction)。

跟 fsetp 双 pred 区别: fsetp 是 pout1 = pred_comb(!cmp, pin0) 同一 cmp 反向组合 (嵌套谓词模式); hsetp2 默认是两 half 各 cmp 独立输出 (packed pair 模式)。

src_iswz: hilo / lo / hi / b32, 跟 hadd_pk 一致。

type_fp16_bf16: fp16 或 bf16; bf16 下 ftz 行为由 implementation 决定。

fp_cmp_type: 16 种 (lt / le / eq / ne / ge / gt / num / nan / f / t 等), 跟 fset 一致。

pred_comb: and / or / xor; pin0=PT 时退化为 identity。

ftz=ftz: input denorm flush 0 后再比较。

Packed imm 编码: 32-bit imm 装 2 个 fp16/bf16, [15:0]=H0, [31:16]=H1。

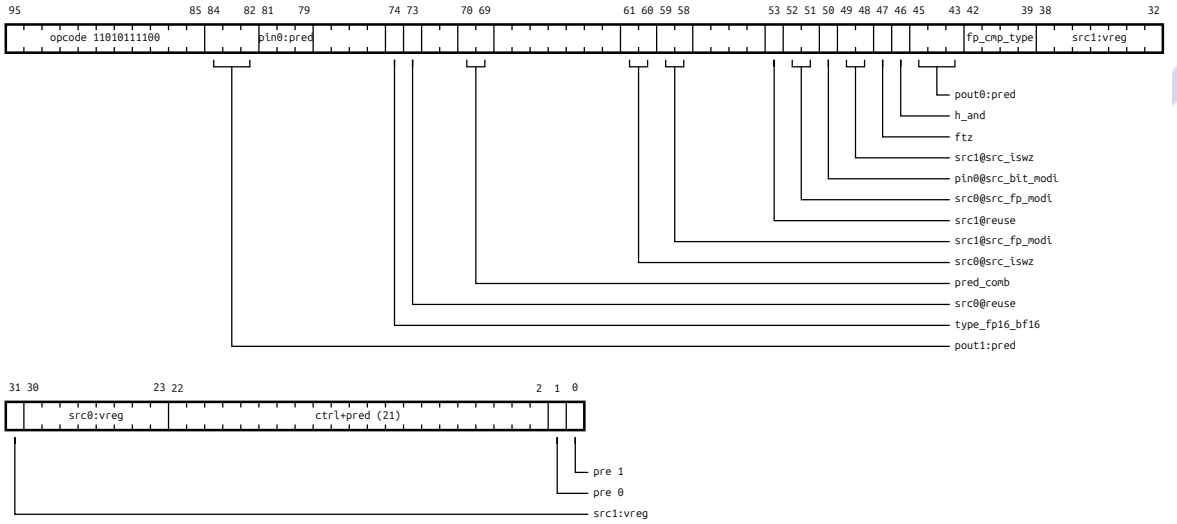
Leaf: hsetp2__pp_vvp

Format:

```
hsetp2.ftz.fp_cmp_type.pred_comb.type_fp16_bf16{.h_and}  pout0, pout1,
↪  src0{.src_fp_modi}{.src_iswz}{.reuse}, src1{.src_fp_modi}{.src_iswz}{.reuse},
↪  pin0{.src_bit_modi}
```

Operands: pout0: pred; pout1: pred; src0: vreg; src1: vreg; pin0: pred

Encoding Diagram: 96b, prefix 10, opcode 11010111100



Notes:

Packed fp16/bf16 比较 → 双 pred: 每对 fp16 各自做 (src0[half] CMP src1[half]), 得 cmp_lo / cmp_hi 两 boolean。

h_and=disable (默认): pout0 = pred_comb(cmp_lo, pin0), pout1 = pred_comb(cmp_hi, pin0), 两 half 各自独立输出 pred。

h_and=h_and: 先做 cmp_and = cmp_lo and cmp_hi 横向 reduce, 然后 pout0 = pred_comb(cmp_and, pin0), 单 reduced 输出 (用于 “两 half 都满足” 的 packed reduction)。

跟 fsetp 双 pred 区别: fsetp 是 pout1 = pred_comb(!cmp, pin0) 同一 cmp 反向组合 (嵌套谓词模式); hsetp2 默认是两 half 各 cmp 独立输出 (packed pair 模式)。

src_iswz: hilo / lo / hi / b32, 跟 hadd_pk 一致。

type_fp16_bf16: fp16 或 bf16; bf16 下 ftz 行为由 implementation 决定。

fp_cmp_type: 16 种 (lt / le / eq / ne / ge / gt / num / nan / f / t 等), 跟 fset 一致。

pred_comb: and / or / xor; pin0=PT 时退化为 identity。

ftz=ftz: input denorm flush 0 后再比较。

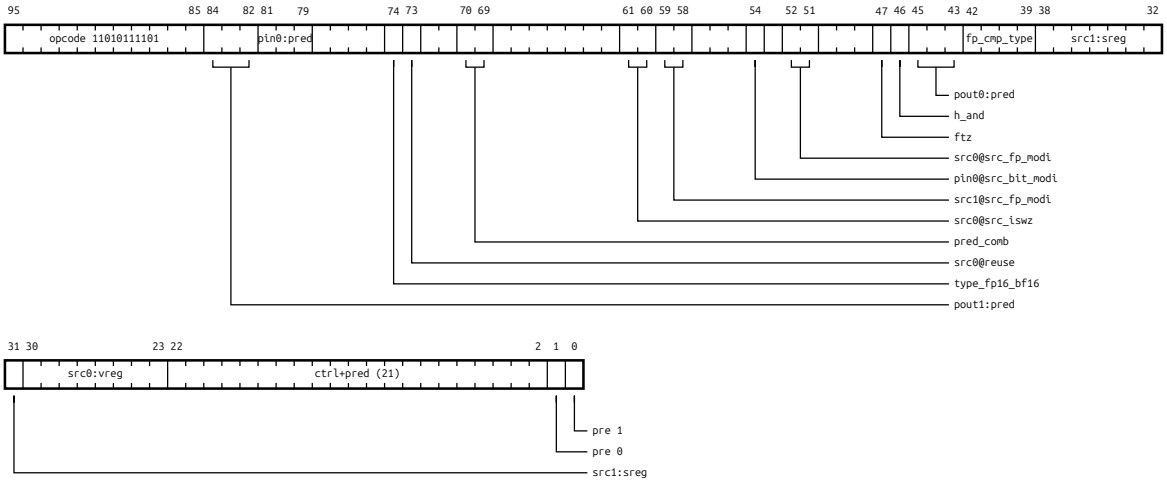
Leaf: hsetp2__pp_vxp

Format:

```
hsetp2.ftz.fp_cmp_type.pred_comb.type_fp16_bf16{.h_and}  pout0, pout1,
↪  src0{.src_fp_modi}{.src_iswz}{.reuse}, src1{.src_fp_modi}, pin0{.src_bit_modi}
```

Operands: pout0: pred; pout1: pred; src0: vreg; src1: sreg; pin0: pred

Encoding Diagram: 96b, prefix 10, opcode 11010111101



Notes:

Packed fp16/bf16 比较 → 双 pred: 每对 fp16 各自做 (src0[half] CMP src1[half]), 得 cmp_lo / cmp_hi 两 boolean。

h_and=disable (默认): pout0 = pred_comb(cmp_lo, pin0), pout1 = pred_comb(cmp_hi, pin0), 两 half 各自独立输出 pred。

h_and=h_and: 先做 cmp_and = cmp_lo and cmp_hi 横向 reduce, 然后 pout0 = pred_comb(cmp_and, pin0), 单 reduced 输出 (用于 “两 half 都满足” 的 packed reduction)。

跟 fsetp 双 pred 区别: fsetp 是 pout1 = pred_comb(!cmp, pin0) 同一 cmp 反向组合 (嵌套谓词模式); hsetp2 默认是两 half 各 cmp 独立输出 (packed pair 模式)。

src_iswz: hilo / lo / hi / b32, 跟 hadd_pk 一致。

type_fp16_bf16: fp16 或 bf16; bf16 下 ftz 行为由 implementation 决定。

fp_cmp_type: 16 种 (lt / le / eq / ne / ge / gt / num / nan / f / t 等), 跟 fset 一致。

pred_comb: and / or / xor; pin0=PT 时退化为 identity。

ftz=ftz: input denorm flush 0 后再比较。

14.7.4 Integer compute

iabs **category**: Integer compute **predicate_mode**: simt

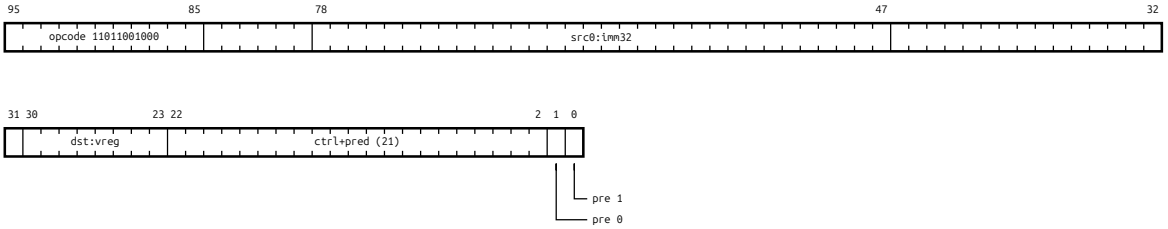
Leaf: iabs__v_i

Format:

```
iabs  dst, src0
```

Operands: dst: vreg; src0: imm32u

Encoding Diagram: 96b, prefix 10, opcode 11011001000



Notes:

32-bit 有符号整数绝对值: dst = abs(src0)。整 warp per-lane 各自计算。

src0 = INT_MIN (-2^31 = 0x80000000) 时无对应正数表示, 结果按 wrap 仍是 INT_MIN (常见 2's complement 行为, 编译器需要时自行处理 overflow)。

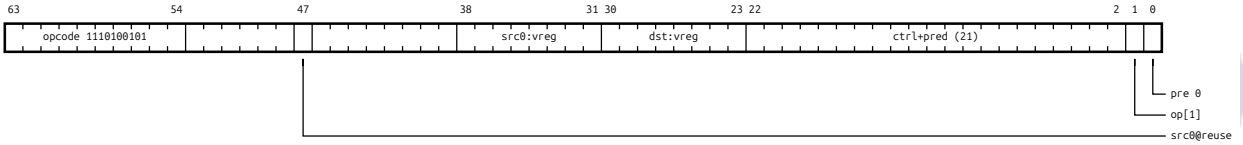
Leaf: iabs__v_v

Format:

```
iabs  dst, src0{.reuse}
```

Operands: dst: vreg; src0: vreg

Encoding Diagram: 64b, prefix 0, opcode 11110100101



Notes:

32-bit 有符号整数绝对值: dst = abs(src0)。整 warp per-lane 各自计算。

src0 = INT_MIN (-2^31 = 0x80000000) 时无对应正数表示, 结果按 wrap 仍是 INT_MIN (常见 2's complement 行为, 编译器需要时自行处理 overflow)。

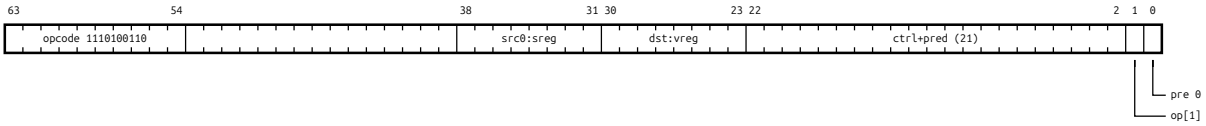
Leaf: iabs__v_x

Format:

```
iabs  dst, src0
```

Operands: dst: vreg; src0: sreg

Encoding Diagram: 64b, prefix 0, opcode 11110100110



Notes:

32-bit 有符号整数绝对值: dst = abs(src0)。整 warp per-lane 各自计算。

src0 = INT_MIN (-2^31 = 0x80000000) 时无对应正数表示, 结果按 wrap 仍是 INT_MIN (常见 2's complement 行为, 编译器需要时自行处理 overflow)。

iadd3　**category**: Integer compute　**predicate_mode**: simt

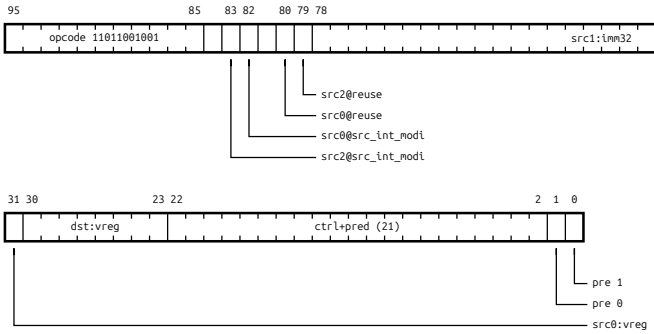
Leaf: iadd3__v_viv

Format:

```
iadd3  dst, src0{.src_int_modi}{.reuse}, src1, src2{.src_int_modi}{.reuse}
```

Operands: dst: vreg; src0: vreg; src1: imm32u; src2: vreg

Encoding Diagram: 96b, prefix 10, opcode 11011001001



Notes:

三入整数加法: 普通指令形态 (v_*) dst = src0 + src1 + src2 (mod 2^32, 无 carry); carry mode 指令形态 (vpp_*pp) {pout1, pout0, dst} = src0 + src1 + src2 + (2·pin1 + pin0), 双 cout / 双 cin 形成 multi-precision 链。整 warp per-lane 各自计算。

包含 2-src add 用例: src2=RZ 退化为 dst = src0 + src1 (替代单独的 iadd / iaddc mnemonic)。

普通指令形态 src 修饰符: src_int_modi=id / neg / abs / abs_then_neg, 实现 a-b-c / a+|b|+c 等变体 (neg = 取负 = 加补码, 加法层面跟 a + (-b) 等价)。

双 carry chain 物理含义: 3-integer add 最大值 3·(2^32-1) 自然产生 2-bit cout (∈ {0,1,2}), 所以下一 chunk 也接 2-bit cin (cout = 2·pout1 + pout0, cin = 2·pin1 + pin0)。

carry mode src 修饰符: src_bit_modi=id / inv (按位取反); 扩展精度减法用 a + ~b + cin chain 自然实现 (跟普通指令形态的 src_int_modi 不同, carry mode 不接受 neg/abs)。

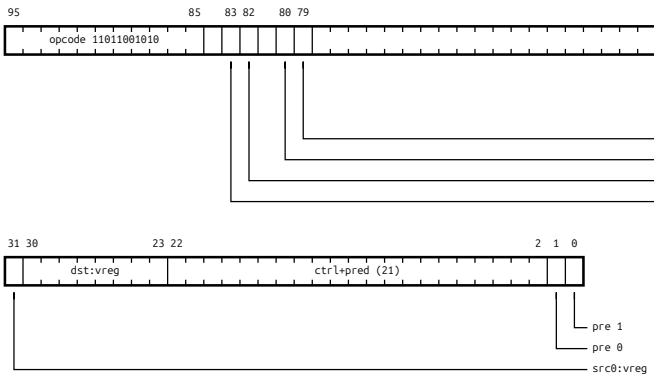
Leaf: iadd3__v_vvv

Format:

```
iadd3  dst, src0{.src_int_modi}{.reuse}, src1{.src_int_modi}{.reuse},
↪  src2{.src_int_modi}{.reuse}
```

Operands: dst: vreg; src0: vreg; src1: vreg; src2: vreg

Encoding Diagram: 96b, prefix 10, opcode 11011001010



Notes:

三入整数加法: 普通指令形态 (v_*) dst = src0 + src1 + src2 (mod 2^32, 无 carry); carry mode 指令形态

(vpp_*pp) {pout1, pout0, dst} = src0 + src1 + src2 + (2·pin1 + pin0), 双 cout / 双 cin 形成 multi-precision 链。整 warp per-lane 各自计算。

包含 2-src add 用例: src2=RZ 退化为 dst = src0 + src1 (替代单独的 iadd / iaddc mnemonic)。

普通指令形态 src 修饰符: src_int_modi=id / neg / abs / abs_then_neg, 实现 a-b-c / a+|b|+c 等变体 (neg = 取负 = 加补码, 加法层面跟 a + (-b) 等价)。

双 carry chain 物理含义: 3-integer add 最大值 3·(2^32-1) 自然产生 2-bit cout (∈ {0,1,2}), 所以下一 chunk 也接 2-bit cin (cout = 2·pout1 + pout0, cin = 2·pin1 + pin0)。

carry mode src 修饰符: src_bit_modi=id / inv (按位取反); 扩展精度减法用 a + ~b + cin chain 自然实现 (跟普通指令形态的 src_int_modi 不同, carry mode 不接受 neg/abs)。

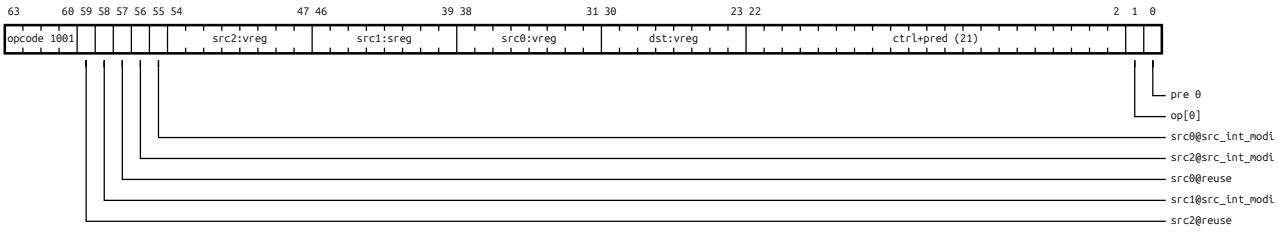
Leaf: iadd3__v_vxv

Format:

```
iadd3  dst, src0{.src_int_modi}{.reuse}, src1{.src_int_modi}, src2{.src_int_modi}{.reuse}
```

Operands: dst: vreg; src0: vreg; src1: sreg; src2: vreg

Encoding Diagram: 64b, prefix 0, opcode 01001



Notes:

三入整数加法: 普通指令形态 (v_*) dst = src0 + src1 + src2 (mod 2^32, 无 carry); carry mode 指令形态 (vpp_*pp) {pout1, pout0, dst} = src0 + src1 + src2 + (2·pin1 + pin0), 双 cout / 双 cin 形成 multi-precision 链。整 warp per-lane 各自计算。

包含 2-src add 用例: src2=RZ 退化为 dst = src0 + src1 (替代单独的 iadd / iaddc mnemonic)。

普通指令形态 src 修饰符: src_int_modi=id / neg / abs / abs_then_neg, 实现 a-b-c / a+|b|+c 等变体 (neg = 取负 = 加补码, 加法层面跟 a + (-b) 等价)。

双 carry chain 物理含义: 3-integer add 最大值 3·(2^32-1) 自然产生 2-bit cout (∈ {0,1,2}), 所以下一 chunk 也接 2-bit cin (cout = 2·pout1 + pout0, cin = 2·pin1 + pin0)。

carry mode src 修饰符: src_bit_modi=id / inv (按位取反); 扩展精度减法用 a + ~b + cin chain 自然实现 (跟普通指令形态的 src_int_modi 不同, carry mode 不接受 neg/abs)。

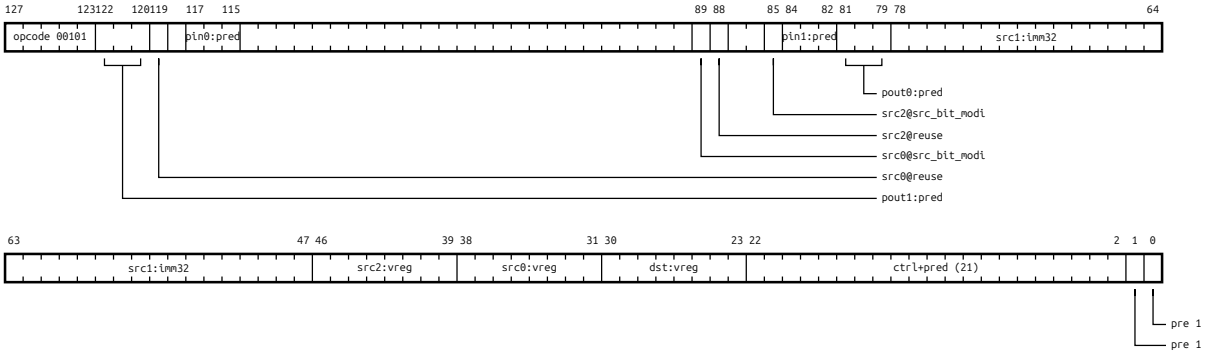
Leaf: iadd3__vpp_vivpp

Format:

```
iadd3  dst, pout0, pout1, src0{.src_bit_modi}{.reuse}, src1, src2{.src_bit_modi}{.reuse},
↪  pin0, pin1
```

Operands: dst: vreg; pout0: pred; pout1: pred; src0: vreg; src1: imm32u; src2: vreg; pin0: pred; pin1: pred

Encoding Diagram: 128b, prefix 11, opcode 00101



Notes:

三入整数加法: 普通指令形态 (v_*) dst = src0 + src1 + src2 (mod 2^32, 无 carry); carry mode 指令形态 (vpp_*pp) {pout1, pout0, dst} = src0 + src1 + src2 + (2·pin1 + pin0), 双 cout / 双 cin 形成 multi-precision 链。整 warp per-lane 各自计算。

包含 2-src add 用例: src2=RZ 退化为 dst = src0 + src1 (替代单独的 iadd / iaddc mnemonic)。

普通指令形态 src 修饰符: src_int_modi=id / neg / abs / abs_then_neg, 实现 a-b-c / a+|b|+c 等变体 (neg = 取负 = 加补码, 加法层面跟 a + (-b) 等价)。

双 carry chain 物理含义: 3-integer add 最大值 3·(2^32-1) 自然产生 2-bit cout (∈ {0,1,2}), 所以下一 chunk 也接 2-bit cin (cout = 2·pout1 + pout0, cin = 2·pin1 + pin0)。

carry mode src 修饰符: src_bit_modi=id / inv (按位取反); 扩展精度减法用 a + ~b + cin chain 自然实现 (跟普通指令形态的 src_int_modi 不同, carry mode 不接受 neg/abs)。

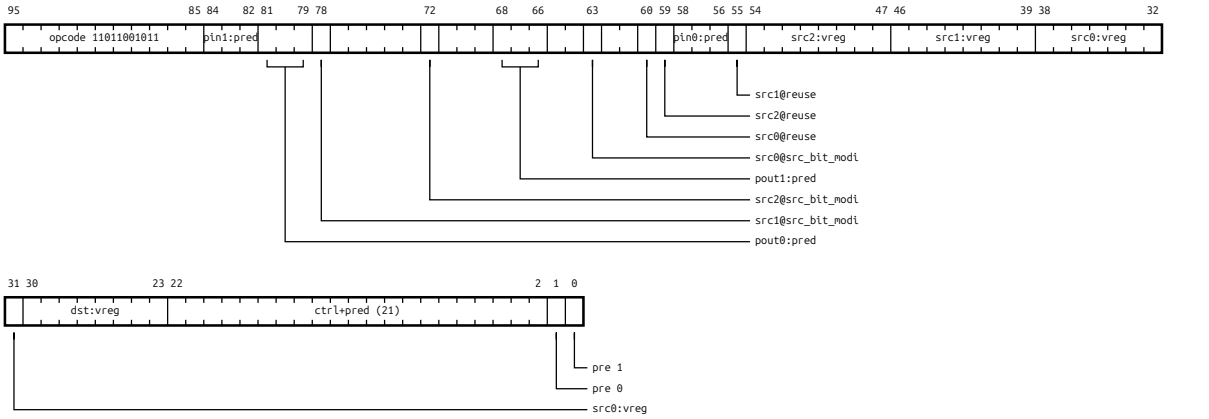
Leaf: iadd3__vpp_vvvpp

Format:

```
iadd3  dst, pout0, pout1, src0{.src_bit_modi}{.reuse}, src1{.src_bit_modi}{.reuse},
↪  src2{.src_bit_modi}{.reuse}, pin0, pin1
```

Operands: dst: vreg; pout0: pred; pout1: pred; src0: vreg; src1: vreg; src2: vreg; pin0: pred; pin1: pred

Encoding Diagram: 96b, prefix 10, opcode 11011001011



Notes:

三入整数加法: 普通指令形态 (v_*) dst = src0 + src1 + src2 (mod 2^32, 无 carry); carry mode 指令形态 (vpp_*pp) {pout1, pout0, dst} = src0 + src1 + src2 + (2·pin1 + pin0), 双 cout / 双 cin 形成 multi-precision 链。整 warp per-lane 各自计算。

包含 2-src add 用例: src2=RZ 退化为 dst = src0 + src1 (替代单独的 iadd / iaddc mnemonic)。

普通指令形态 src 修饰符: src_int_modi=id / neg / abs / abs_then_neg, 实现 a-b-c / a+|b|+c 等变体 (neg = 取负 = 加补码, 加法层面跟 a + (-b) 等价)。

双 carry chain 物理含义: 3-integer add 最大值 3·(2^32-1) 自然产生 2-bit cout (∈ {0,1,2}), 所以下一 chunk 也接 2-bit cin (cout = 2·pout1 + pout0, cin = 2·pin1 + pin0)。

carry mode src 修饰符: src_bit_modi=id / inv (按位取反); 扩展精度减法用 a + ~b + cin chain 自然实现 (跟普通指令形态的 src_int_modi 不同, carry mode 不接受 neg/abs)。

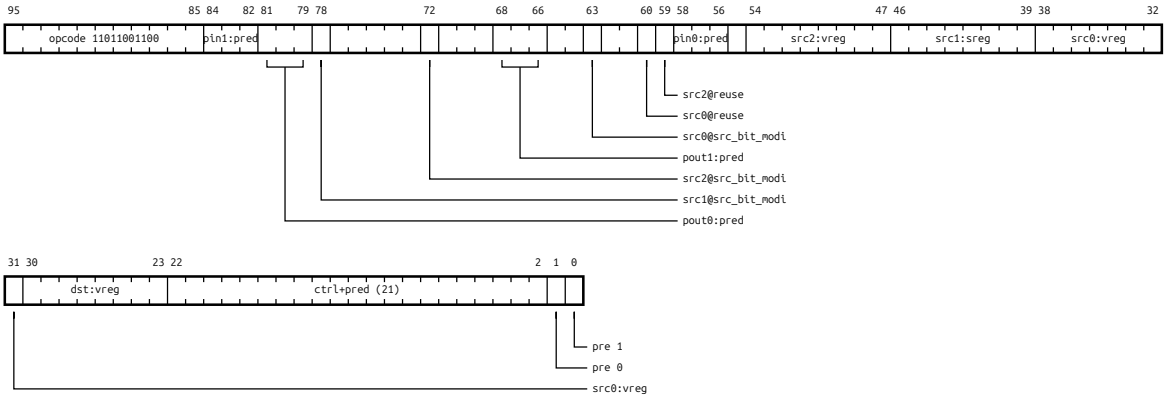
Leaf: iadd3__vpp_vxvpp

Format:

```
iadd3  dst, pout0, pout1, src0{.src_bit_modi}{.reuse}, src1{.src_bit_modi},
↪  src2{.src_bit_modi}{.reuse}, pin0, pin1
```

Operands: dst: vreg; pout0: pred; pout1: pred; src0: vreg; src1: sreg; src2: vreg; pin0: pred; pin1: pred

Encoding Diagram: 96b, prefix 10, opcode 11011001100



Notes:

三入整数加法: 普通指令形态 (v_*) dst = src0 + src1 + src2 (mod 2^32, 无 carry); carry mode 指令形态 (vpp_*pp) {pout1, pout0, dst} = src0 + src1 + src2 + (2·pin1 + pin0), 双 cout / 双 cin 形成 multi-precision 链。整 warp per-lane 各自计算。

包含 2-src add 用例: src2=RZ 退化为 dst = src0 + src1 (替代单独的 iadd / iaddc mnemonic)。

普通指令形态 src 修饰符: src_int_modi=id / neg / abs / abs_then_neg, 实现 a-b-c / a+|b|+c 等变体 (neg = 取负 = 加补码, 加法层面跟 a + (-b) 等价)。

双 carry chain 物理含义: 3-integer add 最大值 3·(2^32-1) 自然产生 2-bit cout (∈ {0,1,2}), 所以下一 chunk 也接 2-bit cin (cout = 2·pout1 + pout0, cin = 2·pin1 + pin0)。

carry mode src 修饰符: src_bit_modi=id / inv (按位取反); 扩展精度减法用 a + ~b + cin chain 自然实现 (跟普通指令形态的 src_int_modi 不同, carry mode 不接受 neg/abs)。

iadd_pk **category**: Integer compute **predicate_mode**: simt

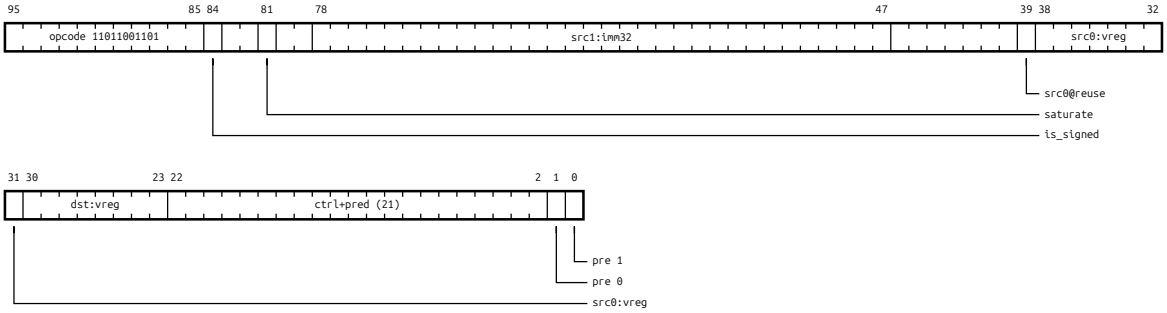
Leaf: iadd_pk__v_vi

Format:

```
iadd_pk.is_signed{.saturate}  dst, src0{.reuse}, src1
```

Operands: dst: vreg; src0: vreg; src1: imm32u

Encoding Diagram: 96b, prefix 10, opcode 11011001101



Notes:

Packed int16 加法 (2-lane / 32-bit reg, 每 lane 16-bit 整数): dst[15:0] = src0[15:0] + src1[15:0]; dst[31:16] = src0[31:16] + src1[31:16]。整 warp per-lane 各自计算。ML int16 量化推理 / 图像处理 / 短整数 SIMD 高频。

is_signed=signed: 按 s16 解读 (符号位参与); is_signed=unsigned: 按 u16 解读 (位模式)。

saturate=saturate: 按 is_signed 饱和到 [INT16_MIN, INT16_MAX] 或 [0, UINT16_MAX], 溢出 clamp 到边界 (默认无 saturate 时 wrap)。

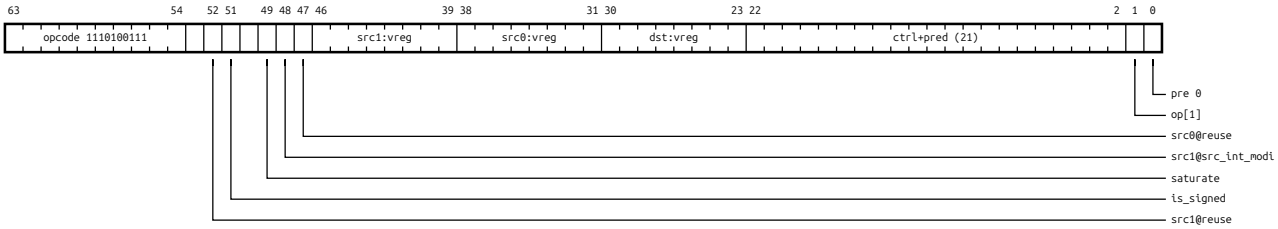
Leaf: iadd_pk__v_vv

Format:

```
iadd_pk.is_signed{.saturate}  dst, src0{.reuse}, src1{.src_int_modi}{.reuse}
```

Operands: dst: vreg; src0: vreg; src1: vreg

Encoding Diagram: 64b, prefix 0, opcode 11110100111



Notes:

Packed int16 加法 (2-lane / 32-bit reg, 每 lane 16-bit 整数): dst[15:0] = src0[15:0] + src1[15:0]; dst[31:16] = src0[31:16] + src1[31:16]。整 warp per-lane 各自计算。ML int16 量化推理 / 图像处理 / 短整数 SIMD 高频。

is_signed=signed: 按 s16 解读 (符号位参与); is_signed=unsigned: 按 u16 解读 (位模式)。

saturate=saturate: 按 is_signed 饱和到 [INT16_MIN, INT16_MAX] 或 [0, UINT16_MAX], 溢出 clamp 到边界 (默认无 saturate 时 wrap)。

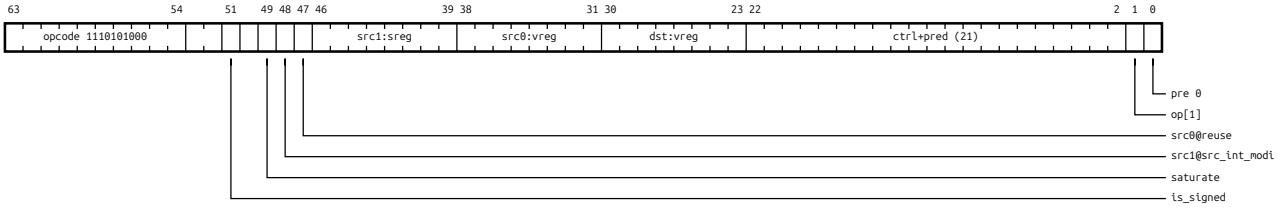
Leaf: iadd_pk__v_vx

Format:

```
iadd_pk.is_signed{.saturate}  dst, src0{.reuse}, src1{.src_int_modi}
```

Operands: dst: vreg; src0: vreg; src1: sreg

Encoding Diagram: 64b, prefix 0, opcode 11110101000



Notes:

Packed int16 加法 (2-lane / 32-bit reg, 每 lane 16-bit 整数): dst[15:0] = src0[15:0] + src1[15:0]; dst[31:16] = src0[31:16] + src1[31:16]。整 warp per-lane 各自计算。ML int16 量化推理 / 图像处理 / 短整数 SIMD 高频。

is_signed=signed: 按 s16 解读 (符号位参与); is_signed=unsigned: 按 u16 解读 (位模式)。

saturate=saturate: 按 is_signed 饱和到 [INT16_MIN, INT16_MAX] 或 [0, UINT16_MAX], 溢出 clamp 到边界 (默认无 saturate 时 wrap)。

idp **category**: Integer compute **predicate_mode**: simt

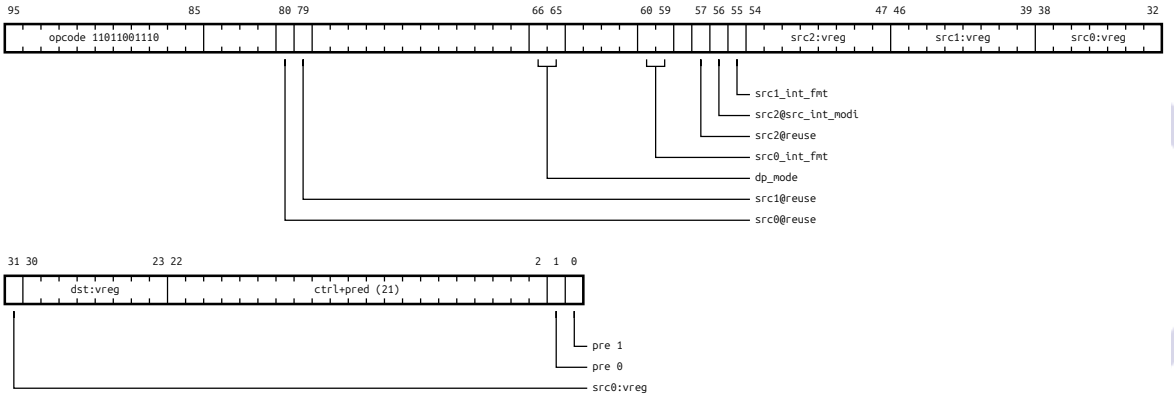
Leaf: idp__v_vvv

Format:

```
idp.dp_mode.src0_int_fmt.src1_int_fmt  dst, src0{.reuse}, src1{.reuse},
↪  src2{.src_int_modi}{.reuse}
```

Operands: dst: vreg; src0: vreg; src1: vreg; src2: vreg

Encoding Diagram: 96b, prefix 10, opcode 11011001110



Notes:

整数点积带累加 (INT8 / INT16 量化高频): dst = Σ (src0.lane_i × src1.lane_i) + src2, 把 src0 / src1 当 packed 整数向量做点积, 32-bit 累加到 src2 写 dst。整 warp per-lane 各自计算。

dp_mode=4a: src0 / src1 都拆成 4 个 8-bit, 4 lane 点积; dst = src0.b0×src1.b0 + src0.b1×src1.b1 + src0.b2×src1.b2 + src0.b3×src1.b3 + src2。

dp_mode=2A.lo: src0 拆 2 个 16-bit, src1 取低字节对 (b0, b2); dst = src0.hw0×src1.b0 + src0.hw1×src1.b2 + src2。

dp_mode=2A.hi: src0 拆 2 个 16-bit, src1 取高字节对 (b1, b3); dst = src0.hw0×src1.b1 + src0.hw1×src1.b3 + src2。

src0_int_fmt / src1_int_fmt: 独立选 src0 / src1 的整数 fmt (4a 模式两 src 都 8-bit u8/s8; 2A 模式 src0 是 16-bit u16/s16, src1 是 8-bit u8/s8)。dp_mode 限制有效组合。

src2 可挂 src_int_modi (neg / abs) 表达 -src2 / |src2| 累加。

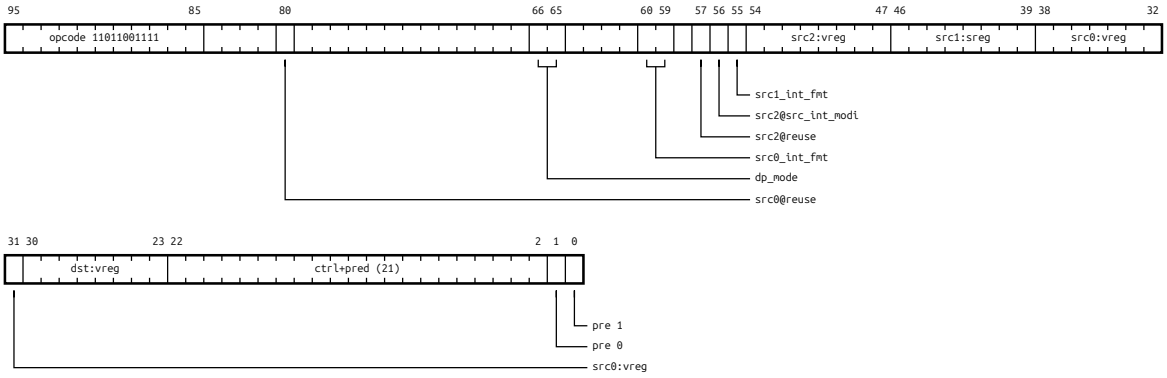
Leaf: idp__v_vxv

Format:

```
idp.dp_mode.src0_int_fmt.src1_int_fmt  dst, src0{.reuse}, src1,
↪  src2{.src_int_modi}{.reuse}
```

Operands: dst: vreg; src0: vreg; src1: sreg; src2: vreg

Encoding Diagram: 96b, prefix 10, opcode 11011001111



Notes:

整数点积带累加 (INT8 / INT16 量化高频): dst = Σ (src0.lane_i × src1.lane_i) + src2, 把 src0 / src1 当 packed 整数向量做点积, 32-bit 累加到 src2 写 dst。整 warp per-lane 各自计算。

dp_mode=4a: src0 / src1 都拆成 4 个 8-bit, 4 lane 点积; dst = src0.b0×src1.b0 + src0.b1×src1.b1 + src0.b2×src1.b2 + src0.b3×src1.b3 + src2。

dp_mode=2A.lo: src0 拆 2 个 16-bit, src1 取低字节对 (b0, b2); dst = src0.hw0×src1.b0 + src0.hw1×src1.b2 + src2。

dp_mode=2A.hi: src0 拆 2 个 16-bit, src1 取高字节对 (b1, b3); dst = src0.hw0×src1.b1 + src0.hw1×src1.b3 + src2。

src0_int_fmt / src1_int_fmt: 独立选 src0 / src1 的整数 fmt (4a 模式两 src 都 8-bit u8/s8; 2A 模式 src0 是 16-bit u16/s16, src1 是 8-bit u8/s8)。dp_mode 限制有效组合。

src2 可挂 src_int_modi (neg / abs) 表达 -src2 / |src2| 累加。

imad **category**: Integer compute **predicate_mode**: simt

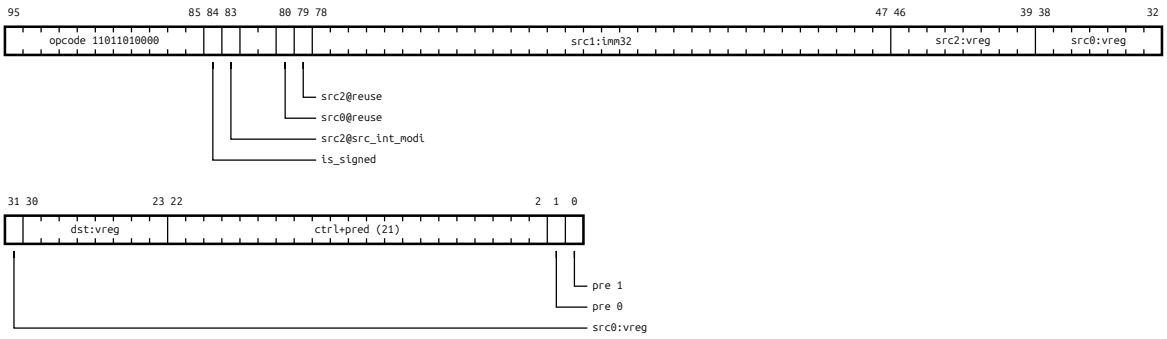
Leaf: imad__v_viv

Format:

```
imad.is_signed  dst, src0{.reuse}, src1, src2{.src_int_modi}{.reuse}
```

Operands: dst: vreg; src0: vreg; src1: imm32u; src2: vreg

Encoding Diagram: 96b, prefix 10, opcode 11011010000



Notes:

32-bit 整数 fused multiply-add: dst = (src0 × src1 + src2) mod 2^32, 结果 wrap 到 32 bit (无 carry-out)。整 warp per-lane 各自计算。

is_signed=signed / unsigned: 作用在乘法的符号解释上 (加法是补码加法, 跟 signed / unsigned 无关; 同 src0/src1 用同一 fmt 解读)。

若需 64-bit 完整结果 (mul 不溢出) 走 imad_wide; 需要 carry-out 做 multi-precision 走 imad + iadd3.x 序列 (无 imad_x / imad_hi 单条变体)。

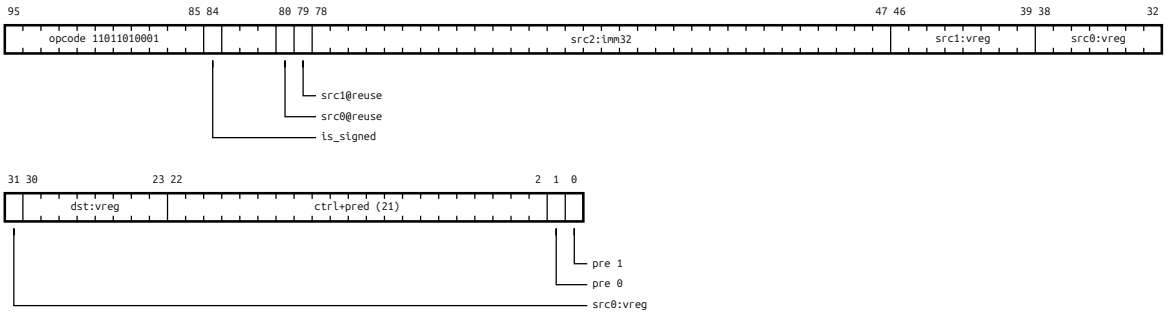
Leaf: imad__v_vvi

Format:

```
imad.is_signed  dst, src0{.reuse}, src1{.reuse}, src2
```

Operands: dst: vreg; src0: vreg; src1: vreg; src2: imm32u

Encoding Diagram: 96b, prefix 10, opcode 11011010001



Notes:

32-bit 整数 fused multiply-add: dst = (src0 × src1 + src2) mod 2^32, 结果 wrap 到 32 bit (无 carry-out)。整 warp per-lane 各自计算。

is_signed=signed / unsigned: 作用在乘法的符号解释上 (加法是补码加法, 跟 signed / unsigned 无关; 同 src0/src1 用同一 fmt 解读)。

若需 64-bit 完整结果 (mul 不溢出) 走 imad_wide; 需要 carry-out 做 multi-precision 走 imad + iadd3.x 序列 (无 imad_x / imad_hi 单条变体)。

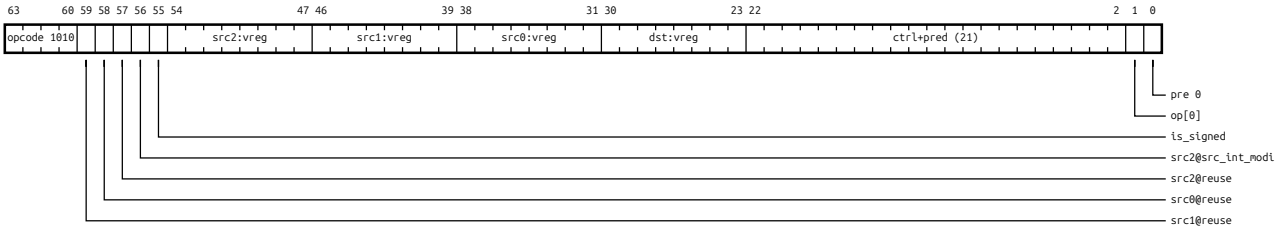
Leaf: imad__v_vvv

Format:

```
imad.is_signed  dst, src0{.reuse}, src1{.reuse}, src2{.src_int_modi}{.reuse}
```

Operands: dst: vreg; src0: vreg; src1: vreg; src2: vreg

Encoding Diagram: 64b, prefix 0, opcode 01010



Notes:

32-bit 整数 fused multiply-add: dst = (src0 × src1 + src2) mod 2^32, 结果 wrap 到 32 bit (无 carry-out)。整 warp per-lane 各自计算。

is_signed=signed / unsigned: 作用在乘法的符号解释上 (加法是补码加法, 跟 signed / unsigned 无关; 同 src0/src1 用同一 fmt 解读)。

若需 64-bit 完整结果 (mul 不溢出) 走 imad_wide; 需要 carry-out 做 multi-precision 走 imad + iadd3.x 序列 (无 imad_x / imad_hi 单条变体)。

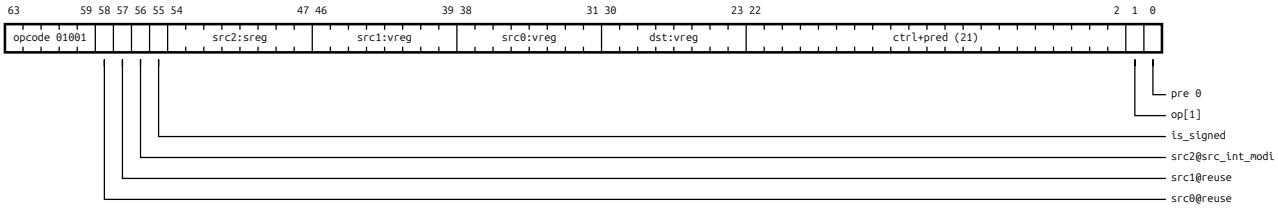
Leaf: imad__v_vvx

Format:

```
imad.is_signed  dst, src0{.reuse}, src1{.reuse}, src2{.src_int_modi}
```

Operands: dst: vreg; src0: vreg; src1: vreg; src2: sreg

Encoding Diagram: 64b, prefix 0, opcode 101001



Notes:

32-bit 整数 fused multiply-add: dst = (src0 × src1 + src2) mod 2^32, 结果 wrap 到 32 bit (无 carry-out)。整 warp per-lane 各自计算。

is_signed=signed / unsigned: 作用在乘法的符号解释上 (加法是补码加法, 跟 signed / unsigned 无关; 同 src0/src1 用同一 fmt 解读)。

若需 64-bit 完整结果 (mul 不溢出) 走 imad_wide; 需要 carry-out 做 multi-precision 走 imad + iadd3.x 序列 (无 imad_x / imad_hi 单条变体)。

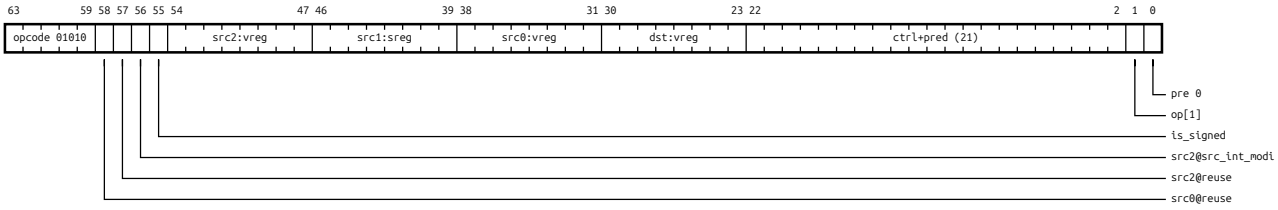
Leaf: imad__v_vxv

Format:

```
imad.is_signed  dst, src0{.reuse}, src1, src2{.src_int_modi}{.reuse}
```

Operands: dst: vreg; src0: vreg; src1: sreg; src2: vreg

Encoding Diagram: 64b, prefix 0, opcode 101010



Notes:

32-bit 整数 fused multiply-add: dst = (src0 × src1 + src2) mod 2^32, 结果 wrap 到 32 bit (无 carry-out)。整 warp per-lane 各自计算。

is_signed=signed / unsigned: 作用在乘法的符号解释上 (加法是补码加法, 跟 signed / unsigned 无关; 同 src0/src1 用同一 fmt 解读)。

若需 64-bit 完整结果 (mul 不溢出) 走 imad_wide; 需要 carry-out 做 multi-precision 走 imad + iadd3.x 序列 (无 imad_x / imad_hi 单条变体)。

imad_wide　**category**: Integer compute　**predicate_mode**: simt

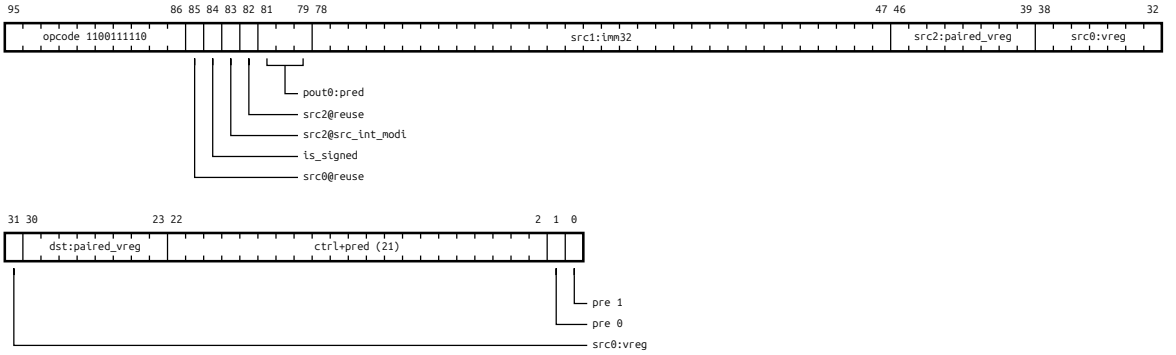
Leaf: imad_wide__vp_viv

Format:

```
imad_wide.is_signed  dst, pout0, src0{.reuse}, src1, src2{.src_int_modi}{.reuse}
```

Operands: dst: paired_vreg; pout0: pred; src0: vreg; src1: imm32u; src2: paired_vreg

Encoding Diagram: 96b, prefix 10, opcode 1100111110



Notes:

Wide 整数 multiply-add: 32-bit × 32-bit 乘法扩展到 64-bit, 加 64-bit accumulator: {dst, dst+1} = src0 × src1 + {src2, src2+1} (mod 2^64), pout0 = 64-bit add 的 carry-out。整 warp per-lane 各自计算。

dst 占 2 个相邻 vreg (偶对齐); src2 是 reg 时也占 2 个相邻 vreg。pout0 是 1-bit predicate, 接收 64-bit 加法的 carry-out (PT 抑制输出)。

is_signed=signed / unsigned: 选择乘法解释 (signed 的两 src 都按 s32 sign-extend 到 64-bit; unsigned 都按 u32 zero-extend)。加法是补码加法跟符号无关。

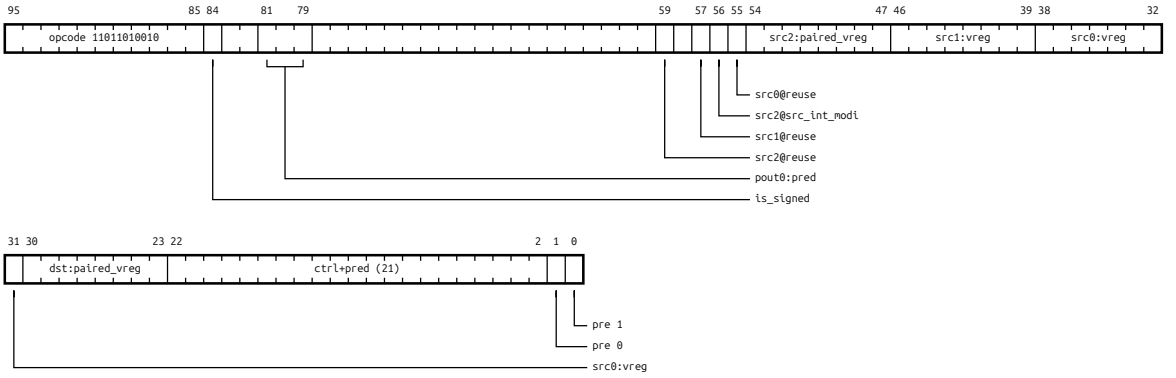
Leaf: imad_wide__vp_vvv

Format:

```
imad_wide.is_signed  dst, pout0, src0{.reuse}, src1{.reuse}, src2{.src_int_modi}{.reuse}
```

Operands: dst: paired_vreg; pout0: pred; src0: vreg; src1: vreg; src2: paired_vreg

Encoding Diagram: 96b, prefix 10, opcode 11011010010



Notes:

Wide 整数 multiply-add: 32-bit × 32-bit 乘法扩展到 64-bit, 加 64-bit accumulator: {dst, dst+1} = src0 × src1 + {src2, src2+1} (mod 2^64), pout0 = 64-bit add 的 carry-out。整 warp per-lane 各自计算。

dst 占 2 个相邻 vreg (偶对齐); src2 是 reg 时也占 2 个相邻 vreg。pout0 是 1-bit predicate, 接收 64-bit 加法的 carry-out (PT 抑制输出)。

is_signed=signed / unsigned: 选择乘法解释 (signed 的两 src 都按 s32 sign-extend 到 64-bit; unsigned 都按 u32 zero-extend)。加法是补码加法跟符号无关。

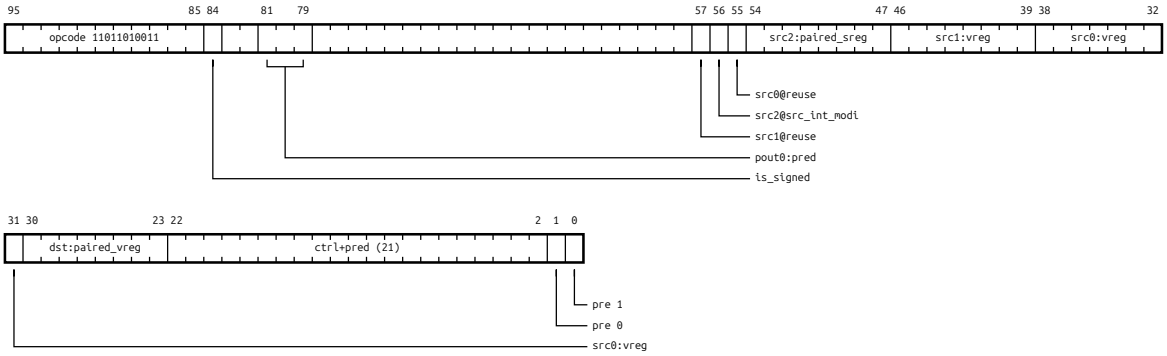
Leaf: imad_wide__vp_vvx

Format:

```
imad_wide.is_signed  dst, pout0, src0{.reuse}, src1{.reuse}, src2{.src_int_modi}
```

Operands: dst: paired_vreg; pout0: pred; src0: vreg; src1: vreg; src2: paired_sreg

Encoding Diagram: 96b, prefix 10, opcode 11011010011



Notes:

Wide 整数 multiply-add: 32-bit × 32-bit 乘法扩展到 64-bit, 加 64-bit accumulator: {dst, dst+1} = src0 × src1 + {src2, src2+1} (mod 2^64), pout0 = 64-bit add 的 carry-out。整 warp per-lane 各自计算。

dst 占 2 个相邻 vreg (偶对齐); src2 是 reg 时也占 2 个相邻 vreg。pout0 是 1-bit predicate, 接收 64-bit 加法的 carry-out (PT 抑制输出)。

is_signed=signed / unsigned: 选择乘法解释 (signed 的两 src 都按 s32 sign-extend 到 64-bit; unsigned 都按 u32 zero-extend)。加法是补码加法跟符号无关。

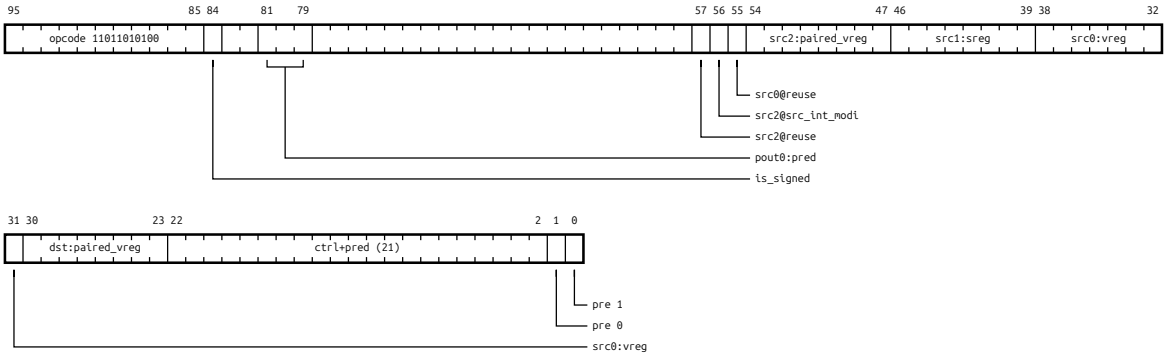
Leaf: imad_wide__vp_vxv

Format:

```
imad_wide.is_signed  dst, pout0, src0{.reuse}, src1, src2{.src_int_modi}{.reuse}
```

Operands: dst: paired_vreg; pout0: pred; src0: vreg; src1: sreg; src2: paired_vreg

Encoding Diagram: 96b, prefix 10, opcode 11011010100



Notes:

Wide 整数 multiply-add: 32-bit × 32-bit 乘法扩展到 64-bit, 加 64-bit accumulator: {dst, dst+1} = src0 × src1 + {src2, src2+1} (mod 2^64), pout0 = 64-bit add 的 carry-out。整 warp per-lane 各自计算。

dst 占 2 个相邻 vreg (偶对齐); src2 是 reg 时也占 2 个相邻 vreg。pout0 是 1-bit predicate, 接收 64-bit 加法的 carry-out (PT 抑制输出)。

is_signed=signed / unsigned: 选择乘法解释 (signed 的两 src 都按 s32 sign-extend 到 64-bit; unsigned 都按 u32 zero-extend)。加法是补码加法跟符号无关。

imnmx **category**: Integer compute **predicate_mode**: simt

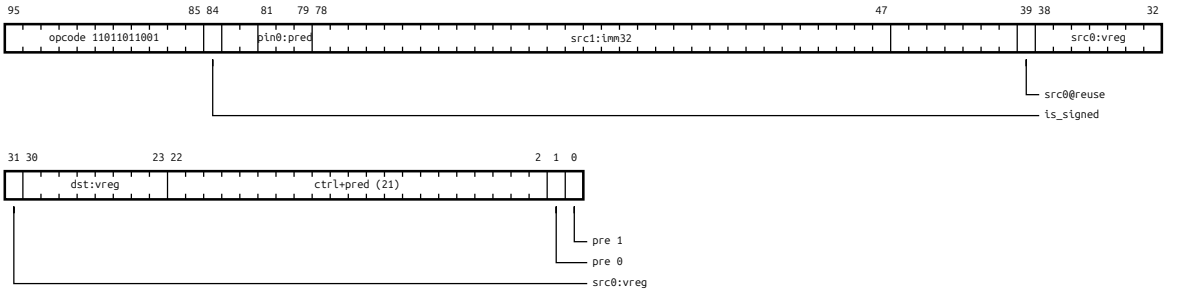
Leaf: imnmx__v_vi

Format:

```
imnmx.is_signed  dst, src0{.reuse}, src1, pin0
```

Operands: dst: vreg; src0: vreg; src1: imm32u; pin0: pred

Encoding Diagram: 96b, prefix 10, opcode 11011011001



Notes:

32-bit 整数 per-lane min/max: dst = pin0 ? min(src0, src1) : max(src0, src1), 由 pin0 predicate 控制 min/max 选择。pin0=PT 永远 min, pin0=!PT 永远 max。整 warp per-lane 各自独立。

is_signed=signed: 按 s32 比较 (符号位参与, 负数 < 正数)。

is_signed=unsigned: 按 u32 比较 (位模式直接比, 0xFFFFFFFF > 0x7FFFFFFF)。

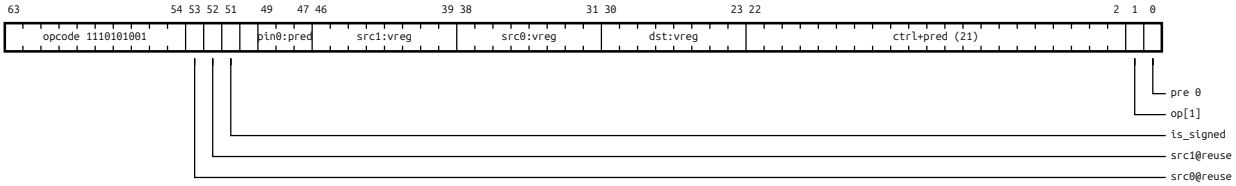
Leaf: imnmx__v_vv

Format:

```
imnmx.is_signed  dst, src0{.reuse}, src1{.reuse}, pin0
```

Operands: dst: vreg; src0: vreg; src1: vreg; pin0: pred

Encoding Diagram: 64b, prefix 0, opcode 11110101001



Notes:

32-bit 整数 per-lane min/max: dst = pin0 ? min(src0, src1) : max(src0, src1), 由 pin0 predicate 控制 min/max 选择。pin0=PT 永远 min, pin0=!PT 永远 max。整 warp per-lane 各自独立。

is_signed=signed: 按 s32 比较 (符号位参与, 负数 < 正数)。

is_signed=unsigned: 按 u32 比较 (位模式直接比, 0xFFFFFFFF > 0x7FFFFFFF)。

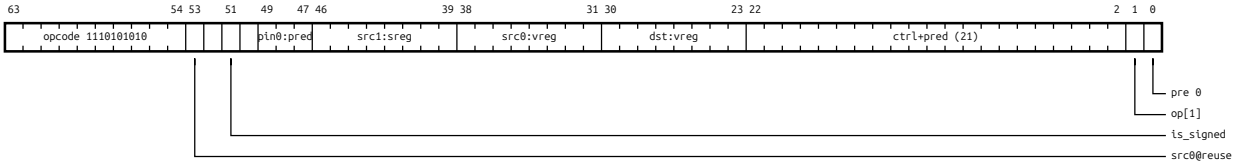
Leaf: imnmx__v_vx

Format:

```
imnmx.is_signed  dst, src0{.reuse}, src1, pin0
```

Operands: dst: vreg; src0: vreg; src1: sreg; pin0: pred

Encoding Diagram: 64b, prefix 0, opcode 11110101010



Notes:

32-bit 整数 per-lane min/max: dst = pin0 ? min(src0, src1) : max(src0, src1), 由 pin0 predicate 控制 min/max 选择。pin0=PT 永远 min, pin0=!PT 永远 max。整 warp per-lane 各自独立。

is_signed=signed: 按 s32 比较 (符号位参与, 负数 < 正数)。

is_signed=unsigned: 按 u32 比较 (位模式直接比, 0xFFFFFFFF > 0x7FFFFFFF)。

imnmx3 **category**: Integer compute **predicate_mode**: simt

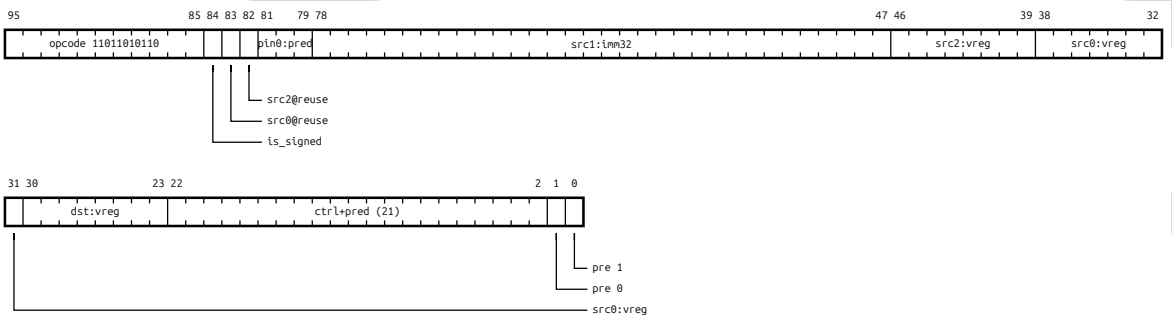
Leaf: imnmx3__v_viv

Format:

```
imnmx3.is_signed  dst, src0{.reuse}, src1, src2{.reuse}, pin0
```

Operands: dst: vreg; src0: vreg; src1: imm32u; src2: vreg; pin0: pred

Encoding Diagram: 96b, prefix 10, opcode 11011010110



Notes:

三入 32-bit 整数 per-lane min/max: dst = pin0 ? min(src0, src1, src2) : max(src0, src1, src2), pin0 控制 min/max 选择。pin0=PT 永远 min, pin0=!PT 永远 max。常用于 clamp (min(max(x, lo), hi))、sorting network。整 warp per-lane 各自计算。

is_signed=signed: 按 s32 比较。

is_signed=unsigned: 按 u32 比较。

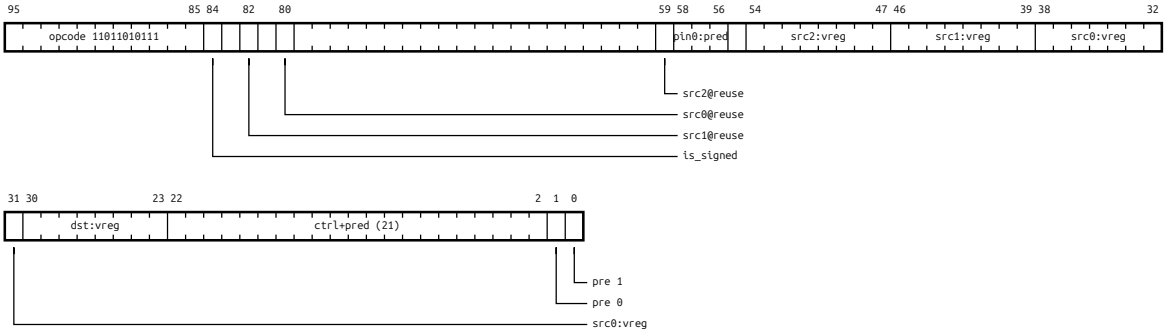
Leaf: imnmx3__v_vvv

Format:

```
imnmx3.is_signed  dst, src0{.reuse}, src1{.reuse}, src2{.reuse}, pin0
```

Operands: dst: vreg; src0: vreg; src1: vreg; src2: vreg; pin0: pred

Encoding Diagram: 96b, prefix 10, opcode 11011010111



Notes:

三入 32-bit 整数 per-lane min/max: dst = pin0 ? min(src0, src1, src2) : max(src0, src1, src2), pin0 控制 min/max 选择。pin0=PT 永远 min, pin0=!PT 永远 max。常用于 clamp (min(max(x, lo), hi))、sorting network。整 warp per-lane 各自计算。

is_signed=signed: 按 s32 比较。

is_signed=unsigned: 按 u32 比较。

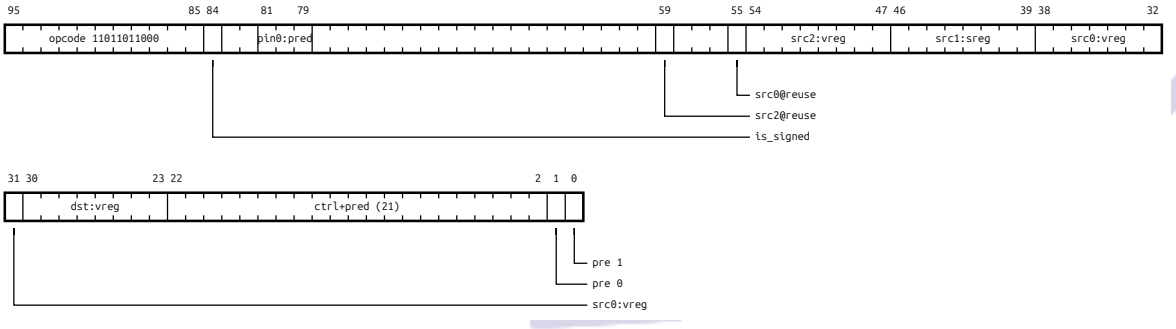
Leaf: imnmx3__v_vxv

Format:

```
imnmx3.is_signed  dst, src0{.reuse}, src1, src2{.reuse}, pin0
```

Operands: dst: vreg; src0: vreg; src1: sreg; src2: vreg; pin0: pred

Encoding Diagram: 96b, prefix 10, opcode 11011011000



Notes:

三入 32-bit 整数 per-lane min/max: dst = pin0 ? min(src0, src1, src2) : max(src0, src1, src2), pin0 控制 min/max 选择。pin0=PT 永远 min, pin0=!PT 永远 max。常用于 clamp (min(max(x, lo), hi))、sorting network。整 warp per-lane 各自计算。

is_signed=signed: 按 s32 比较。

is_signed=unsigned: 按 u32 比较。

imnmx_pk **category**: Integer compute **predicate_mode**: simt

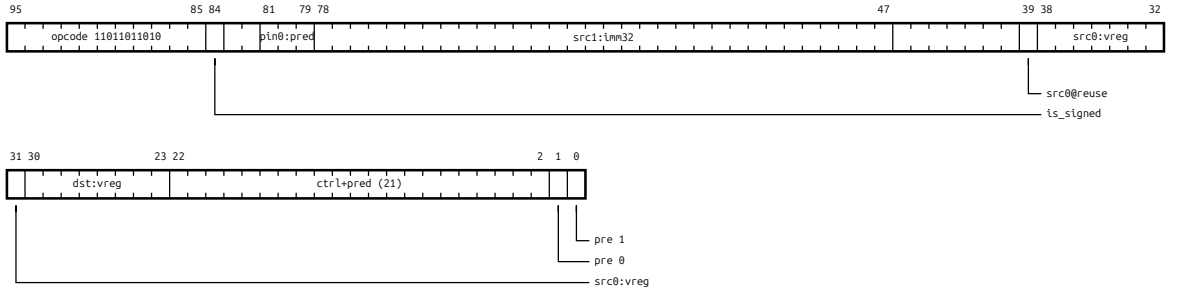
Leaf: imnmx_pk__v_vip

Format:

```
imnmx_pk.is_signed  dst, src0{.reuse}, src1, pin0
```

Operands: dst: vreg; src0: vreg; src1: imm32u; pin0: pred

Encoding Diagram: 96b, prefix 10, opcode 11011011010



Notes:

Packed int16 per-lane min/max (2-lane / 32-bit reg): dst[15:0] = pin0 ? min : max(src0[15:0], src1[15:0]); 高 16-bit 同理独立计算。pin0=PT 永远 min, pin0=!PT 永远 max。整 warp per-lane 各自计算。

is_signed=signed / unsigned: 选 s16 / u16 比较 (s16 时符号位参与, u16 时位模式比较)。

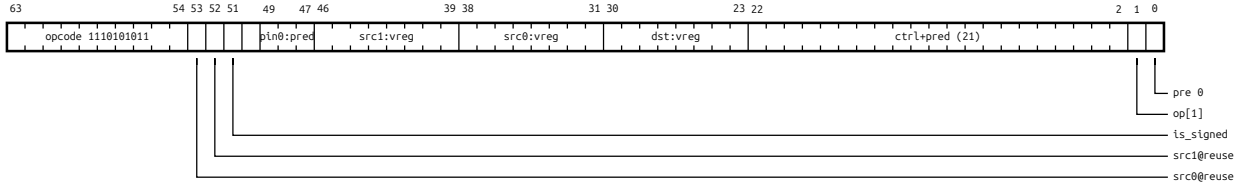
Leaf: imnmx_pk__v_vvp

Format:

```
imnmx_pk.is_signed  dst, src0{.reuse}, src1{.reuse}, pin0
```

Operands: dst: vreg; src0: vreg; src1: vreg; pin0: pred

Encoding Diagram: 64b, prefix 0, opcode 11110101011



Notes:

Packed int16 per-lane min/max (2-lane / 32-bit reg): dst[15:0] = pin0 ? min : max(src0[15:0], src1[15:0]); 高 16-bit 同理独立计算。pin0=PT 永远 min, pin0=!PT 永远 max。整 warp per-lane 各自计算。

is_signed=signed / unsigned: 选 s16 / u16 比较 (s16 时符号位参与, u16 时位模式比较)。

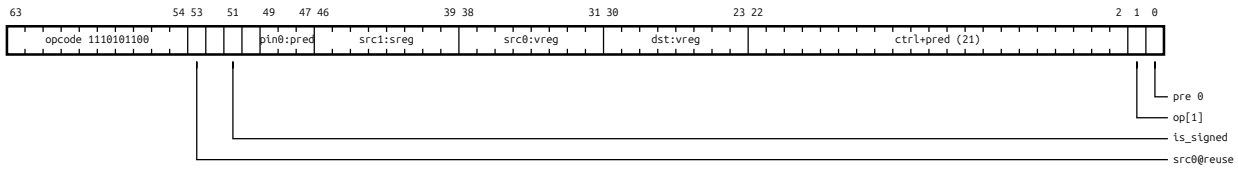
Leaf: imnmx_pk__v_vxp

Format:

```
imnmx_pk.is_signed  dst, src0{.reuse}, src1, pin0
```

Operands: dst: vreg; src0: vreg; src1: sreg; pin0: pred

Encoding Diagram: 64b, prefix 0, opcode 11110101100



Notes:

Packed int16 per-lane min/max (2-lane / 32-bit reg): dst[15:0] = pin0 ? min : max(src0[15:0], src1[15:0]); 高 16-bit 同理独立计算。pin0=PT 永远 min, pin0=!PT 永远 max。整 warp per-lane 各自计算。

is_signed=signed / unsigned: 选 s16 / u16 比较 (s16 时符号位参与, u16 时位模式比较)。

isetp **category**: Integer compute **predicate_mode**: simt

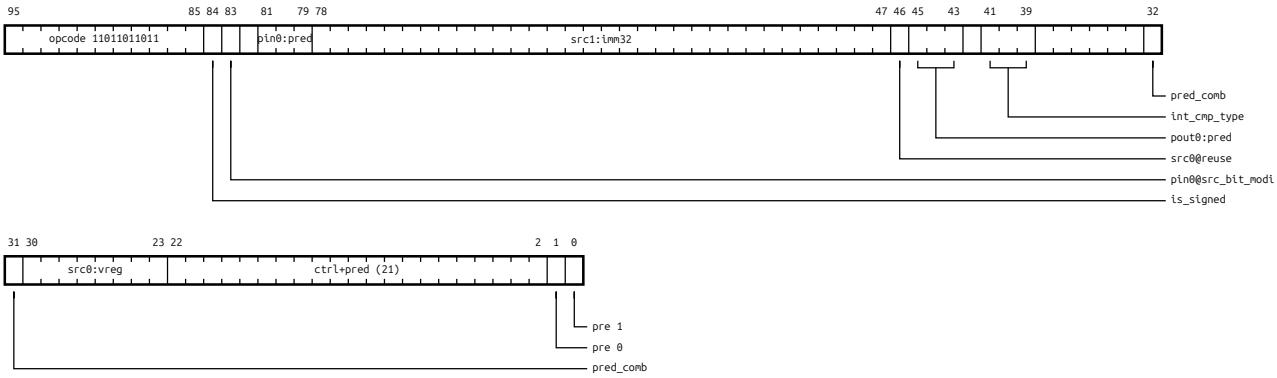
Leaf: isetp__p_vip

Format:

```
isetp.int_cmp_type.pred_comb.is_signed  pout0, src0{.reuse}, src1, pin0{.src_bit_modi}
```

Operands: pout0: pred; src0: vreg; src1: imm32u; pin0: pred

Encoding Diagram: 96b, prefix 10, opcode 11011011011



Notes:

32-bit 整数比较 → predicate: 先做 (src0 INT_CMP src1) 得 1-bit cmp; pout0 = pred_comb(cmp, pin0), pout1 (若声明) = pred_comb(!cmp, pin0)。整 warp per-lane 各自独立。

pout1 不是!pout0, 而是用同一 pred_comb 跟同一 pin0 把!cmp 组合一次 (嵌套谓词模式: 外层 pin0 控制内层 cmp)。pin0=PT + pred_comb=and 时退化为 pout0 = cmp / pout1 = !cmp。

int_cmp_type: 8 valid (f 恒假 / lt / eq / le / gt / ne / ge / t 恒真)。

is_signed=signed / unsigned: s32 用符号位参与比较, u32 按位模式比较。

pred_comb: and / or / xor 三种 cmp 跟 pin0 的组合方式。

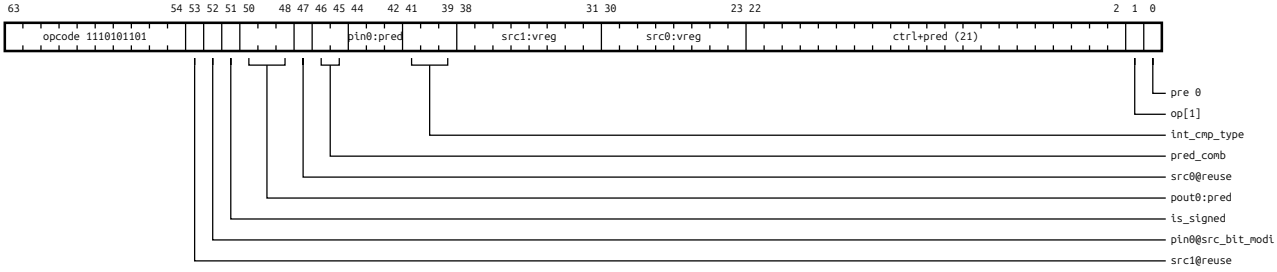
Leaf: isetp__p_vvp

Format:

```
isetp.int_cmp_type.pred_comb.is_signed  pout0, src0{.reuse}, src1{.reuse},
↪  pin0{.src_bit_modi}
```

Operands: pout0: pred; src0: vreg; src1: vreg; pin0: pred

Encoding Diagram: 64b, prefix 0, opcode 11110101101



Notes:

32-bit 整数比较 → predicate: 先做 (src0 INT_CMP src1) 得 1-bit cmp; pout0 = pred_comb(cmp, pin0), pout1 (若声明) = pred_comb(!cmp, pin0)。整 warp per-lane 各自独立。

pout1 不是!pout0, 而是用同一 pred_comb 跟同一 pin0 把!cmp 组合一次 (嵌套谓词模式: 外层 pin0 控制内层 cmp)。pin0=PT + pred_comb=and 时退化为 pout0 = cmp / pout1 = !cmp。

int_cmp_type: 8 valid (f 恒假 / lt / eq / le / gt / ne / ge / t 恒真)。

is_signed=signed / unsigned: s32 用符号位参与比较, u32 按位模式比较。

pred_comb: and / or / xor 三种 cmp 跟 pin0 的组合方式。

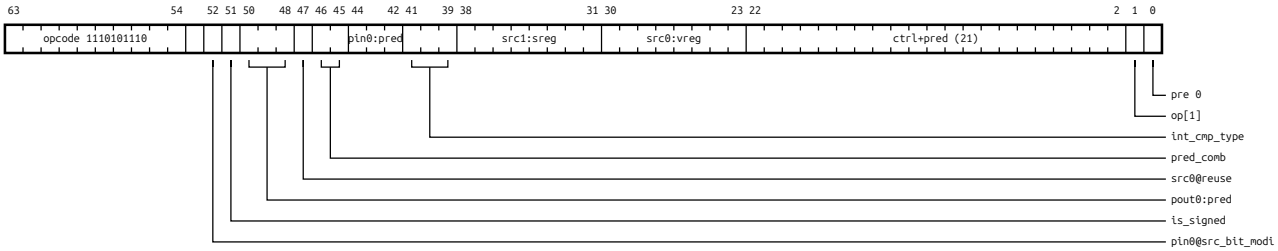
Leaf: isetp__p_vxp

Format:

```
isetp.int_cmp_type.pred_comb.is_signed  pout0, src0{.reuse}, src1, pin0{.src_bit_modi}
```

Operands: pout0: pred; src0: vreg; src1: sreg; pin0: pred

Encoding Diagram: 64b, prefix 0, opcode 11110101110



Notes:

32-bit 整数比较 → predicate: 先做 (src0 INT_CMP src1) 得 1-bit cmp; pout0 = pred_comb(cmp, pin0), pout1 (若声明) = pred_comb(!cmp, pin0)。整 warp per-lane 各自独立。

pout1 不是!pout0, 而是用同一 pred_comb 跟同一 pin0 把!cmp 组合一次 (嵌套谓词模式: 外层 pin0 控制内层 cmp)。pin0=PT + pred_comb=and 时退化为 pout0 = cmp / pout1 = !cmp。

int_cmp_type: 8 valid (f 恒假 / lt / eq / le / gt / ne / ge / t 恒真)。

is_signed=signed / unsigned: s32 用符号位参与比较, u32 按位模式比较。

pred_comb: and / or / xor 三种 cmp 跟 pin0 的组合方式。

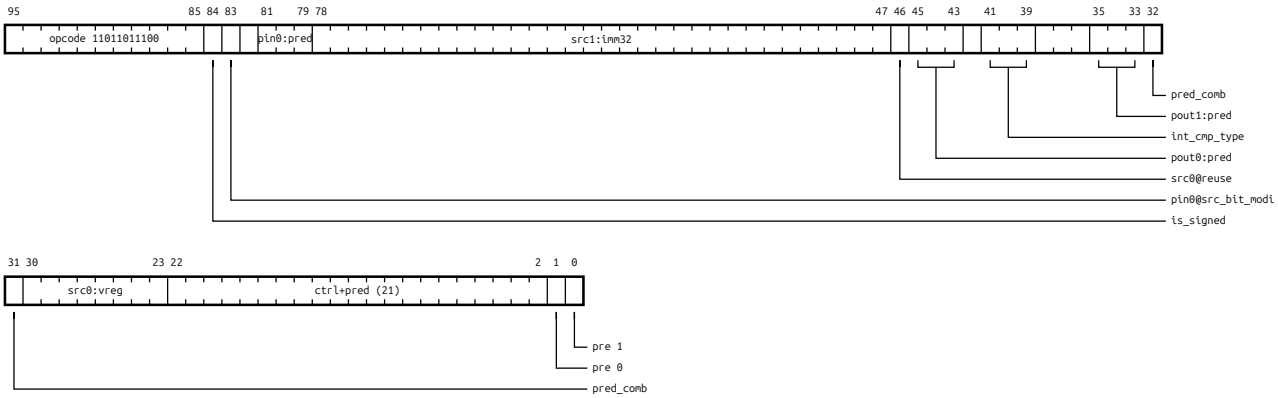
Leaf: isetp__pp_vip

Format:

```
isetp.int_cmp_type.pred_comb.is_signed  pout0, pout1, src0{.reuse}, src1,
↪  pin0{.src_bit_modi}
```

Operands: pout0: pred; pout1: pred; src0: vreg; src1: imm32u; pin0: pred

Encoding Diagram: 96b, prefix 10, opcode 11011011100



Notes:

32-bit 整数比较 → predicate: 先做 (src0 INT_CMP src1) 得 1-bit cmp; pout0 = pred_comb(cmp, pin0), pout1 (若声明) = pred_comb(!cmp, pin0)。整 warp per-lane 各自独立。

pout1 不是!pout0, 而是用同一 pred_comb 跟同一 pin0 把!cmp 组合一次 (嵌套谓词模式: 外层 pin0 控制内层 cmp)。pin0=PT + pred_comb=and 时退化为 pout0 = cmp / pout1 = !cmp。

int_cmp_type: 8 valid (f 恒假 / lt / eq / le / gt / ne / ge / t 恒真)。

is_signed=signed / unsigned: s32 用符号位参与比较, u32 按位模式比较。

pred_comb: and / or / xor 三种 cmp 跟 pin0 的组合方式。

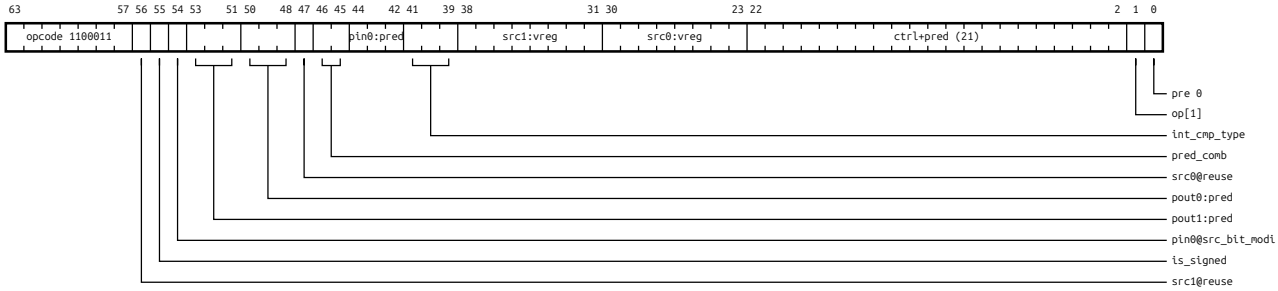
Leaf: isetp__pp_vvp

Format:

```
isetp.int_cmp_type.pred_comb.is_signed  pout0, pout1, src0{.reuse}, src1{.reuse},
↪  pin0{.src_bit_modi}
```

Operands: pout0: pred; pout1: pred; src0: vreg; src1: vreg; pin0: pred

Encoding Diagram: 64b, prefix 0, opcode 11100011



Notes:

32-bit 整数比较 → predicate: 先做 (src0 INT_CMP src1) 得 1-bit cmp; pout0 = pred_comb(cmp, pin0), pout1 (若声明) = pred_comb(!cmp, pin0)。整 warp per-lane 各自独立。

pout1 不是!pout0, 而是用同一 pred_comb 跟同一 pin0 把!cmp 组合一次 (嵌套谓词模式: 外层 pin0 控制内层 cmp)。pin0=PT + pred_comb=and 时退化为 pout0 = cmp / pout1 = !cmp。

int_cmp_type: 8 valid (f 恒假 / lt / eq / le / gt / ne / ge / t 恒真)。

is_signed=signed / unsigned: s32 用符号位参与比较, u32 按位模式比较。

pred_comb: and / or / xor 三种 cmp 跟 pin0 的组合方式。

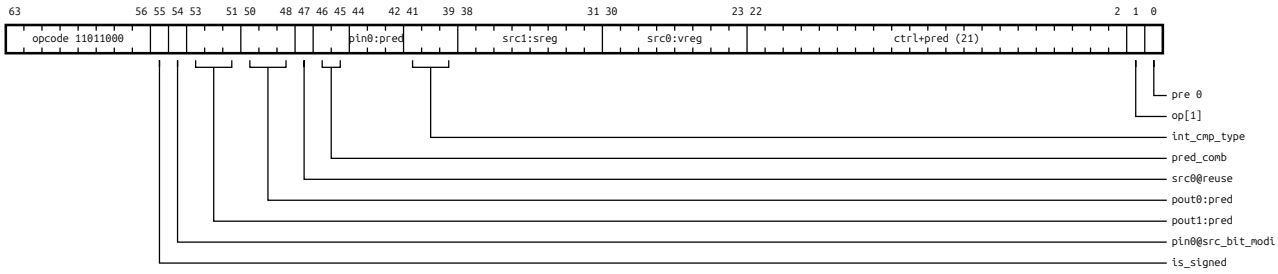
Leaf: isetp__pp_vxp

Format:

```
isetp.int_cmp_type.pred_comb.is_signed  pout0, pout1, src0{.reuse}, src1,
↪  pin0{.src_bit_modi}
```

Operands: pout0: pred; pout1: pred; src0: vreg; src1: sreg; pin0: pred

Encoding Diagram: 64b, prefix 0, opcode 111011000



Notes:

32-bit 整数比较 → predicate: 先做 (src0 INT_CMP src1) 得 1-bit cmp; pout0 = pred_comb(cmp, pin0), pout1 (若声明) = pred_comb(!cmp, pin0)。整 warp per-lane 各自独立。

pout1 不是!pout0, 而是用同一 pred_comb 跟同一 pin0 把!cmp 组合一次 (嵌套谓词模式: 外层 pin0 控制内层 cmp)。pin0=PT + pred_comb=and 时退化为 pout0 = cmp / pout1 = !cmp。

int_cmp_type: 8 valid (f 恒假 / lt / eq / le / gt / ne / ge / t 恒真)。

is_signed=signed / unsigned: s32 用符号位参与比较, u32 按位模式比较。

pred_comb: and / or / xor 三种 cmp 跟 pin0 的组合方式。

shl_add **category**: Integer compute **predicate_mode**: simt

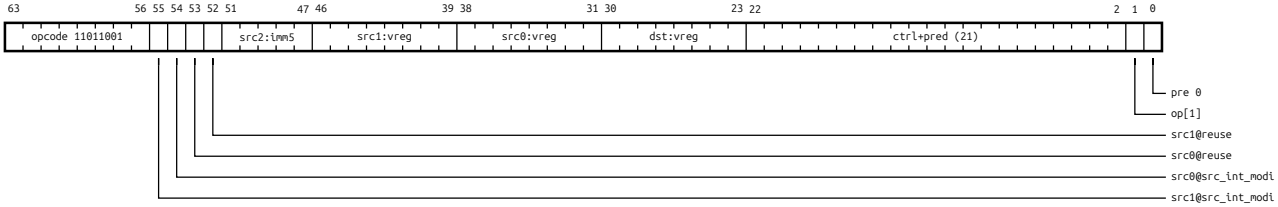
Leaf: shl_add__v_vvi

Format:

```
shl_add  dst, src0{.src_int_modi}{.reuse}, src1{.src_int_modi}{.reuse}, src2
```

Operands: dst: vreg; src0: vreg; src1: vreg; src2: imm5u

Encoding Diagram: 64b, prefix 0, opcode 111011001



Notes:

Shifted-add: dst = src0 + (src1 « src2) (mod 2^32), src2 是 5-bit 立即数 shift count (0-31)。常用于地址计算 (base + offset * stride, stride 是 2 的幂时 shift 表达)。整 warp per-lane 各自计算。

pin0 (carry-in) + pout0 (carry-out): 含 pin0 的指令形态 (v_*p / vp_*) 加入 32-bit carry chain (cin pin0 + cout pout0), 用于 multi-precision shifted-add。

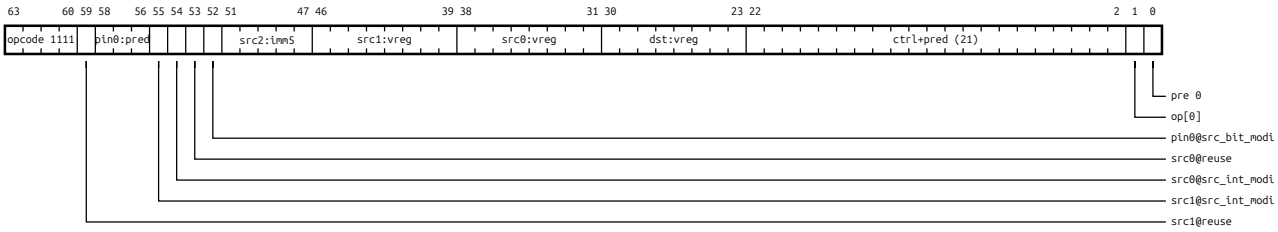
Leaf: shl_add__v_vvip

Format:

```
shl_add  dst, src0{.src_int_modi}{.reuse}, src1{.src_int_modi}{.reuse}, src2,
↪  pin0{.src_bit_modi}
```

Operands: dst: vreg; src0: vreg; src1: vreg; src2: imm5u; pin0: pred

Encoding Diagram: 64b, prefix 0, opcode 01111



Notes:

Shifted-add: dst = src0 + (src1 « src2) (mod 2^32), src2 是 5-bit 立即数 shift count (0-31)。常用于地址计算 (base + offset * stride, stride 是 2 的幂时 shift 表达)。整 warp per-lane 各自计算。

pin0 (carry-in) + pout0 (carry-out): 含 pin0 的指令形态 (v_*p / vp_*) 加入 32-bit carry chain (cin pin0 + cout pout0), 用于 multi-precision shifted-add。

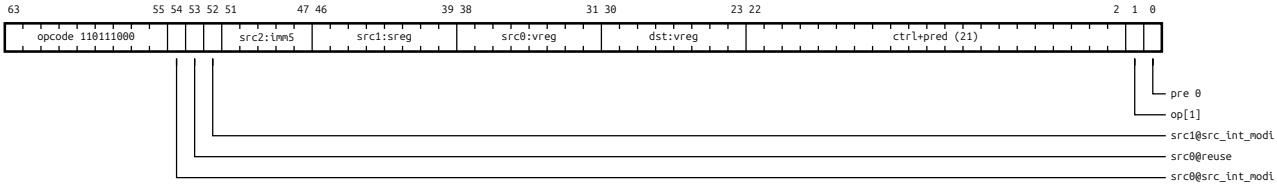
Leaf: shl_add__v_vxi

Format:

```
shl_add  dst, src0{.src_int_modi}{.reuse}, src1{.src_int_modi}, src2
```

Operands: dst: vreg; src0: vreg; src1: sreg; src2: imm5u

Encoding Diagram: 64b, prefix 0, opcode 1110111000



Notes:

Shifted-add: dst = src0 + (src1 « src2) (mod 2^32), src2 是 5-bit 立即数 shift count (0-31)。常用于地址计算 (base + offset * stride, stride 是 2 的幂时 shift 表达)。整 warp per-lane 各自计算。

pin0 (carry-in) + pout0 (carry-out): 含 pin0 的指令形态 (v_*p / vp_*) 加入 32-bit carry chain (cin pin0 + cout pout0), 用于 multi-precision shifted-add。

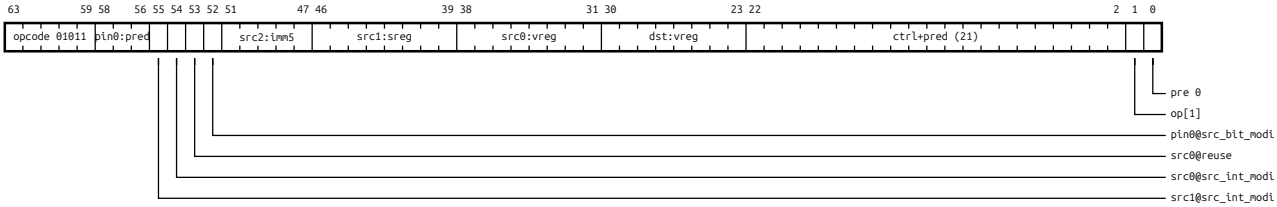
Leaf: shl_add__v_vxip

Format:

```
shl_add  dst, src0{.src_int_modi}{.reuse}, src1{.src_int_modi}, src2, pin0{.src_bit_modi}
```

Operands: dst: vreg; src0: vreg; src1: sreg; src2: imm5u; pin0: pred

Encoding Diagram: 64b, prefix 0, opcode 101011



Notes:

Shifted-add: dst = src0 + (src1 « src2) (mod 2^32), src2 是 5-bit 立即数 shift count (0-31)。常用于地址计算 (base + offset * stride, stride 是 2 的幂时 shift 表达)。整 warp per-lane 各自计算。

pin0 (carry-in) + pout0 (carry-out): 含 pin0 的指令形态 (v_*p / vp_*) 加入 32-bit carry chain (cin pin0 + cout pout0), 用于 multi-precision shifted-add。

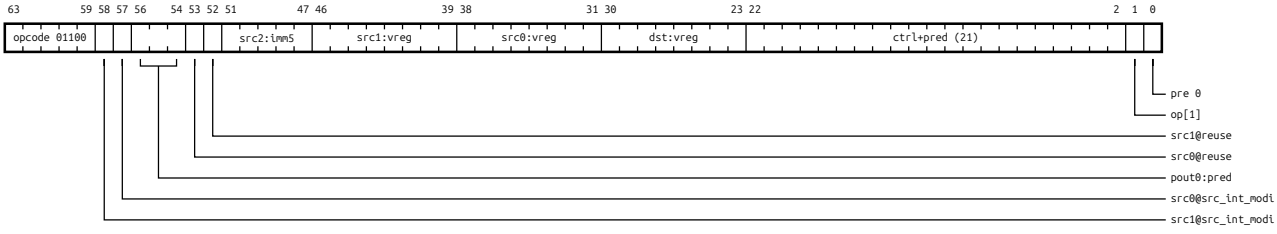
Leaf: shl_add__vp_vvi

Format:

```
shl_add  dst, pout0, src0{.src_int_modi}{.reuse}, src1{.src_int_modi}{.reuse}, src2
```

Operands: dst: vreg; pout0: pred; src0: vreg; src1: vreg; src2: imm5u

Encoding Diagram: 64b, prefix 0, opcode 101100



Notes:

Shifted-add: dst = src0 + (src1 « src2) (mod 2^32), src2 是 5-bit 立即数 shift count (0-31)。常用于地址计算 (base + offset * stride, stride 是 2 的幂时 shift 表达)。整 warp per-lane 各自计算。

pin0 (carry-in) + pout0 (carry-out): 含 pin0 的指令形态 (v_*p / vp_*) 加入 32-bit carry chain (cin pin0 + cout pout0), 用于 multi-precision shifted-add。

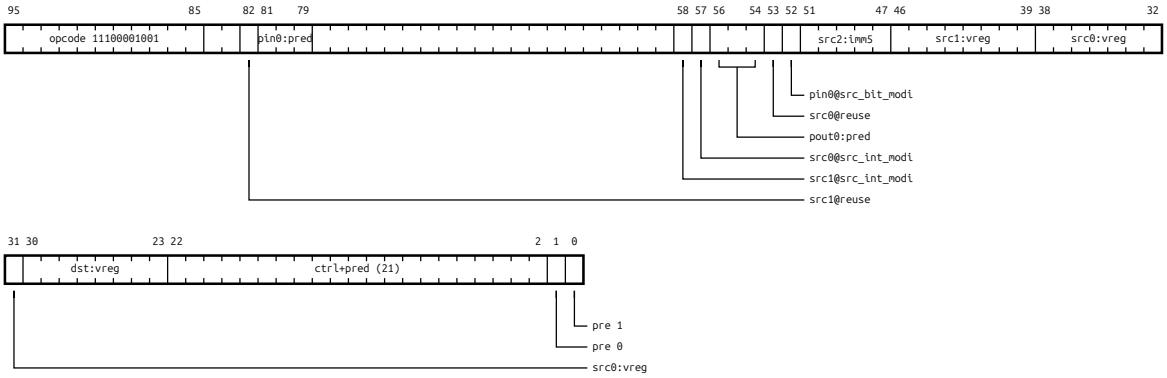
Leaf: shl_add__vp_vvip

Format:

```
shl_add  dst, pout0, src0{.src_int_modi}{.reuse}, src1{.src_int_modi}{.reuse}, src2,
↪  pin0{.src_bit_modi}
```

Operands: dst: vreg; pout0: pred; src0: vreg; src1: vreg; src2: imm5u; pin0: pred

Encoding Diagram: 96b, prefix 10, opcode 11100001001



Notes:

Shifted-add: dst = src0 + (src1 « src2) (mod 2^32), src2 是 5-bit 立即数 shift count (0-31)。常用于地址计算 (base + offset * stride, stride 是 2 的幂时 shift 表达)。整 warp per-lane 各自计算。

pin0 (carry-in) + pout0 (carry-out): 含 pin0 的指令形态 (v_*p / vp_*) 加入 32-bit carry chain (cin pin0 + cout pout0), 用于 multi-precision shifted-add。

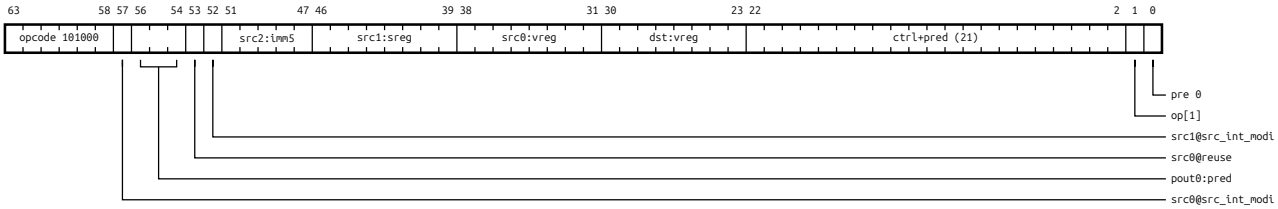
Leaf: shl_add__vp_vxi

Format:

```
shl_add  dst, pout0, src0{.src_int_modi}{.reuse}, src1{.src_int_modi}, src2
```

Operands: dst: vreg; pout0: pred; src0: vreg; src1: sreg; src2: imm5u

Encoding Diagram: 64b, prefix 0, opcode 1101000



Notes:

Shifted-add: dst = src0 + (src1 « src2) (mod 2^32), src2 是 5-bit 立即数 shift count (0-31)。常用于地址计算 (base + offset * stride, stride 是 2 的幂时 shift 表达)。整 warp per-lane 各自计算。

pin0 (carry-in) + pout0 (carry-out): 含 pin0 的指令形态 (v_*p / vp_*) 加入 32-bit carry chain (cin pin0 + cout pout0), 用于 multi-precision shifted-add。

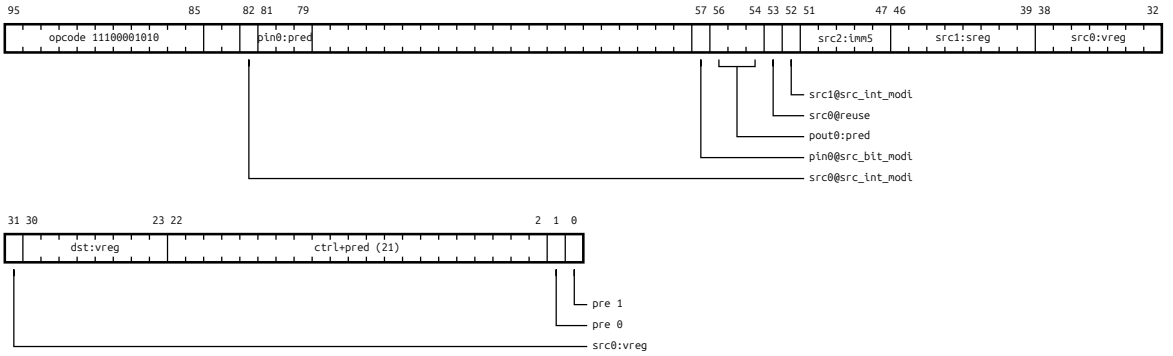
Leaf: shl_add__vp_vxip

Format:

```
shl_add  dst, pout0, src0{.src_int_modi}{.reuse}, src1{.src_int_modi}, src2,
↪  pin0{.src_bit_modi}
```

Operands: dst: vreg; pout0: pred; src0: vreg; src1: sreg; src2: imm5u; pin0: pred

Encoding Diagram: 96b, prefix 10, opcode 11100001010



Notes:

Shifted-add: dst = src0 + (src1 « src2) (mod 2^32), src2 是 5-bit 立即数 shift count (0-31)。常用于地址计算 (base + offset * stride, stride 是 2 的幂时 shift 表达)。整 warp per-lane 各自计算。

pin0 (carry-in) + pout0 (carry-out): 含 pin0 的指令形态 (v_*p / vp_*) 加入 32-bit carry chain (cin pin0 + cout pout0), 用于 multi-precision shifted-add。

uiadd3　**category**: Integer compute　**predicate_mode**: uniform

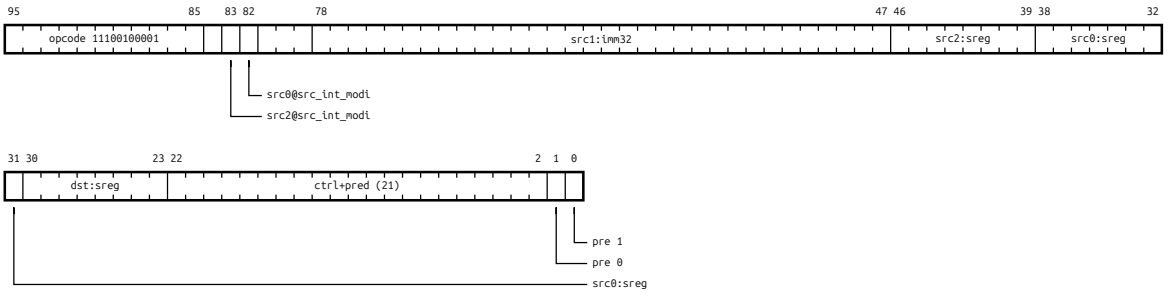
Leaf: uiadd3__x_xix

Format:

```
uiadd3  dst, src0{.src_int_modi}, src1, src2{.src_int_modi}
```

Operands: dst: sreg; src0: sreg; src1: imm32u; src2: sreg

Encoding Diagram: 96b, prefix 10, opcode 11100100001



Notes:

三入整数加法 uniform 变体: 普通指令形态 dst = src0 + src1 + src2; carry mode 指令形态 {pout1, pout0, dst} = src0 + src1 + src2 + (2·pin1 + pin0)。所有 operand 都是 sreg / imm + upred, 整 warp 共享。

包含 2-src add 用例: src2=URZ 退化为 dst = src0 + src1。

双 carry chain 物理含义: 跟 iadd3 一致 (cout / cin 各 2-bit, 编码 0..3 之和), 但 pred 类型为 upred (uniform predicate)。

src 修饰符: 普通指令形态 src_int_modi (neg / abs), carry mode src_bit_modi (id / inv)。跟 iadd3 一致。

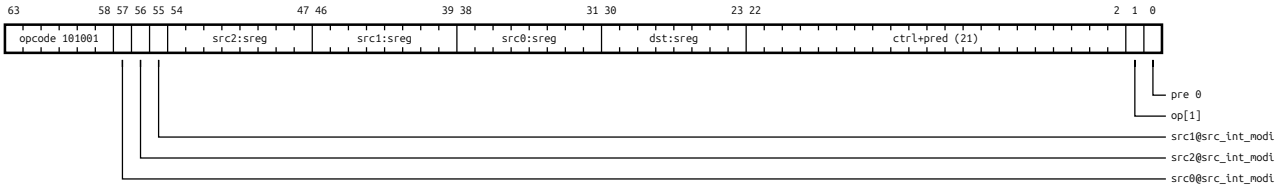
Leaf: uiadd3__x_xxx

Format:

```
uiadd3  dst, src0{.src_int_modi}, src1{.src_int_modi}, src2{.src_int_modi}
```

Operands: dst: sreg; src0: sreg; src1: sreg; src2: sreg

Encoding Diagram: 64b, prefix 0, opcode 1101001



Notes:

三入整数加法 uniform 变体: 普通指令形态 dst = src0 + src1 + src2; carry mode 指令形态 {pout1, pout0, dst} = src0 + src1 + src2 + (2·pin1 + pin0)。所有 operand 都是 sreg / imm + upred, 整 warp 共享。

包含 2-src add 用例: src2=URZ 退化为 dst = src0 + src1。

双 carry chain 物理含义: 跟 iadd3 一致 (cout / cin 各 2-bit, 编码 0..3 之和), 但 pred 类型为 upred (uniform predicate)。

src 修饰符: 普通指令形态 src_int_modi (neg / abs), carry mode src_bit_modi (id / inv)。跟 iadd3 一致。

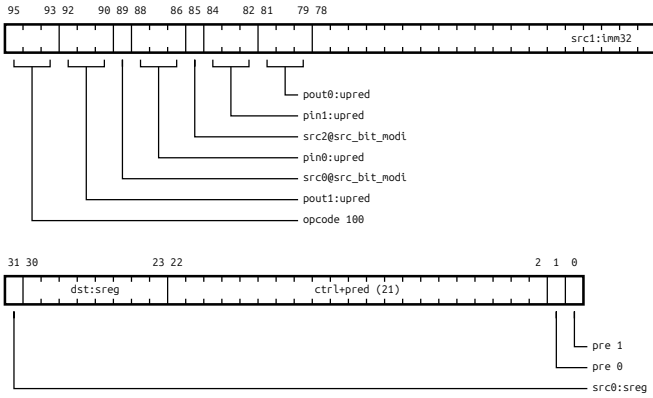
Leaf: uiadd3__xpp_xixpp

Format:

```
uiadd3  dst, pout0, pout1, src0{.src_bit_modi}, src1, src2{.src_bit_modi}, pin0, pin1
```

Operands: dst: sreg; pout0: upred; pout1: upred; src0: sreg; src1: imm32u; src2: sreg; pin0: upred; pin1: upred

Encoding Diagram: 96b, prefix 10, opcode 100



Notes:

三入整数加法 uniform 变体: 普通指令形态 dst = src0 + src1 + src2; carry mode 指令形态 {pout1, pout0, dst} = src0 + src1 + src2 + (2·pin1 + pin0)。所有 operand 都是 sreg / imm + upred, 整 warp 共享。

包含 2-src add 用例: src2=URZ 退化为 dst = src0 + src1。

双 carry chain 物理含义: 跟 iadd3 一致 (cout / cin 各 2-bit, 编码 0..3 之和), 但 pred 类型为 upred (uniform predicate)。

src 修饰符: 普通指令形态 src_int_modi (neg / abs), carry mode src_bit_modi (id / inv)。跟 iadd3 一致。

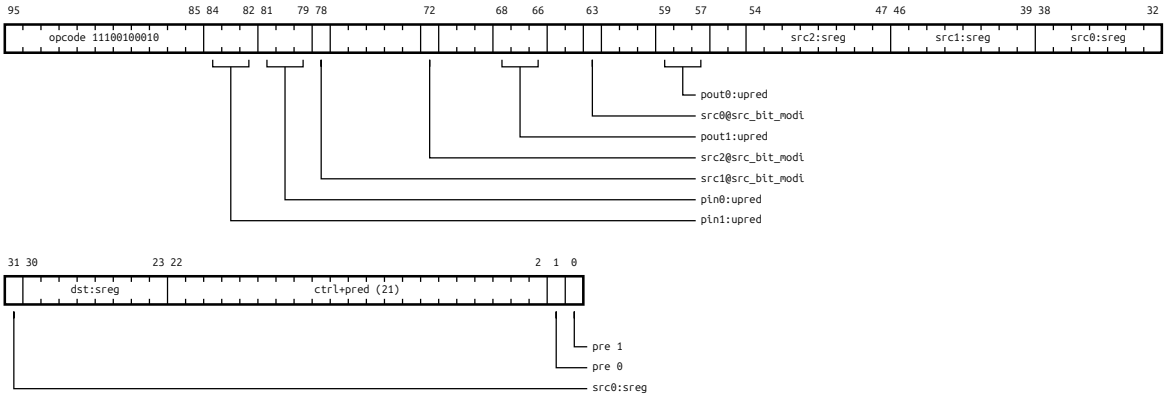
Leaf: uiadd3__xpp_xxxpp

Format:

```
uiadd3  dst, pout0, pout1, src0{.src_bit_modi}, src1{.src_bit_modi}, src2{.src_bit_modi},
↪  pin0, pin1
```

Operands: dst: sreg; pout0: upred; pout1: upred; src0: sreg; src1: sreg; src2: sreg; pin0: upred; pin1: upred

Encoding Diagram: 96b, prefix 10, opcode 11100100010



Notes:

三入整数加法 uniform 变体: 普通指令形态 dst = src0 + src1 + src2; carry mode 指令形态 {pout1, pout0, dst} = src0 + src1 + src2 + (2·pin1 + pin0)。所有 operand 都是 sreg / imm + upred, 整 warp 共享。

包含 2-src add 用例: src2=URZ 退化为 dst = src0 + src1。

双 carry chain 物理含义: 跟 iadd3 一致 (cout / cin 各 2-bit, 编码 0..3 之和), 但 pred 类型为 upred (uniform predicate)。

src 修饰符: 普通指令形态 src_int_modi (neg / abs), carry mode src_bit_modi (id / inv)。跟 iadd3 一致。

uimad　**category**: Integer compute　**predicate_mode**: uniform

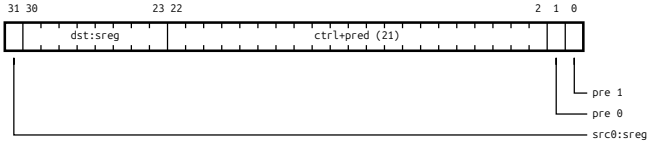
Leaf: uimad__x_xix

Format:

```
uimad.is_signed  dst, src0{.src_int_modi}, src1, src2{.src_int_modi}
```

Operands: dst: sreg; src0: sreg; src1: imm32u; src2: sreg

Encoding Diagram: 96b, prefix 10, opcode 11100100011



Notes:

32-bit 整数 fused multiply-add uniform 变体: dst = (src0 × src1 + src2) mod 2^32, 结果 wrap 32 bit 无 carry-out。所有 operand sreg / imm, 整 warp 共享。

is_signed=signed / unsigned: 跟 imad 一致 (作用在乘法的符号解释)。

需要 64-bit uniform 结果走 uimad_wide。

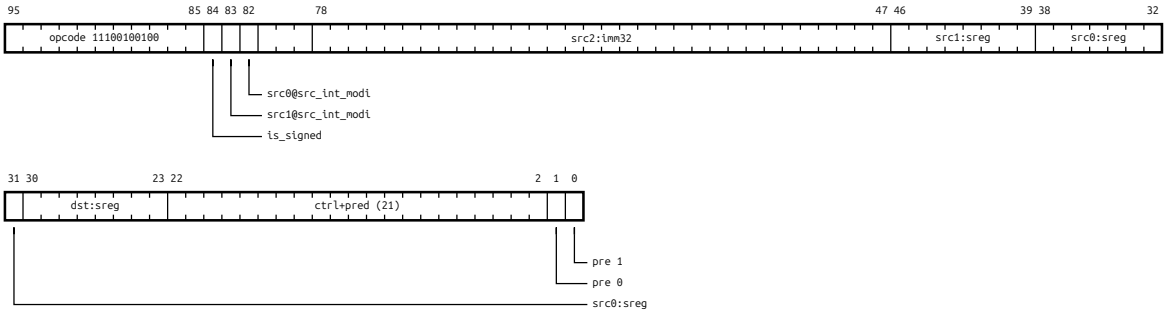
Leaf: uimad__x_xxi

Format:

```
uimad.is_signed  dst, src0{.src_int_modi}, src1{.src_int_modi}, src2
```

Operands: dst: sreg; src0: sreg; src1: sreg; src2: imm32u

Encoding Diagram: 96b, prefix 10, opcode 11100100100



Notes:

32-bit 整数 fused multiply-add uniform 变体: dst = (src0 × src1 + src2) mod 2^32, 结果 wrap 32 bit 无 carry-out。所有 operand sreg / imm, 整 warp 共享。

is_signed=signed / unsigned: 跟 imad 一致 (作用在乘法的符号解释)。

需要 64-bit uniform 结果走 uimad_wide。

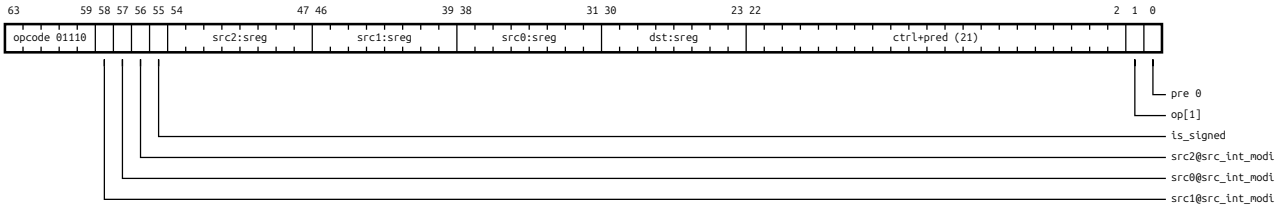
Leaf: uimad__x_xxx

Format:

```
uimad.is_signed  dst, src0{.src_int_modi}, src1{.src_int_modi}, src2{.src_int_modi}
```

Operands: dst: sreg; src0: sreg; src1: sreg; src2: sreg

Encoding Diagram: 64b, prefix 0, opcode 101110



Notes:

32-bit 整数 fused multiply-add uniform 变体: dst = (src0 × src1 + src2) mod 2^32, 结果 wrap 32 bit 无 carry-out。所有 operand sreg / imm, 整 warp 共享。

is_signed=signed / unsigned: 跟 imad 一致 (作用在乘法的符号解释)。

需要 64-bit uniform 结果走 uimad_wide。

uimad_wide　**category**: Integer compute　**predicate_mode**: uniform

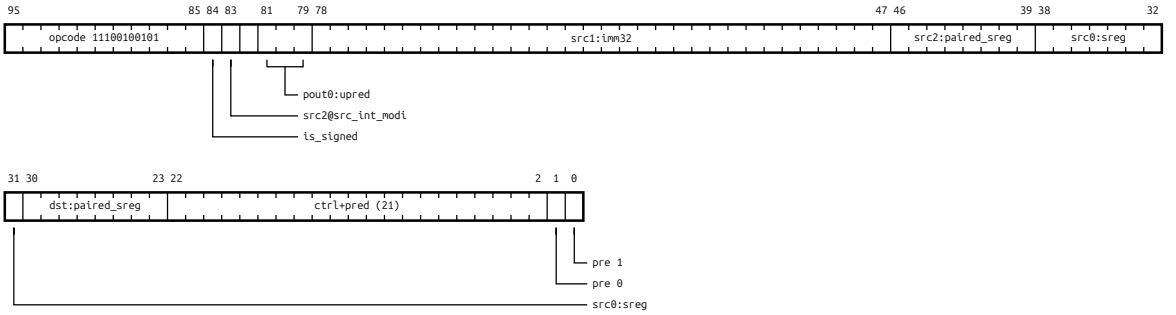
Leaf: uimad_wide__xp_xix

Format:

```
uimad_wide.is_signed  dst, pout0, src0, src1, src2{.src_int_modi}
```

Operands: dst: paired_sreg; pout0: upred; src0: sreg; src1: imm32u; src2: paired_sreg

Encoding Diagram: 96b, prefix 10, opcode 11100100101



Notes:

Wide 整数 multiply-add uniform 变体: {dst, dst+1} = src0 × src1 + {src2, src2+1}, pout0 = carry-out。所有 operand 都是 sreg / imm + upred, 整 warp 共享。

dst 占 2 个相邻 sreg (偶对齐); src2 是 reg 时也占 2 个相邻 sreg。pout0 是 upred (UPT 可抑制输出)。

is_signed=signed / unsigned: 跟 imad_wide 一致。

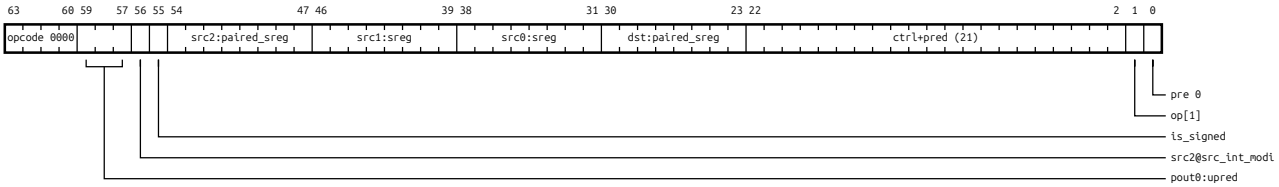
Leaf: uimad_wide__xp_xxx

Format:

```
uimad_wide.is_signed  dst, pout0, src0, src1, src2{.src_int_modi}
```

Operands: dst: paired_sreg; pout0: upred; src0: sreg; src1: sreg; src2: paired_sreg

Encoding Diagram: 64b, prefix 0, opcode 10000



Notes:

Wide 整数 multiply-add uniform 变体: {dst, dst+1} = src0 × src1 + {src2, src2+1}, pout0 = carry-out。所有 operand 都是 sreg / imm + upred, 整 warp 共享。

dst 占 2 个相邻 sreg (偶对齐); src2 是 reg 时也占 2 个相邻 sreg。pout0 是 upred (UPT 可抑制输出)。

is_signed=signed / unsigned: 跟 imad_wide 一致。

uimnmx **category**: Integer compute **predicate_mode**: uniform

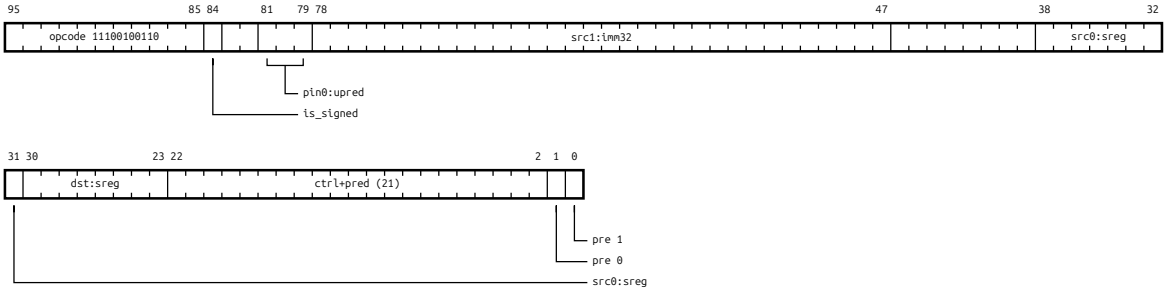
Leaf: uimnmx__x_xi

Format:

```
uimnmx.is_signed  dst, src0, src1, pin0
```

Operands: dst: sreg; src0: sreg; src1: imm32u; pin0: upred

Encoding Diagram: 96b, prefix 10, opcode 11100100110



Notes:

32-bit 整数 min/max uniform 变体: dst = pin0 ? min(src0, src1) : max(src0, src1), pin0 是 upred (per-warp), 整 warp 同步取 min 或 max。

is_signed=signed / unsigned: 跟 imnmx 一致。

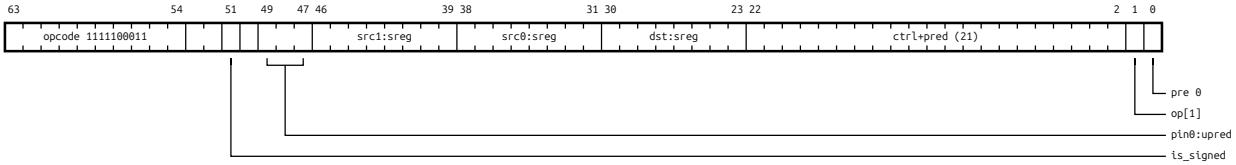
Leaf: uimnmx__x_xx

Format:

```
uimnmx.is_signed  dst, src0, src1, pin0
```

Operands: dst: sreg; src0: sreg; src1: sreg; pin0: upred

Encoding Diagram: 64b, prefix 0, opcode 11111100011



Notes:

32-bit 整数 min/max uniform 变体: dst = pin0 ? min(src0, src1) : max(src0, src1), pin0 是 upred (per-warp), 整 warp 同步取 min 或 max。

is_signed=signed / unsigned: 跟 imnmx 一致。

uisetp **category**: Integer compute **predicate_mode**: uniform

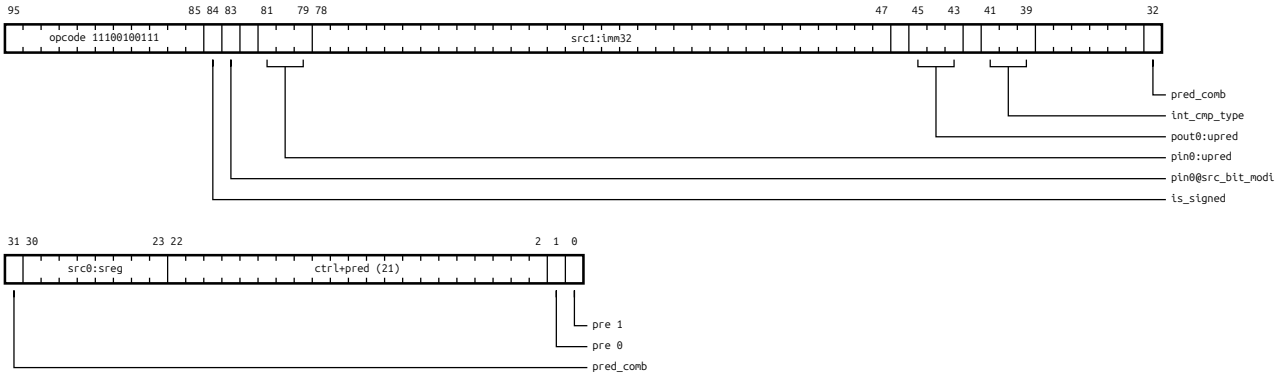
Leaf: uisetp__p_xip

Format:

```
uisetp.int_cmp_type.pred_comb.is_signed  pout0, src0, src1, pin0{.src_bit_modi}
```

Operands: pout0: upred; src0: sreg; src1: imm32u; pin0: upred

Encoding Diagram: 96b, prefix 10, opcode 11100100111



Notes:

32-bit 整数比较 → upred 的 uniform 变体: 所有 src 是 sreg / imm, 所有 pred 是 upred (per-warp)。整 warp 共享一份比较结果。

int_cmp_type: 8 valid (f / lt / eq / le / gt / ne / ge / t), 跟 isetp 一致。

is_signed / pred_comb: 跟 isetp 完全一致 (双 pred 输出语义同源, pout1 是!cmp 跟 pin0 组合)。

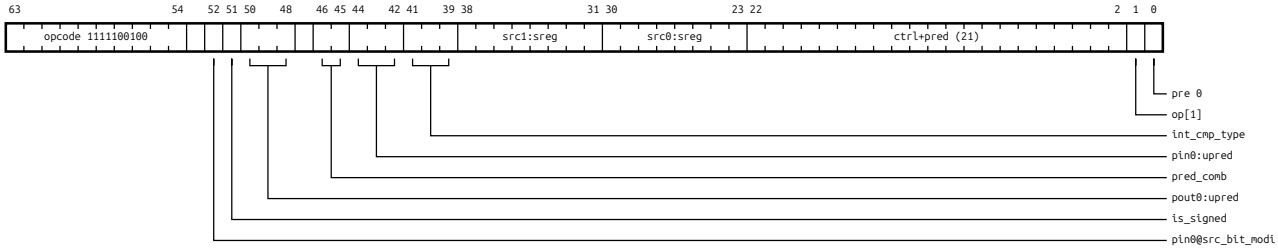
Leaf: uisetp__p_xxp

Format:

```
uisetp.int_cmp_type.pred_comb.is_signed  pout0, src0, src1, pin0{.src_bit_modi}
```

Operands: pout0: upred; src0: sreg; src1: sreg; pin0: upred

Encoding Diagram: 64b, prefix 0, opcode 11111100100



Notes:

32-bit 整数比较 → upred 的 uniform 变体: 所有 src 是 sreg / imm, 所有 pred 是 upred (per-warp)。整 warp 共享一份比较结果。

int_cmp_type: 8 valid (f / lt / eq / le / gt / ne / ge / t), 跟 isetp 一致。

is_signed / pred_comb: 跟 isetp 完全一致 (双 pred 输出语义同源, pout1 是!cmp 跟 pin0 组合)。

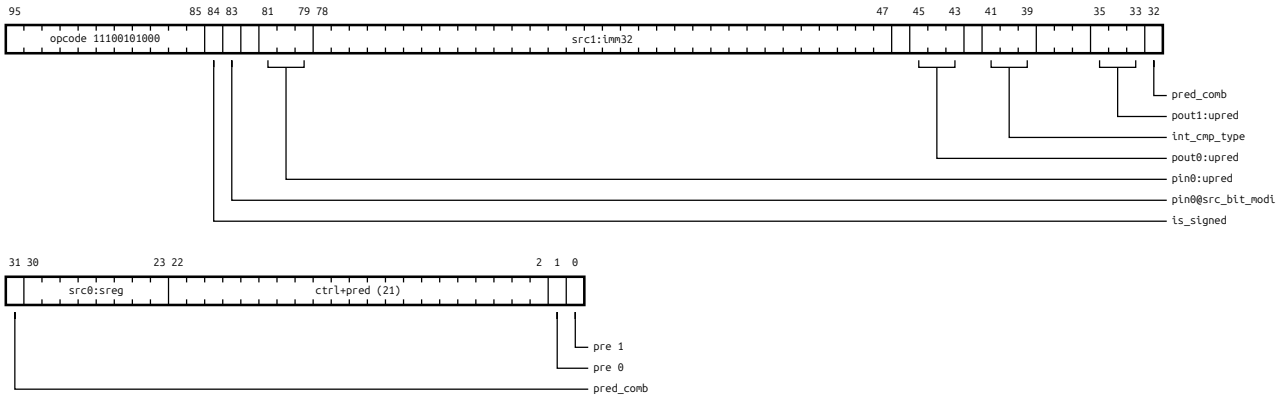
Leaf: uisetp__pp_xip

Format:

```
uisetp.int_cmp_type.pred_comb.is_signed  pout0, pout1, src0, src1, pin0{.src_bit_modi}
```

Operands: pout0: upred; pout1: upred; src0: sreg; src1: imm32u; pin0: upred

Encoding Diagram: 96b, prefix 10, opcode 11100101000



Notes:

32-bit 整数比较 → upred 的 uniform 变体: 所有 src 是 sreg / imm, 所有 pred 是 upred (per-warp)。整 warp 共享一份比较结果。

int_cmp_type: 8 valid (f / lt / eq / le / gt / ne / ge / t), 跟 isetp 一致。

is_signed / pred_comb: 跟 isetp 完全一致 (双 pred 输出语义同源, pout1 是!cmp 跟 pin0 组合)。

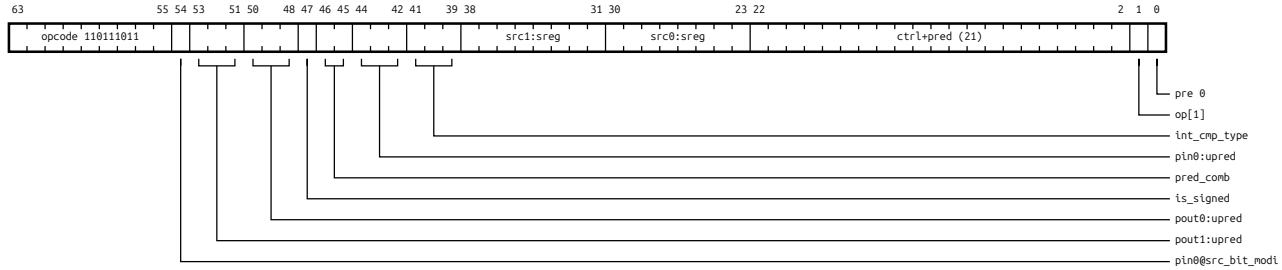
Leaf: uisetp__pp_xxp

Format:

```
uisetp.int_cmp_type.pred_comb.is_signed  pout0, pout1, src0, src1, pin0{.src_bit_modi}
```

Operands: pout0: upred; pout1: upred; src0: sreg; src1: sreg; pin0: upred

Encoding Diagram: 64b, prefix 0, opcode 1110111011



Notes:

32-bit 整数比较 → upred 的 uniform 变体: 所有 src 是 sreg / imm, 所有 pred 是 upred (per-warp)。整 warp 共享一份比较结果。

int_cmp_type: 8 valid (f / lt / eq / le / gt / ne / ge / t), 跟 isetp 一致。

is_signed / pred_comb: 跟 isetp 完全一致 (双 pred 输出语义同源, pout1 是!cmp 跟 pin0 组合)。

ushl_add **category**: Integer compute **predicate_mode**: uniform

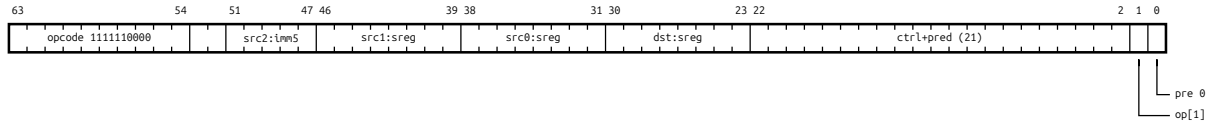
Leaf: ushl_add__x_xxi

Format:

```
ushl_add  dst, src0, src1, src2
```

Operands: dst: sreg; src0: sreg; src1: sreg; src2: imm5u

Encoding Diagram: 64b, prefix 0, opcode 11111110000



Notes:

Shifted-add uniform 变体: dst = src0 + (src1 « src2)。所有 operand 都是 sreg / imm, 整 warp 共享。

uniform 路径仅保留 1 个指令形态 (无 pin0/pout0 carry chain), 编译器要 multi-precision 走 simt shl_add 序列代偿。

14.7.5 Bitwise and logic

bfe **category**: Bitwise and logic **predicate_mode**: simt

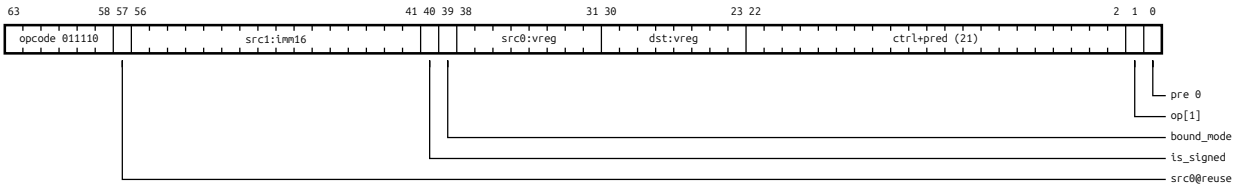
Leaf: bfe__v_vi

Format:

```
bfe.is_signed{.bound_mode}  dst, src0{.reuse}, src1
```

Operands: dst: vreg; src0: vreg; src1: imm16u

Encoding Diagram: 64b, prefix 0, opcode 1011110



Notes:

Bit field extract: 从 src0 抽取 [pos+len-1 : pos] 位段写 dst。整 warp per-lane 各自计算。

src1 编码 pos 和 len: src1[7:0] = pos (起始位置), src1[15:8] = len (位段长度)。

is_signed=signed: 按抽出位段的最高位做符号扩展到 32-bit (s8/s16 → s32 的位段路径)。

is_signed=unsigned: 高位补 0 (zero-extend)。

len=0 时 dst=0; len > 32 - pos 时硬件按 bound_mode 处理。

bound_mode=clamp: pos+len 越界时 clamp 到 32 (取到末尾停)。

bound_mode=wrap: 取低 5-bit 绕回。

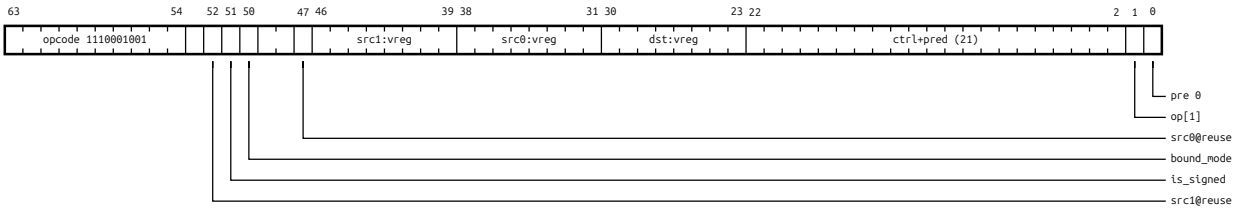
Leaf: bfe__v_vv

Format:

```
bfe.is_signed{.bound_mode}  dst, src0{.reuse}, src1{.reuse}
```

Operands: dst: vreg; src0: vreg; src1: vreg

Encoding Diagram: 64b, prefix 0, opcode 11110001001



Notes:

Bit field extract: 从 src0 抽取 [pos+len-1 : pos] 位段写 dst。整 warp per-lane 各自计算。

src1 编码 pos 和 len: src1[7:0] = pos (起始位置), src1[15:8] = len (位段长度)。

is_signed=signed: 按抽出位段的最高位做符号扩展到 32-bit (s8/s16 → s32 的位段路径)。

is_signed=unsigned: 高位补 0 (zero-extend)。

len=0 时 dst=0; len > 32 - pos 时硬件按 bound_mode 处理。

bound_mode=clamp: pos+len 越界时 clamp 到 32 (取到末尾停)。

bound_mode=wrap: 取低 5-bit 绕回。

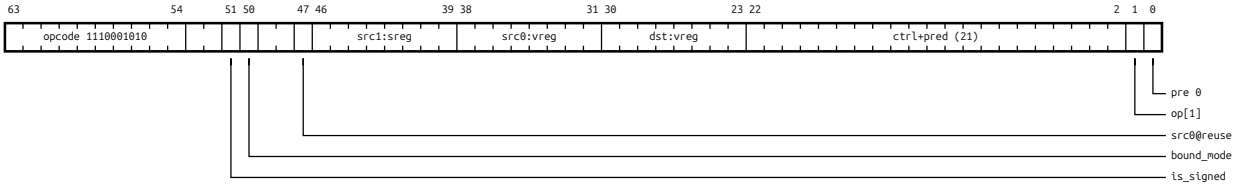
Leaf: bfe__v_vx

Format:

```
bfe.is_signed{.bound_mode}  dst, src0{.reuse}, src1
```

Operands: dst: vreg; src0: vreg; src1: sreg

Encoding Diagram: 64b, prefix 0, opcode 11110001010



Notes:

Bit field extract: 从 src0 抽取 [pos+len-1 : pos] 位段写 dst。整 warp per-lane 各自计算。

src1 编码 pos 和 len: src1[7:0] = pos (起始位置), src1[15:8] = len (位段长度)。

is_signed=signed: 按抽出位段的最高位做符号扩展到 32-bit (s8/s16 → s32 的位段路径)。

is_signed=unsigned: 高位补 0 (zero-extend)。

len=0 时 dst=0; len > 32 - pos 时硬件按 bound_mode 处理。

bound_mode=clamp: pos+len 越界时 clamp 到 32 (取到末尾停)。

bound_mode=wrap: 取低 5-bit 绕回。

bfi **category**: Bitwise and logic **predicate_mode**: simt

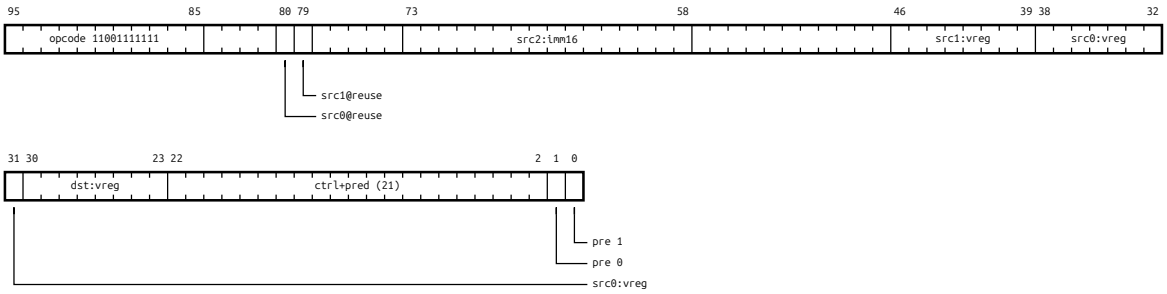
Leaf: bfi__v_vvi

Format:

```
bfi  dst, src0{.reuse}, src1{.reuse}, src2
```

Operands: dst: vreg; src0: vreg; src1: vreg; src2: imm16u

Encoding Diagram: 96b, prefix 10, opcode 11001111111



Notes:

Bit field insert: 把 src0 的低 len 位插入到 src1 的 [pos+len-1 : pos] 区间, 其它位保留 src1 原值, 结果写 dst。等价 lop3 + shf + lop3 三条序列, 单条更紧。整 warp per-lane 各自计算。

src2 编码 pos 和 len: src2[7:0] = pos, src2[15:8] = len (跟 bfe 一致)。

边界: len=0 时 dst = src1 不变; pos ≥ 32 时 dst = src1 不变; len > 32 时按 32 处理。

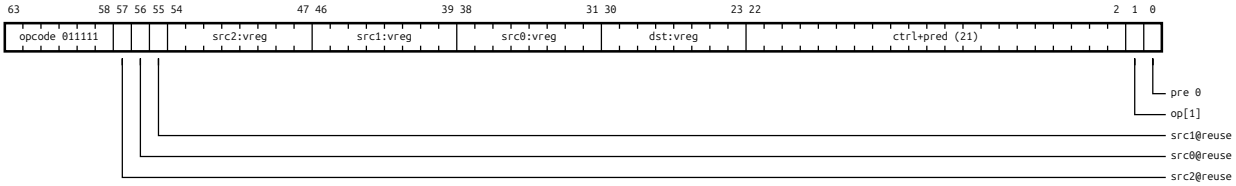
Leaf: bfi__v_vvv

Format:

```
bfi  dst, src0{.reuse}, src1{.reuse}, src2{.reuse}
```

Operands: dst: vreg; src0: vreg; src1: vreg; src2: vreg

Encoding Diagram: 64b, prefix 0, opcode 1011111



Notes:

Bit field insert: 把 src0 的低 len 位插入到 src1 的 [pos+len-1 : pos] 区间, 其它位保留 src1 原值, 结果写 dst。等价 lop3 + shf + lop3 三条序列, 单条更紧。整 warp per-lane 各自计算。

src2 编码 pos 和 len: src2[7:0] = pos, src2[15:8] = len (跟 bfe 一致)。

边界: len=0 时 dst = src1 不变; pos ≥ 32 时 dst = src1 不变; len > 32 时按 32 处理。

bmsk **category**: Bitwise and logic **predicate_mode**: simt

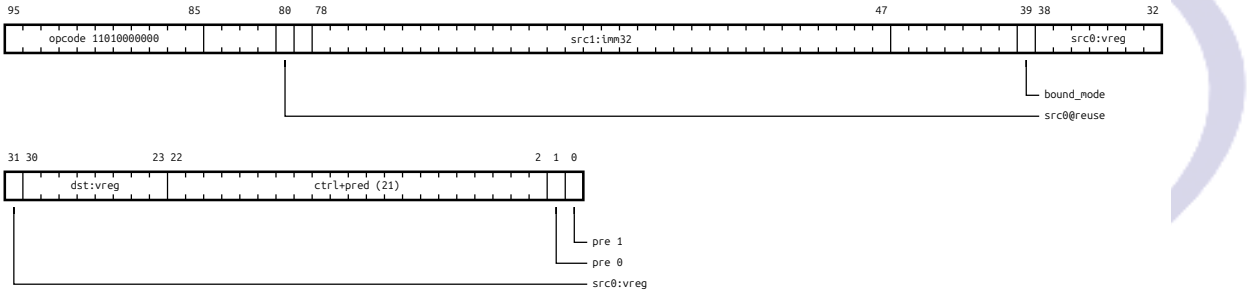
Leaf: bmsk__v_vi

Format:

```
bmsk{.bound_mode}  dst, src0{.reuse}, src1
```

Operands: dst: vreg; src0: vreg; src1: imm32u

Encoding Diagram: 96b, prefix 10, opcode 11010000000



Notes:

Bit mask 生成: dst[pos+len-1 : pos] 全 1, 其它位 0。src0 = len (mask 长度), src1 = pos (起始位置)。例: bmsk(len=4, pos=8) → 0x0F00。整 warp 32 个 active thread 各自计算。

bound_mode=clamp: len > 32 时 mask 全 1 (饱和到 32-bit); pos >= 32 时 dst = 0 (mask 整体被推出寄存器)。

bound_mode=wrap: 硬件取 src0[4:0] / src1[4:0] 做 mod-32 绕回 (不做越界检查, 编译器要保证 len/pos 在范围内)。

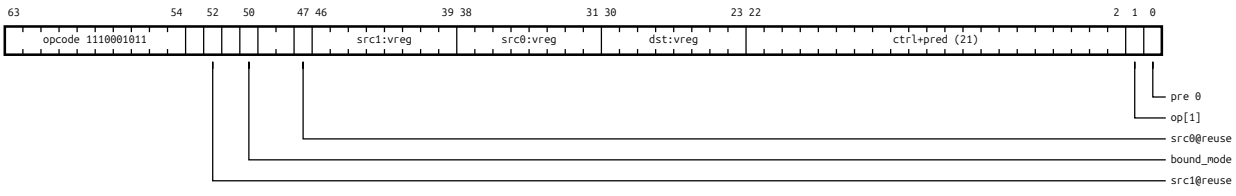
Leaf: bmsk__v_vv

Format:

```
bmsk{.bound_mode}  dst, src0{.reuse}, src1{.reuse}
```

Operands: dst: vreg; src0: vreg; src1: vreg

Encoding Diagram: 64b, prefix 0, opcode 11110001011



Notes:

Bit mask 生成: dst[pos+len-1 : pos] 全 1, 其它位 0。src0 = len (mask 长度), src1 = pos (起始位置)。例: bmsk(len=4, pos=8) → 0x0F00。整 warp 32 个 active thread 各自计算。

bound_mode=clamp: len > 32 时 mask 全 1 (饱和到 32-bit); pos >= 32 时 dst = 0 (mask 整体被推出寄存器)。

bound_mode=wrap: 硬件取 src0[4:0] / src1[4:0] 做 mod-32 绕回 (不做越界检查, 编译器要保证 len/pos 在范围内)。

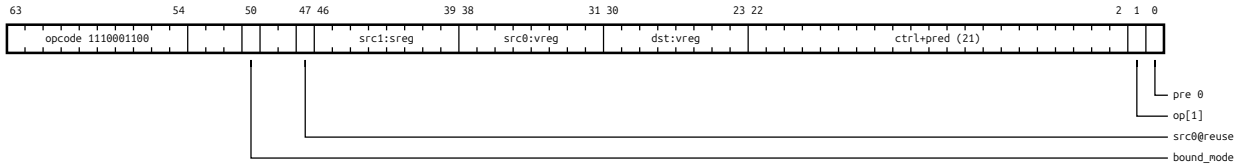
Leaf: bmsk__v_vx

Format:

```
bmsk{.bound_mode}  dst, src0{.reuse}, src1
```

Operands: dst: vreg; src0: vreg; src1: sreg

Encoding Diagram: 64b, prefix 0, opcode 11110001100



Notes:

Bit mask 生成: dst[pos+len-1 : pos] 全 1, 其它位 0。src0 = len (mask 长度), src1 = pos (起始位置)。例: bmsk(len=4, pos=8) → 0x0F00。整 warp 32 个 active thread 各自计算。

bound_mode=clamp: len > 32 时 mask 全 1 (饱和到 32-bit); pos >= 32 时 dst = 0 (mask 整体被推出寄存器)。

bound_mode=wrap: 硬件取 src0[4:0] / src1[4:0] 做 mod-32 绕回 (不做越界检查, 编译器要保证 len/pos 在范围内)。

brev　**category**: Bitwise and logic　**predicate_mode**: simt

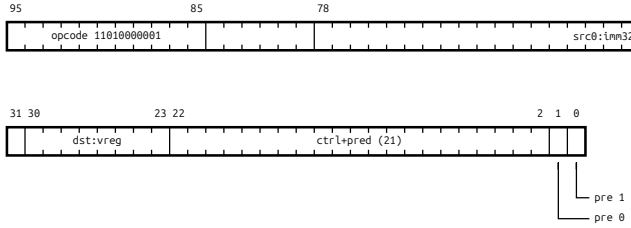
Leaf: brev__v_i

Format:

```
brev  dst, src0
```

Operands: dst: vreg; src0: imm32u

Encoding Diagram: 96b, prefix 10, opcode 11010000001



Notes:

32-bit 位反转: dst[i] = src0[31-i] for i in 0..31 (bit[0] ↔ bit[31], bit[1] ↔ bit[30], 依次类推)。整 warp per-lane 各自计算。

无 modifier; 单输入位变换。

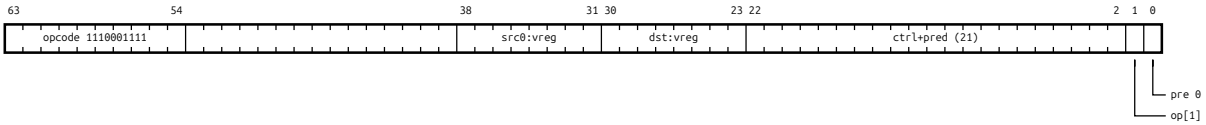
Leaf: brev__v_v

Format:

```
brev  dst, src0
```

Operands: dst: vreg; src0: vreg

Encoding Diagram: 64b, prefix 0, opcode 11110001111



Notes:

32-bit 位反转: dst[i] = src0[31-i] for i in 0..31 (bit[0] ↔ bit[31], bit[1] ↔ bit[30], 依次类推)。整 warp per-lane 各自计算。

无 modifier; 单输入位变换。

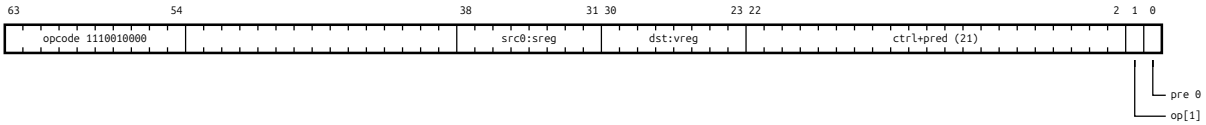
Leaf: brev__v_x

Format:

```
brev  dst, src0
```

Operands: dst: vreg; src0: sreg

Encoding Diagram: 64b, prefix 0, opcode 11110010000



Notes:

32-bit 位反转: dst[i] = src0[31-i] for i in 0..31 (bit[0] ↔ bit[31], bit[1] ↔ bit[30], 依次类推)。整 warp per-lane 各自计算。

无 modifier; 单输入位变换。

flo **category**: Bitwise and logic **predicate_mode**: simt

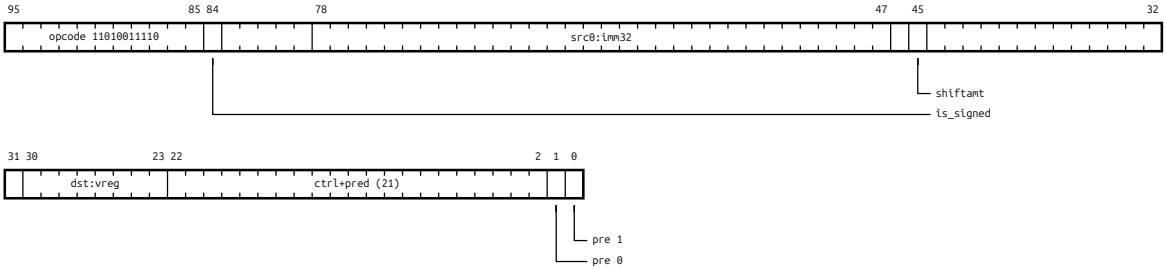
Leaf: flo__v_i

Format:

```
flo.is_signed{.shiftamt}  dst, src0
```

Operands: dst: vreg; src0: imm32u

Encoding Diagram: 96b, prefix 10, opcode 11010011110



Notes:

Find Leading One: 从 MSB (i=31) 向 LSB 扫描, 找第一个 set bit 的位置 (等价 31 - clz(src0))。src0=0 时 dst = -1 (0xFFFFFFFF) 表示' 没找到'。整 warp per-lane 各自计算。

is_signed=unsigned (默认): 找第一个 ‘1’ bit。

is_signed=signed: src0 < 0 (符号位 1) 时找第一个 ‘0’ bit (跳过连续符号位); src0 ≥ 0 时仍找第一个 ‘1’ bit。常用于浮点 normalization 或可变长 int encoding。

shiftamt=disable (默认): 返回 bit position (0..31)。

shiftamt=SH: 返回 shift-left count = 31 - bit_position, 直接拿来把 first set bit shift 到 MSB (一条移位省一条减法)。

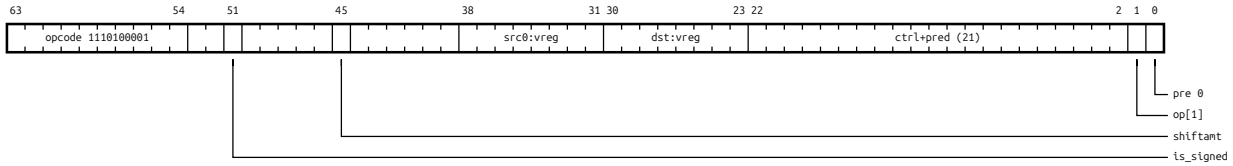
Leaf: flo__v_v

Format:

```
flo.is_signed{.shiftamt}  dst, src0
```

Operands: dst: vreg; src0: vreg

Encoding Diagram: 64b, prefix 0, opcode 11110100001



Notes:

Find Leading One: 从 MSB (i=31) 向 LSB 扫描, 找第一个 set bit 的位置 (等价 31 - clz(src0))。src0=0 时 dst = -1 (0xFFFFFFFF) 表示’ 没找到’。整 warp per-lane 各自计算。

is_signed=unsigned (默认): 找第一个 ‘1’ bit。

is_signed=signed: src0 < 0 (符号位 1) 时找第一个 ‘0’ bit (跳过连续符号位); src0 ≥ 0 时仍找第一个 ‘1’ bit。常用于浮点 normalization 或可变长 int encoding。

shiftamt=disable (默认): 返回 bit position (0..31)。

shiftamt=SH: 返回 shift-left count = 31 - bit_position, 直接拿来把 first set bit shift 到 MSB (一条移位省一条减法)。

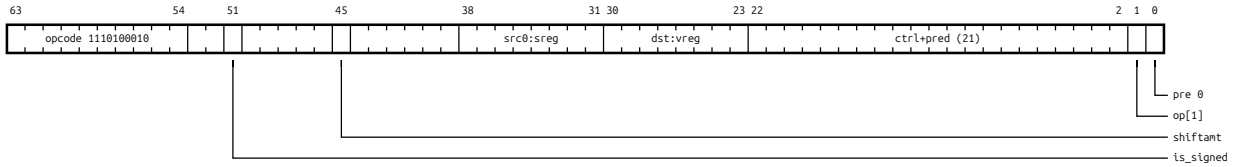
Leaf: flo__v_x

Format:

```
flo.is_signed{.shiftamt}  dst, src0
```

Operands: dst: vreg; src0: sreg

Encoding Diagram: 64b, prefix 0, opcode 11110100010



Notes:

Find Leading One: 从 MSB (i=31) 向 LSB 扫描, 找第一个 set bit 的位置 (等价 31 - clz(src0))。src0=0 时 dst = -1 (0xFFFFFFFF) 表示’ 没找到’。整 warp per-lane 各自计算。

is_signed=unsigned (默认): 找第一个 ‘1’ bit。

is_signed=signed: src0 < 0 (符号位 1) 时找第一个 ‘0’ bit (跳过连续符号位); src0 ≥ 0 时仍找第一个 ‘1’ bit。常用于浮点 normalization 或可变长 int encoding。

shiftamt=disable (默认): 返回 bit position (0..31)。

shiftamt=SH: 返回 shift-left count = 31 - bit_position, 直接拿来把 first set bit shift 到 MSB (一条移位省一条减法)。

lop3 **category**: Bitwise and logic **predicate_mode**: simt

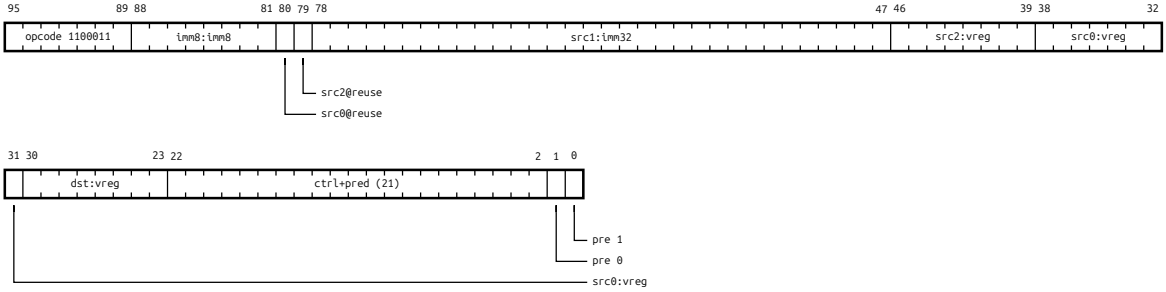
Leaf: lop3__v_vivi

Format:

```
lop3  dst, src0{.reuse}, src1, src2{.reuse}, imm8
```

Operands: dst: vreg; src0: vreg; src1: imm32u; src2: vreg; imm8: imm8u

Encoding Diagram: 96b, prefix 10, opcode 1100011



Notes:

3-input 位逻辑: dst[i] = LUT[(src0[i] « 2) | (src1[i] « 1) | src2[i]] for i in 0..31。8-bit imm LUT 完整表达任意 3-input boolean function (2^3=8 种输入组合 → 8-bit 输出表)。整 warp per-lane 各自独立计算。

常用 LUT 值: AND3=0x80 (a&b&c), OR3=0xFE (a|b|c), XOR3=0x96 (a⊕b⊕c), NOR=0x01 (~(a|b|c)), and-NOT=0x60 ((a&b)&~c), MUX=0xCA ((a&b)|(~a&c) 即 a?b:c)。一条 lop3 替代多条 and/or/xor/not 序列。

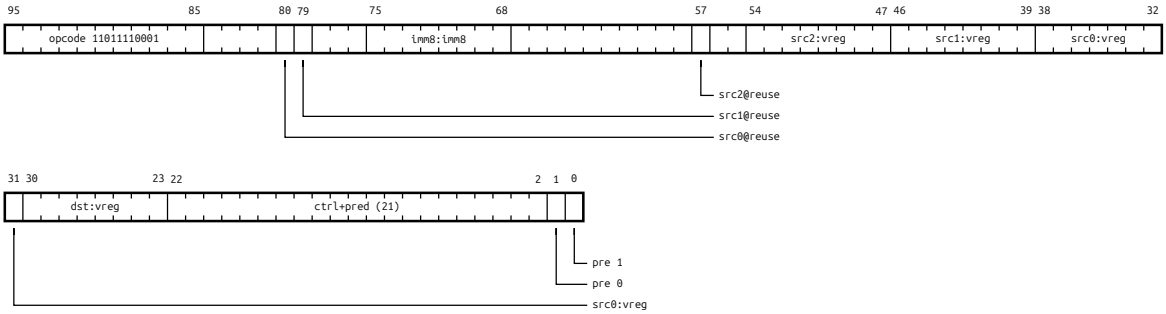
Leaf: lop3__v_vvvi

Format:

```
lop3  dst, src0{.reuse}, src1{.reuse}, src2{.reuse}, imm8
```

Operands: dst: vreg; src0: vreg; src1: vreg; src2: vreg; imm8: imm8u

Encoding Diagram: 96b, prefix 10, opcode 11011110001



Notes:

3-input 位逻辑: dst[i] = LUT[(src0[i] « 2) | (src1[i] « 1) | src2[i]] for i in 0..31。8-bit imm LUT 完整表达任意 3-input boolean function (2^3=8 种输入组合 → 8-bit 输出表)。整 warp per-lane 各自独立计算。

常用 LUT 值: AND3=0x80 (a&b&c), OR3=0xFE (a|b|c), XOR3=0x96 (a⊕b⊕c), NOR=0x01 (~(a|b|c)), and-NOT=0x60 ((a&b)&~c), MUX=0xCA ((a&b)|(~a&c) 即 a?b:c)。一条 lop3 替代多条 and/or/xor/not 序列。

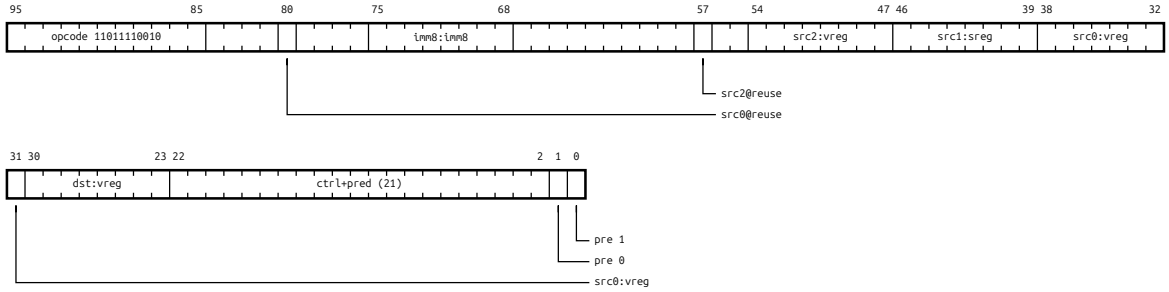
Leaf: lop3__v_vxvi

Format:

```
lop3  dst, src0{.reuse}, src1, src2{.reuse}, imm8
```

Operands: dst: vreg; src0: vreg; src1: sreg; src2: vreg; imm8: imm8u

Encoding Diagram: 96b, prefix 10, opcode 11011110010



Notes:

3-input 位逻辑: dst[i] = LUT[(src0[i] « 2) | (src1[i] « 1) | src2[i]] for i in 0..31。8-bit imm LUT 完整表达任意 3-input boolean function (2^3=8 种输入组合 → 8-bit 输出表)。整 warp per-lane 各自独立计算。

常用 LUT 值: AND3=0x80 (a&b&c), OR3=0xFE (a|b|c), XOR3=0x96 (a⊕b⊕c), NOR=0x01 (~(a|b|c)), and-NOT=0x60 ((a&b)&~c), MUX=0xCA ((a&b)|(~a&c) 即 a?b:c)。一条 lop3 替代多条 and/or/xor/not 序列。

popc **category**: Bitwise and logic **predicate_mode**: simt

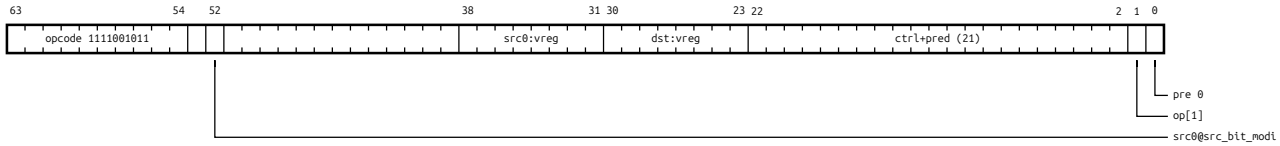
Leaf: popc__v_v

Format:

```
popc  dst, src0{.src_bit_modi}
```

Operands: dst: vreg; src0: vreg

Encoding Diagram: 64b, prefix 0, opcode 11111001011



Notes:

Population count: dst = popcount(src0), 统计 src0 中 1 bit 的个数, dst ∈ [0, 32]。整 warp per-lane 各自计算。

src_bit_modi=id (默认): 直接 popcount(src0)。

src_bit_modi=inv: popcount(~src0) = 32 - popcount(src0), 省一条 not 指令。

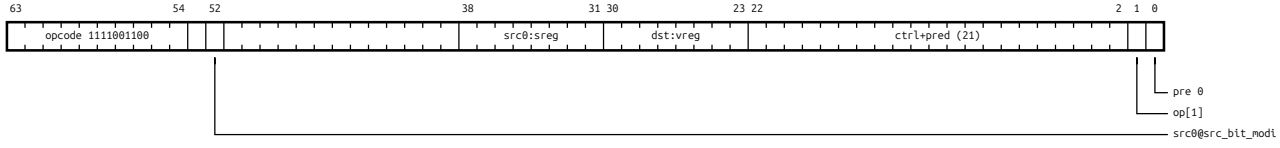
Leaf: popc__v_x

Format:

```
popc  dst, src0{.src_bit_modi}
```

Operands: dst: vreg; src0: sreg

Encoding Diagram: 64b, prefix 0, opcode 11111001100



Notes:

Population count: dst = popcount(src0), 统计 src0 中 1 bit 的个数, dst ∈ [0, 32]。整 warp per-lane 各自计算。

src_bit_modi=id (默认): 直接 popcount(src0)。

src_bit_modi=inv: popcount(~src0) = 32 - popcount(src0), 省一条 not 指令。

prmt **category**: Bitwise and logic **predicate_mode**: simt

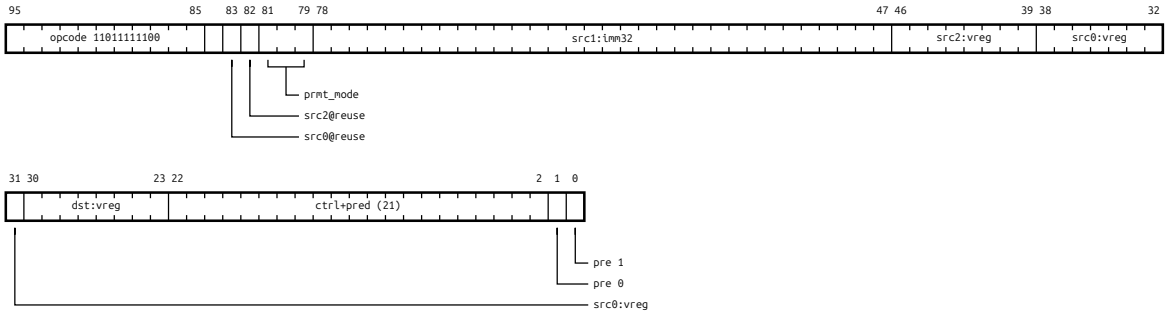
Leaf: prmt__v_viv

Format:

```
prmt{.prmt_mode}  dst, src0{.reuse}, src1, src2{.reuse}
```

Operands: dst: vreg; src0: vreg; src1: imm32u; src2: vreg

Encoding Diagram: 96b, prefix 10, opcode 11011111100



Notes:

Byte permute: 把 src0 (低 4 byte) 和 src2 (高 4 byte) 拼成 8-byte 源 ({src2, src0}, byte 0..7), 按 src1[15:0] control 位选 4 byte 重组写 dst。整 warp per-lane 各自计算。

operand 角色: src0 = byte source 低半; src2 = byte source 高半; src1 = control (仅低 16 位用)。8-byte 源编号: byte 0 = src0[7:0] 到 byte 7 = src2[31:24]。

prmt_mode=idx (默认, 最通用): src1 拆 4 个 4-bit selector, dst.byteN = sel(src1[4N+3:4N])。每个 4-bit selector: bits[2:0] 选 8-byte 源中某 byte (0..7); bit[3]=0 直接 copy byte, bit[3]=1 sign-replicate (复制该 byte 的 msb 到整个 8-bit, 即 byte sign-extend)。仅 idx mode 支持 sign-replicate。

prmt_mode=f4e (forward 4 extract): src1[1:0] 选 base offset, 从 base 起连续 4 byte; f4e.0 → {3,2,1,0} / f4e.1 → {4,3,2,1} / f4e.2 → {5,4,3,2} / f4e.3 → {6,5,4,3}。

prmt_mode=b4e (backward 4 extract): src1[1:0] 选 base; b4e.0 → {5,6,7,0} / b4e.1 → {6,7,0,1} / b4e.2 → {7,0,1,2} / b4e.3 → {0,1,2,3}。

prmt_mode=rc8 (replicate 8): src1[1:0] 选某 byte 复制 4 次到 dst。

prmt_mode=ecl (edge clamp left) / ecr (edge clamp right): 从 src1[1:0] 起向左 / 向右 clamp 边界饱和。

prmt_mode=rc16 (replicate 16): src1[1:0] 选 0 / 2, 复制 src 的 lo half 或 hi half (16-bit) 两次。

典型用例: byte 重排 (big-endian / little-endian 转换); 字节级 swizzle (lo/hi byte 交换); broadcast 单 byte 到整 32-bit; pack 4 个独立 byte 成 packed int8。

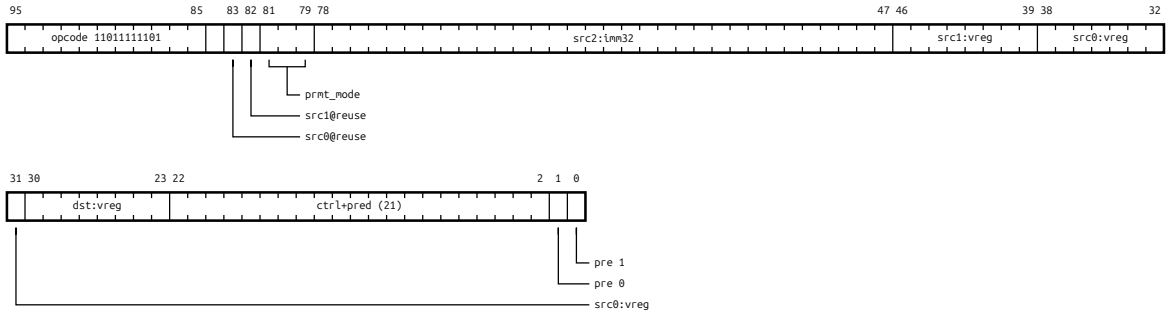
Leaf: prmt__v_vvi

Format:

```
prmt{.prmt_mode}  dst, src0{.reuse}, src1{.reuse}, src2
```

Operands: dst: vreg; src0: vreg; src1: vreg; src2: imm32u

Encoding Diagram: 96b, prefix 10, opcode 11011111101



Notes:

Byte permute: 把 src0 (低 4 byte) 和 src2 (高 4 byte) 拼成 8-byte 源 ({src2, src0}, byte 0..7), 按 src1[15:0] control 位选 4 byte 重组写 dst。整 warp per-lane 各自计算。

operand 角色: src0 = byte source 低半; src2 = byte source 高半; src1 = control (仅低 16 位用)。8-byte 源编号: byte 0 = src0[7:0] 到 byte 7 = src2[31:24]。

prmt_mode=idx (默认, 最通用): src1 拆 4 个 4-bit selector, dst.byteN = sel(src1[4N+3:4N])。每个 4-bit selector: bits[2:0] 选 8-byte 源中某 byte (0..7); bit[3]=0 直接 copy byte, bit[3]=1 sign-replicate (复制该 byte 的 msb 到整个 8-bit, 即 byte sign-extend)。仅 idx mode 支持 sign-replicate。

prmt_mode=f4e (forward 4 extract): src1[1:0] 选 base offset, 从 base 起连续 4 byte; f4e.0 → {3,2,1,0} / f4e.1 → {4,3,2,1} / f4e.2 → {5,4,3,2} / f4e.3 → {6,5,4,3}。

prmt_mode=b4e (backward 4 extract): src1[1:0] 选 base; b4e.0 → {5,6,7,0} / b4e.1 → {6,7,0,1} / b4e.2 → {7,0,1,2} / b4e.3 → {0,1,2,3}。

prmt_mode=rc8 (replicate 8): src1[1:0] 选某 byte 复制 4 次到 dst。

prmt_mode=ecl (edge clamp left) / ecr (edge clamp right): 从 src1[1:0] 起向左 / 向右 clamp 边界饱和。

prmt_mode=rc16 (replicate 16): src1[1:0] 选 0 / 2, 复制 src 的 lo half 或 hi half (16-bit) 两次。

典型用例: byte 重排 (big-endian / little-endian 转换); 字节级 swizzle (lo/hi byte 交换); broadcast 单 byte 到整 32-bit; pack 4 个独立 byte 成 packed int8。

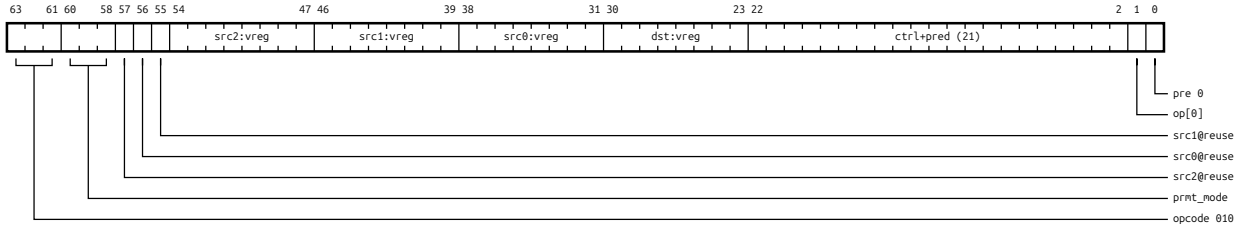
Leaf: prmt__v_vvv

Format:

```
prmt{.prmt_mode}  dst, src0{.reuse}, src1{.reuse}, src2{.reuse}
```

Operands: dst: vreg; src0: vreg; src1: vreg; src2: vreg

Encoding Diagram: 64b, prefix 0, opcode 0010



Notes:

Byte permute: 把 src0 (低 4 byte) 和 src2 (高 4 byte) 拼成 8-byte 源 ({src2, src0}, byte 0..7), 按 src1[15:0] control 位选 4 byte 重组写 dst。整 warp per-lane 各自计算。

operand 角色: src0 = byte source 低半; src2 = byte source 高半; src1 = control (仅低 16 位用)。8-byte 源编号: byte 0 = src0[7:0] 到 byte 7 = src2[31:24]。

prmt_mode=idx (默认, 最通用): src1 拆 4 个 4-bit selector, dst.byteN = sel(src1[4N+3:4N])。每个 4-bit selector:

bits[2:0] 选 8-byte 源中某 byte (0..7); bit[3]=0 直接 copy byte, bit[3]=1 sign-replicate (复制该 byte 的 msb 到整个 8-bit, 即 byte sign-extend)。仅 idx mode 支持 sign-replicate。

prmt_mode=f4e (forward 4 extract): src1[1:0] 选 base offset, 从 base 起连续 4 byte; f4e.0 → {3,2,1,0} / f4e.1 → {4,3,2,1} / f4e.2 → {5,4,3,2} / f4e.3 → {6,5,4,3}。

prmt_mode=b4e (backward 4 extract): src1[1:0] 选 base; b4e.0 → {5,6,7,0} / b4e.1 → {6,7,0,1} / b4e.2 → {7,0,1,2} / b4e.3 → {0,1,2,3}。

prmt_mode=rc8 (replicate 8): src1[1:0] 选某 byte 复制 4 次到 dst。

prmt_mode=ecl (edge clamp left) / ecr (edge clamp right): 从 src1[1:0] 起向左 / 向右 clamp 边界饱和。

prmt_mode=rc16 (replicate 16): src1[1:0] 选 0 / 2, 复制 src 的 lo half 或 hi half (16-bit) 两次。

典型用例: byte 重排 (big-endian / little-endian 转换); 字节级 swizzle (lo/hi byte 交换); broadcast 单 byte 到整 32-bit; pack 4 个独立 byte 成 packed int8。

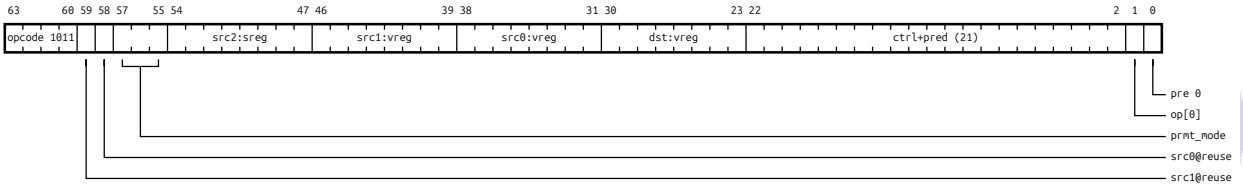
Leaf: prmt__v_vvx

Format:

```
prmt{.prmt_mode}  dst, src0{.reuse}, src1{.reuse}, src2
```

Operands: dst: vreg; src0: vreg; src1: vreg; src2: sreg

Encoding Diagram: 64b, prefix 0, opcode 01011



Notes:

Byte permute: 把 src0 (低 4 byte) 和 src2 (高 4 byte) 拼成 8-byte 源 ({src2, src0}, byte 0..7), 按 src1[15:0] control 位选 4 byte 重组写 dst。整 warp per-lane 各自计算。

operand 角色: src0 = byte source 低半; src2 = byte source 高半; src1 = control (仅低 16 位用)。8-byte 源编号: byte 0 = src0[7:0] 到 byte 7 = src2[31:24]。

prmt_mode=idx (默认, 最通用): src1 拆 4 个 4-bit selector, dst.byteN = sel(src1[4N+3:4N])。每个 4-bit selector: bits[2:0] 选 8-byte 源中某 byte (0..7); bit[3]=0 直接 copy byte, bit[3]=1 sign-replicate (复制该 byte 的 msb 到整个 8-bit, 即 byte sign-extend)。仅 idx mode 支持 sign-replicate。

prmt_mode=f4e (forward 4 extract): src1[1:0] 选 base offset, 从 base 起连续 4 byte; f4e.0 → {3,2,1,0} / f4e.1 → {4,3,2,1} / f4e.2 → {5,4,3,2} / f4e.3 → {6,5,4,3}。

prmt_mode=b4e (backward 4 extract): src1[1:0] 选 base; b4e.0 → {5,6,7,0} / b4e.1 → {6,7,0,1} / b4e.2 → {7,0,1,2} / b4e.3 → {0,1,2,3}。

prmt_mode=rc8 (replicate 8): src1[1:0] 选某 byte 复制 4 次到 dst。

prmt_mode=ecl (edge clamp left) / ecr (edge clamp right): 从 src1[1:0] 起向左 / 向右 clamp 边界饱和。

prmt_mode=rc16 (replicate 16): src1[1:0] 选 0 / 2, 复制 src 的 lo half 或 hi half (16-bit) 两次。

典型用例: byte 重排 (big-endian / little-endian 转换); 字节级 swizzle (lo/hi byte 交换); broadcast 单 byte 到整 32-bit; pack 4 个独立 byte 成 packed int8。

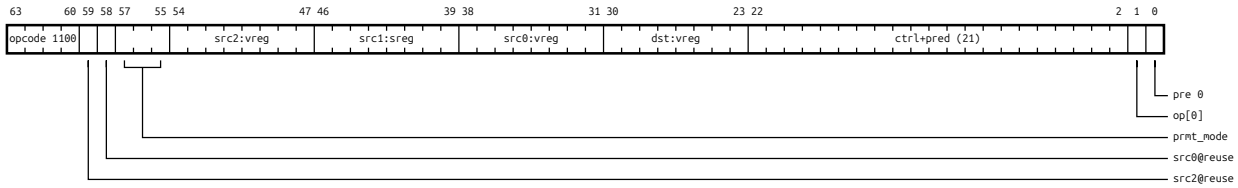
Leaf: prmt__v_vxv

Format:

```
prmt{.prmt_mode}  dst, src0{.reuse}, src1, src2{.reuse}
```

Operands: dst: vreg; src0: vreg; src1: sreg; src2: vreg

Encoding Diagram: 64b, prefix 0, opcode 01100



Notes:

Byte permute: 把 src0 (低 4 byte) 和 src2 (高 4 byte) 拼成 8-byte 源 ({src2, src0}, byte 0..7), 按 src1[15:0] control 位选 4 byte 重组写 dst。整 warp per-lane 各自计算。

operand 角色: src0 = byte source 低半; src2 = byte source 高半; src1 = control (仅低 16 位用)。8-byte 源编号: byte 0 = src0[7:0] 到 byte 7 = src2[31:24]。

prmt_mode=idx (默认, 最通用): src1 拆 4 个 4-bit selector, dst.byteN = sel(src1[4N+3:4N])。每个 4-bit selector: bits[2:0] 选 8-byte 源中某 byte (0..7); bit[3]=0 直接 copy byte, bit[3]=1 sign-replicate (复制该 byte 的 msb 到整个 8-bit, 即 byte sign-extend)。仅 idx mode 支持 sign-replicate。

prmt_mode=f4e (forward 4 extract): src1[1:0] 选 base offset, 从 base 起连续 4 byte; f4e.0 → {3,2,1,0} / f4e.1 → {4,3,2,1} / f4e.2 → {5,4,3,2} / f4e.3 → {6,5,4,3}。

prmt_mode=b4e (backward 4 extract): src1[1:0] 选 base; b4e.0 → {5,6,7,0} / b4e.1 → {6,7,0,1} / b4e.2 → {7,0,1,2} / b4e.3 → {0,1,2,3}。

prmt_mode=rc8 (replicate 8): src1[1:0] 选某 byte 复制 4 次到 dst。

prmt_mode=ecl (edge clamp left) / ecr (edge clamp right): 从 src1[1:0] 起向左 / 向右 clamp 边界饱和。

prmt_mode=rc16 (replicate 16): src1[1:0] 选 0 / 2, 复制 src 的 lo half 或 hi half (16-bit) 两次。

典型用例: byte 重排 (big-endian / little-endian 转换); 字节级 swizzle (lo/hi byte 交换); broadcast 单 byte 到整 32-bit; pack 4 个独立 byte 成 packed int8。

shf **category**: Bitwise and logic **predicate_mode**: simt

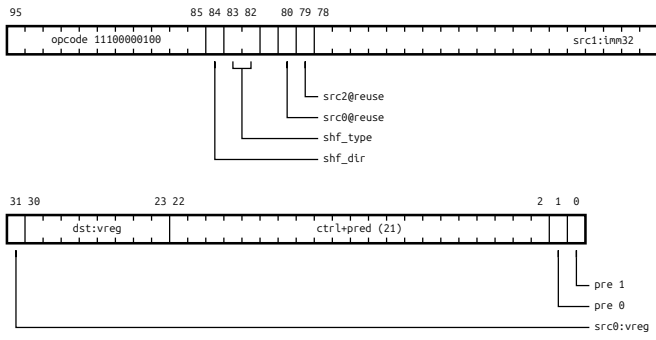
Leaf: shf__v_viv

Format:

```
shf.shf_dir.shf_type  dst, src0{.reuse}, src1, src2{.reuse}
```

Operands: dst: vreg; src0: vreg; src1: imm32u; src2: vreg

Encoding Diagram: 96b, prefix 10, opcode 11100000100



Notes:

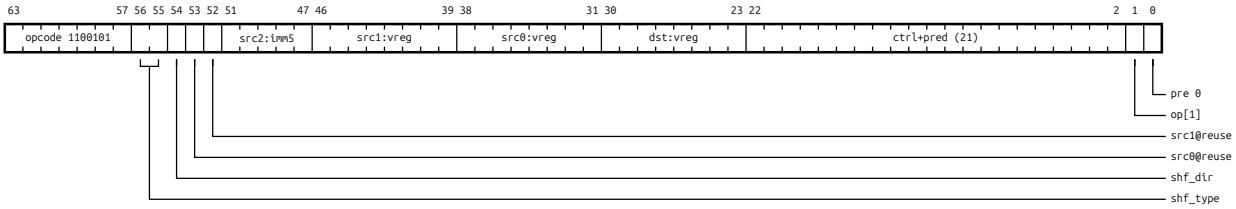
Funnel shift: 把 src1 (高 32-bit) : src0 (低 32-bit) 拼成 64-bit pair, 按 src2 移位, 取一个 32-bit window 写 dst。常用于 multi-precision shift 和 unaligned bit-field extract。整 warp per-lane 各自计算。

operand 语义: src0 = 低 32-bit, src1 = 高 32-bit, src2 = 移位数 0..31。
shf_dir=l (左移): 把 64-bit pair 左移 shamt 位; shf_dir=r (右移): 右移。
shf_type=lo: 取移位结果的低 32-bit。
shf_type=hi: 取高 32-bit。
shf_type=u64: 64-bit unsigned 模式 (右移补 0)。
shf_type=s64: 64-bit signed 模式 (右移按 hi MSB 符号扩展, arithmetic right shift)。

Leaf: shf__v_vvi
Format:

```
shf.shf_dir.shf_type  dst, src0{.reuse}, src1{.reuse}, src2
```

Operands: dst: vreg; src0: vreg; src1: vreg; src2: imm5u
Encoding Diagram: 64b, prefix 0, opcode 11100101

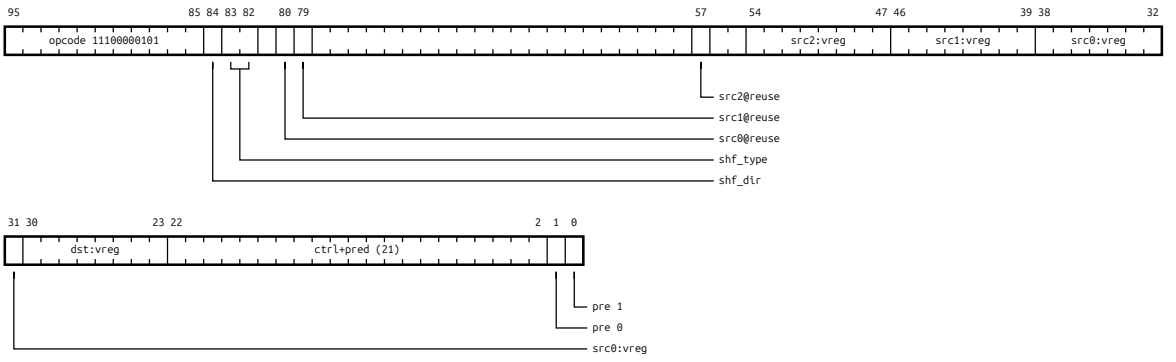


Notes:
Funnel shift: 把 src1 (高 32-bit) : src0 (低 32-bit) 拼成 64-bit pair, 按 src2 移位, 取一个 32-bit window 写 dst。常用于 multi-precision shift 和 unaligned bit-field extract。整 warp per-lane 各自计算。
operand 语义: src0 = 低 32-bit, src1 = 高 32-bit, src2 = 移位数 0..31。
shf_dir=l (左移): 把 64-bit pair 左移 shamt 位; shf_dir=r (右移): 右移。
shf_type=lo: 取移位结果的低 32-bit。
shf_type=hi: 取高 32-bit。
shf_type=u64: 64-bit unsigned 模式 (右移补 0)。
shf_type=s64: 64-bit signed 模式 (右移按 hi MSB 符号扩展, arithmetic right shift)。

Leaf: shf__v_vvv
Format:

```
shf.shf_dir.shf_type  dst, src0{.reuse}, src1{.reuse}, src2{.reuse}
```

Operands: dst: vreg; src0: vreg; src1: vreg; src2: vreg
Encoding Diagram: 96b, prefix 10, opcode 11100000101



Notes:

Funnel shift: 把 src1 (高 32-bit) : src0 (低 32-bit) 拼成 64-bit pair, 按 src2 移位, 取一个 32-bit window 写 dst。常用于 multi-precision shift 和 unaligned bit-field extract。整 warp per-lane 各自计算。

operand 语义: src0 = 低 32-bit, src1 = 高 32-bit, src2 = 移位数 0..31。

shf_dir=l (左移): 把 64-bit pair 左移 shamt 位; shf_dir=r (右移): 右移。

shf_type=lo: 取移位结果的低 32-bit。

shf_type=hi: 取高 32-bit。

shf_type=u64: 64-bit unsigned 模式 (右移补 0)。

shf_type=s64: 64-bit signed 模式 (右移按 hi MSB 符号扩展, arithmetic right shift)。

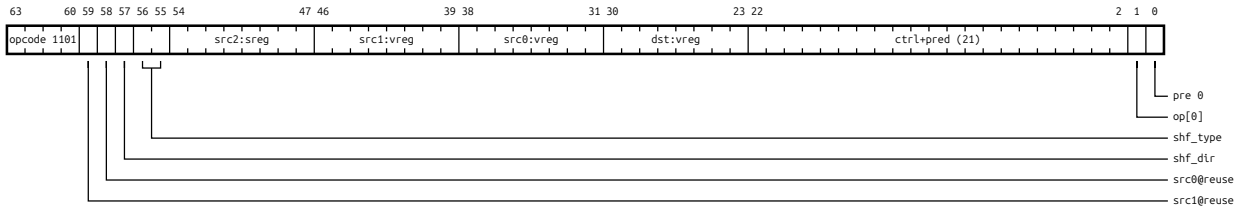
Leaf: shf__v_vvx

Format:

```
shf.shf_dir.shf_type  dst, src0{.reuse}, src1{.reuse}, src2
```

Operands: dst: vreg; src0: vreg; src1: vreg; src2: sreg

Encoding Diagram: 64b, prefix 0, opcode 01101



Notes:

Funnel shift: 把 src1 (高 32-bit) : src0 (低 32-bit) 拼成 64-bit pair, 按 src2 移位, 取一个 32-bit window 写 dst。常用于 multi-precision shift 和 unaligned bit-field extract。整 warp per-lane 各自计算。

operand 语义: src0 = 低 32-bit, src1 = 高 32-bit, src2 = 移位数 0..31。

shf_dir=l (左移): 把 64-bit pair 左移 shamt 位; shf_dir=r (右移): 右移。

shf_type=lo: 取移位结果的低 32-bit。

shf_type=hi: 取高 32-bit。

shf_type=u64: 64-bit unsigned 模式 (右移补 0)。

shf_type=s64: 64-bit signed 模式 (右移按 hi MSB 符号扩展, arithmetic right shift)。

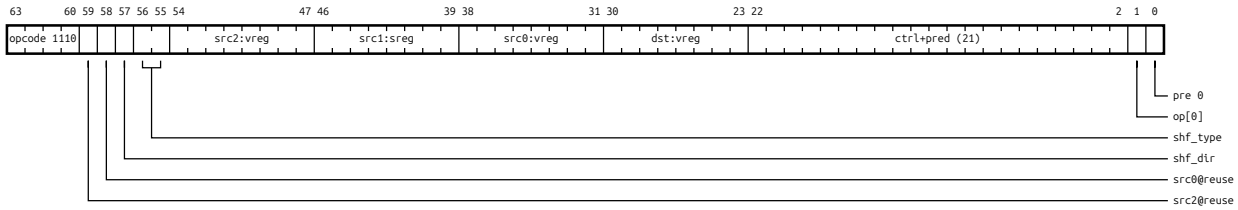
Leaf: shf__v_vxv

Format:

```
shf.shf_dir.shf_type  dst, src0{.reuse}, src1, src2{.reuse}
```

Operands: dst: vreg; src0: vreg; src1: sreg; src2: vreg

Encoding Diagram: 64b, prefix 0, opcode 01110



Notes:

Funnel shift: 把 src1 (高 32-bit) : src0 (低 32-bit) 拼成 64-bit pair, 按 src2 移位, 取一个 32-bit window 写 dst。常用于 multi-precision shift 和 unaligned bit-field extract。整 warp per-lane 各自计算。

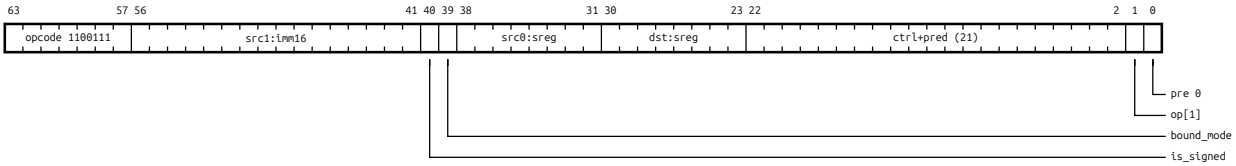
operand 语义: src0 = 低 32-bit, src1 = 高 32-bit, src2 = 移位数 0..31。
shf_dir=l (左移): 把 64-bit pair 左移 shamt 位; shf_dir=r (右移): 右移。
shf_type=lo: 取移位结果的低 32-bit。
shf_type=hi: 取高 32-bit。
shf_type=u64: 64-bit unsigned 模式 (右移补 0)。
shf_type=s64: 64-bit signed 模式 (右移按 hi MSB 符号扩展, arithmetic right shift)。

ubfe **category**: Bitwise and logic **predicate_mode**: uniform

Leaf: ubfe__x_xi
Format:

```
ubfe.is_signed{.bound_mode}  dst, src0, src1
```

Operands: dst: sreg; src0: sreg; src1: imm16u
Encoding Diagram: 64b, prefix 0, opcode 11100111

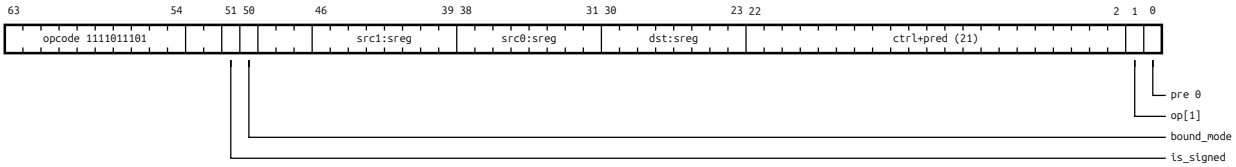


Notes:
Bit field extract uniform 变体: 从 src0 抽取 [pos+len-1 : pos] 位段写 dst。所有 operand 都是 sreg / imm, 整 warp 共享。
src1 编码 pos 和 len: 跟 bfe 一致 (src1[7:0]=pos, src1[15:8]=len)。
is_signed=signed / unsigned: 跟 bfe 一致。bound_mode=clamp / wrap: 跟 bfe 一致。

Leaf: ubfe__x_xx
Format:

```
ubfe.is_signed{.bound_mode}  dst, src0, src1
```

Operands: dst: sreg; src0: sreg; src1: sreg
Encoding Diagram: 64b, prefix 0, opcode 11111011101



Notes:
Bit field extract uniform 变体: 从 src0 抽取 [pos+len-1 : pos] 位段写 dst。所有 operand 都是 sreg / imm, 整 warp 共享。
src1 编码 pos 和 len: 跟 bfe 一致 (src1[7:0]=pos, src1[15:8]=len)。
is_signed=signed / unsigned: 跟 bfe 一致。bound_mode=clamp / wrap: 跟 bfe 一致。

ubfi **category**: Bitwise and logic **predicate_mode**: uniform

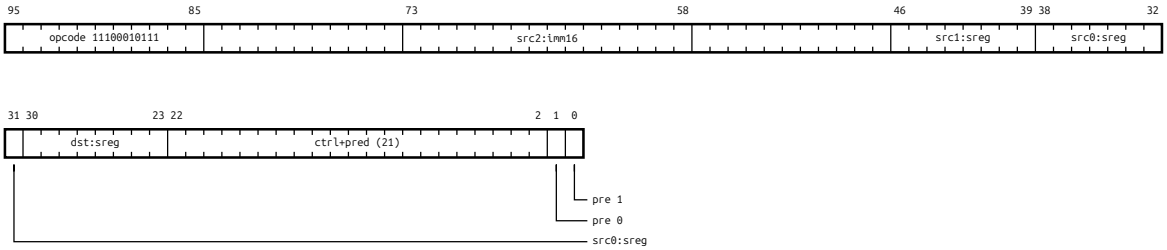
Leaf: ubfi__x_xxi

Format:

```
ubfi  dst, src0, src1, src2
```

Operands: dst: sreg; src0: sreg; src1: sreg; src2: imm16u

Encoding Diagram: 96b, prefix 10, opcode 11100010111



Notes:

Bit field insert uniform 变体: 把 src0 的低 len 位插入 src1 的 [pos+len-1 : pos] 区间写 dst。所有 operand 都是 sreg / imm, 整 warp 共享。

src2 编码 pos 和 len: 跟 bfi 一致 (src2[7:0]=pos, src2[15:8]=len)。边界条件跟 bfi 一致。

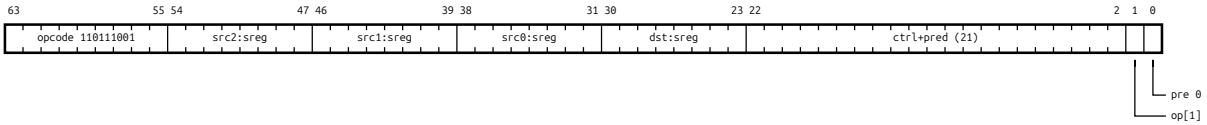
Leaf: ubfi__x_xxx

Format:

```
ubfi  dst, src0, src1, src2
```

Operands: dst: sreg; src0: sreg; src1: sreg; src2: sreg

Encoding Diagram: 64b, prefix 0, opcode 1110111001



Notes:

Bit field insert uniform 变体: 把 src0 的低 len 位插入 src1 的 [pos+len-1 : pos] 区间写 dst。所有 operand 都是 sreg / imm, 整 warp 共享。

src2 编码 pos 和 len: 跟 bfi 一致 (src2[7:0]=pos, src2[15:8]=len)。边界条件跟 bfi 一致。

ubmsk **category**: Bitwise and logic **predicate_mode**: uniform

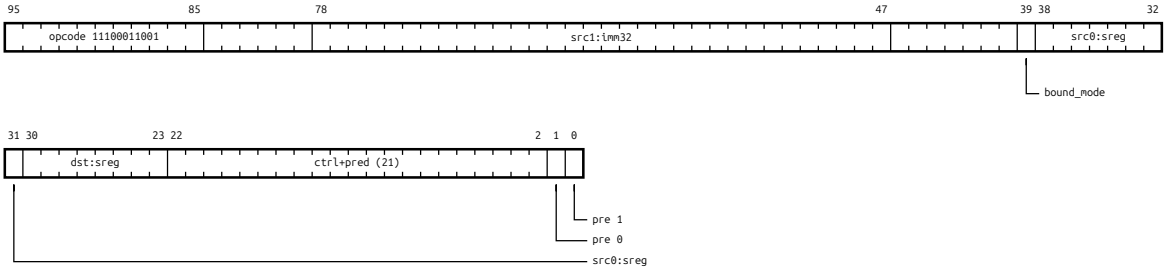
Leaf: ubmsk__x_xi

Format:

```
ubmsk{.bound_mode}  dst, src0, src1
```

Operands: dst: sreg; src0: sreg; src1: imm32u

Encoding Diagram: 96b, prefix 10, opcode 11100011001



Notes:

Bit mask 生成 uniform 变体: dst[pos+len-1 : pos] 全 1, 其它位 0。所有 operand 都是 sreg / imm, 整 warp 共享一份结果。

bound_mode=clamp / wrap: 跟 bmsk 一致 (clamp 越界饱和, wrap 取低 5-bit mod-32)。

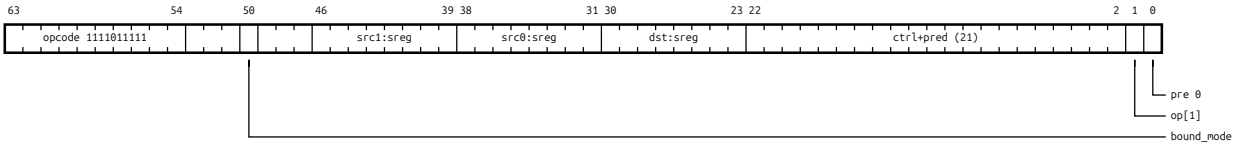
Leaf: ubmsk__x_xx

Format:

```
ubmsk{.bound_mode}  dst, src0, src1
```

Operands: dst: sreg; src0: sreg; src1: sreg

Encoding Diagram: 64b, prefix 0, opcode 11111011111



Notes:

Bit mask 生成 uniform 变体: dst[pos+len-1 : pos] 全 1, 其它位 0。所有 operand 都是 sreg / imm, 整 warp 共享一份结果。

bound_mode=clamp / wrap: 跟 bmsk 一致 (clamp 越界饱和, wrap 取低 5-bit mod-32)。

ubrev　**category**: Bitwise and logic　**predicate_mode**: uniform

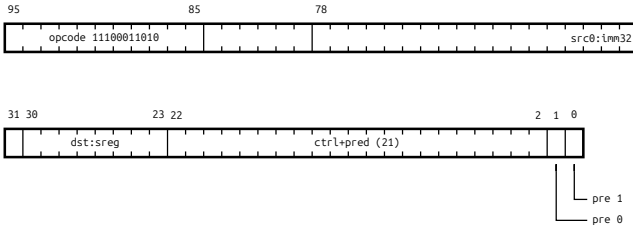
Leaf: ubrev__x_i

Format:

```
ubrev  dst, src0
```

Operands: dst: sreg; src0: imm32u

Encoding Diagram: 96b, prefix 10, opcode 11100011010



Notes:

32-bit 位反转 uniform 变体: dst[i] = src0[31-i] for i in 0..31。所有 operand 都是 sreg / imm, 整 warp 共享一份结果。

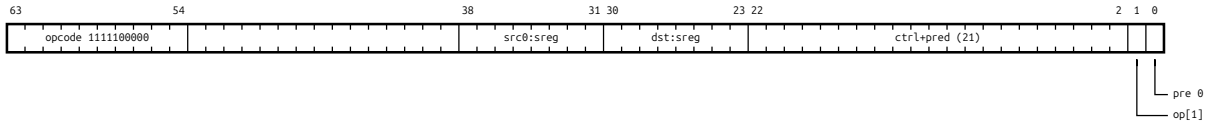
Leaf: ubrev__x_x

Format:

```
ubrev  dst, src0
```

Operands: dst: sreg; src0: sreg

Encoding Diagram: 64b, prefix 0, opcode 11111100000



Notes:

32-bit 位反转 uniform 变体: dst[i] = src0[31-i] for i in 0..31。所有 operand 都是 sreg / imm, 整 warp 共享一份结果。

uflo **category**: Bitwise and logic **predicate_mode**: uniform

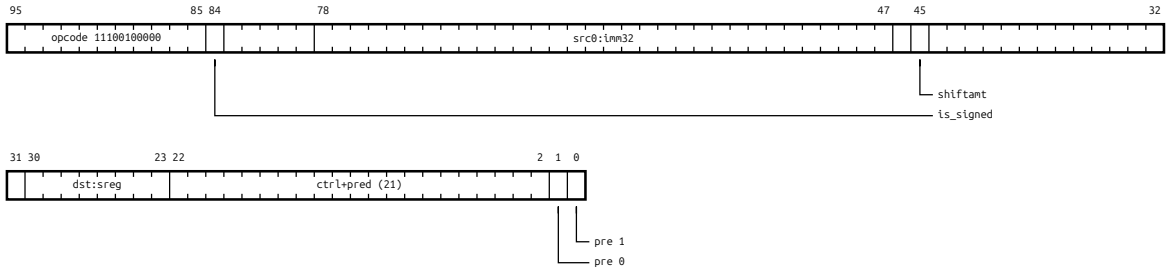
Leaf: uflo__x_i

Format:

```
uflo.is_signed{.shiftamt}  dst, src0
```

Operands: dst: sreg; src0: imm32u

Encoding Diagram: 96b, prefix 10, opcode 11100100000



Notes:

Find Leading One uniform 变体: 从 MSB 扫描找第一个 set bit 位置, src0=0 时 dst = -1。所有 operand 都是 sreg / imm, 整 warp 共享。

is_signed=unsigned / signed: 跟 flo 一致 (signed 负数找第一个 0)。

shiftamt=disable / SH: 跟 flo 一致 (SH 返回 shift count)。

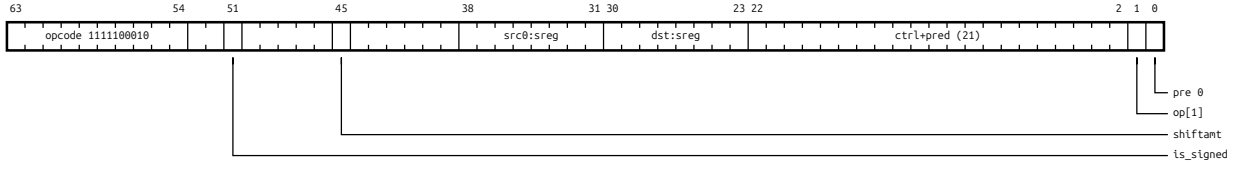
Leaf: uflo__x_x

Format:

```
uflo.is_signed{.shiftamt}  dst, src0
```

Operands: dst: sreg; src0: sreg

Encoding Diagram: 64b, prefix 0, opcode 11111100010



Notes:

Find Leading One uniform 变体: 从 MSB 扫描找第一个 set bit 位置, src0=0 时 dst = -1。所有 operand 都是 sreg / imm, 整 warp 共享。

is_signed=unsigned / signed: 跟 flo 一致 (signed 负数找第一个 0)。

shiftamt=disable / SH: 跟 flo 一致 (SH 返回 shift count)。

ulop3 **category**: Bitwise and logic **predicate_mode**: uniform

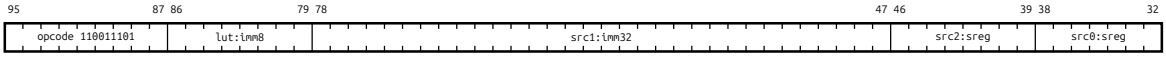
Leaf: ulop3__x_xixi

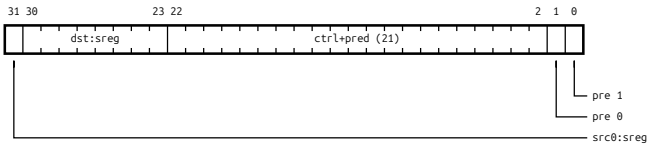
Format:

```
ulop3  dst, src0, src1, src2, lut
```

Operands: dst: sreg; src0: sreg; src1: imm32u; src2: sreg; lut: imm8u

Encoding Diagram: 96b, prefix 10, opcode 110011101





Notes:

3-input 位逻辑 uniform 变体: dst[i] = LUT[(src0[i] « 2) | (src1[i] « 1) | src2[i]], 所有 operand 都是 sreg / imm + 8-bit LUT imm, 整 warp 共享。

常用 LUT 值: 跟 lop3 一致 (AND3=0x80 / OR3=0xFE / XOR3=0x96 / MUX=0xCA 等)。

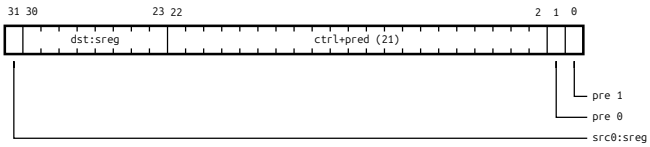
Leaf: ulop3__x_xxxi

Format:

```
ulop3  dst, src0, src1, src2, lut
```

Operands: dst: sreg; src0: sreg; src1: sreg; src2: sreg; lut: imm8u

Encoding Diagram: 96b, prefix 10, opcode 11100110000



Notes:

3-input 位逻辑 uniform 变体: dst[i] = LUT[(src0[i] « 2) | (src1[i] « 1) | src2[i]], 所有 operand 都是 sreg / imm + 8-bit LUT imm, 整 warp 共享。

常用 LUT 值: 跟 lop3 一致 (AND3=0x80 / OR3=0xFE / XOR3=0x96 / MUX=0xCA 等)。

upopc **category**: Bitwise and logic **predicate_mode**: uniform

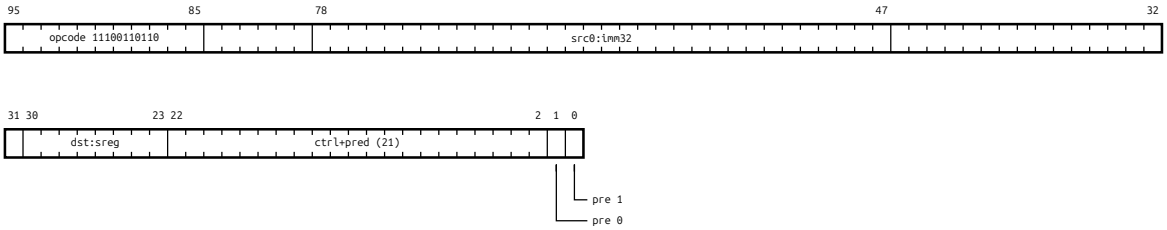
Leaf: upopc__x_i

Format:

```
upopc  dst, src0
```

Operands: dst: sreg; src0: imm32u

Encoding Diagram: 96b, prefix 10, opcode 11100110110



Notes:

Population count uniform 变体: dst = popcount(src0)。所有 operand 都是 sreg / imm, 整 warp 共享。

src_bit_modi=id / inv: 跟 popc 一致 (inv 表示先取反再 popcount)。

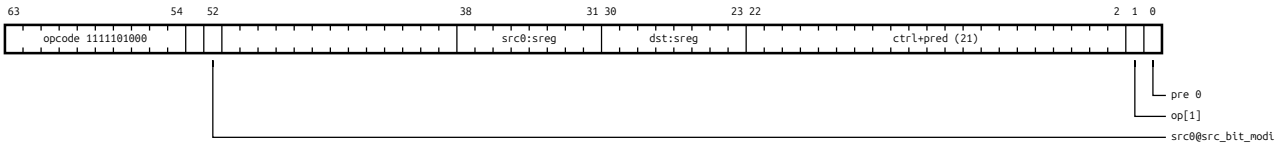
Leaf: upopc__x_x

Format:

```
upopc  dst, src0{.src_bit_modi}
```

Operands: dst: sreg; src0: sreg

Encoding Diagram: 64b, prefix 0, opcode 11111101000



Notes:

Population count uniform 变体: dst = popcount(src0)。所有 operand 都是 sreg / imm, 整 warp 共享。

src_bit_modi=id / inv: 跟 popc 一致 (inv 表示先取反再 popcount)。

uprmt **category**: Bitwise and logic **predicate_mode**: uniform

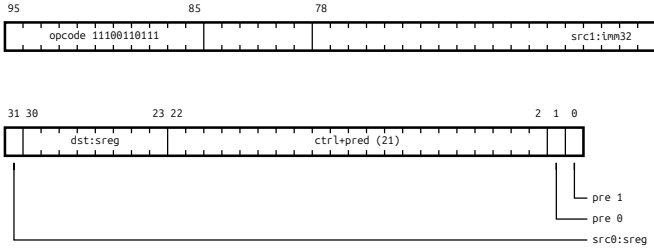
Leaf: uprmt__x_xix

Format:

```
uprmt  dst, src0, src1, src2
```

Operands: dst: sreg; src0: sreg; src1: imm32u; src2: sreg

Encoding Diagram: 96b, prefix 10, opcode 11100110111



Notes:

Byte permute uniform 变体, 固定 idx mode: 把 src0 (低 4 byte) 和 src2 (高 4 byte) 拼 8-byte 源, 按 src1[15:0] 拆 4 个 4-bit selector 选 byte 写 dst。所有 operand 都是 sreg / imm, 整 warp 共享。

idx selector 规则: bits[2:0] 选 byte 0..7 索引; bit[3]=0 直接 copy byte, bit[3]=1 sign-replicate (msb 复制成 byte)。跟 simt prmt 的 idx mode 完全一致。

uprmt 仅暴露 idx mode (无 prmt_mode modifier); f4e / b4e / rc8 / ecl / ecr / rc16 等 6 个其它 mode 仅 simt prmt 支持, uniform 路径需要时走 simt prmt + r2ur 序列代偿。

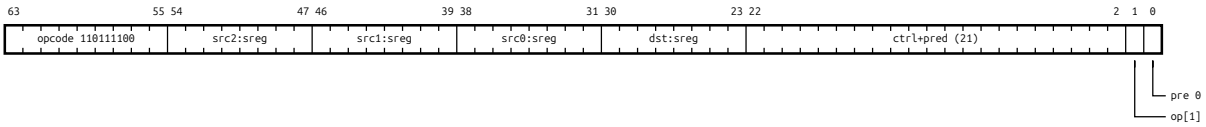
Leaf: uprmt__x_xxx

Format:

```
uprmt  dst, src0, src1, src2
```

Operands: dst: sreg; src0: sreg; src1: sreg; src2: sreg

Encoding Diagram: 64b, prefix 0, opcode 1110111100



Notes:

Byte permute uniform 变体, 固定 idx mode: 把 src0 (低 4 byte) 和 src2 (高 4 byte) 拼 8-byte 源, 按 src1[15:0] 拆 4 个 4-bit selector 选 byte 写 dst。所有 operand 都是 sreg / imm, 整 warp 共享。

idx selector 规则: bits[2:0] 选 byte 0..7 索引; bit[3]=0 直接 copy byte, bit[3]=1 sign-replicate (msb 复制成 byte)。跟 simt prmt 的 idx mode 完全一致。

uprmt 仅暴露 idx mode (无 prmt_mode modifier); f4e / b4e / rc8 / ecl / ecr / rc16 等 6 个其它 mode 仅 simt prmt 支持, uniform 路径需要时走 simt prmt + r2ur 序列代偿。

ushf **category**: Bitwise and logic **predicate_mode**: uniform

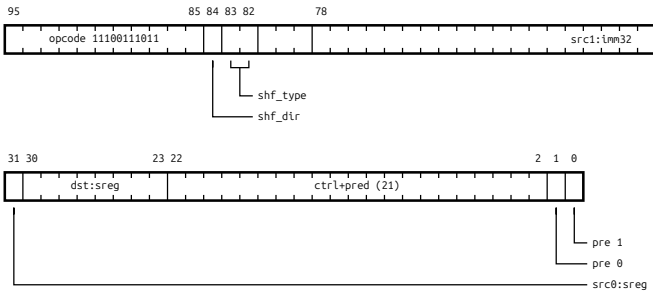
Leaf: ushf__x_xix

Format:

```
ushf.shf_dir.shf_type  dst, src0, src1, src2
```

Operands: dst: sreg; src0: sreg; src1: imm32u; src2: sreg

Encoding Diagram: 96b, prefix 10, opcode 11100111011



Notes:

Funnel shift uniform 变体: 把 src1 (hi) : src0 (lo) 拼成 64-bit pair 移位后取 32-bit window 写 dst。所有 operand 都是 sreg / imm, 整 warp 共享。

shf_dir=l / r: 左移 / 右移。

shf_type=lo / hi / u64 / s64: 跟 shf 一致 (lo/hi 取窗口位置; u64/s64 控制 64-bit 模式与符号扩展)。

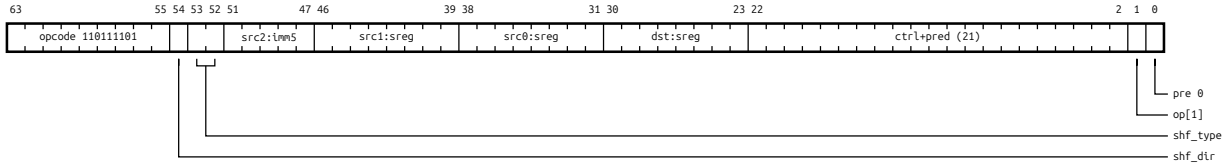
Leaf: ushf__x_xxi

Format:

```
ushf.shf_dir.shf_type  dst, src0, src1, src2
```

Operands: dst: sreg; src0: sreg; src1: sreg; src2: imm5u

Encoding Diagram: 64b, prefix 0, opcode 1110111101



Notes:

Funnel shift uniform 变体: 把 src1 (hi) : src0 (lo) 拼成 64-bit pair 移位后取 32-bit window 写 dst。所有 operand 都是 sreg / imm, 整 warp 共享。

shf_dir=l / r: 左移 / 右移。

shf_type=lo / hi / u64 / s64: 跟 shf 一致 (lo/hi 取窗口位置; u64/s64 控制 64-bit 模式与符号扩展)。

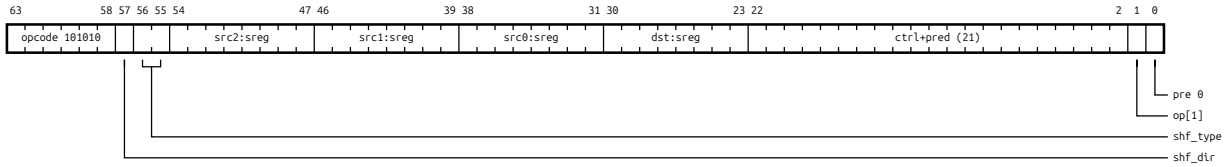
Leaf: ushf__x_xxx

Format:

```
ushf.shf_dir.shf_type  dst, src0, src1, src2
```

Operands: dst: sreg; src0: sreg; src1: sreg; src2: sreg

Encoding Diagram: 64b, prefix 0, opcode 1101010



Notes:

Funnel shift uniform 变体: 把 src1 (hi) : src0 (lo) 拼成 64-bit pair 移位后取 32-bit window 写 dst。所有 operand 都是 sreg / imm, 整 warp 共享。

shf_dir=l / r: 左移 / 右移。

shf_type=lo / hi / u64 / s64: 跟 shf 一致 (lo/hi 取窗口位置; u64/s64 控制 64-bit 模式与符号扩展)。

14.7.6 Conversion

f2f **category**: Conversion **predicate_mode**: simt

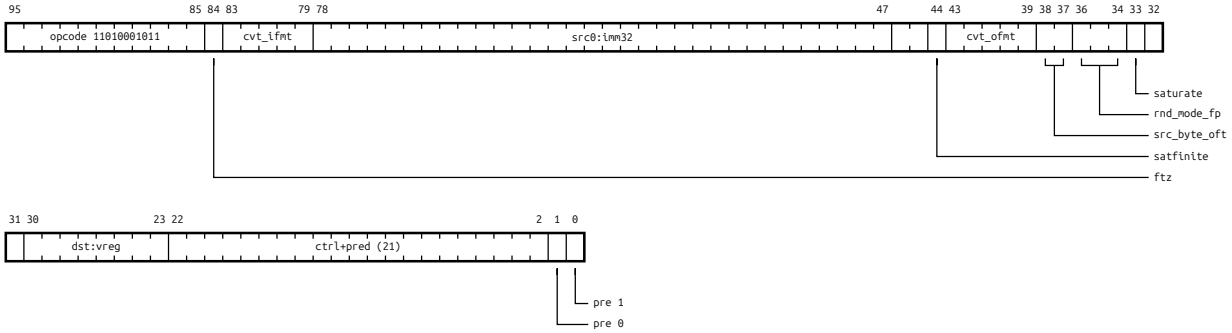
Leaf: f2f__v_i

Format:

```
f2f.cvt_ifmt.cvt_ofmt.rnd_mode_fp.ftz{.saturate}{.satfinite}{.src_byte_oft}  dst, src0
```

Operands: dst: vreg; src0: imm32u

Encoding Diagram: 96b, prefix 10, opcode 11010001011



Notes:

浮点 → 浮点类型转换: 把 src0 按 cvt_ifmt 解读的浮点数据转成 cvt_ofmt 类型, 按 rnd_mode_fp 舍入, 结果写 dst。整 warp per-lane 各自计算。

src 子词路径: src0 < 32-bit 时用 src_byte_oft 选 byte 起始位置 (单 byte / 双 byte 起始); src0 = 32-bit (f32 / tf32) 忽略 src_byte_oft。

cvt_ifmt / cvt_ofmt 两端都是 fp 类型: f16 / bf16 / f32 / tf32 / e5m2 / e4m3 / e3m4 (fp8) / e3m2 / e2m3 (fp6) / e2m1 / e0m3 (fp4) / e8 (UE8M0)。

rnd_mode_fp: 4 valid (rne 默认 / rtz / rup / rdn); f32→tf32 / f32→bf16 等 down-convert 高频用。f2f 不支持 rs (stochastic), 那是 f2fp / uf2fp 的 v_vvs 指令形态专用。

ftz=ftz: fp32 input / output denorm flush 到 0 (sign-preserving)。

saturate=saturate: fp dst 溢出 clamp 到 [-max_finite, +max_finite], NaN 保持 NaN (不变 +0)。

satfinite=satfinite: fp dst 溢出不变 ±Inf 而 clamp 到 ±max_finite, NaN 保持。saturate / satfinite 区别在 ±Inf 处理。

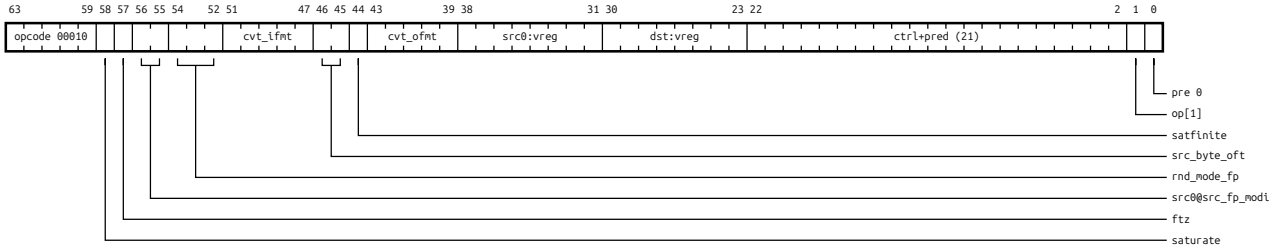
Leaf: f2f__v_v

Format:

```
f2f.cvt_ifmt.cvt_ofmt.rnd_mode_fp.ftz{.saturate}{.satfinite}{.src_byte_oft}  dst,
↪  src0{.src_fp_modi}
```

Operands: dst: vreg; src0: vreg

Encoding Diagram: 64b, prefix 0, opcode 100010



Notes:

浮点 → 浮点类型转换: 把 src0 按 cvt_ifmt 解读的浮点数据转成 cvt_ofmt 类型, 按 rnd_mode_fp 舍入, 结果写 dst。整 warp per-lane 各自计算。

src 子词路径: src0 < 32-bit 时用 src_byte_oft 选 byte 起始位置 (单 byte / 双 byte 起始); src0 = 32-bit (f32 / tf32) 忽略 src_byte_oft。

cvt_ifmt / cvt_ofmt 两端都是 fp 类型: f16 / bf16 / f32 / tf32 / e5m2 / e4m3 / e3m4 (fp8) / e3m2 / e2m3 (fp6) / e2m1 / e0m3 (fp4) / e8 (UE8M0)。

rnd_mode_fp: 4 valid (rne 默认 / rtz / rup / rdn); f32→tf32 / f32→bf16 等 down-convert 高频用。f2f 不支持 rs (stochastic), 那是 f2fp / uf2fp 的 v_vvs 指令形态专用。

ftz=ftz: fp32 input / output denorm flush 到 0 (sign-preserving)。

saturate=saturate: fp dst 溢出 clamp 到 [-max_finite, +max_finite], NaN 保持 NaN (不变 +0)。

satfinite=satfinite: fp dst 溢出不变 ±Inf 而 clamp 到 ±max_finite, NaN 保持。saturate / satfinite 区别在 ±Inf 处理。

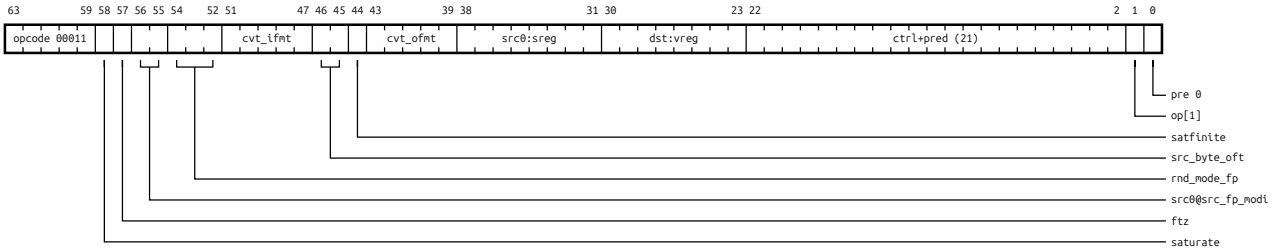
Leaf: f2f__v_x

Format:

```
f2f.cvt_ifmt.cvt_ofmt.rnd_mode_fp.ftz{.saturate}{.satfinite}{.src_byte_oft}  dst,
↪  src0{.src_fp_modi}
```

Operands: dst: vreg; src0: sreg

Encoding Diagram: 64b, prefix 0, opcode 100011



Notes:

浮点 → 浮点类型转换: 把 src0 按 cvt_ifmt 解读的浮点数据转成 cvt_ofmt 类型, 按 rnd_mode_fp 舍入, 结果写 dst。整 warp per-lane 各自计算。

src 子词路径: src0 < 32-bit 时用 src_byte_oft 选 byte 起始位置 (单 byte / 双 byte 起始); src0 = 32-bit (f32 / tf32) 忽略 src_byte_oft。

cvt_ifmt / cvt_ofmt 两端都是 fp 类型: f16 / bf16 / f32 / tf32 / e5m2 / e4m3 / e3m4 (fp8) / e3m2 / e2m3 (fp6) / e2m1 / e0m3 (fp4) / e8 (UE8M0)。

rnd_mode_fp: 4 valid (rne 默认 / rtz / rup / rdn); f32→tf32 / f32→bf16 等 down-convert 高频用。f2f 不支持 rs (stochastic), 那是 f2fp / uf2fp 的 v_vvs 指令形态专用。

ftz=ftz: fp32 input / output denorm flush 到 0 (sign-preserving)。

saturate=saturate: fp dst 溢出 clamp 到 [-max_finite, +max_finite], NaN 保持 NaN (不变 +0)。

satfinite=satfinite: fp dst 溢出不变 ±Inf 而 clamp 到 ±max_finite, NaN 保持。saturate / satfinite 区别在 ±Inf 处理。

f2fp　**category**: Conversion　**predicate_mode**: simt

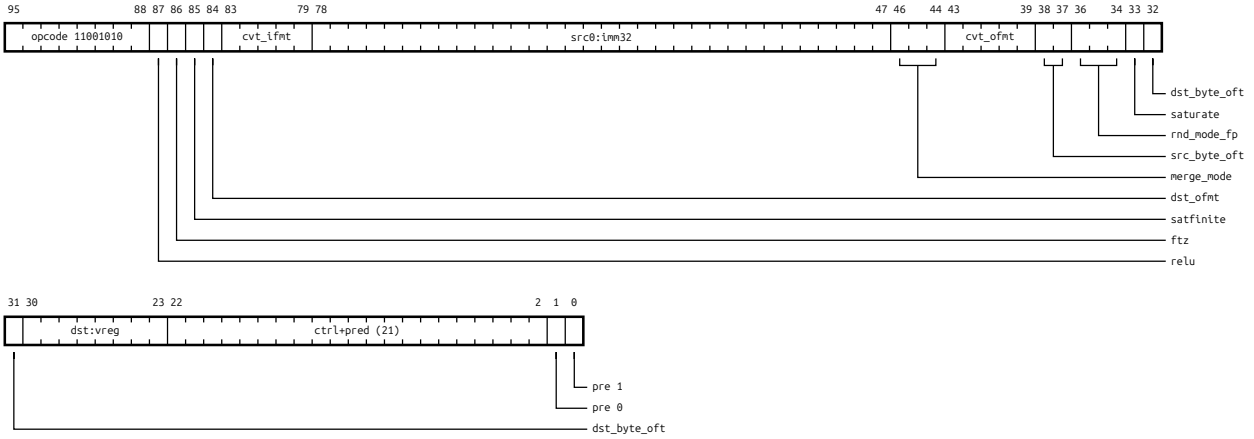
Leaf: f2fp__v_i

Format:

```
f2fp.cvt_ifmt.cvt_ofmt.rnd_mode_fp.ftz{.saturate}{.satfinite}{.relu}{.dst_ofmt}{.src_byt」
↪  e_oft}{.dst_byte_oft}{.merge_mode}  dst,
↪  src0
```

Operands: dst: vreg; src0: imm32u

Encoding Diagram: 96b, prefix 10, opcode 11001010



Notes:

浮点 → 浮点 packed 转换 (多 lane 写入): 覆盖单 src widen / 双 src pack / 子词 unpack / merge-with-old-dst / stochastic round + block-scale 等所有 fp 转换路径。整 warp per-lane 各自计算。

指令形态 / merge_mode 组合语义 (精确 dst 写法):

- v_v + normal: 单 src pad-zero 覆盖整 dst (dst.lo = convert(src0), dst.hi = 0)。
- v_v + unpack: 单 src 子词 unpack (dst[byte=dst_byte_oft, len=ofmt_size] = convert(src0[byte=src_byte_oft, len=ifmt_size]), dst 其它 byte 默认 0)。
- v_v + pack_hi: 单 src 写 dst 高 16-bit (dst.hi = convert(src0), dst.lo 保留旧值)。
- v_vv + normal (= PACK_AB): 双 src pack 覆盖整 dst (dst.lo = convert(src0), dst.hi = convert(src1)) — FP32+FP32 → packed fp16/bf16 主路径。
- v_vvc + merge: 单 src 转换 + 跟 src2 (旧 dst) merge 部分写。
- v_vvc + pack_merge: 双 src pack + 跟 src2 merge。
- v_vvc + unpack_merge: unpack + 跟 src2 merge (src1 不参与, 指令形态 schema 仍要 3 src)。
- v_vvs (3 src, src2 角色由 rnd_mode_fp 决定): rnd=rs → src2 是 stochastic random seed (fp8 量化训练用); rnd=rne/rtz/rup/rdn → src2 是 block scale factor (MX 标准 fp8/fp4 缩放)。

典型用例: FP32+FP32 → packed fp16/bf16 用 normal.v_vv; fp4/fp8 → fp16 unpack 用 unpack.v_v; fp32 → fp8 stochastic round 用 normal.v_vvs (rnd=rs); fp16+fp16 → packed fp8 with MX scale 用 normal.v_vvs (rnd=rne)。

rnd_mode_fp: 5 valid (rne / rtz / rup / rdn / rs); rs 仅 v_vvs 指令形态用 (stochastic round 走 src2 seed)。

ftz=ftz: fp32 denorm flush 0。saturate=saturate: fp dst clamp [-max_finite, +max_finite]。satfinite=satfinite: 溢出 clamp ±max_finite (不变 ±Inf)。relu=relu: 负数 clamp +0.0, NaN→+0.0。

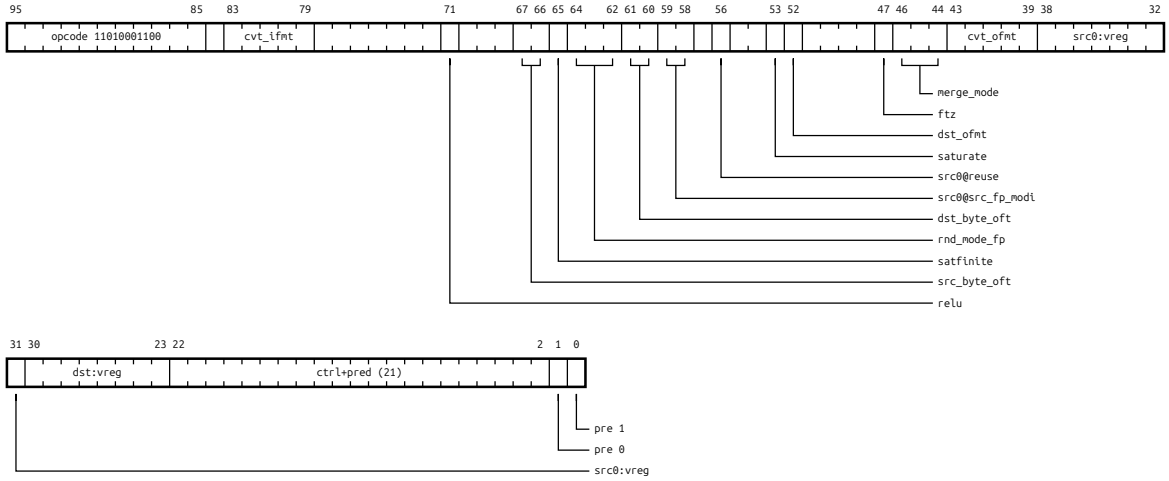
Leaf: f2fp__v_v

Format:

```
f2fp.cvt_ifmt.cvt_ofmt.rnd_mode_fp.ftz{.saturate}{.satfinite}{.relu}{.dst_ofmt}{.src_byt」
↪  e_oft}{.dst_byte_oft}{.merge_mode}  dst,
↪  src0{.src_fp_modi}{.reuse}
```

Operands: dst: vreg; src0: vreg

Encoding Diagram: 96b, prefix 10, opcode 11010001100



Notes:

浮点 → 浮点 packed 转换 (多 lane 写入): 覆盖单 src widen / 双 src pack / 子词 unpack / merge-with-old-dst / stochastic round + block-scale 等所有 fp 转换路径。整 warp per-lane 各自计算。

指令形态 / merge_mode 组合语义 (精确 dst 写法):

- v_v + normal: 单 src pad-zero 覆盖整 dst (dst.lo = convert(src0), dst.hi = 0)。
- v_v + unpack: 单 src 子词 unpack (dst[byte=dst_byte_oft, len=ofmt_size] = convert(src0[byte=src_byte_oft, len=ifmt_size]), dst 其它 byte 默认 0)。
- v_v + pack_hi: 单 src 写 dst 高 16-bit (dst.hi = convert(src0), dst.lo 保留旧值)。
- v_vv + normal (= PACK_AB): 双 src pack 覆盖整 dst (dst.lo = convert(src0), dst.hi = convert(src1)) — FP32+FP32 → packed fp16/bf16 主路径。
- v_vvc + merge: 单 src 转换 + 跟 src2 (旧 dst) merge 部分写。
- v_vvc + pack_merge: 双 src pack + 跟 src2 merge。
- v_vvc + unpack_merge: unpack + 跟 src2 merge (src1 不参与, 指令形态 schema 仍要 3 src)。
- v_vvs (3 src, src2 角色由 rnd_mode_fp 决定): rnd=rs → src2 是 stochastic random seed (fp8 量化训练用); rnd=rne/rtz/rup/rdn → src2 是 block scale factor (MX 标准 fp8/fp4 缩放)。

典型用例: FP32+FP32 → packed fp16/bf16 用 normal.v_vv; fp4/fp8 → fp16 unpack 用 unpack.v_v; fp32 → fp8 stochastic round 用 normal.v_vvs (rnd=rs); fp16+fp16 → packed fp8 with MX scale 用 normal.v_vvs (rnd=rne)。

rnd_mode_fp: 5 valid (rne / rtz / rup / rdn / rs); rs 仅 v_vvs 指令形态用 (stochastic round 走 src2 seed)。

ftz=ftz: fp32 denorm flush 0。saturate=saturate: fp dst clamp [-max_finite, +max_finite]。satfinite=satfinite: 溢出 clamp ±max_finite (不变 ±Inf)。relu=relu: 负数 clamp +0.0, NaN→+0.0。

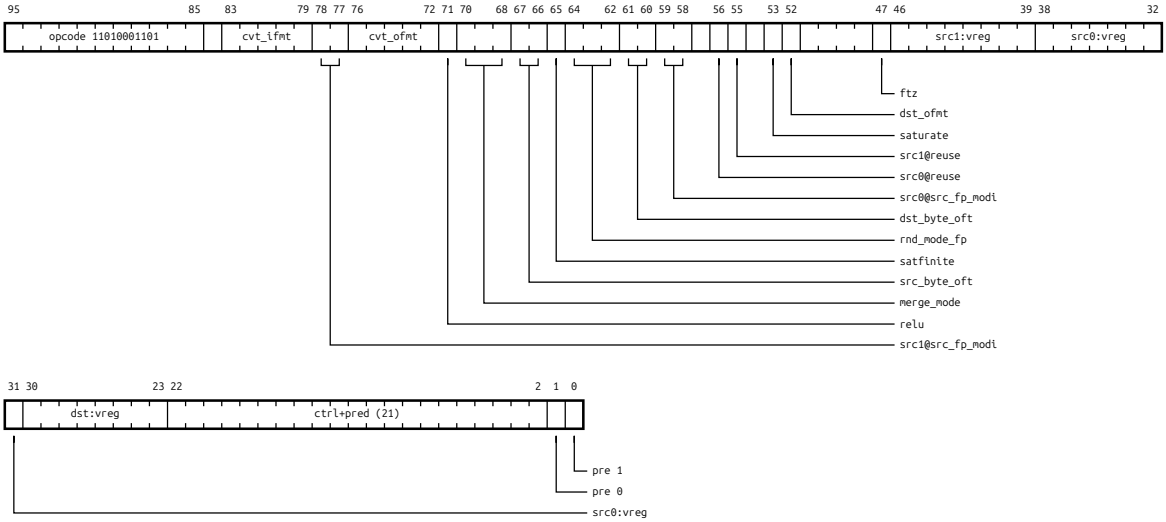
Leaf: f2fp__v_vv

Format:

```
f2fp.cvt_ifmt.cvt_ofmt.rnd_mode_fp.ftz{.saturate}{.satfinite}{.relu}{.dst_ofmt}{.src_byt」
↪  e_oft}{.dst_byte_oft}{.merge_mode}  dst, src0{.src_fp_modi}{.reuse},
↪  src1{.src_fp_modi}{.reuse}
```

Operands: dst: vreg; src0: vreg; src1: vreg

Encoding Diagram: 96b, prefix 10, opcode 11010001101



Notes:

浮点 → 浮点 packed 转换 (多 lane 写入): 覆盖单 src widen / 双 src pack / 子词 unpack / merge-with-old-dst / stochastic round + block-scale 等所有 fp 转换路径。整 warp per-lane 各自计算。

指令形态 / merge_mode 组合语义 (精确 dst 写法):

- v_v + normal: 单 src pad-zero 覆盖整 dst (dst.lo = convert(src0), dst.hi = 0)。
- v_v + unpack: 单 src 子词 unpack (dst[byte=dst_byte_oft, len=ofmt_size] = convert(src0[byte=src_byte_oft, len=ifmt_size]), dst 其它 byte 默认 0)。
- v_v + pack_hi: 单 src 写 dst 高 16-bit (dst.hi = convert(src0), dst.lo 保留旧值)。
- v_vv + normal (= PACK_AB): 双 src pack 覆盖整 dst (dst.lo = convert(src0), dst.hi = convert(src1)) — FP32+FP32 → packed fp16/bf16 主路径。
- v_vvc + merge: 单 src 转换 + 跟 src2 (旧 dst) merge 部分写。
- v_vvc + pack_merge: 双 src pack + 跟 src2 merge。
- v_vvc + unpack_merge: unpack + 跟 src2 merge (src1 不参与, 指令形态 schema 仍要 3 src)。
- v_vvs (3 src, src2 角色由 rnd_mode_fp 决定): rnd=rs → src2 是 stochastic random seed (fp8 量化训练用); rnd=rne/rtz/rup/rdn → src2 是 block scale factor (MX 标准 fp8/fp4 缩放)。

典型用例: FP32+FP32 → packed fp16/bf16 用 normal.v_vv; fp4/fp8 → fp16 unpack 用 unpack.v_v; fp32 → fp8 stochastic round 用 normal.v_vvs (rnd=rs); fp16+fp16 → packed fp8 with MX scale 用 normal.v_vvs (rnd=rne)。

rnd_mode_fp: 5 valid (rne / rtz / rup / rdn / rs); rs 仅 v_vvs 指令形态用 (stochastic round 走 src2 seed)。

ftz=ftz: fp32 denorm flush 0。saturate=saturate: fp dst clamp [-max_finite, +max_finite]。satfinite=satfinite: 溢出 clamp ±max_finite (不变 ±Inf)。relu=relu: 负数 clamp +0.0, NaN→+0.0。

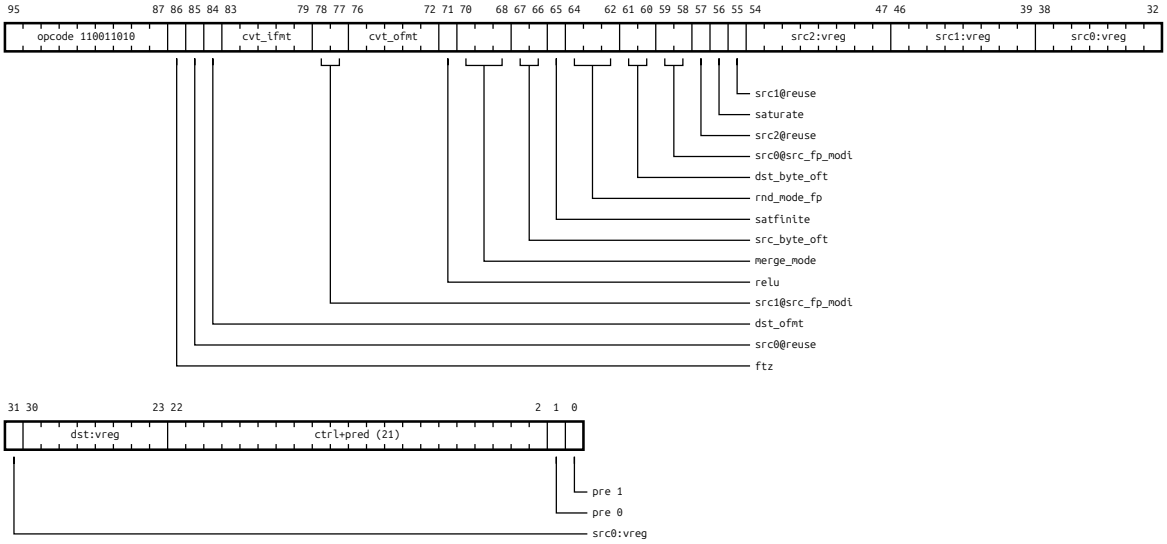
Leaf: f2fp__v_vvc

Format:

```
f2fp.cvt_ifmt.cvt_ofmt.rnd_mode_fp.ftz{.saturate}{.satfinite}{.relu}{.dst_ofmt}{.src_byt」
↪  e_oft}{.dst_byte_oft}{.merge_mode}  dst, src0{.src_fp_modi}{.reuse},
↪  src1{.src_fp_modi}{.reuse}, src2{.reuse}
```

Operands: dst: vreg; src0: vreg; src1: vreg; src2: vreg

Encoding Diagram: 96b, prefix 10, opcode 110011010



Notes:

浮点 → 浮点 packed 转换 (多 lane 写入): 覆盖单 src widen / 双 src pack / 子词 unpack / merge-with-old-dst / stochastic round + block-scale 等所有 fp 转换路径。整 warp per-lane 各自计算。

指令形态 / merge_mode 组合语义 (精确 dst 写法):

- v_v + normal: 单 src pad-zero 覆盖整 dst (dst.lo = convert(src0), dst.hi = 0)。
- v_v + unpack: 单 src 子词 unpack (dst[byte=dst_byte_oft, len=ofmt_size] = convert(src0[byte=src_byte_oft, len=ifmt_size]), dst 其它 byte 默认 0)。
- v_v + pack_hi: 单 src 写 dst 高 16-bit (dst.hi = convert(src0), dst.lo 保留旧值)。
- v_vv + normal (= PACK_AB): 双 src pack 覆盖整 dst (dst.lo = convert(src0), dst.hi = convert(src1)) — FP32+FP32 → packed fp16/bf16 主路径。
- v_vvc + merge: 单 src 转换 + 跟 src2 (旧 dst) merge 部分写。
- v_vvc + pack_merge: 双 src pack + 跟 src2 merge。
- v_vvc + unpack_merge: unpack + 跟 src2 merge (src1 不参与, 指令形态 schema 仍要 3 src)。
- v_vvs (3 src, src2 角色由 rnd_mode_fp 决定): rnd=rs → src2 是 stochastic random seed (fp8 量化训练用); rnd=rne/rtz/rup/rdn → src2 是 block scale factor (MX 标准 fp8/fp4 缩放)。

典型用例: FP32+FP32 → packed fp16/bf16 用 normal.v_vv; fp4/fp8 → fp16 unpack 用 unpack.v_v; fp32 → fp8 stochastic round 用 normal.v_vvs (rnd=rs); fp16+fp16 → packed fp8 with MX scale 用 normal.v_vvs (rnd=rne)。

rnd_mode_fp: 5 valid (rne / rtz / rup / rdn / rs); rs 仅 v_vvs 指令形态用 (stochastic round 走 src2 seed)。

ftz=ftz: fp32 denorm flush 0。saturate=saturate: fp dst clamp [-max_finite, +max_finite]。satfinite=satfinite: 溢出 clamp ±max_finite (不变 ±Inf)。relu=relu: 负数 clamp +0.0, NaN→+0.0。

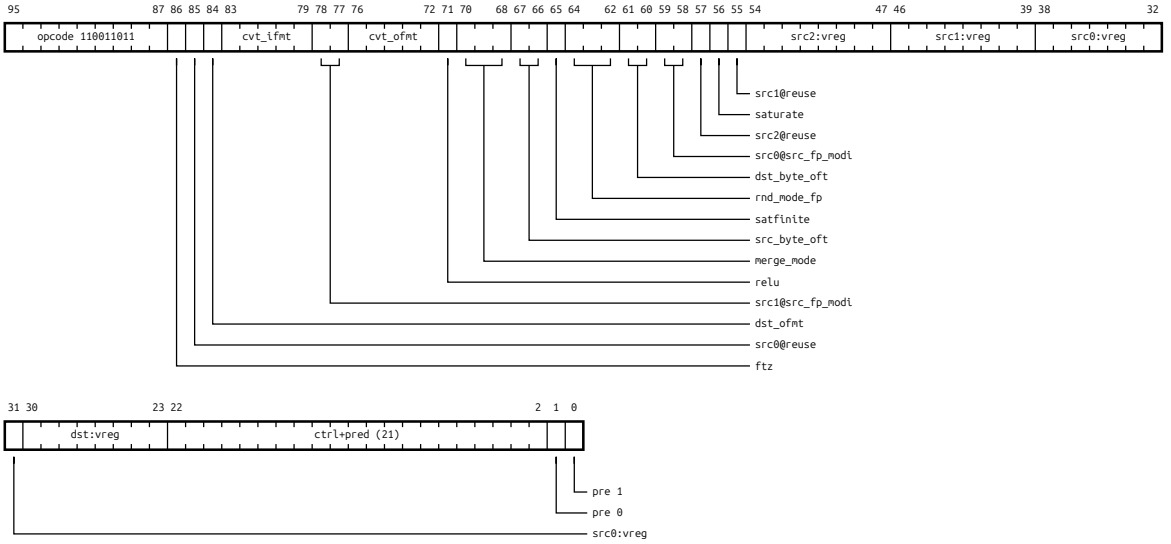
Leaf: f2fp__v_vvs

Format:

```
f2fp.cvt_ifmt.cvt_ofmt.rnd_mode_fp.ftz{.saturate}{.satfinite}{.relu}{.dst_ofmt}{.src_byt」
↪  e_oft}{.dst_byte_oft}{.merge_mode}  dst, src0{.src_fp_modi}{.reuse},
↪  src1{.src_fp_modi}{.reuse}, src2{.reuse}
```

Operands: dst: vreg; src0: vreg; src1: vreg; src2: vreg

Encoding Diagram: 96b, prefix 10, opcode 110011011



Notes:

浮点 → 浮点 packed 转换 (多 lane 写入): 覆盖单 src widen / 双 src pack / 子词 unpack / merge-with-old-dst / stochastic round + block-scale 等所有 fp 转换路径。整 warp per-lane 各自计算。

指令形态 / merge_mode 组合语义 (精确 dst 写法):

- v_v + normal: 单 src pad-zero 覆盖整 dst (dst.lo = convert(src0), dst.hi = 0)。
- v_v + unpack: 单 src 子词 unpack (dst[byte=dst_byte_oft, len=ofmt_size] = convert(src0[byte=src_byte_oft, len=ifmt_size]), dst 其它 byte 默认 0)。
- v_v + pack_hi: 单 src 写 dst 高 16-bit (dst.hi = convert(src0), dst.lo 保留旧值)。
- v_vv + normal (= PACK_AB): 双 src pack 覆盖整 dst (dst.lo = convert(src0), dst.hi = convert(src1)) — FP32+FP32 → packed fp16/bf16 主路径。
- v_vvc + merge: 单 src 转换 + 跟 src2 (旧 dst) merge 部分写。
- v_vvc + pack_merge: 双 src pack + 跟 src2 merge。
- v_vvc + unpack_merge: unpack + 跟 src2 merge (src1 不参与, 指令形态 schema 仍要 3 src)。
- v_vvs (3 src, src2 角色由 rnd_mode_fp 决定): rnd=rs → src2 是 stochastic random seed (fp8 量化训练用); rnd=rne/rtz/rup/rdn → src2 是 block scale factor (MX 标准 fp8/fp4 缩放)。

典型用例: FP32+FP32 → packed fp16/bf16 用 normal.v_vv; fp4/fp8 → fp16 unpack 用 unpack.v_v; fp32 → fp8 stochastic round 用 normal.v_vvs (rnd=rs); fp16+fp16 → packed fp8 with MX scale 用 normal.v_vvs (rnd=rne)。

rnd_mode_fp: 5 valid (rne / rtz / rup / rdn / rs); rs 仅 v_vvs 指令形态用 (stochastic round 走 src2 seed)。

ftz=ftz: fp32 denorm flush 0。saturate=saturate: fp dst clamp [-max_finite, +max_finite]。satfinite=satfinite: 溢出 clamp ±max_finite (不变 ±Inf)。relu=relu: 负数 clamp +0.0, NaN→+0.0。

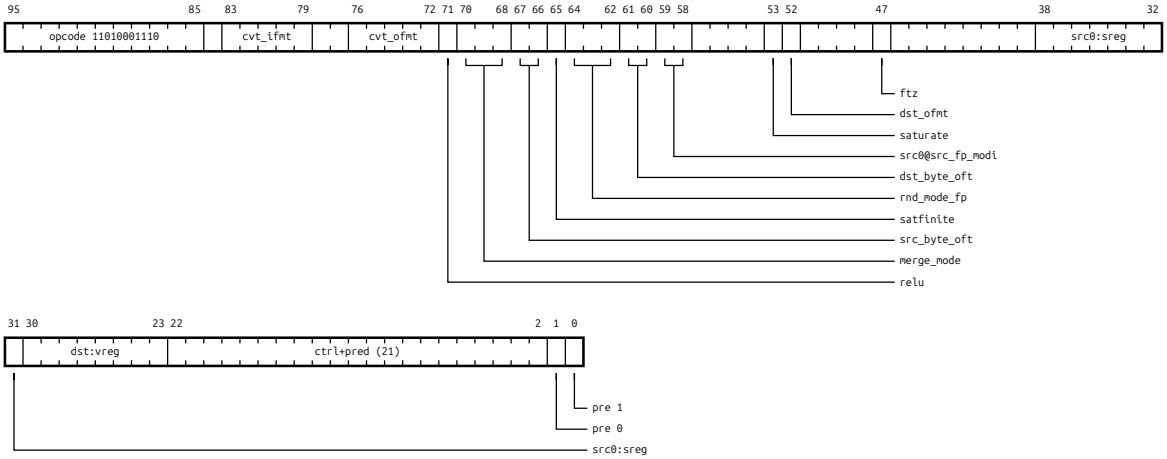
Leaf: f2fp__v_x

Format:

```
f2fp.cvt_ifmt.cvt_ofmt.rnd_mode_fp.ftz{.saturate}{.satfinite}{.relu}{.dst_ofmt}{.src_byt」
↪  e_oft}{.dst_byte_oft}{.merge_mode}  dst,
↪  src0{.src_fp_modi}
```

Operands: dst: vreg; src0: sreg

Encoding Diagram: 96b, prefix 10, opcode 11010001110



Notes:

浮点 → 浮点 packed 转换 (多 lane 写入): 覆盖单 src widen / 双 src pack / 子词 unpack / merge-with-old-dst / stochastic round + block-scale 等所有 fp 转换路径。整 warp per-lane 各自计算。

指令形态 / merge_mode 组合语义 (精确 dst 写法):

- v_v + normal: 单 src pad-zero 覆盖整 dst (dst.lo = convert(src0), dst.hi = 0)。
- v_v + unpack: 单 src 子词 unpack (dst[byte=dst_byte_oft, len=ofmt_size] = convert(src0[byte=src_byte_oft, len=ifmt_size]), dst 其它 byte 默认 0)。
- v_v + pack_hi: 单 src 写 dst 高 16-bit (dst.hi = convert(src0), dst.lo 保留旧值)。
- v_vv + normal (= PACK_AB): 双 src pack 覆盖整 dst (dst.lo = convert(src0), dst.hi = convert(src1)) — FP32+FP32 → packed fp16/bf16 主路径。
- v_vvc + merge: 单 src 转换 + 跟 src2 (旧 dst) merge 部分写。
- v_vvc + pack_merge: 双 src pack + 跟 src2 merge。
- v_vvc + unpack_merge: unpack + 跟 src2 merge (src1 不参与, 指令形态 schema 仍要 3 src)。
- v_vvs (3 src, src2 角色由 rnd_mode_fp 决定): rnd=rs → src2 是 stochastic random seed (fp8 量化训练用); rnd=rne/rtz/rup/rdn → src2 是 block scale factor (MX 标准 fp8/fp4 缩放)。

典型用例: FP32+FP32 → packed fp16/bf16 用 normal.v_vv; fp4/fp8 → fp16 unpack 用 unpack.v_v; fp32 → fp8 stochastic round 用 normal.v_vvs (rnd=rs); fp16+fp16 → packed fp8 with MX scale 用 normal.v_vvs (rnd=rne)。

rnd_mode_fp: 5 valid (rne / rtz / rup / rdn / rs); rs 仅 v_vvs 指令形态用 (stochastic round 走 src2 seed)。

ftz=ftz: fp32 denorm flush 0。saturate=saturate: fp dst clamp [-max_finite, +max_finite]。satfinite=satfinite: 溢出 clamp ±max_finite (不变 ±Inf)。relu=relu: 负数 clamp +0.0, NaN→+0.0。

f2i **category**: Conversion **predicate_mode**: simt

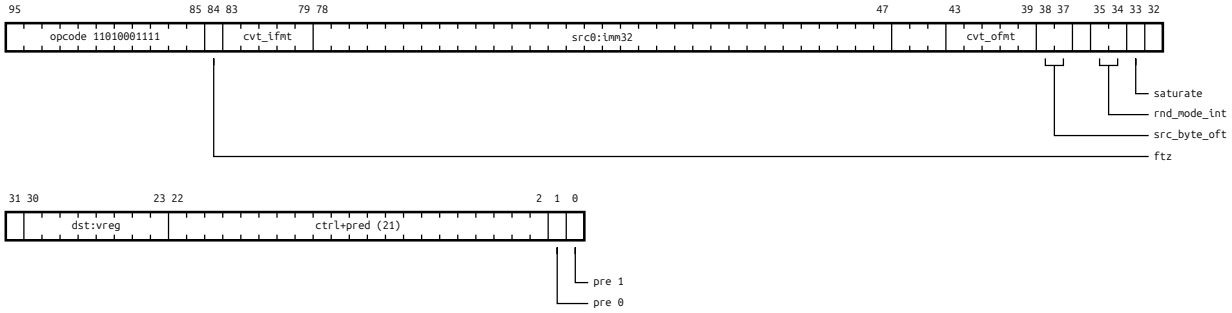
Leaf: f2i__v_i

Format:

```
f2i.cvt_ifmt.cvt_ofmt.rnd_mode_int.ftz{.saturate}{.src_byte_oft}  dst, src0
```

Operands: dst: vreg; src0: imm32u

Encoding Diagram: 96b, prefix 10, opcode 11010001111



Notes:

浮点 → 整数转换: 把 src0 按 cvt_ifmt 解读的浮点数据按 rnd_mode_int 取整到 cvt_ofmt 整数类型, 结果写 dst。整 warp per-lane 各自计算。

cvt_ifmt: fp 类型 (fp16 / bf16 / FP32 / tf32 + FP8 / FP6 / FP4 / e8 微缩格式)。cvt_ofmt: int 类型 (u8 / s8 / u16 / s16 / u32 / s32 / u64 / s64 + u4 / s4 / u2 / s2)。

src 子词 / 64-bit 路径: src0 < 32-bit 用 src_byte_oft 选 byte 起始; s64/u64 dst 占 2 个相邻 reg。

rnd_mode_int: 4 valid (rni round-to-nearest-int / rzi truncate / rupi ceil / rdni floor) — 取整到整数值, 跟 rnd_mode_fp 不同语义。

ftz=ftz: fp32 input denorm flush 0 后再取整。

saturate=saturate: int dst 溢出饱和到 dst 类型边界 ([INT_MIN, INT_MAX] 或 [0, UINT_MAX]); fp NaN → 0 (IEEE FP-to-int 默认行为)。

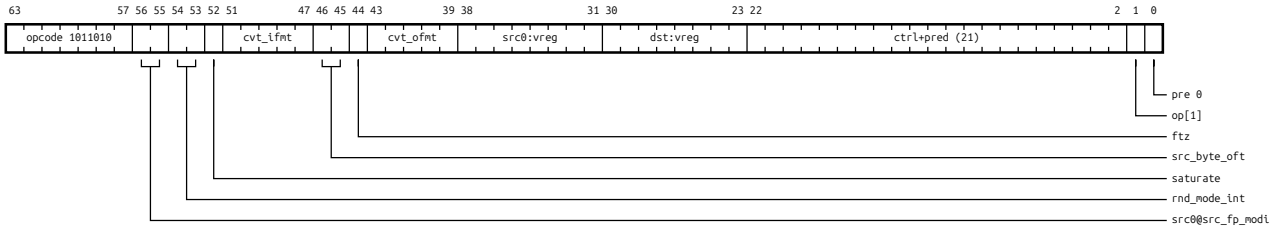
Leaf: f2i__v_v

Format:

```
f2i.cvt_ifmt.cvt_ofmt.rnd_mode_int.ftz{.saturate}{.src_byte_oft}  dst, src0{.src_fp_modi}
```

Operands: dst: vreg; src0: vreg

Encoding Diagram: 64b, prefix 0, opcode 11011010



Notes:

浮点 → 整数转换: 把 src0 按 cvt_ifmt 解读的浮点数据按 rnd_mode_int 取整到 cvt_ofmt 整数类型, 结果写 dst。整 warp per-lane 各自计算。

cvt_ifmt: fp 类型 (fp16 / bf16 / FP32 / tf32 + FP8 / FP6 / FP4 / e8 微缩格式)。cvt_ofmt: int 类型 (u8 / s8 / u16 / s16 / u32 / s32 / u64 / s64 + u4 / s4 / u2 / s2)。

src 子词 / 64-bit 路径: src0 < 32-bit 用 src_byte_oft 选 byte 起始; s64/u64 dst 占 2 个相邻 reg。

rnd_mode_int: 4 valid (rni round-to-nearest-int / rzi truncate / rupi ceil / rdni floor) — 取整到整数值, 跟 rnd_mode_fp 不同语义。

ftz=ftz: fp32 input denorm flush 0 后再取整。

saturate=saturate: int dst 溢出饱和到 dst 类型边界 ([INT_MIN, INT_MAX] 或 [0, UINT_MAX]); fp NaN → 0 (IEEE FP-to-int 默认行为)。

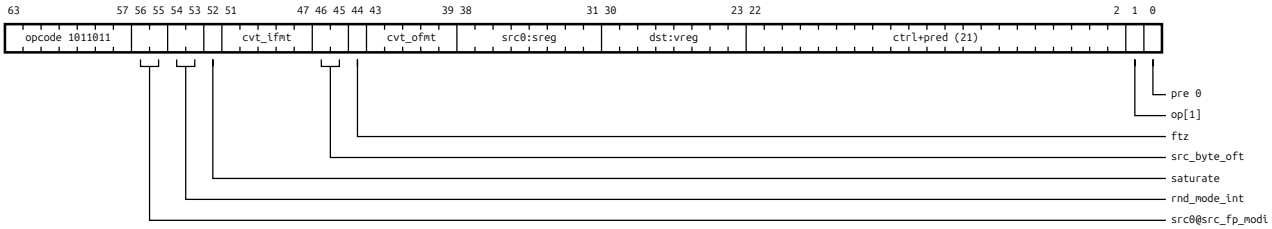
Leaf: f2i__v_x

Format:

```
f2i.cvt_ifmt.cvt_ofmt.rnd_mode_int.ftz{.saturate}{.src_byte_oft}  dst, src0{.src_fp_modi}
```

Operands: dst: vreg; src0: sreg

Encoding Diagram: 64b, prefix 0, opcode 11011011



Notes:

浮点 → 整数转换: 把 src0 按 cvt_ifmt 解读的浮点数据按 rnd_mode_int 取整到 cvt_ofmt 整数类型, 结果写 dst。整 warp per-lane 各自计算。

cvt_ifmt: fp 类型 (fp16 / bf16 / FP32 / tf32 + FP8 / FP6 / FP4 / e8 微缩格式)。cvt_ofmt: int 类型 (u8 / s8 / u16 / s16 / u32 / s32 / u64 / s64 + u4 / s4 / u2 / s2)。

src 子词 / 64-bit 路径: src0 < 32-bit 用 src_byte_oft 选 byte 起始; s64/u64 dst 占 2 个相邻 reg。

rnd_mode_int: 4 valid (rni round-to-nearest-int / rzi truncate / rupi ceil / rdni floor) — 取整到整数值, 跟 rnd_mode_fp 不同语义。

ftz=ftz: fp32 input denorm flush 0 后再取整。

saturate=saturate: int dst 溢出饱和到 dst 类型边界 ([INT_MIN, INT_MAX] 或 [0, UINT_MAX]); fp NaN → 0 (IEEE FP-to-int 默认行为)。

f2ip　**category**: Conversion　**predicate_mode**: simt

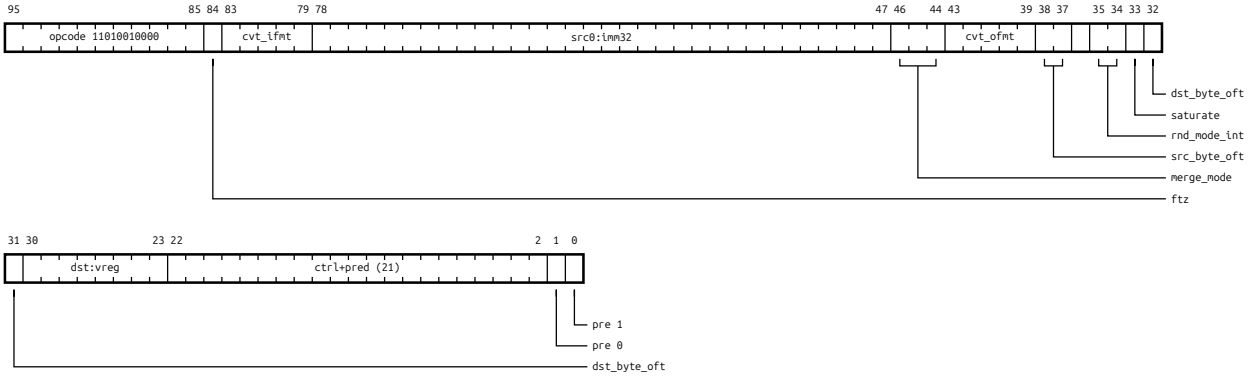
Leaf: f2ip__v_i

Format:

```
f2ip.cvt_ifmt.cvt_ofmt.rnd_mode_int.ftz{.saturate}{.src_byte_oft}{.dst_byte_oft}{.merge_」
↪  mode}  dst,
↪  src0
```

Operands: dst: vreg; src0: imm32u

Encoding Diagram: 96b, prefix 10, opcode 11010010000



Notes:

浮点 → 整数 packed 转换 (多 lane 写入): 跟 f2fp 同结构, 仅 dst 类型从 fp 改 int, 按 rnd_mode_int 取整。整 warp per-lane 各自计算。

指令形态 / merge_mode 组合语义:

- v_v + unpack: 单 src fp → int 子词 unpack (典型 fp16 → s8 byte 写入 dst[byte=dst_byte_oft])。
- v_v + normal: 单 src fp → int pad-zero 覆盖整 dst。
- v_vv + normal (= PACK_AB): 双 src fp → packed int (dst.lo = convert(src0), dst.hi = convert(src1)) — fp16+fp16 → packed s16/u16 主路径。
- v_vvc + merge / pack_merge / unpack_merge: 跟 src2 (旧 dst) merge 部分写。

典型用例: FP32 → INT8/INT16 packed 量化 (ML 推理); fp16+fp16 → packed s16/u16; fp16 → s8 byte unpack。

rnd_mode_int: 4 valid (rni / rzi truncate / rupi / rdni)。

ftz=ftz: fp32 input denorm flush 0。saturate=saturate: int dst 溢出饱和到 dst 类型边界 (fp NaN → 0 IEEE 默认)。

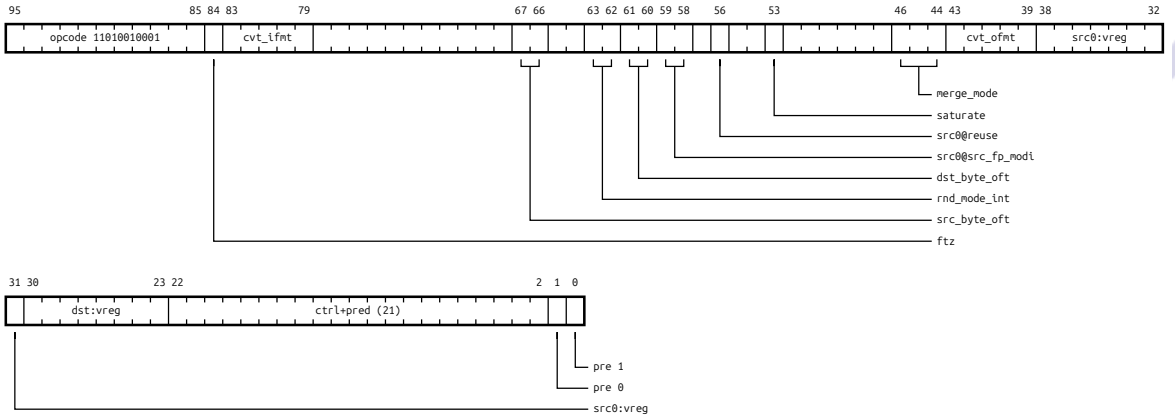
Leaf: f2ip__v_v

Format:

```
f2ip.cvt_ifmt.cvt_ofmt.rnd_mode_int.ftz{.saturate}{.src_byte_oft}{.dst_byte_oft}{.merge_」
↪  mode}  dst,
↪  src0{.src_fp_modi}{.reuse}
```

Operands: dst: vreg; src0: vreg

Encoding Diagram: 96b, prefix 10, opcode 11010010001



Notes:

浮点 → 整数 packed 转换 (多 lane 写入): 跟 f2fp 同结构, 仅 dst 类型从 fp 改 int, 按 rnd_mode_int 取整。整 warp per-lane 各自计算。

指令形态 / merge_mode 组合语义:

- v_v + unpack: 单 src fp → int 子词 unpack (典型 fp16 → s8 byte 写入 dst[byte=dst_byte_oft])。
- v_v + normal: 单 src fp → int pad-zero 覆盖整 dst。
- v_vv + normal (= PACK_AB): 双 src fp → packed int (dst.lo = convert(src0), dst.hi = convert(src1)) — fp16+fp16 → packed s16/u16 主路径。
- v_vvc + merge / pack_merge / unpack_merge: 跟 src2 (旧 dst) merge 部分写。

典型用例: FP32 → INT8/INT16 packed 量化 (ML 推理); fp16+fp16 → packed s16/u16; fp16 → s8 byte unpack。

rnd_mode_int: 4 valid (rni / rzi truncate / rupi / rdni)。

ftz=ftz: fp32 input denorm flush 0。saturate=saturate: int dst 溢出饱和到 dst 类型边界 (fp NaN → 0 IEEE 默认)。

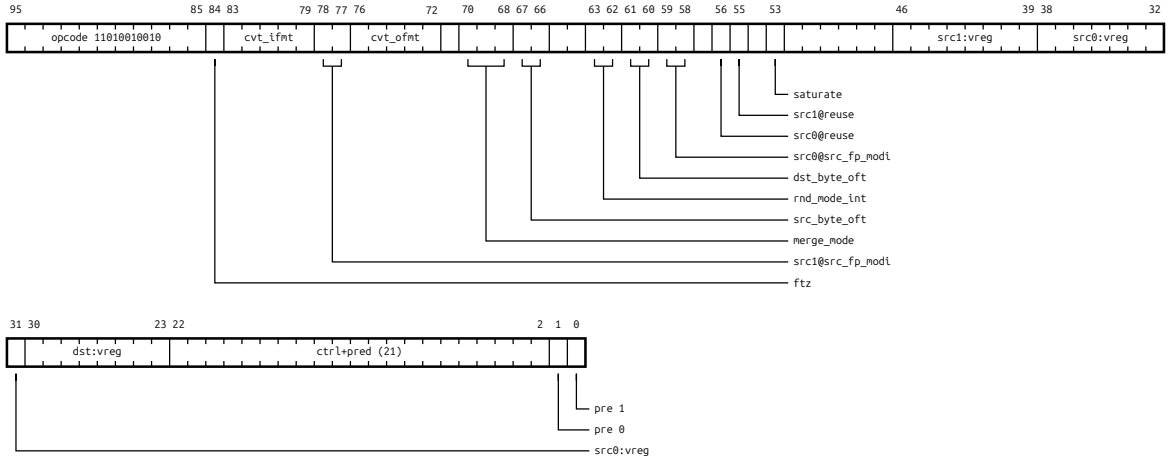
Leaf: f2ip__v_vv

Format:

```
f2ip.cvt_ifmt.cvt_ofmt.rnd_mode_int.ftz{.saturate}{.src_byte_oft}{.dst_byte_oft}{.merge_」
↪  mode}  dst, src0{.src_fp_modi}{.reuse},
↪  src1{.src_fp_modi}{.reuse}
```

Operands: dst: vreg; src0: vreg; src1: vreg

Encoding Diagram: 96b, prefix 10, opcode 11010010010



Notes:

浮点 → 整数 packed 转换 (多 lane 写入): 跟 f2fp 同结构, 仅 dst 类型从 fp 改 int, 按 rnd_mode_int 取整。整 warp per-lane 各自计算。

指令形态 / merge_mode 组合语义:

- v_v + unpack: 单 src fp → int 子词 unpack (典型 fp16 → s8 byte 写入 dst[byte=dst_byte_oft])。
- v_v + normal: 单 src fp → int pad-zero 覆盖整 dst。
- v_vv + normal (= PACK_AB): 双 src fp → packed int (dst.lo = convert(src0), dst.hi = convert(src1)) — fp16+fp16 → packed s16/u16 主路径。
- v_vvc + merge / pack_merge / unpack_merge: 跟 src2 (旧 dst) merge 部分写。

典型用例: FP32 → INT8/INT16 packed 量化 (ML 推理); fp16+fp16 → packed s16/u16; fp16 → s8 byte unpack。

rnd_mode_int: 4 valid (rni / rzi truncate / rupi / rdni)。

ftz=ftz: fp32 input denorm flush 0。saturate=saturate: int dst 溢出饱和到 dst 类型边界 (fp NaN → 0 IEEE 默认)。

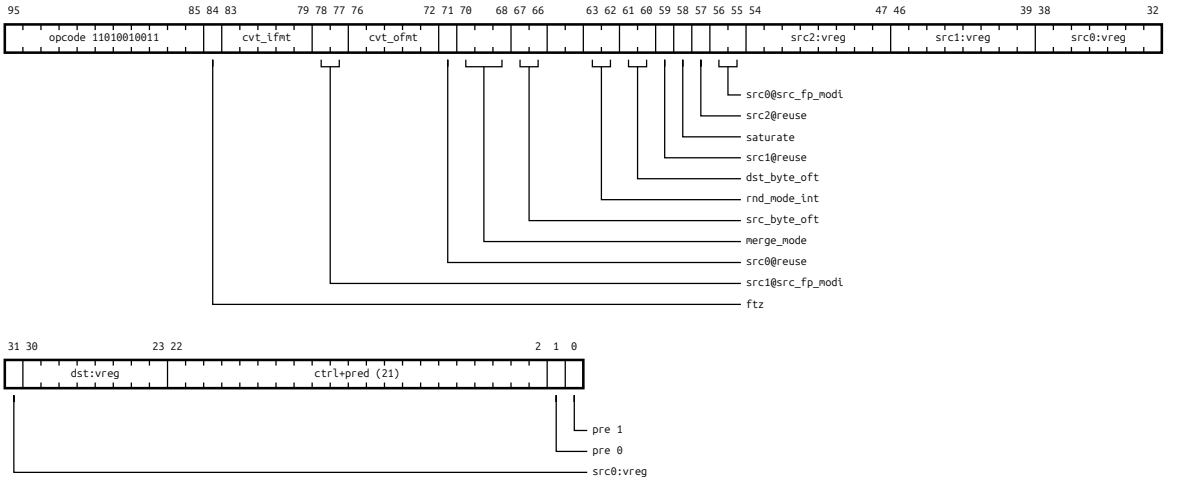
Leaf: f2ip__v_vvc

Format:

```
f2ip.cvt_ifmt.cvt_ofmt.rnd_mode_int.ftz{.saturate}{.src_byte_oft}{.dst_byte_oft}{.merge_」
↪  mode}  dst, src0{.src_fp_modi}{.reuse}, src1{.src_fp_modi}{.reuse},
↪  src2{.reuse}
```

Operands: dst: vreg; src0: vreg; src1: vreg; src2: vreg

Encoding Diagram: 96b, prefix 10, opcode 11010010011



Notes:

浮点 → 整数 packed 转换 (多 lane 写入): 跟 f2fp 同结构, 仅 dst 类型从 fp 改 int, 按 rnd_mode_int 取整。整 warp per-lane 各自计算。

指令形态 / merge_mode 组合语义:

- v_v + unpack: 单 src fp → int 子词 unpack (典型 fp16 → s8 byte 写入 dst[byte=dst_byte_oft])。
- v_v + normal: 单 src fp → int pad-zero 覆盖整 dst。
- v_vv + normal (= PACK_AB): 双 src fp → packed int (dst.lo = convert(src0), dst.hi = convert(src1)) — fp16+fp16 → packed s16/u16 主路径。
- v_vvc + merge / pack_merge / unpack_merge: 跟 src2 (旧 dst) merge 部分写。

典型用例: FP32 → INT8/INT16 packed 量化 (ML 推理); fp16+fp16 → packed s16/u16; fp16 → s8 byte unpack。

rnd_mode_int: 4 valid (rni / rzi truncate / rupi / rdni)。

ftz=ftz: fp32 input denorm flush 0。saturate=saturate: int dst 溢出饱和到 dst 类型边界 (fp NaN → 0 IEEE 默认)。

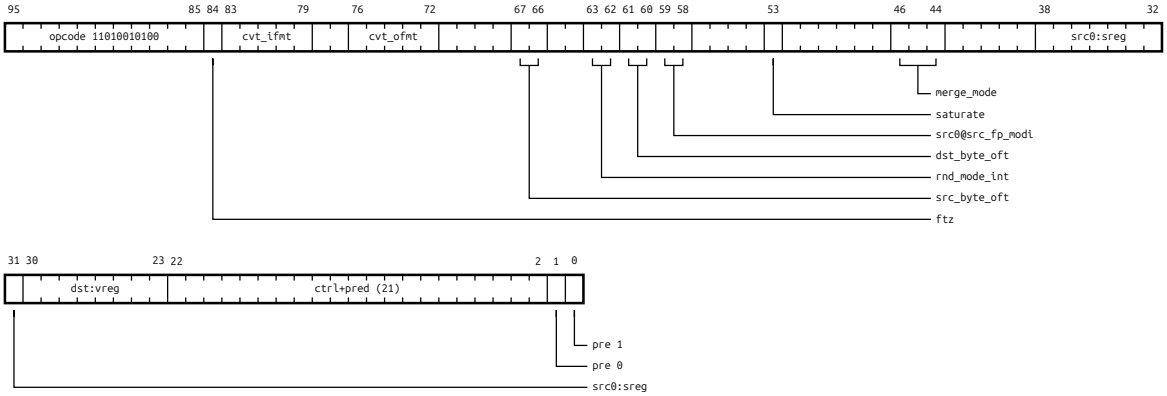
Leaf: f2ip__v_x

Format:

```
f2ip.cvt_ifmt.cvt_ofmt.rnd_mode_int.ftz{.saturate}{.src_byte_oft}{.dst_byte_oft}{.merge_」
↪  mode}  dst,
↪  src0{.src_fp_modi}
```

Operands: dst: vreg; src0: sreg

Encoding Diagram: 96b, prefix 10, opcode 11010010100



Notes:

浮点 → 整数 packed 转换 (多 lane 写入): 跟 f2fp 同结构, 仅 dst 类型从 fp 改 int, 按 rnd_mode_int 取整。整 warp per-lane 各自计算。

指令形态 / merge_mode 组合语义:

- v_v + unpack: 单 src fp → int 子词 unpack (典型 fp16 → s8 byte 写入 dst[byte=dst_byte_oft])。
- v_v + normal: 单 src fp → int pad-zero 覆盖整 dst。
- v_vv + normal (= PACK_AB): 双 src fp → packed int (dst.lo = convert(src0), dst.hi = convert(src1)) — fp16+fp16 → packed s16/u16 主路径。
- v_vvc + merge / pack_merge / unpack_merge: 跟 src2 (旧 dst) merge 部分写。

典型用例: FP32 → INT8/INT16 packed 量化 (ML 推理); fp16+fp16 → packed s16/u16; fp16 → s8 byte unpack。

rnd_mode_int: 4 valid (rni / rzi truncate / rupi / rdni)。

ftz=ftz: fp32 input denorm flush 0。saturate=saturate: int dst 溢出饱和到 dst 类型边界 (fp NaN → 0 IEEE 默认)。

frnd **category**: Conversion **predicate_mode**: simt

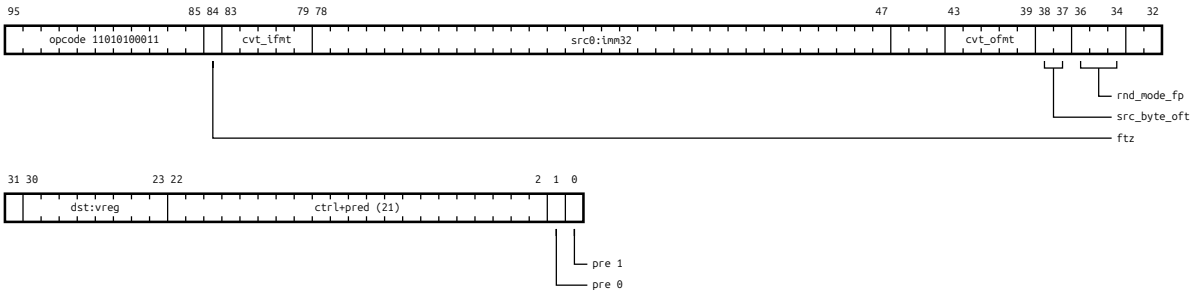
Leaf: frnd__v_i

Format:

```
frnd.cvt_ifmt.cvt_ofmt.rnd_mode_fp.ftz{.src_byte_oft}  dst, src0
```

Operands: dst: vreg; src0: imm32u

Encoding Diagram: 96b, prefix 10, opcode 11010100011



Notes:

浮点同类型舍入到整数值: 保留 fp 类型不变, 把 src0 浮点数按 rnd_mode_fp 舍入到最接近的整数值 (仍是 fp 表示) 写 dst。例: f32 1.7 → f32 2.0 (rne) / f32 1.7 → f32 1.0 (rtz)。整 warp per-lane 各自计算。

cvt_ifmt 跟 cvt_ofmt 必须同 fp 类型 (f32→f32 / f16→f16); 跟 f2f 区别: f2f 改 fp 类型 (如 f32→f16), frnd 不改类型仅做 round-to-integer。

src 子词路径: src0 < 32-bit 用 src_byte_oft 选 byte 起始 (高频用例 R0.b2 fp16 → fp16 round)。

rnd_mode_fp: 4 valid (rne 默认 ML 用 / rtz 整数 cast 用 / rup ceil / rdn floor); frnd 不支持 rs。

ftz=ftz: fp32 input / output denorm flush 0。

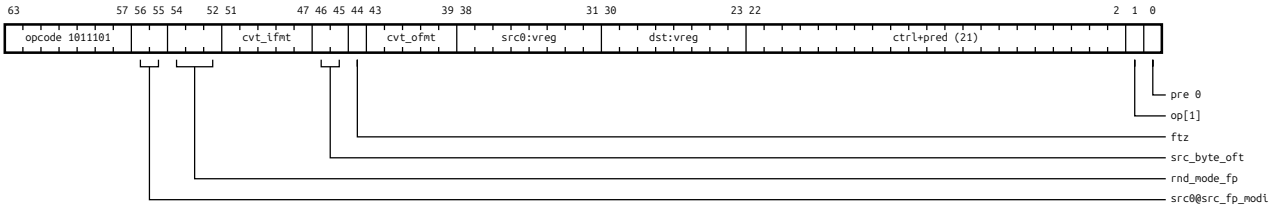
Leaf: frnd__v_v

Format:

```
frnd.cvt_ifmt.cvt_ofmt.rnd_mode_fp.ftz{.src_byte_oft}  dst, src0{.src_fp_modi}
```

Operands: dst: vreg; src0: vreg

Encoding Diagram: 64b, prefix 0, opcode 11011101



Notes:

浮点同类型舍入到整数值: 保留 fp 类型不变, 把 src0 浮点数按 rnd_mode_fp 舍入到最接近的整数值 (仍是 fp 表示) 写 dst。例: f32 1.7 → f32 2.0 (rne) / f32 1.7 → f32 1.0 (rtz)。整 warp per-lane 各自计算。

cvt_ifmt 跟 cvt_ofmt 必须同 fp 类型 (f32→f32 / f16→f16); 跟 f2f 区别: f2f 改 fp 类型 (如 f32→f16), frnd 不改类型仅做 round-to-integer。

src 子词路径: src0 < 32-bit 用 src_byte_oft 选 byte 起始 (高频用例 R0.b2 fp16 → fp16 round)。

rnd_mode_fp: 4 valid (rne 默认 ML 用 / rtz 整数 cast 用 / rup ceil / rdn floor); frnd 不支持 rs。

ftz=ftz: fp32 input / output denorm flush 0。

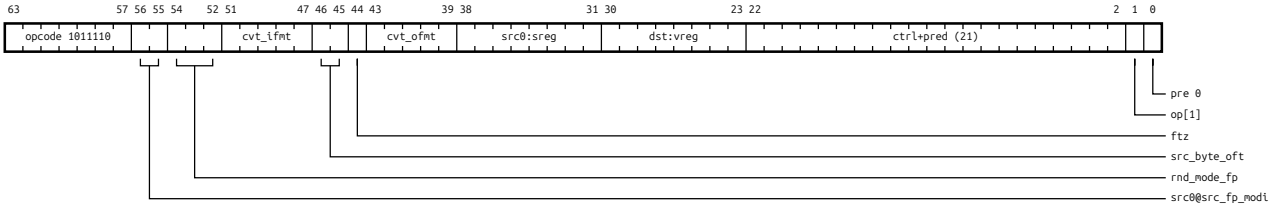
Leaf: frnd__v_x

Format:

```
frnd.cvt_ifmt.cvt_ofmt.rnd_mode_fp.ftz{.src_byte_oft}  dst, src0{.src_fp_modi}
```

Operands: dst: vreg; src0: sreg

Encoding Diagram: 64b, prefix 0, opcode 11011110



Notes:

浮点同类型舍入到整数值: 保留 fp 类型不变, 把 src0 浮点数按 rnd_mode_fp 舍入到最接近的整数值 (仍是 fp 表示) 写 dst。例: f32 1.7 → f32 2.0 (rne) / f32 1.7 → f32 1.0 (rtz)。整 warp per-lane 各自计算。

cvt_ifmt 跟 cvt_ofmt 必须同 fp 类型 (f32→f32 / f16→f16); 跟 f2f 区别: f2f 改 fp 类型 (如 f32→f16), frnd 不改类型仅做 round-to-integer。

src 子词路径: src0 < 32-bit 用 src_byte_oft 选 byte 起始 (高频用例 R0.b2 fp16 → fp16 round)。

rnd_mode_fp: 4 valid (rne 默认 ML 用 / rtz 整数 cast 用 / rup ceil / rdn floor); frnd 不支持 rs。

ftz=ftz: fp32 input / output denorm flush 0。

i2f **category**: Conversion **predicate_mode**: simt

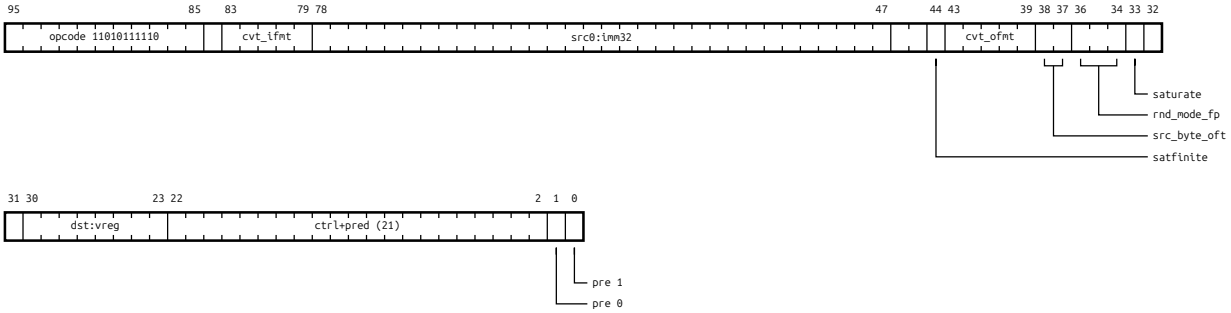
Leaf: i2f__v_i

Format:

```
i2f.cvt_ifmt.cvt_ofmt.rnd_mode_fp{.saturate}{.satfinite}{.src_byte_oft}  dst, src0
```

Operands: dst: vreg; src0: imm32u

Encoding Diagram: 96b, prefix 10, opcode 11010111110



Notes:

整数 → 浮点转换: 把 src0 按 cvt_ifmt 解读的整数数据转成 cvt_ofmt 浮点类型, 按 rnd_mode_fp 舍入 (精度不足时选最接近的 fp 表示), 结果写 dst。整 warp per-lane 各自计算。

cvt_ifmt: int 类型 (u8 / s8 / u16 / s16 / u32 / s32 / u64 / s64 + u4 / s4 / u2 / s2)。cvt_ofmt: fp 类型 (传统 fp16/bf16/fp32/tf32 + fp8/fp6/fp4 微缩格式)。

src 子词 / 64-bit 路径: src0 < 32-bit 用 src_byte_oft 选 byte 起始 (高频用例 R0.b1 s8 → f32); u64/s64 src 占 2 个相邻 reg。

rnd_mode_fp: 4 valid (rne / rtz / rup / rdn), int → fp 精度不足 (如 s32 → f16) 时按 rnd 选最接近 fp 值。

saturate=saturate: fp dst clamp 到 [-max_finite, +max_finite], NaN 保持。

satfinite=satfinite: fp dst 溢出不变 ±Inf 而 clamp 到 ±max_finite。

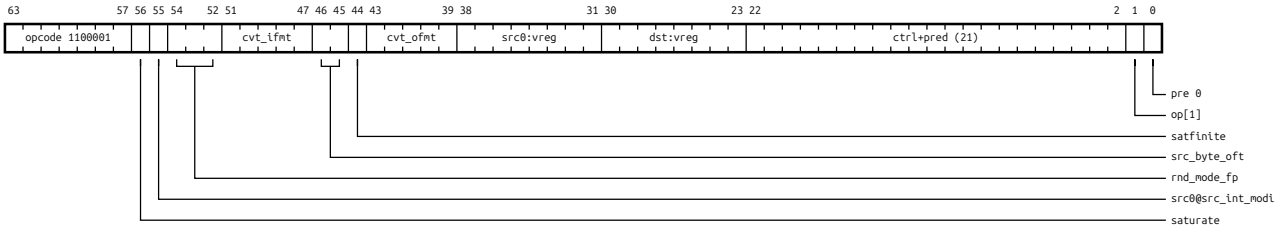
Leaf: i2f__v_v

Format:

```
i2f.cvt_ifmt.cvt_ofmt.rnd_mode_fp{.saturate}{.satfinite}{.src_byte_oft}  dst,
↪  src0{.src_int_modi}
```

Operands: dst: vreg; src0: vreg

Encoding Diagram: 64b, prefix 0, opcode 11100001



Notes:

整数 → 浮点转换: 把 src0 按 cvt_ifmt 解读的整数数据转成 cvt_ofmt 浮点类型, 按 rnd_mode_fp 舍入 (精度不足时选最接近的 fp 表示), 结果写 dst。整 warp per-lane 各自计算。

cvt_ifmt: int 类型 (u8 / s8 / u16 / s16 / u32 / s32 / u64 / s64 + u4 / s4 / u2 / s2)。cvt_ofmt: fp 类型 (传统 fp16/bf16/fp32/tf32 + fp8/fp6/fp4 微缩格式)。

src 子词 / 64-bit 路径: src0 < 32-bit 用 src_byte_oft 选 byte 起始 (高频用例 R0.b1 s8 → f32); u64/s64 src 占 2 个相邻 reg。

rnd_mode_fp: 4 valid (rne / rtz / rup / rdn), int → fp 精度不足 (如 s32 → f16) 时按 rnd 选最接近 fp 值。

saturate=saturate: fp dst clamp 到 [-max_finite, +max_finite], NaN 保持。

satfinite=satfinite: fp dst 溢出不变 ±Inf 而 clamp 到 ±max_finite。

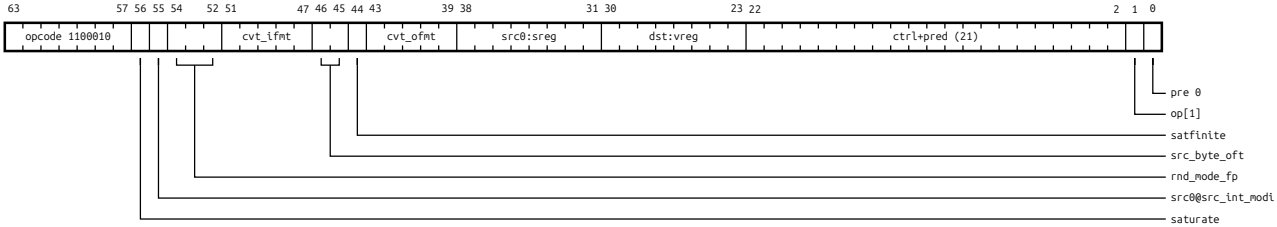
Leaf: i2f__v_x

Format:

```
i2f.cvt_ifmt.cvt_ofmt.rnd_mode_fp{.saturate}{.satfinite}{.src_byte_oft}  dst,
↪  src0{.src_int_modi}
```

Operands: dst: vreg; src0: sreg

Encoding Diagram: 64b, prefix 0, opcode 11100010



Notes:

整数 → 浮点转换: 把 src0 按 cvt_ifmt 解读的整数数据转成 cvt_ofmt 浮点类型, 按 rnd_mode_fp 舍入 (精度不足时选最接近的 fp 表示), 结果写 dst。整 warp per-lane 各自计算。

cvt_ifmt: int 类型 (u8 / s8 / u16 / s16 / u32 / s32 / u64 / s64 + u4 / s4 / u2 / s2)。cvt_ofmt: fp 类型 (传统 fp16/bf16/fp32/tf32 + fp8/fp6/fp4 微缩格式)。

src 子词 / 64-bit 路径: src0 < 32-bit 用 src_byte_oft 选 byte 起始 (高频用例 R0.b1 s8 → f32); u64/s64 src 占 2 个相邻 reg。

rnd_mode_fp: 4 valid (rne / rtz / rup / rdn), int → fp 精度不足 (如 s32 → f16) 时按 rnd 选最接近 fp 值。

saturate=saturate: fp dst clamp 到 [-max_finite, +max_finite], NaN 保持。

satfinite=satfinite: fp dst 溢出不变 ±Inf 而 clamp 到 ±max_finite。

i2fp **category**: Conversion **predicate_mode**: simt

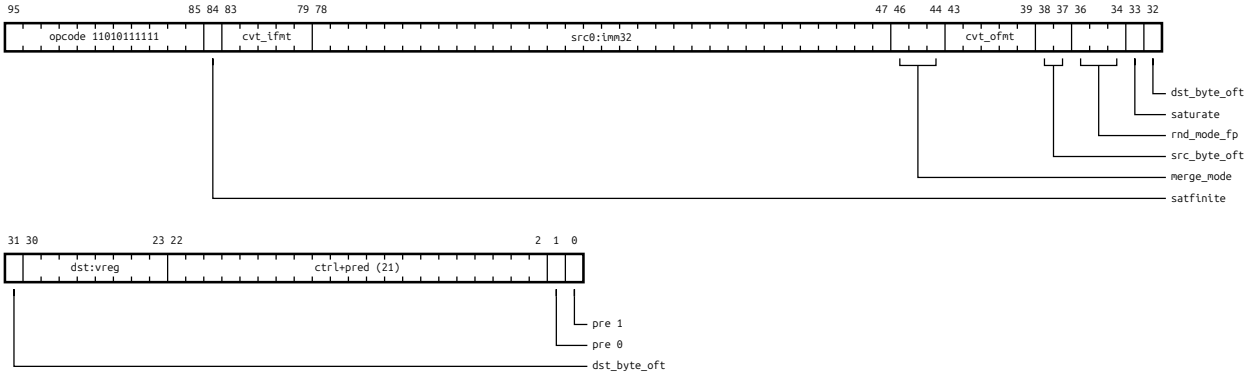
Leaf: i2fp__v_i

Format:

```
i2fp.cvt_ifmt.cvt_ofmt.rnd_mode_fp{.saturate}{.satfinite}{.src_byte_oft}{.dst_byte_oft}{」
↪  .merge_mode}  dst,
↪  src0
```

Operands: dst: vreg; src0: imm32u

Encoding Diagram: 96b, prefix 10, opcode 11010111111



Notes:

整数 → 浮点 packed 转换 (多 lane 写入): 跟 f2fp 同结构, 仅 src/dst 方向从 fp→fp 改 int→fp。整 warp per-lane 各自计算。

指令形态 / merge_mode 组合语义:

- v_v + unpack: 单 src 子词 int → fp unpack (典型 s8 byte → fp16/bf16, ML INT8 反量化主路径)。
- v_v + normal: 单 src int → fp pad-zero 覆盖整 dst。
- v_vv + normal (= PACK_AB): 双 src int → packed fp (dst.lo = convert(src0), dst.hi = convert(src1)) — INT16+INT16 → packed fp16/bf16 主路径。
- v_vvc + merge 系列: 跟 src2 (旧 dst) merge 部分写。

典型用例: INT8 → bf16/fp16 packed 反量化 (ML dequant); INT16 → fp16/bf16 packed widen。

rnd_mode_fp: 4 valid (rne / rtz / rup / rdn), int → fp 精度不足时按 rnd 选最接近 fp 值。

saturate=saturate: fp dst clamp [-max_finite, +max_finite]。satfinite=satfinite: clamp ±max_finite 不变 ±Inf。

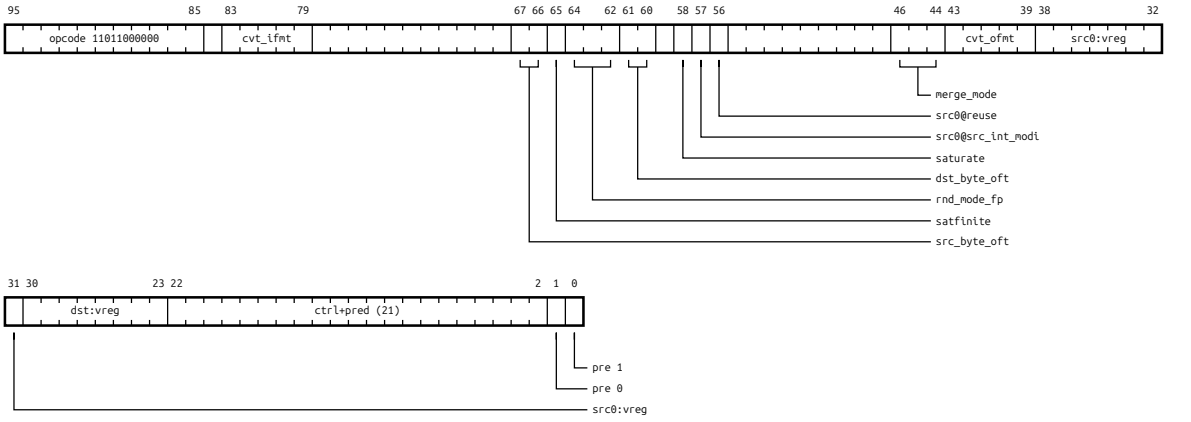
Leaf: i2fp__v_v

Format:

```
i2fp.cvt_ifmt.cvt_ofmt.rnd_mode_fp{.saturate}{.satfinite}{.src_byte_oft}{.dst_byte_oft}{」
↪  .merge_mode}  dst,
↪  src0{.src_int_modi}{.reuse}
```

Operands: dst: vreg; src0: vreg

Encoding Diagram: 96b, prefix 10, opcode 11011000000



Notes:

整数 → 浮点 packed 转换 (多 lane 写入): 跟 f2fp 同结构, 仅 src/dst 方向从 fp→fp 改 int→fp。整 warp per-lane 各自计算。

指令形态 / merge_mode 组合语义:

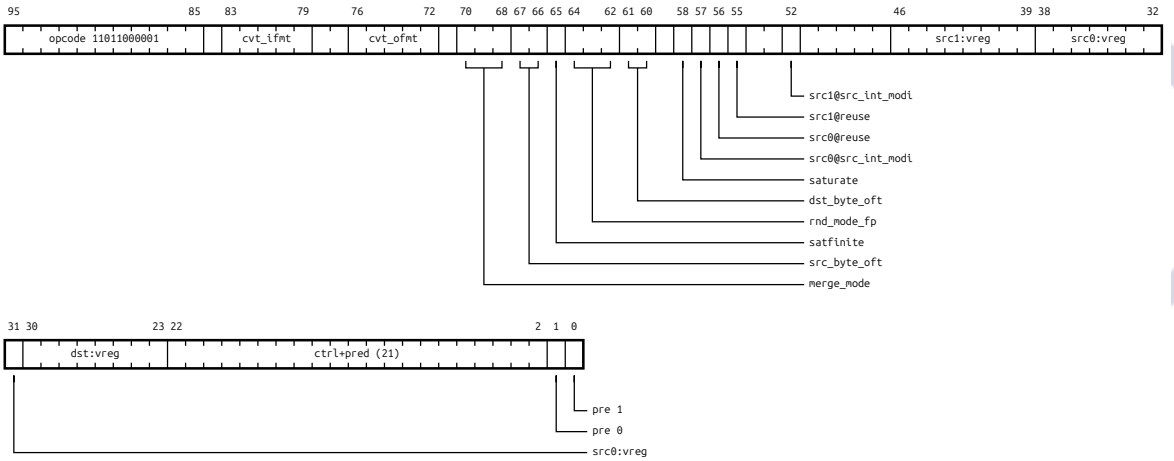
- v_v + unpack: 单 src 子词 int → fp unpack (典型 s8 byte → fp16/bf16, ML INT8 反量化主路径)。
- v_v + normal: 单 src int → fp pad-zero 覆盖整 dst。
- v_vv + normal (= PACK_AB): 双 src int → packed fp (dst.lo = convert(src0), dst.hi = convert(src1)) — INT16+INT16 → packed fp16/bf16 主路径。
- v_vvc + merge 系列: 跟 src2 (旧 dst) merge 部分写。

典型用例: INT8 → bf16/fp16 packed 反量化 (ML dequant); INT16 → fp16/bf16 packed widen。
rnd_mode_fp: 4 valid (rne / rtz / rup / rdn), int → fp 精度不足时按 rnd 选最接近 fp 值。
saturate=saturate: fp dst clamp [-max_finite, +max_finite]。satfinite=satfinite: clamp ±max_finite 不变 ±Inf。

Leaf: i2fp__v_vv
Format:

```
i2fp.cvt_ifmt.cvt_ofmt.rnd_mode_fp{.saturate}{.satfinite}{.src_byte_oft}{.dst_byte_oft}{」
↪  .merge_mode}  dst, src0{.src_int_modi}{.reuse},
↪  src1{.src_int_modi}{.reuse}
```

Operands: dst: vreg; src0: vreg; src1: vreg
Encoding Diagram: 96b, prefix 10, opcode 11011000001



Notes:
整数 → 浮点 packed 转换 (多 lane 写入): 跟 f2fp 同结构, 仅 src/dst 方向从 fp→fp 改 int→fp。整 warp per-lane 各自计算。
指令形态 / merge_mode 组合语义:

- v_v + unpack: 单 src 子词 int → fp unpack (典型 s8 byte → fp16/bf16, ML INT8 反量化主路径)。
- v_v + normal: 单 src int → fp pad-zero 覆盖整 dst。
- v_vv + normal (= PACK_AB): 双 src int → packed fp (dst.lo = convert(src0), dst.hi = convert(src1)) — INT16+INT16 → packed fp16/bf16 主路径。
- v_vvc + merge 系列: 跟 src2 (旧 dst) merge 部分写。

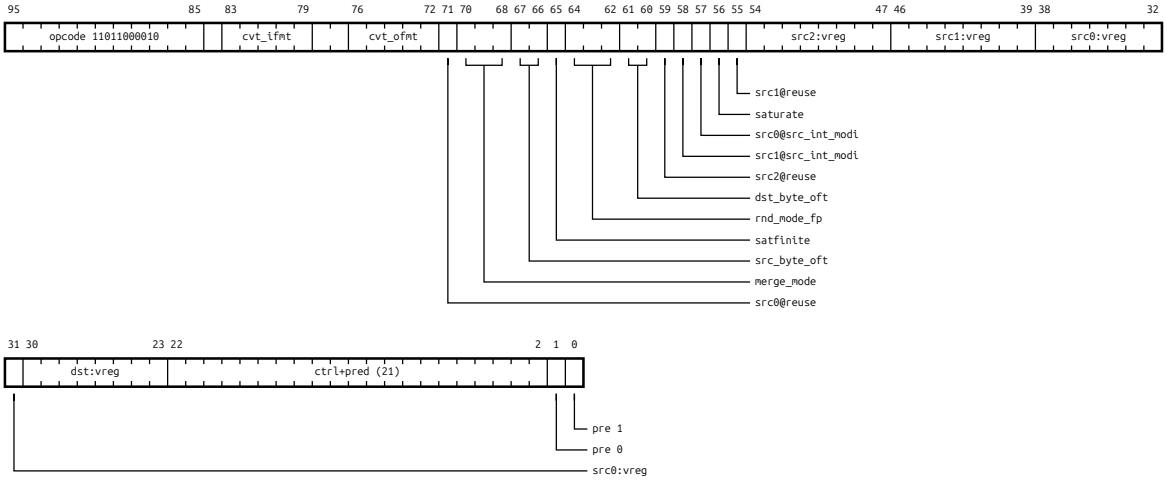
典型用例: INT8 → bf16/fp16 packed 反量化 (ML dequant); INT16 → fp16/bf16 packed widen。
rnd_mode_fp: 4 valid (rne / rtz / rup / rdn), int → fp 精度不足时按 rnd 选最接近 fp 值。
saturate=saturate: fp dst clamp [-max_finite, +max_finite]。satfinite=satfinite: clamp ±max_finite 不变 ±Inf。

Leaf: i2fp__v_vvc
Format:

```
i2fp.cvt_ifmt.cvt_ofmt.rnd_mode_fp{.saturate}{.satfinite}{.src_byte_oft}{.dst_byte_oft}{」
↪  .merge_mode}  dst, src0{.src_int_modi}{.reuse}, src1{.src_int_modi}{.reuse},
↪  src2{.reuse}
```

Operands: dst: vreg; src0: vreg; src1: vreg; src2: vreg

Encoding Diagram: 96b, prefix 10, opcode 11011000010



Notes:

整数 → 浮点 packed 转换 (多 lane 写入): 跟 f2fp 同结构, 仅 src/dst 方向从 fp→fp 改 int→fp。整 warp per-lane 各自计算。

指令形态 / merge_mode 组合语义:

- v_v + unpack: 单 src 子词 int → fp unpack (典型 s8 byte → fp16/bf16, ML INT8 反量化主路径)。
- v_v + normal: 单 src int → fp pad-zero 覆盖整 dst。
- v_vv + normal (= PACK_AB): 双 src int → packed fp (dst.lo = convert(src0), dst.hi = convert(src1)) — INT16+INT16 → packed fp16/bf16 主路径。
- v_vvc + merge 系列: 跟 src2 (旧 dst) merge 部分写。

典型用例: INT8 → bf16/fp16 packed 反量化 (ML dequant); INT16 → fp16/bf16 packed widen。

rnd_mode_fp: 4 valid (rne / rtz / rup / rdn), int → fp 精度不足时按 rnd 选最接近 fp 值。

saturate=saturate: fp dst clamp [-max_finite, +max_finite]。satfinite=satfinite: clamp ±max_finite 不变 ±Inf。

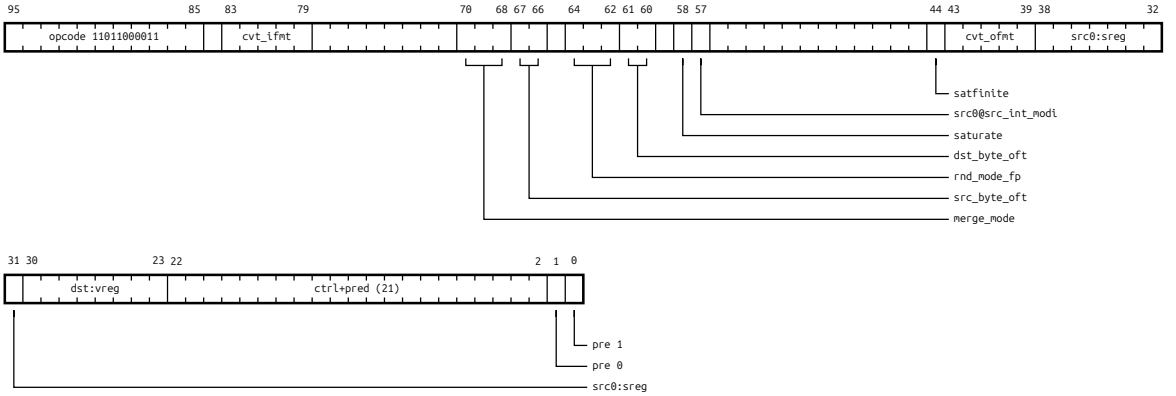
Leaf: i2fp__v_x

Format:

```
i2fp.cvt_ifmt.cvt_ofmt.rnd_mode_fp{.saturate}{.satfinite}{.src_byte_oft}{.dst_byte_oft}{」
↪  .merge_mode}  dst,
↪  src0{.src_int_modi}
```

Operands: dst: vreg; src0: sreg

Encoding Diagram: 96b, prefix 10, opcode 11011000011



Notes:

整数 → 浮点 packed 转换 (多 lane 写入): 跟 f2fp 同结构, 仅 src/dst 方向从 fp→fp 改 int→fp。整 warp per-lane 各自计算。

指令形态 / merge_mode 组合语义:

- v_v + unpack: 单 src 子词 int → fp unpack (典型 s8 byte → fp16/bf16, ML INT8 反量化主路径)。
- v_v + normal: 单 src int → fp pad-zero 覆盖整 dst。
- v_vv + normal (= PACK_AB): 双 src int → packed fp (dst.lo = convert(src0), dst.hi = convert(src1)) — INT16+INT16 → packed fp16/bf16 主路径。
- v_vvc + merge 系列: 跟 src2 (旧 dst) merge 部分写。

典型用例: INT8 → bf16/fp16 packed 反量化 (ML dequant); INT16 → fp16/bf16 packed widen。

rnd_mode_fp: 4 valid (rne / rtz / rup / rdn), int → fp 精度不足时按 rnd 选最接近 fp 值。

saturate=saturate: fp dst clamp [-max_finite, +max_finite]。satfinite=satfinite: clamp ±max_finite 不变 ±Inf。

i2i **category**: Conversion **predicate_mode**: simt

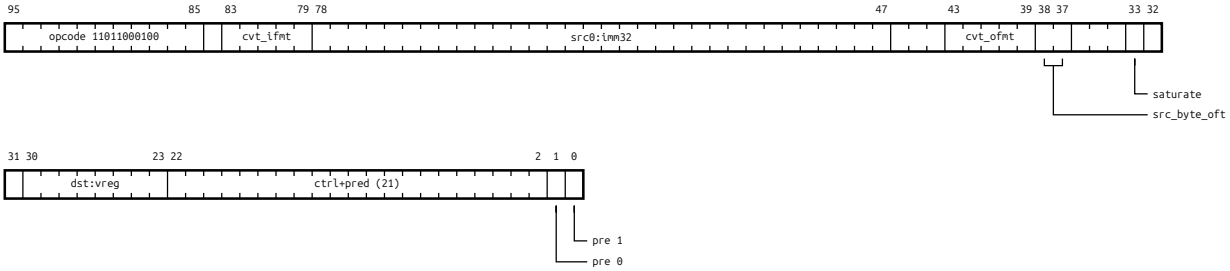
Leaf: i2i__v_i

Format:

```
i2i.cvt_ifmt.cvt_ofmt{.saturate}{.src_byte_oft}  dst, src0
```

Operands: dst: vreg; src0: imm32u

Encoding Diagram: 96b, prefix 10, opcode 11011000100



Notes:

整数 → 整数位宽转换: 把 src0 按 cvt_ifmt 解读的整数数据转成 cvt_ofmt 整数类型 (narrow 截断 / widen 符号或零扩展 / signed-unsigned 互转), 结果写 dst。整 warp per-lane 各自计算。

cvt_ifmt / cvt_ofmt: int 类型 (u8 / s8 / u16 / s16 / u32 / s32 / u64 / s64 + u4 / s4 / u2 / s2)。

三类语义: widen (如 s16 → s32) 按 cvt_ifmt 符号 / 零扩展; narrow (如 s32 → s16) 按 saturate 决定截断或饱和; 同位宽 signed/unsigned 互转 (如 u32 → s32) 二进制等价。

src 子词路径: src0 < 32-bit 用 src_byte_oft 选 byte 起始位置 (32/64-bit 忽略)。

saturate=saturate: narrow 转换时按 cvt_ofmt 边界饱和 (如 s32→s16 时 clamp 到 [INT16_MIN, INT16_-MAX]); 不开 saturate 则按位截断 (modulo 2^N)。

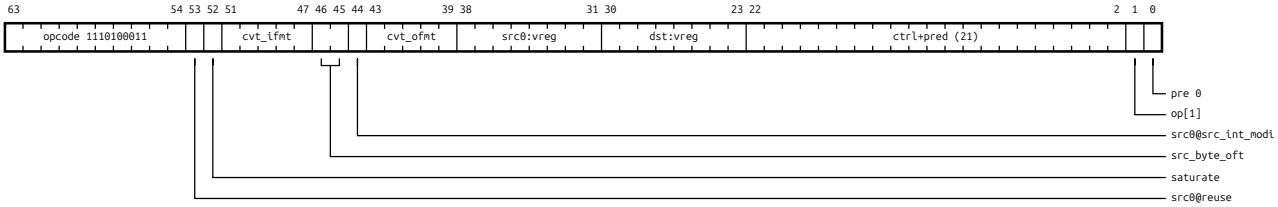
Leaf: i2i__v_v

Format:

```
i2i.cvt_ifmt.cvt_ofmt{.saturate}{.src_byte_oft}  dst, src0{.src_int_modi}{.reuse}
```

Operands: dst: vreg; src0: vreg

Encoding Diagram: 64b, prefix 0, opcode 11110100011



Notes:

整数 → 整数位宽转换: 把 src0 按 cvt_ifmt 解读的整数数据转成 cvt_ofmt 整数类型 (narrow 截断 / widen 符号或零扩展 / signed-unsigned 互转), 结果写 dst。整 warp per-lane 各自计算。

cvt_ifmt / cvt_ofmt: int 类型 (u8 / s8 / u16 / s16 / u32 / s32 / u64 / s64 + u4 / s4 / u2 / s2)。

三类语义: widen (如 s16 → s32) 按 cvt_ifmt 符号 / 零扩展; narrow (如 s32 → s16) 按 saturate 决定截断或饱和; 同位宽 signed/unsigned 互转 (如 u32 → s32) 二进制等价。

src 子词路径: src0 < 32-bit 用 src_byte_oft 选 byte 起始位置 (32/64-bit 忽略)。

saturate=saturate: narrow 转换时按 cvt_ofmt 边界饱和 (如 s32→s16 时 clamp 到 [INT16_MIN, INT16_-MAX]); 不开 saturate 则按位截断 (modulo 2^N)。

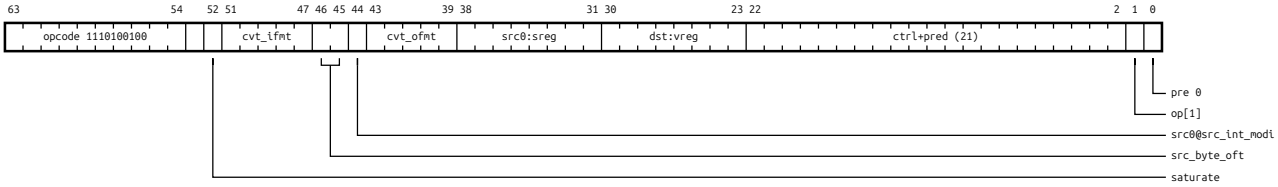
Leaf: i2i__v_x

Format:

```
i2i.cvt_ifmt.cvt_ofmt{.saturate}{.src_byte_oft}  dst, src0{.src_int_modi}
```

Operands: dst: vreg; src0: sreg

Encoding Diagram: 64b, prefix 0, opcode 11110100100



Notes:

整数 → 整数位宽转换: 把 src0 按 cvt_ifmt 解读的整数数据转成 cvt_ofmt 整数类型 (narrow 截断 / widen 符号或零扩展 / signed-unsigned 互转), 结果写 dst。整 warp per-lane 各自计算。

cvt_ifmt / cvt_ofmt: int 类型 (u8 / s8 / u16 / s16 / u32 / s32 / u64 / s64 + u4 / s4 / u2 / s2)。

三类语义: widen (如 s16 → s32) 按 cvt_ifmt 符号 / 零扩展; narrow (如 s32 → s16) 按 saturate 决定截断或饱和; 同位宽 signed/unsigned 互转 (如 u32 → s32) 二进制等价。

src 子词路径: src0 < 32-bit 用 src_byte_oft 选 byte 起始位置 (32/64-bit 忽略)。

saturate=saturate: narrow 转换时按 cvt_ofmt 边界饱和 (如 s32→s16 时 clamp 到 [INT16_MIN, INT16_-MAX]); 不开 saturate 则按位截断 (modulo 2^N)。

i2ip **category**: Conversion **predicate_mode**: simt

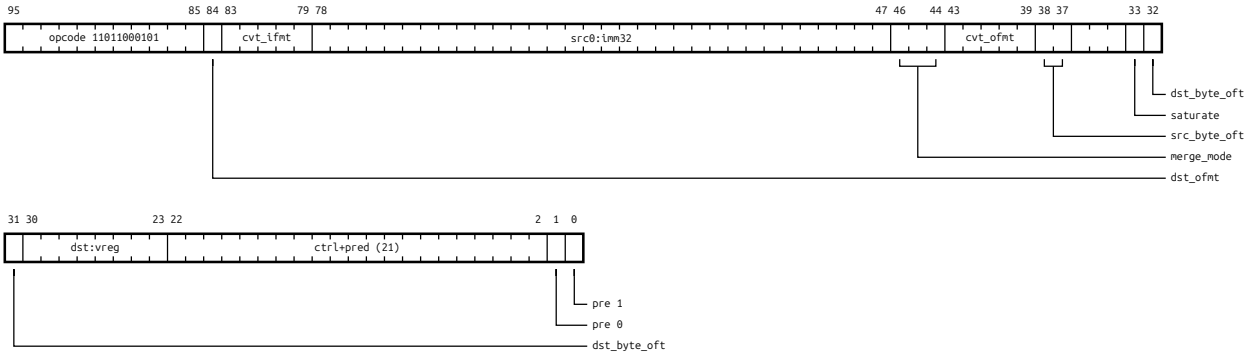
Leaf: i2ip__v_i

Format:

```
i2ip.cvt_ifmt.cvt_ofmt{.saturate}{.dst_ofmt}{.src_byte_oft}{.dst_byte_oft}{.merge_mode}
↪ dst, src0
```

Operands: dst: vreg; src0: imm32u

Encoding Diagram: 96b, prefix 10, opcode 11011000101



Notes:

整数 → 整数 packed 转换 (多 lane 写入): 跟 f2fp 同结构, 仅 src/dst 方向从 fp→fp 改 int→int (narrow / widen / signed-unsigned 互转 + packed 多 lane 写)。整 warp per-lane 各自计算。

指令形态 / merge_mode 组合语义:

- v_v + unpack: 单 src 子词 int → int unpack (典型 INT8 byte → INT32 widen)。
- v_v + normal: 单 src int pad-zero 覆盖整 dst。
- v_vv + normal (= PACK_AB): 双 src 32-bit int → packed int (dst.lo = convert(src0), dst.hi = convert(src1))。

v_vvc + merge 系列: 跟 src2 (旧 dst) merge 部分写。

典型用例: INT32 → INT8/INT16 packed pack (向量打包写 byte/half); INT8 packed → INT32 unpack; INT4/INT2 packed → INT8 widen。

dst_byte_oft: narrow dst 时选 byte 写入位置。

saturate=saturate: narrow 转换按 cvt_ofmt 边界饱和; 不开 saturate 则按位截断 (modulo 2^N)。

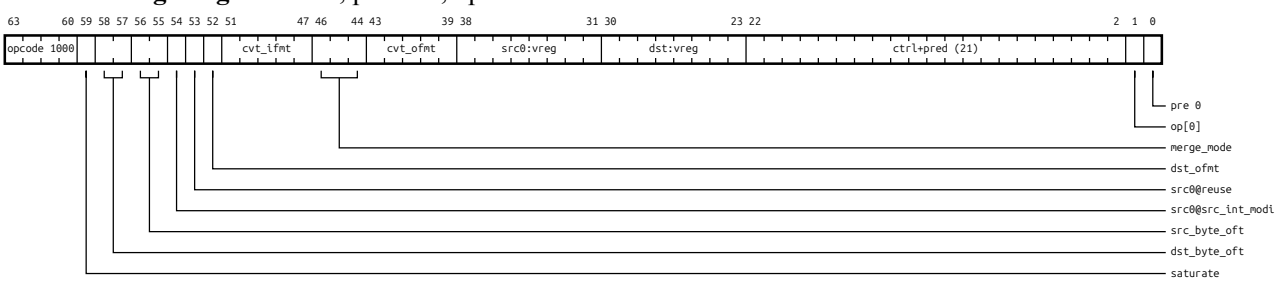
Leaf: i2ip__v_v

Format:

```
i2ip.cvt_ifmt.cvt_ofmt{.saturate}{.dst_ofmt}{.src_byte_oft}{.dst_byte_oft}{.merge_mode}
↪ dst, src0{.src_int_modi}{.reuse}
```

Operands: dst: vreg; src0: vreg

Encoding Diagram: 64b, prefix 0, opcode 01000



Notes:

整数 → 整数 packed 转换 (多 lane 写入): 跟 f2fp 同结构, 仅 src/dst 方向从 fp→fp 改 int→int (narrow / widen / signed-unsigned 互转 + packed 多 lane 写)。整 warp per-lane 各自计算。

指令形态 / merge_mode 组合语义:

- v_v + unpack: 单 src 子词 int → int unpack (典型 INT8 byte → INT32 widen)。
- v_v + normal: 单 src int pad-zero 覆盖整 dst。
- v_vv + normal (= PACK_AB): 双 src 32-bit int → packed int (dst.lo = convert(src0), dst.hi = convert(src1))。

v_vvc + merge 系列: 跟 src2 (旧 dst) merge 部分写。

典型用例: INT32 → INT8/INT16 packed pack (向量打包写 byte/half); INT8 packed → INT32 unpack; INT4/INT2 packed → INT8 widen。

dst_byte_oft: narrow dst 时选 byte 写入位置。

saturate=saturate: narrow 转换按 cvt_ofmt 边界饱和; 不开 saturate 则按位截断 (modulo 2^N)。

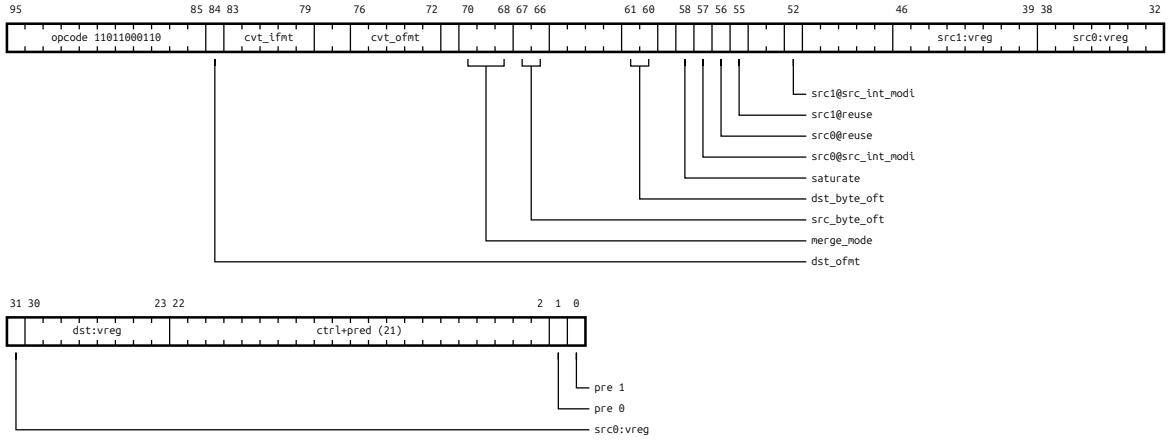
Leaf: i2ip__v_vv

Format:

```
i2ip.cvt_ifmt.cvt_ofmt{.saturate}{.dst_ofmt}{.src_byte_oft}{.dst_byte_oft}{.merge_mode}
↪  dst, src0{.src_int_modi}{.reuse}, src1{.src_int_modi}{.reuse}
```

Operands: dst: vreg; src0: vreg; src1: vreg

Encoding Diagram: 96b, prefix 10, opcode 11011000110



Notes:

整数 → 整数 packed 转换 (多 lane 写入): 跟 f2fp 同结构, 仅 src/dst 方向从 fp→fp 改 int→int (narrow / widen / signed-unsigned 互转 + packed 多 lane 写)。整 warp per-lane 各自计算。

指令形态 / merge_mode 组合语义:

- v_v + unpack: 单 src 子词 int → int unpack (典型 INT8 byte → INT32 widen)。
- v_v + normal: 单 src int pad-zero 覆盖整 dst。
- v_vv + normal (= PACK_AB): 双 src 32-bit int → packed int (dst.lo = convert(src0), dst.hi = convert(src1))。

v_vvc + merge 系列: 跟 src2 (旧 dst) merge 部分写。

典型用例: INT32 → INT8/INT16 packed pack (向量打包写 byte/half); INT8 packed → INT32 unpack; INT4/INT2 packed → INT8 widen。

dst_byte_oft: narrow dst 时选 byte 写入位置。

saturate=saturate: narrow 转换按 cvt_ofmt 边界饱和; 不开 saturate 则按位截断 (modulo 2^N)。

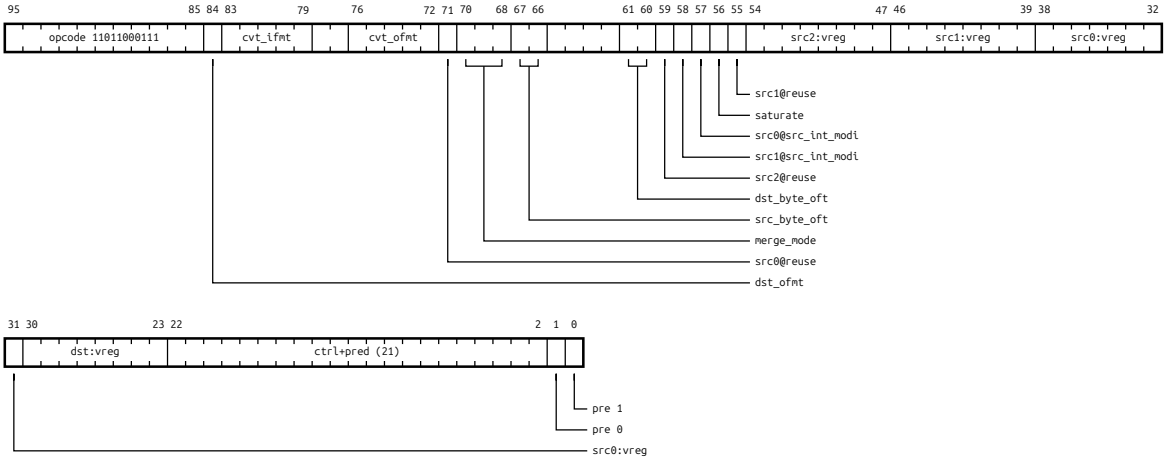
Leaf: i2ip__v_vvc

Format:

```
i2ip.cvt_ifmt.cvt_ofmt{.saturate}{.dst_ofmt}{.src_byte_oft}{.dst_byte_oft}{.merge_mode}
↪  dst, src0{.src_int_modi}{.reuse}, src1{.src_int_modi}{.reuse}, src2{.reuse}
```

Operands: dst: vreg; src0: vreg; src1: vreg; src2: vreg

Encoding Diagram: 96b, prefix 10, opcode 11011000111



Notes:

整数 → 整数 packed 转换 (多 lane 写入): 跟 f2fp 同结构, 仅 src/dst 方向从 fp→fp 改 int→int (narrow / widen / signed-unsigned 互转 + packed 多 lane 写)。整 warp per-lane 各自计算。

指令形态 / merge_mode 组合语义:

- v_v + unpack: 单 src 子词 int → int unpack (典型 INT8 byte → INT32 widen)。
- v_v + normal: 单 src int pad-zero 覆盖整 dst。
- v_vv + normal (= PACK_AB): 双 src 32-bit int → packed int (dst.lo = convert(src0), dst.hi = convert(src1))。

v_vvc + merge 系列: 跟 src2 (旧 dst) merge 部分写。

典型用例: INT32 → INT8/INT16 packed pack (向量打包写 byte/half); INT8 packed → INT32 unpack; INT4/INT2 packed → INT8 widen。

dst_byte_oft: narrow dst 时选 byte 写入位置。

saturate=saturate: narrow 转换按 cvt_ofmt 边界饱和; 不开 saturate 则按位截断 (modulo 2^N)。

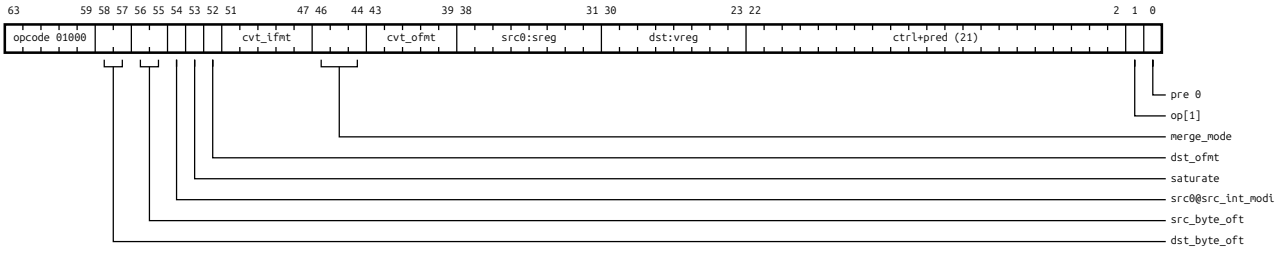
Leaf: i2ip__v_x

Format:

```
i2ip.cvt_ifmt.cvt_ofmt{.saturate}{.dst_ofmt}{.src_byte_oft}{.dst_byte_oft}{.merge_mode}
↪  dst, src0{.src_int_modi}
```

Operands: dst: vreg; src0: sreg

Encoding Diagram: 64b, prefix 0, opcode 101000



Notes:

整数 → 整数 packed 转换 (多 lane 写入): 跟 f2fp 同结构, 仅 src/dst 方向从 fp→fp 改 int→int (narrow / widen / signed-unsigned 互转 + packed 多 lane 写)。整 warp per-lane 各自计算。

指令形态 / merge_mode 组合语义:

- v_v + unpack: 单 src 子词 int → int unpack (典型 INT8 byte → INT32 widen)。
- v_v + normal: 单 src int pad-zero 覆盖整 dst。
- v_vv + normal (= PACK_AB): 双 src 32-bit int → packed int (dst.lo = convert(src0), dst.hi = convert(src1))。

v_vvc + merge 系列: 跟 src2 (旧 dst) merge 部分写。

典型用例: INT32 → INT8/INT16 packed pack (向量打包写 byte/half); INT8 packed → INT32 unpack; INT4/INT2 packed → INT8 widen。

dst_byte_oft: narrow dst 时选 byte 写入位置。

saturate=saturate: narrow 转换按 cvt_ofmt 边界饱和; 不开 saturate 则按位截断 (modulo 2^N)。

uf2fp **category**: Conversion **predicate_mode**: uniform

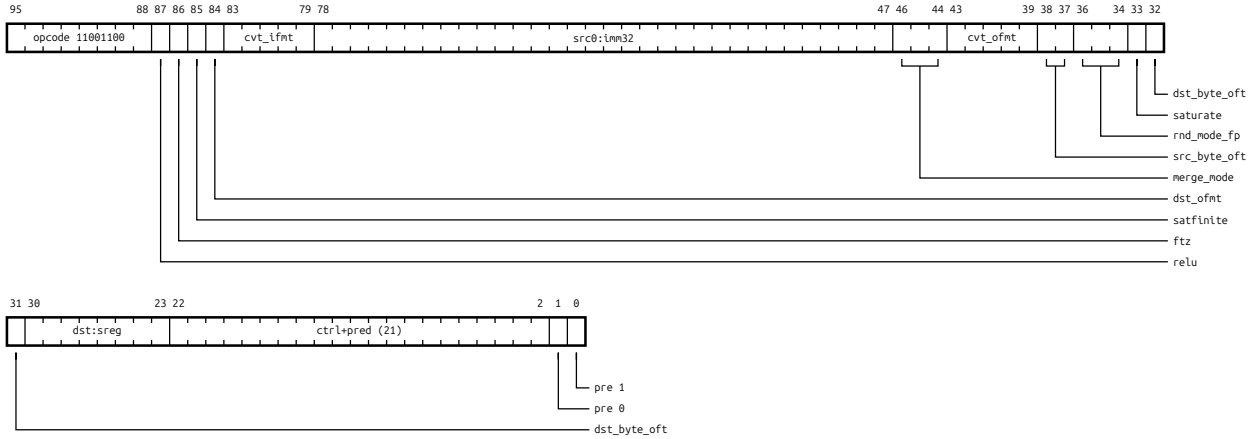
Leaf: uf2fp__x_i

Format:

```
uf2fp.cvt_ifmt.cvt_ofmt.rnd_mode_fp.ftz{.saturate}{.satfinite}{.relu}{.dst_ofmt}{.src_by
↪ te_oft}{.dst_byte_oft}{.merge_mode}  dst,
↪ src0
```

Operands: dst: sreg; src0: imm32u

Encoding Diagram: 96b, prefix 10, opcode 11001100



Notes:

浮点 → 浮点 packed 转换 uniform 变体: 完全跟 simt f2fp 同结构, 所有 src/dst 都是 sreg。整 warp 共享一份转换结果。

指令形态 / merge_mode 组合语义 (跟 f2fp 一一对应):

- x_x + unpack: 单 src 子词 unpack (典型 fp4/fp8 sreg → fp16 sreg)。
- x_x + normal / pack_hi: 单 src 写整 dst 或写 dst 高 16-bit。
- x_xx + normal (= PACK_AB): 双 src pack (典型 fp32+fp32 sreg → packed fp16/bf16 sreg)。
- x_xxc + merge 系列: 跟 src2 (旧 dst) merge 部分写。
- x_xxs: 3 src, src2 = scale (rnd=rne/rtz/rup/rdn) 或 stochastic seed (rnd=rs), 跟 simt v_vvs 同语义。

uniform cvt 路径仅暴露 uf2fp 一条 (含 merge_c); 其他 uniform cvt (uf2f / uf2i / ui2f / ui2i) 编译器走 simt

cvt + r2ur 序列代偿。

rnd_mode_fp / ftz / saturate / satfinite / relu: 跟 simt f2fp 一致。

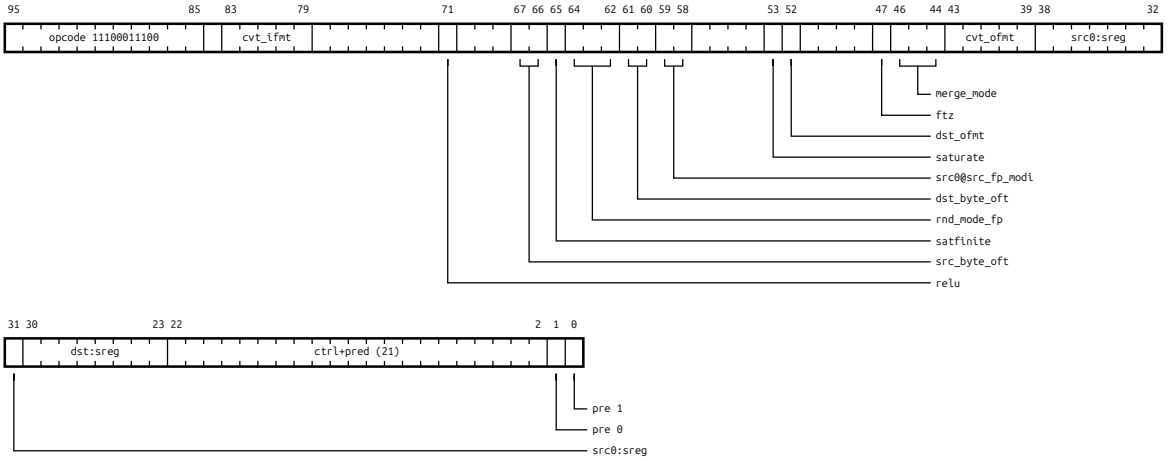
Leaf: uf2fp__x_x

Format:

```
uf2fp.cvt_ifmt.cvt_ofmt.rnd_mode_fp.ftz{.saturate}{.satfinite}{.relu}{.dst_ofmt}{.src_by」
↪  te_oft}{.dst_byte_oft}{.merge_mode}  dst,
↪  src0{.src_fp_modi}
```

Operands: dst: sreg; src0: sreg

Encoding Diagram: 96b, prefix 10, opcode 11100011100



Notes:

浮点 → 浮点 packed 转换 uniform 变体: 完全跟 simt f2fp 同结构, 所有 src/dst 都是 sreg。整 warp 共享一份转换结果。

指令形态 / merge_mode 组合语义 (跟 f2fp 一一对应):

- x_x + unpack: 单 src 子词 unpack (典型 fp4/fp8 sreg → fp16 sreg)。
- x_x + normal / pack_hi: 单 src 写整 dst 或写 dst 高 16-bit。
- x_xx + normal (= PACK_AB): 双 src pack (典型 fp32+fp32 sreg → packed fp16/bf16 sreg)。
- x_xxc + merge 系列: 跟 src2 (旧 dst) merge 部分写。
- x_xxs: 3 src, src2 = scale (rnd=rne/rtz/rup/rdn) 或 stochastic seed (rnd=rs), 跟 simt v_vvs 同语义。

uniform cvt 路径仅暴露 uf2fp 一条 (含 merge_c); 其他 uniform cvt (uf2f / uf2i / ui2f / ui2i) 编译器走 simt cvt + r2ur 序列代偿。

rnd_mode_fp / ftz / saturate / satfinite / relu: 跟 simt f2fp 一致。

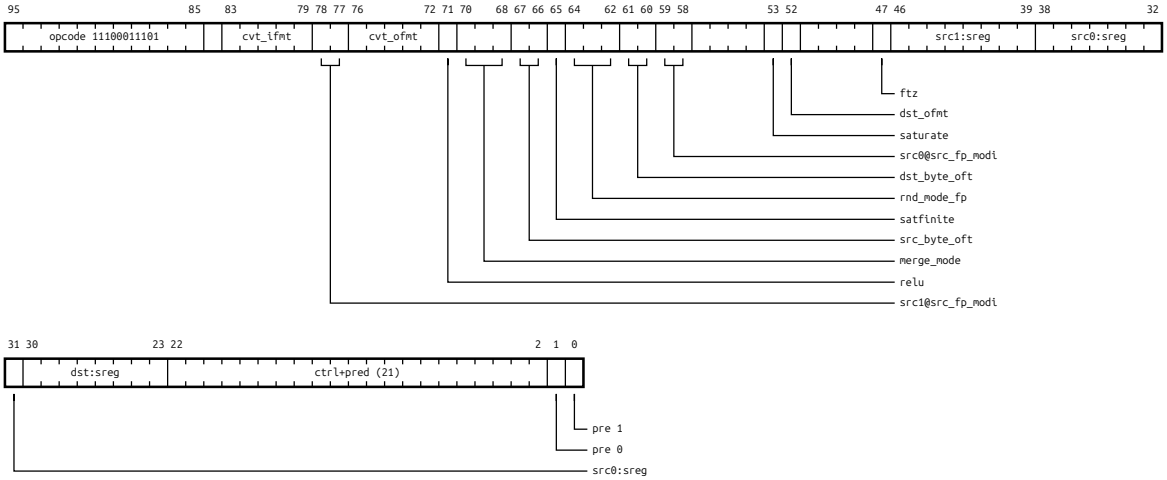
Leaf: uf2fp__x_xx

Format:

```
uf2fp.cvt_ifmt.cvt_ofmt.rnd_mode_fp.ftz{.saturate}{.satfinite}{.relu}{.dst_ofmt}{.src_by」
↪  te_oft}{.dst_byte_oft}{.merge_mode}  dst, src0{.src_fp_modi},
↪  src1{.src_fp_modi}
```

Operands: dst: sreg; src0: sreg; src1: sreg

Encoding Diagram: 96b, prefix 10, opcode 11100011101



Notes:

浮点 → 浮点 packed 转换 uniform 变体: 完全跟 simt f2fp 同结构, 所有 src/dst 都是 sreg。整 warp 共享一份转换结果。

指令形态 / merge_mode 组合语义 (跟 f2fp 一一对应):

- x_x + unpack: 单 src 子词 unpack (典型 fp4/fp8 sreg → fp16 sreg)。
- x_x + normal / pack_hi: 单 src 写整 dst 或写 dst 高 16-bit。
- x_xx + normal (= PACK_AB): 双 src pack (典型 fp32+fp32 sreg → packed fp16/bf16 sreg)。
- x_xxc + merge 系列: 跟 src2 (旧 dst) merge 部分写。
- x_xxs: 3 src, src2 = scale (rnd=rne/rtz/rup/rdn) 或 stochastic seed (rnd=rs), 跟 simt v_vvs 同语义。

uniform cvt 路径仅暴露 uf2fp 一条 (含 merge_c); 其他 uniform cvt (uf2f / uf2i / ui2f / ui2i) 编译器走 simt cvt + r2ur 序列代偿。

rnd_mode_fp / ftz / saturate / satfinite / relu: 跟 simt f2fp 一致。

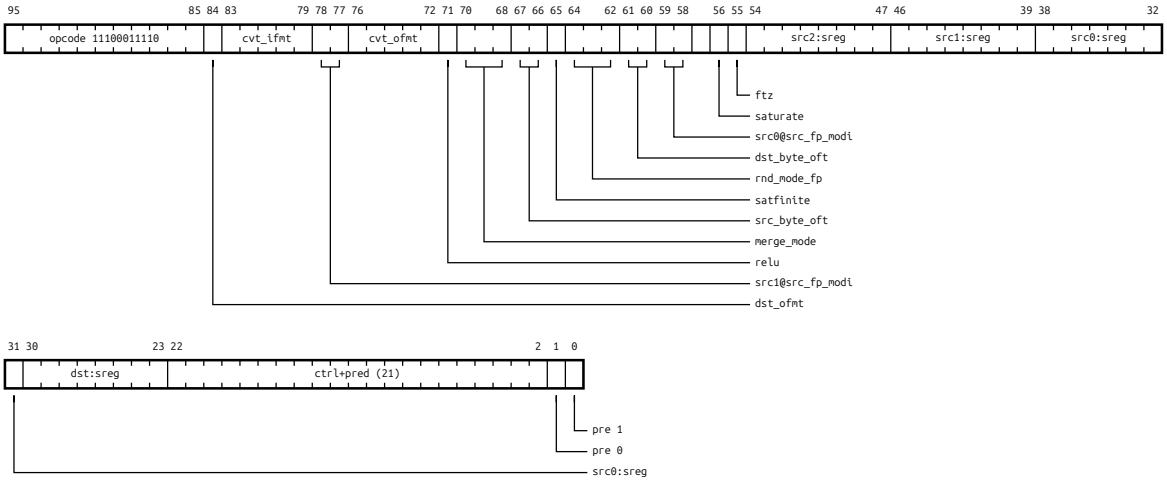
Leaf: uf2fp__x_xxc

Format:

```
uf2fp.cvt_ifmt.cvt_ofmt.rnd_mode_fp.ftz{.saturate}{.satfinite}{.relu}{.dst_ofmt}{.src_by」
↪  te_oft}{.dst_byte_oft}{.merge_mode}  dst, src0{.src_fp_modi}, src1{.src_fp_modi},
↪  src2
```

Operands: dst: sreg; src0: sreg; src1: sreg; src2: sreg

Encoding Diagram: 96b, prefix 10, opcode 11100011110



Notes:

浮点 → 浮点 packed 转换 uniform 变体: 完全跟 simt f2fp 同结构, 所有 src/dst 都是 sreg。整 warp 共享一份转换结果。

指令形态 / merge_mode 组合语义 (跟 f2fp 一一对应):

- x_x + unpack: 单 src 子词 unpack (典型 fp4/fp8 sreg → fp16 sreg)。
- x_x + normal / pack_hi: 单 src 写整 dst 或写 dst 高 16-bit。
- x_xx + normal (= PACK_AB): 双 src pack (典型 fp32+fp32 sreg → packed fp16/bf16 sreg)。
- x_xxc + merge 系列: 跟 src2 (旧 dst) merge 部分写。
- x_xxs: 3 src, src2 = scale (rnd=rne/rtz/rup/rdn) 或 stochastic seed (rnd=rs), 跟 simt v_vvs 同语义。

uniform cvt 路径仅暴露 uf2fp 一条 (含 merge_c); 其他 uniform cvt (uf2f / uf2i / ui2f / ui2i) 编译器走 simt cvt + r2ur 序列代偿。

rnd_mode_fp / ftz / saturate / satfinite / relu: 跟 simt f2fp 一致。

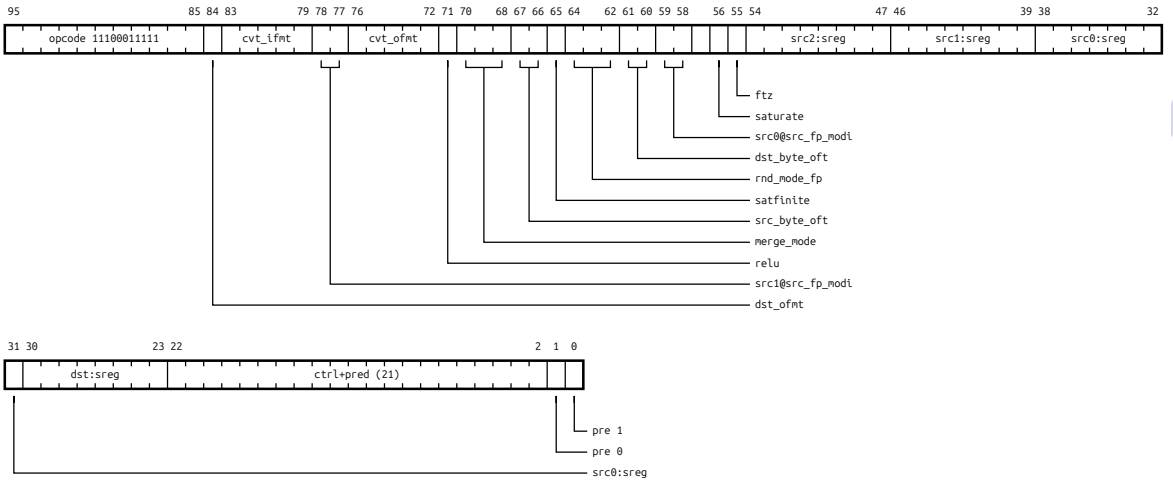
Leaf: uf2fp__x_xxs

Format:

```
uf2fp.cvt_ifmt.cvt_ofmt.rnd_mode_fp.ftz{.saturate}{.satfinite}{.relu}{.dst_ofmt}{.src_by
↪  te_oft}{.dst_byte_oft}{.merge_mode}  dst, src0{.src_fp_modi}, src1{.src_fp_modi},
↪  src2
```

Operands: dst: sreg; src0: sreg; src1: sreg; src2: sreg

Encoding Diagram: 96b, prefix 10, opcode 11100011111



Notes:

浮点 → 浮点 packed 转换 uniform 变体: 完全跟 simt f2fp 同结构, 所有 src/dst 都是 sreg。整 warp 共享一份转换结果。

指令形态 / merge_mode 组合语义 (跟 f2fp 一一对应):

- x_x + unpack: 单 src 子词 unpack (典型 fp4/fp8 sreg → fp16 sreg)。
- x_x + normal / pack_hi: 单 src 写整 dst 或写 dst 高 16-bit。
- x_xx + normal (= PACK_AB): 双 src pack (典型 fp32+fp32 sreg → packed fp16/bf16 sreg)。
- x_xxc + merge 系列: 跟 src2 (旧 dst) merge 部分写。
- x_xxs: 3 src, src2 = scale (rnd=rne/rtz/rup/rdn) 或 stochastic seed (rnd=rs), 跟 simt v_vvs 同语义。

uniform cvt 路径仅暴露 uf2fp 一条 (含 merge_c); 其他 uniform cvt (uf2f / uf2i / ui2f / ui2i) 编译器走 simt cvt + r2ur 序列代偿。

rnd_mode_fp / ftz / saturate / satfinite / relu: 跟 simt f2fp 一致。

14.7.7 Data move and selection

mov **category**: Data move and selection **predicate_mode**: simt

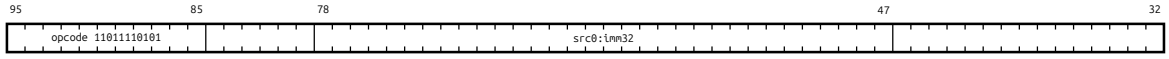
Leaf: mov__v_i

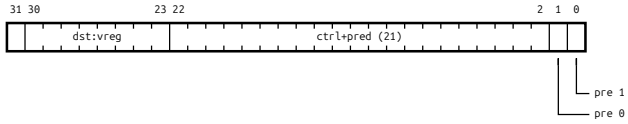
Format:

```
mov  dst, src0
```

Operands: dst: vreg; src0: imm32u

Encoding Diagram: 96b, prefix 10, opcode 11011110101





Notes:

32-bit 数据复制: dst = src0, 整 32-bit 原样复制 (不做类型转换 / 位运算)。最简单的数据移动指令, 编译器寄存器分配 + spill/reload 主路径。整 warp per-lane 各自计算。

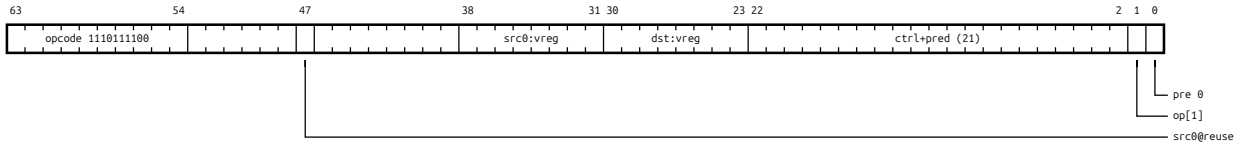
Leaf: mov__v_v

Format:

```
mov  dst, src0{.reuse}
```

Operands: dst: vreg; src0: vreg

Encoding Diagram: 64b, prefix 0, opcode 11110111100



Notes:

32-bit 数据复制: dst = src0, 整 32-bit 原样复制 (不做类型转换 / 位运算)。最简单的数据移动指令, 编译器寄存器分配 + spill/reload 主路径。整 warp per-lane 各自计算。

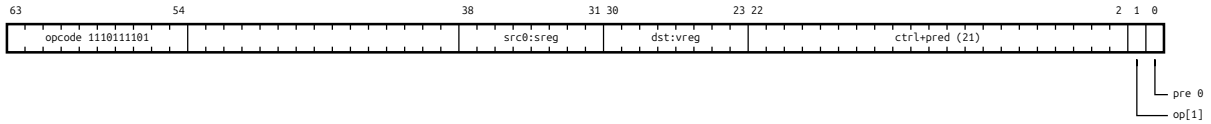
Leaf: mov__v_x

Format:

```
mov  dst, src0
```

Operands: dst: vreg; src0: sreg

Encoding Diagram: 64b, prefix 0, opcode 11110111101



Notes:

32-bit 数据复制: dst = src0, 整 32-bit 原样复制 (不做类型转换 / 位运算)。最简单的数据移动指令, 编译器寄存器分配 + spill/reload 主路径。整 warp per-lane 各自计算。

p2r **category**: Data move and selection **predicate_mode**: simt

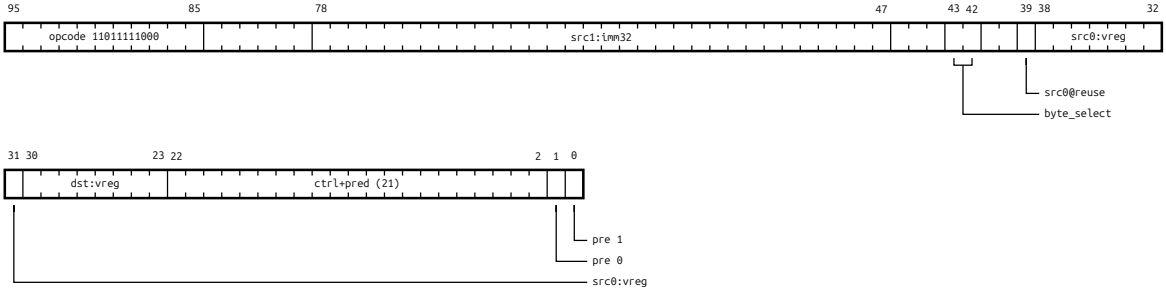
Leaf: p2r__v_vi

Format:

```
p2r{.byte_select}  dst, src0{.reuse}, src1
```

Operands: dst: vreg; src0: vreg; src1: imm32u

Encoding Diagram: 96b, prefix 10, opcode 11011111000



Notes:

把当前 thread 自己的 7 个 predicate (P0..P6) 有选择地搬到 r 寄存器: dst = src0 但 byte_select 那个 byte 里逐 bit 替换 — SbMask[j]=1 取 PR[j], SbMask[j]=0 保留 src0 该 byte 的 bit j。整 warp per-lane 严格独立 (每 thread 用自己的 PR, 不是 warp ballot)。

operand 对应: asm “P2R{.insert} Rd, PR, Ra, SbMask” → spec dst = Rd, src0 = Ra (原值寄存器), src1 = SbMask (8-bit 选择器)。PR 是 architectural state (整个 predicate file 当 8-bit 整体读), 不占编码 bit、不放 operands 列表; 汇编渲染时仍输出 ‘PR’ literal。

PR 8-bit 布局: [reserved, P6, P5, P4, P3, P2, P1, P0]。每 thread 一份。

byte_select=b0 / b1 / b2 / b3: 选 dst / src0 的哪个 byte 是合并目标。

伪代码 (per-lane): byte_b = (src0 » (8 * byte_select)) & 0xFF; merged = 0; for j in 0..7: merged |= (SbMask[j] ? PR[j] : byte_b[j]) « j; dst = (src0 & ~(0xFF « (8 * byte_select))) | (merged « (8 * byte_select))。

例子: P2R.b0 R5, PR, RZ, 0xFF 把 PR 完整搬到 R5 低 8 位 (高 24 位 = 0); P2R.b0 R0, PR, R0, 0x01 只把 P0 写 R0[0] 其它位不变; P2R.b0 R0, PR, R5, 0x05 用 SbMask=0x05 选 P0/P2 替换 R5 对应 bit, 其它位取 R5。

典型用途: 编译器后端做 predicate state 的 spill — 谓词寄存器不够用时把 7 个 P 暂存 r 里, 配合 r2p 装回。应用代码不会直接生成。

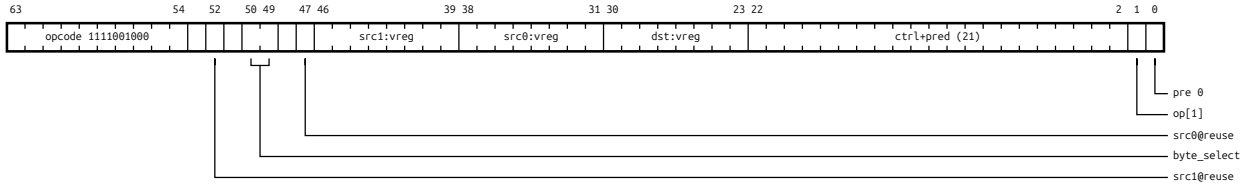
Leaf: p2r__v_vv

Format:

```
p2r{.byte_select}  dst, src0{.reuse}, src1{.reuse}
```

Operands: dst: vreg; src0: vreg; src1: vreg

Encoding Diagram: 64b, prefix 0, opcode 11111001000



Notes:

把当前 thread 自己的 7 个 predicate (P0..P6) 有选择地搬到 r 寄存器: dst = src0 但 byte_select 那个 byte 里逐 bit 替换 — SbMask[j]=1 取 PR[j], SbMask[j]=0 保留 src0 该 byte 的 bit j。整 warp per-lane 严格独立 (每 thread 用自己的 PR, 不是 warp ballot)。

operand 对应: asm “P2R{.insert} Rd, PR, Ra, SbMask” → spec dst = Rd, src0 = Ra (原值寄存器), src1 = SbMask (8-bit 选择器)。PR 是 architectural state (整个 predicate file 当 8-bit 整体读), 不占编码 bit、不放 operands 列表; 汇编渲染时仍输出 ‘PR’ literal。

PR 8-bit 布局: [reserved, P6, P5, P4, P3, P2, P1, P0]。每 thread 一份。

byte_select=b0 / b1 / b2 / b3: 选 dst / src0 的哪个 byte 是合并目标。

伪代码 (per-lane): byte_b = (src0 » (8 * byte_select)) & 0xFF; merged = 0; for j in 0..7: merged |= (SbMask[j] ? PR[j] : byte_b[j]) « j; dst = (src0 & ~(0xFF « (8 * byte_select))) | (merged « (8 * byte_select))。

例子: P2R.b0 R5, PR, RZ, 0xFF 把 PR 完整搬到 R5 低 8 位 (高 24 位 = 0); P2R.b0 R0, PR, R0, 0x01 只把 P0 写 R0[0] 其它位不变; P2R.b0 R0, PR, R5, 0x05 用 SbMask=0x05 选 P0/P2 替换 R5 对应 bit, 其它位取 R5。

典型用途: 编译器后端做 predicate state 的 spill — 谓词寄存器不够用时把 7 个 P 暂存 r 里, 配合 r2p 装回。应用代码不会直接生成。

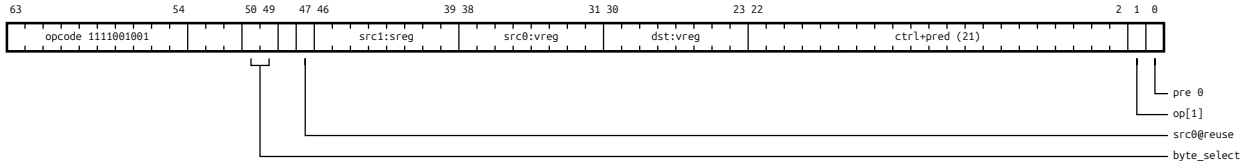
Leaf: p2r__v_vx

Format:

```
p2r{.byte_select}  dst, src0{.reuse}, src1
```

Operands: dst: vreg; src0: vreg; src1: sreg

Encoding Diagram: 64b, prefix 0, opcode 11111001001



Notes:

把当前 thread 自己的 7 个 predicate (P0..P6) 有选择地搬到 r 寄存器: dst = src0 但 byte_select 那个 byte 里逐 bit 替换 — SbMask[j]=1 取 PR[j], SbMask[j]=0 保留 src0 该 byte 的 bit j。整 warp per-lane 严格独立 (每 thread 用自己的 PR, 不是 warp ballot)。

operand 对应: asm “P2R{.insert} Rd, PR, Ra, SbMask” → spec dst = Rd, src0 = Ra (原值寄存器), src1 = SbMask (8-bit 选择器)。PR 是 architectural state (整个 predicate file 当 8-bit 整体读), 不占编码 bit、不放 operands 列表; 汇编渲染时仍输出 ‘PR’ literal。

PR 8-bit 布局: [reserved, P6, P5, P4, P3, P2, P1, P0]。每 thread 一份。

byte_select=b0 / b1 / b2 / b3: 选 dst / src0 的哪个 byte 是合并目标。

伪代码 (per-lane): byte_b = (src0 » (8 * byte_select)) & 0xFF; merged = 0; for j in 0..7: merged |= (SbMask[j] ? PR[j] : byte_b[j]) « j; dst = (src0 & ~(0xFF « (8 * byte_select))) | (merged « (8 * byte_select))。

例子: P2R.b0 R5, PR, RZ, 0xFF 把 PR 完整搬到 R5 低 8 位 (高 24 位 = 0); P2R.b0 R0, PR, R0, 0x01 只把 P0 写 R0[0] 其它位不变; P2R.b0 R0, PR, R5, 0x05 用 SbMask=0x05 选 P0/P2 替换 R5 对应 bit, 其它位取 R5。

典型用途: 编译器后端做 predicate state 的 spill — 谓词寄存器不够用时把 7 个 P 暂存 r 里, 配合 r2p 装回。应用代码不会直接生成。

r2p **category**: Data move and selection **predicate_mode**: simt

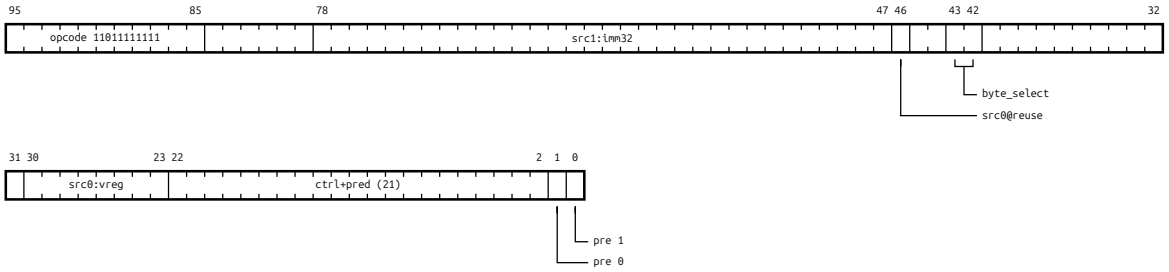
Leaf: r2p___vi

Format:

```
r2p{.byte_select}  src0{.reuse}, src1
```

Operands: src0: vreg; src1: imm32u

Encoding Diagram: 96b, prefix 10, opcode 11011111111



Notes:

从 r 寄存器某个 byte 取 8 bit, 按 mask 选择性写回当前 thread 的 7 个 predicate: 对每 bit j, SbMask[j]=1 → PR[j] = src0 的 byte_select byte 的 bit j; SbMask[j]=0 → PR[j] 保持原值。是 p2r 的反向, 整 warp per-lane 各自独立 (每 thread 写自己的 PR)。

operand 对应: asm “R2P PR, Ra{.b0/.b1/.b2/.b3}, SbMask” → spec src0 = Ra (输入 r 寄存器), src1 = SbMask (8-bit 选择器, vreg / sreg / imm)。PR 是 architectural state (整个 predicate file 当 8-bit 整体写), 不占编码 bit、不放 operands 列表。

byte_select=b0 / b1 / b2 / b3: 选 src0 的哪个 byte 作为输入。

伪代码 (per-lane): byte_b = (src0 » (8 * byte_select)) & 0xFF; for j in 0..7: if SbMask[j]: PR[j] = byte_b[j] (否则 PR[j] 不变)。

例子: R2P PR, R5.b0, 0xFF 把 R5 低 8 位完整写 7 个 P (+reserved bit); R2P PR, R5.b0, 0x01 只用 R5[0] 覆盖 P0 其它 P 不变; R2P PR, R5.b2, 0x7F 用 R5[22:16] (低 7 位) 覆盖 P0..P6。

典型用途: 跟 p2r 配对做 predicate state 的 reload (谓词寄存器分配压力时的 spill/reload 主路径)。

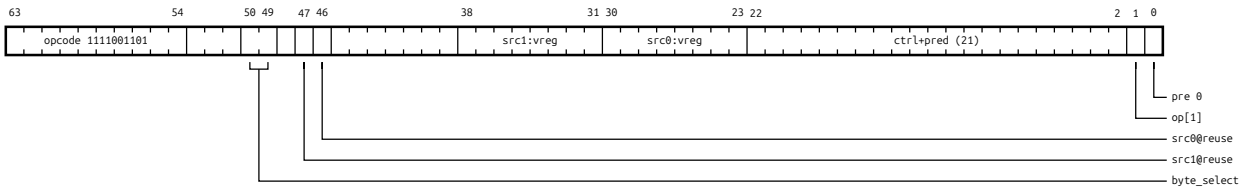
Leaf: r2p___vv

Format:

```
r2p{.byte_select}  src0{.reuse}, src1{.reuse}
```

Operands: src0: vreg; src1: vreg

Encoding Diagram: 64b, prefix 0, opcode 11111001101



Notes:

从 r 寄存器某个 byte 取 8 bit, 按 mask 选择性写回当前 thread 的 7 个 predicate: 对每 bit j, SbMask[j]=1 → PR[j] = src0 的 byte_select byte 的 bit j; SbMask[j]=0 → PR[j] 保持原值。是 p2r 的反向, 整 warp per-lane 各自独立 (每 thread 写自己的 PR)。

operand 对应: asm “R2P PR, Ra{.b0/.b1/.b2/.b3}, SbMask” → spec src0 = Ra (输入 r 寄存器), src1 = SbMask (8-bit 选择器, vreg / sreg / imm)。PR 是 architectural state (整个 predicate file 当 8-bit 整体写), 不占编码 bit、不放 operands 列表。

byte_select=b0 / b1 / b2 / b3: 选 src0 的哪个 byte 作为输入。

伪代码 (per-lane): byte_b = (src0 » (8 * byte_select)) & 0xFF; for j in 0..7: if SbMask[j]: PR[j] = byte_b[j] (否则 PR[j] 不变)。

例子: R2P PR, R5.b0, 0xFF 把 R5 低 8 位完整写 7 个 P (+reserved bit); R2P PR, R5.b0, 0x01 只用 R5[0] 覆盖 P0 其它 P 不变; R2P PR, R5.b2, 0x7F 用 R5[22:16] (低 7 位) 覆盖 P0..P6。

典型用途: 跟 p2r 配对做 predicate state 的 reload (谓词寄存器分配压力时的 spill/reload 主路径)。

Leaf: r2p___vx

Format:

```
r2p{.byte_select}  src0{.reuse}, src1
```

Operands: src0: vreg; src1: sreg

Encoding Diagram: 64b, prefix 0, opcode 11111001110

Notes:

从 r 寄存器某个 byte 取 8 bit, 按 mask 选择性写回当前 thread 的 7 个 predicate: 对每 bit j, SbMask[j]=1 → PR[j] = src0 的 byte_select byte 的 bit j; SbMask[j]=0 → PR[j] 保持原值。是 p2r 的反向, 整 warp per-lane 各自独立 (每 thread 写自己的 PR)。

operand 对应: asm “R2P PR, Ra{.b0/.b1/.b2/.b3}, SbMask” → spec src0 = Ra (输入 r 寄存器), src1 = SbMask (8-bit 选择器, vreg / sreg / imm)。PR 是 architectural state (整个 predicate file 当 8-bit 整体写), 不占编码 bit、不放 operands 列表。

byte_select=b0 / b1 / b2 / b3: 选 src0 的哪个 byte 作为输入。

伪代码 (per-lane): byte_b = (src0 » (8 * byte_select)) & 0xFF; for j in 0..7: if SbMask[j]: PR[j] = byte_b[j] (否则 PR[j] 不变)。

例子: R2P PR, R5.b0, 0xFF 把 R5 低 8 位完整写 7 个 P (+reserved bit); R2P PR, R5.b0, 0x01 只用 R5[0] 覆盖 P0 其它 P 不变; R2P PR, R5.b2, 0x7F 用 R5[22:16] (低 7 位) 覆盖 P0..P6。

典型用途: 跟 p2r 配对做 predicate state 的 reload (谓词寄存器分配压力时的 spill/reload 主路径)。

r2ur **category**: Data move and selection **predicate_mode**: simt

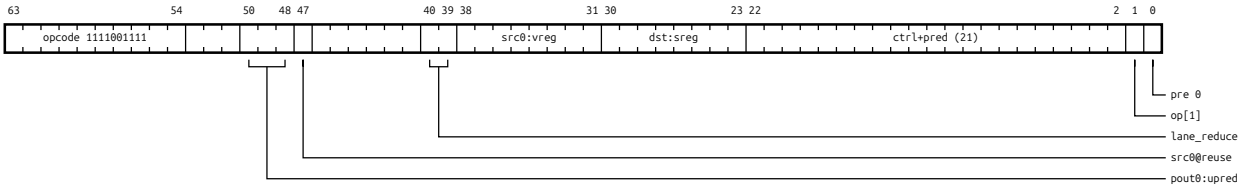
Leaf: r2ur__xp_v

Format:

```
r2ur{.lane_reduce}  pout0, dst, src0{.reuse}
```

Operands: pout0: upred; dst: sreg; src0: vreg

Encoding Diagram: 64b, prefix 0, opcode 11111001111



Notes:

Per-lane vreg → warp-uniform sreg reduction: 把 per-lane vreg 缩到 warp 共享的 sreg, 同时输出 'all active lanes equal' uniformity check predicate。核心问题 — 32 active lane 32 个值怎么缩? lane_reduce modifier 4 valid 收敛 4 种策略。pout0 永远输出 (uniformity 状态), 让编译器 / runtime 检测一致性。

lane_reduce 4 valid: broadcast (default, 取 lane0 给 dst, assume 一致, 编译器 'trust me' 路径; 若不一致 dst 静默拿 lane0 = silent truncate) / fill (一致时 broadcast lane0, 不一致时 dst=0, safety-critical 推荐, 避免悄悄拿 lane0 错值) / or (所有 active lane 的 vreg 按位 or reduce 到 dst, 主用例 32 lane mask 字段聚合) / or_fill (or reduce + 不一致 fallback fill, 收敛 or + fill 组合到单 modifier)。

operand 模型: pout0 = upred (uniformity check, 永远输出); dst = sreg (缩并结果); src0 = vreg (per-lane 数据)。predicate_mode=simt (源 vreg 是 per-lane 数据 + 显式读 EXEC mask 决定 active lane)。

典型用例: (1) idx 收窄 — ldg.v_vi R5, …; r2ur.broadcast pout0, UR0, R5; @!pout0 bra divergent_path (编译器假设 idx uniform 收到 ureg, runtime 验证); (2) lane mask 聚合 — vote.any P0, P1, …; r2p R10, P0; r2ur.or pout0, UR0, R10 (or 收所有 lane vote 结果到 ureg); (3) safety — r2ur.fill pout0, UR0, R5 (若 lane 不一致 dst=0 让后续操作分支安全)。

s2r **category**: Data move and selection **predicate_mode**: simt

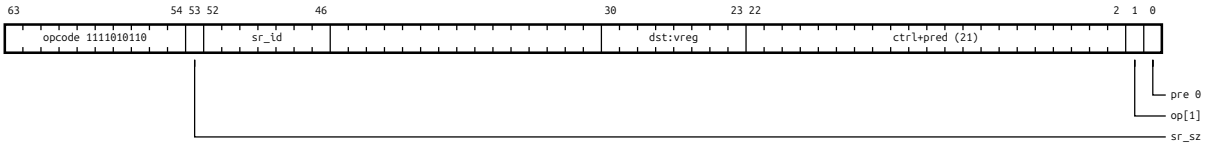
Leaf: s2r__v_

Format:

```
s2r.sr_id{.sr_sz}  dst
```

Operands: dst: vreg [notes: span=default=1; sr_sz=b64→2, align=default=1; sr_sz=b64→2]

Encoding Diagram: 64b, prefix 0, opcode 11111010110



Notes:

Special register read (per-lane): dst = SpecialRegister[sr_id], 整 warp 32 个 active lane 各自读取 SR 写到 per-lane vreg。SR 包括 laneid / TID / CTAID / CLOCK / smid / GLOBALTIMER 等 44 个硬件状态值。

sr_id (modifier): 选择目标 SR, 共 44 valid (laneid / tid_x / tid_y / tid_z / ctaid_x / ntid_x / gridid / warpid / smid / clocklo / clockhi / clock64 / globaltimer64 / local_base64 等)。

sr_sz: b32 (默认, dst 占 1 vreg) / b64 (dst 占 vreg pair r / r+1 偶对齐)。sr_sz=b64 valid_combo: 仅 clock64 / globaltimer64 / local_base64 合法 (其他 sr_id 配 b64 硬件 trap, 走 prose 文档化)。

SIMT 行为: per-lane 读, 仅 active lane 写 dst。laneid / TID 等 per-thread SR 每 lane 不同值; CTAID / NTID 等 warp-uniform SR 每 lane 同值 (仍写 vreg, 但若已知 uniform 优先用 s2ur 节省 vreg)。

典型用例: s2r.laneid R0 (lane 内位置) / s2r.tid_x R0 (thread id x) / s2r.ctaid_x R0 (cta id x) / s2r.clock64 R0 sz=b64 (64-bit clock) / s2r.local_base64 R0 sz=b64 (local memory aperture base)。

s2ur **category**: Data move and selection **predicate_mode**: uniform

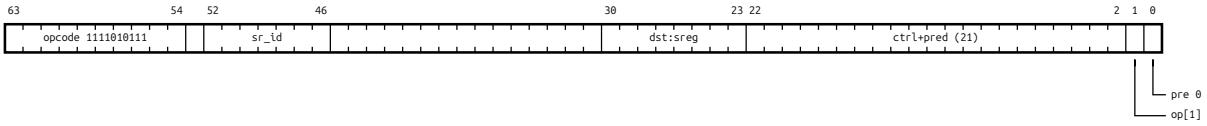
Leaf: s2ur__x_

Format:

```
s2ur.sr_id  dst
```

Operands: dst: sreg

Encoding Diagram: 64b, prefix 0, opcode 11111010111



Notes:

Special register read (warp-uniform): dst = SpecialRegister[sr_id], warp 集体读一次 SR 写到 sreg (warp 内 1 个 32-bit slot, 32 lane 共享)。仅支持 32-bit 路径 (没 b64 sr_sz, b64 paired alias 走 s2r)。

跟 s2r 区别: s2r 写 vreg (per-lane 32 slots, 适合 per-lane SR 如 laneid / TID), s2ur 写 sreg (warp 内 1 slot, 适合 warp-uniform SR 如 CTAID / smid); warp-uniform SR 用 s2ur 节省 31 个 vreg slot。

valid sr_id (warp-uniform 子集): ctaid_x / Y / Z, ntid_x / Y / Z, nctaid_x / Y / Z, gridid, warpid, nwarpid, smid, nsmid, virtcfg, cluster_ctaid_x / Y / Z, cluster_nctaid_x / Y / Z, cluster_ctarank, clusterid_x / Y / Z, nclusterid_x / Y / Z, total_smem_size, dynamic_smem_size, regalloc, barrieralloc, ctareg_pool_sz, clocklo, clockhi (clock 在 warp 内一致), globaltimerlo, globaltimerhi, local_base_lo, local_base_hi。

invalid sr_id (per-lane SR, 必须用 s2r 写 vreg): laneid, tid_x / Y / Z (per-thread 独立值), virtid, exec_mask (走 mset 读), pc_lo / pc_hi / pc64 (走 ugetpc 专用快捷指令)。

invalid sr_id (b64 paired alias, 必须用 s2r): clock64, globaltimer64, local_base64。

使用 invalid sr_id assembler 报错或 runtime 硬件 trap。

典型用例: s2ur.ctaid_x UR0 (block id x) / s2ur.ntid_x UR0 (block dim x) / s2ur.smid UR0 (sm id) / s2ur.cluster_ctaid_x UR0 / s2ur.ctareg_pool_sz UR0。

sel **category**: Data move and selection **predicate_mode**: simt

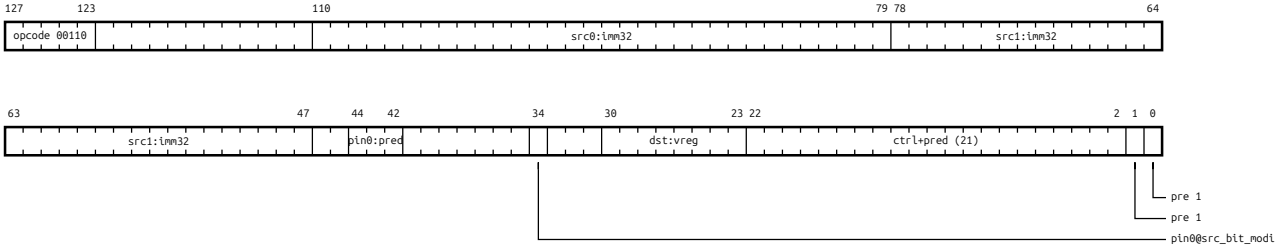
Leaf: sel__v_iip

Format:

```
sel  dst, src0, src1, pin0{.src_bit_modi}
```

Operands: dst: vreg; src0: imm32u; src1: imm32u; pin0: pred

Encoding Diagram: 128b, prefix 11, opcode 00110



Notes:

Predicate-controlled select: dst = pin0 ? src0 : src1, 根据 pin0 在 src0 / src1 之间选一个写整 32-bit dst (无类型转换)。整 warp per-lane 各自独立。

pin0 可挂 src_bit_modi: id (默认) / inv ([!]pin0 取反), 配合 cmp 逻辑反转。

src 类型组合: S1 / s2 任意 (vreg / sreg / imm 共 9 种), 总共 9 指令形态 = 3×3。覆盖 per-lane pred 选 2 个 uniform、选 2 个常量、混合 vreg+imm 等用例。

典型用例: 条件赋值 dst = pred ? a : b 一条搞定 (替代 isetp + sel 序列); dst = pred ? imm_a : imm_b 用 v_iip 不占临时 reg; conditional move (CMOV) 风格。

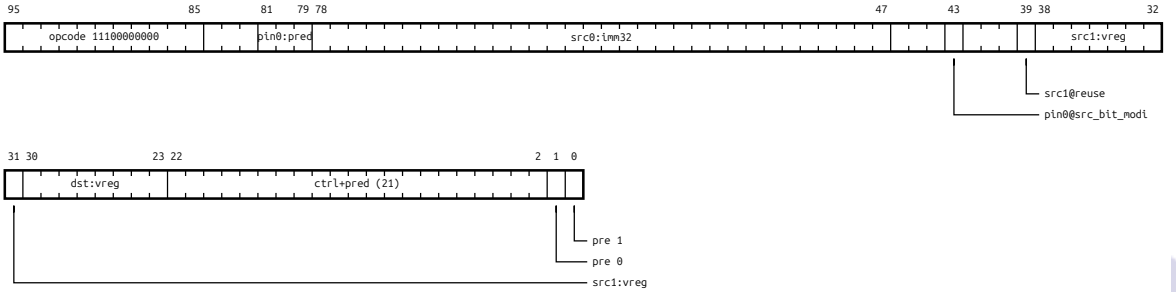
Leaf: sel__v_ivp

Format:

```
sel  dst, src0, src1{.reuse}, pin0{.src_bit_modi}
```

Operands: dst: vreg; src0: imm32u; src1: vreg; pin0: pred

Encoding Diagram: 96b, prefix 10, opcode 11100000000



Notes:

Predicate-controlled select: dst = pin0 ? src0 : src1, 根据 pin0 在 src0 / src1 之间选一个写整 32-bit dst (无类型转换)。整 warp per-lane 各自独立。

pin0 可挂 src_bit_modi: id (默认) / inv ([!]pin0 取反), 配合 cmp 逻辑反转。

src 类型组合: S1 / s2 任意 (vreg / sreg / imm 共 9 种), 总共 9 指令形态 = 3×3。覆盖 per-lane pred 选 2 个 uniform、选 2 个常量、混合 vreg+imm 等用例。

典型用例: 条件赋值 dst = pred ? a : b 一条搞定 (替代 isetp + sel 序列); dst = pred ? imm_a : imm_b 用 v_iip 不占临时 reg; conditional move (CMOV) 风格。

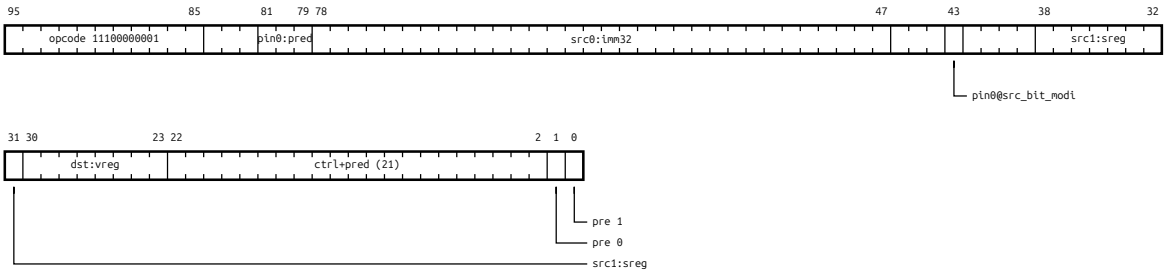
Leaf: sel__v_ixp

Format:

```
sel  dst, src0, src1, pin0{.src_bit_modi}
```

Operands: dst: vreg; src0: imm32u; src1: sreg; pin0: pred

Encoding Diagram: 96b, prefix 10, opcode 11100000001



Notes:

Predicate-controlled select: dst = pin0 ? src0 : src1, 根据 pin0 在 src0 / src1 之间选一个写整 32-bit dst (无类型转换)。整 warp per-lane 各自独立。

pin0 可挂 src_bit_modi: id (默认) / inv ([!]pin0 取反), 配合 cmp 逻辑反转。

src 类型组合: S1 / s2 任意 (vreg / sreg / imm 共 9 种), 总共 9 指令形态 = 3×3。覆盖 per-lane pred 选 2 个 uniform、选 2 个常量、混合 vreg+imm 等用例。

典型用例: 条件赋值 dst = pred ? a : b 一条搞定 (替代 isetp + sel 序列); dst = pred ? imm_a : imm_b 用 v_iip 不占临时 reg; conditional move (CMOV) 风格。

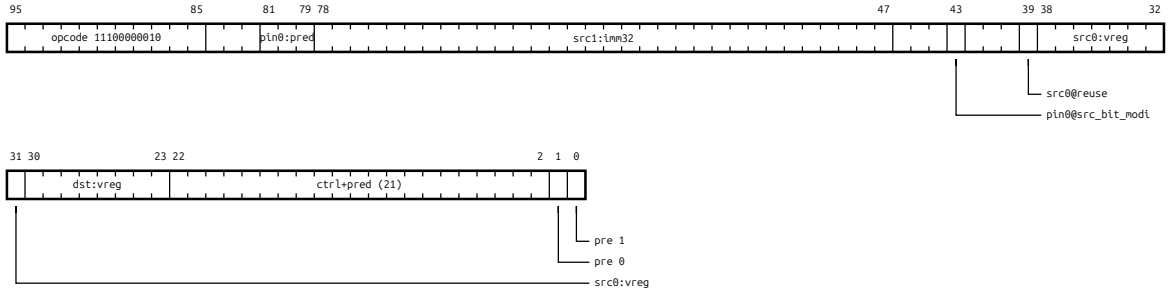
Leaf: sel__v_vip

Format:

```
sel  dst, src0{.reuse}, src1, pin0{.src_bit_modi}
```

Operands: dst: vreg; src0: vreg; src1: imm32u; pin0: pred

Encoding Diagram: 96b, prefix 10, opcode 11100000010



Notes:

Predicate-controlled select: dst = pin0 ? src0 : src1, 根据 pin0 在 src0 / src1 之间选一个写整 32-bit dst (无类型转换)。整 warp per-lane 各自独立。

pin0 可挂 src_bit_modi: id (默认) / inv ([!]pin0 取反), 配合 cmp 逻辑反转。

src 类型组合: S1 / s2 任意 (vreg / sreg / imm 共 9 种), 总共 9 指令形态 = 3×3。覆盖 per-lane pred 选 2 个 uniform、选 2 个常量、混合 vreg+imm 等用例。

典型用例: 条件赋值 dst = pred ? a : b 一条搞定 (替代 isetp + sel 序列); dst = pred ? imm_a : imm_b 用 v_iip 不占临时 reg; conditional move (CMOV) 风格。

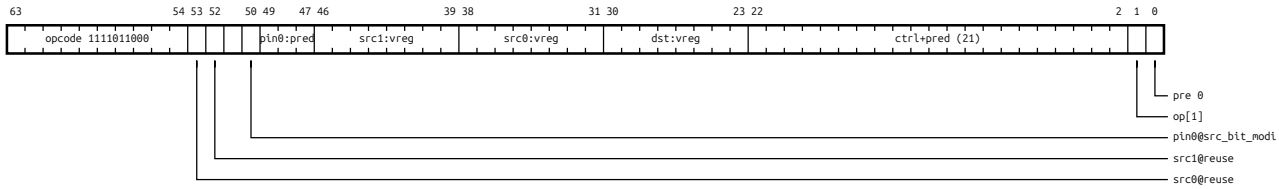
Leaf: sel__v_vvp

Format:

```
sel  dst, src0{.reuse}, src1{.reuse}, pin0{.src_bit_modi}
```

Operands: dst: vreg; src0: vreg; src1: vreg; pin0: pred

Encoding Diagram: 64b, prefix 0, opcode 11111011000



Notes:

Predicate-controlled select: dst = pin0 ? src0 : src1, 根据 pin0 在 src0 / src1 之间选一个写整 32-bit dst (无类型转换)。整 warp per-lane 各自独立。

pin0 可挂 src_bit_modi: id (默认) / inv ([!]pin0 取反), 配合 cmp 逻辑反转。

src 类型组合: S1 / s2 任意 (vreg / sreg / imm 共 9 种), 总共 9 指令形态 = 3×3。覆盖 per-lane pred 选 2 个 uniform、选 2 个常量、混合 vreg+imm 等用例。

典型用例: 条件赋值 dst = pred ? a : b 一条搞定 (替代 isetp + sel 序列); dst = pred ? imm_a : imm_b 用 v_iip 不占临时 reg; conditional move (CMOV) 风格。

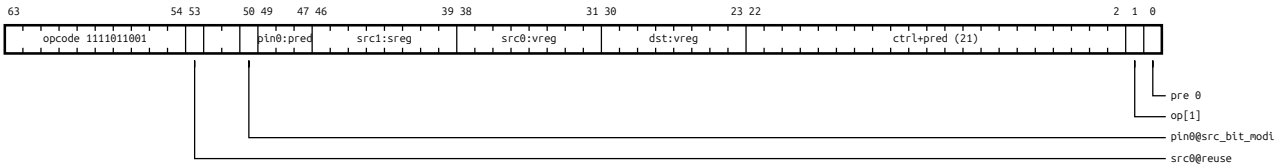
Leaf: sel__v_vxp

Format:

```
sel  dst, src0{.reuse}, src1, pin0{.src_bit_modi}
```

Operands: dst: vreg; src0: vreg; src1: sreg; pin0: pred

Encoding Diagram: 64b, prefix 0, opcode 11111011001



Notes:

Predicate-controlled select: dst = pin0 ? src0 : src1, 根据 pin0 在 src0 / src1 之间选一个写整 32-bit dst (无类型转换)。整 warp per-lane 各自独立。

pin0 可挂 src_bit_modi: id (默认) / inv ([!]pin0 取反), 配合 cmp 逻辑反转。

src 类型组合: S1 / s2 任意 (vreg / sreg / imm 共 9 种), 总共 9 指令形态 = 3×3。覆盖 per-lane pred 选 2 个 uniform、选 2 个常量、混合 vreg+imm 等用例。

典型用例: 条件赋值 dst = pred ? a : b 一条搞定 (替代 isetp + sel 序列); dst = pred ? imm_a : imm_b 用 v_iip 不占临时 reg; conditional move (CMOV) 风格。

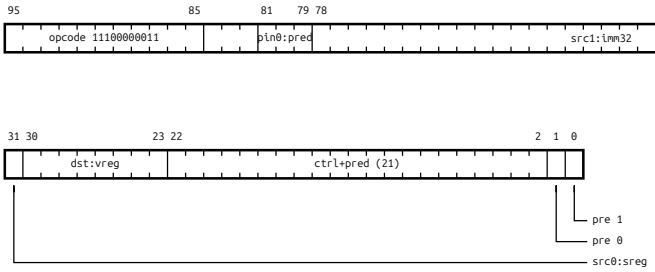
Leaf: sel__v_xip

Format:

```
sel  dst, src0, src1, pin0{.src_bit_modi}
```

Operands: dst: vreg; src0: sreg; src1: imm32u; pin0: pred

Encoding Diagram: 96b, prefix 10, opcode 11100000011



Notes:

Predicate-controlled select: dst = pin0 ? src0 : src1, 根据 pin0 在 src0 / src1 之间选一个写整 32-bit dst (无类型转换)。整 warp per-lane 各自独立。

pin0 可挂 src_bit_modi: id (默认) / inv ([!]pin0 取反), 配合 cmp 逻辑反转。

src 类型组合: S1 / s2 任意 (vreg / sreg / imm 共 9 种), 总共 9 指令形态 = 3×3。覆盖 per-lane pred 选 2 个 uniform、选 2 个常量、混合 vreg+imm 等用例。

典型用例: 条件赋值 dst = pred ? a : b 一条搞定 (替代 isetp + sel 序列); dst = pred ? imm_a : imm_b 用 v_iip 不占临时 reg; conditional move (CMOV) 风格。

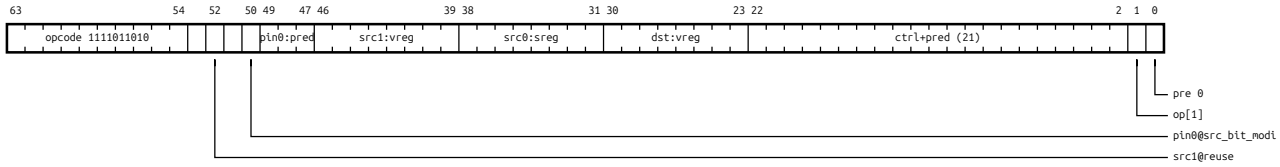
Leaf: sel__v_xvp

Format:

```
sel  dst, src0, src1{.reuse}, pin0{.src_bit_modi}
```

Operands: dst: vreg; src0: sreg; src1: vreg; pin0: pred

Encoding Diagram: 64b, prefix 0, opcode 11111011010



Notes:

Predicate-controlled select: dst = pin0 ? src0 : src1, 根据 pin0 在 src0 / src1 之间选一个写整 32-bit dst (无类型转换)。整 warp per-lane 各自独立。

pin0 可挂 src_bit_modi: id (默认) / inv ([!]pin0 取反), 配合 cmp 逻辑反转。

src 类型组合: S1 / s2 任意 (vreg / sreg / imm 共 9 种), 总共 9 指令形态 = 3×3。覆盖 per-lane pred 选 2 个 uniform、选 2 个常量、混合 vreg+imm 等用例。

典型用例: 条件赋值 dst = pred ? a : b 一条搞定 (替代 isetp + sel 序列); dst = pred ? imm_a : imm_b 用 v_iip 不占临时 reg; conditional move (CMOV) 风格。

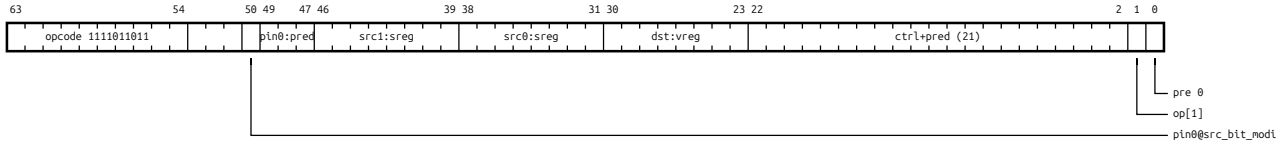
Leaf: sel__v_xxp

Format:

```
sel  dst, src0, src1, pin0{.src_bit_modi}
```

Operands: dst: vreg; src0: sreg; src1: sreg; pin0: pred

Encoding Diagram: 64b, prefix 0, opcode 11111011011



Notes:

Predicate-controlled select: dst = pin0 ? src0 : src1, 根据 pin0 在 src0 / src1 之间选一个写整 32-bit dst (无类型转换)。整 warp per-lane 各自独立。

pin0 可挂 src_bit_modi: id (默认) / inv ([!]pin0 取反), 配合 cmp 逻辑反转。

src 类型组合: S1 / s2 任意 (vreg / sreg / imm 共 9 种), 总共 9 指令形态 = 3×3。覆盖 per-lane pred 选 2 个 uniform、选 2 个常量、混合 vreg+imm 等用例。

典型用例: 条件赋值 dst = pred ? a : b 一条搞定 (替代 isetp + sel 序列); dst = pred ? imm_a : imm_b 用 v_iip 不占临时 reg; conditional move (CMOV) 风格。

umov　**category**: Data move and selection　**predicate_mode**: uniform

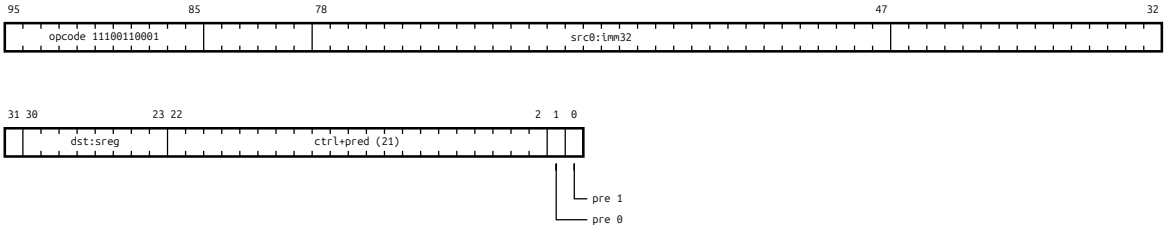
Leaf: umov__x_i

Format:

```
umov  dst, src0
```

Operands: dst: sreg; src0: imm32u

Encoding Diagram: 96b, prefix 10, opcode 11100110001



Notes:

32-bit 数据复制 uniform 变体: dst = src0, 所有 operand 都是 sreg / imm, 整 warp 共享。mov 的 uniform 镜像。

无 64-bit 变体: 64-bit uniform move 走 2 条 umov 序列 (没有单条 umov_64)。

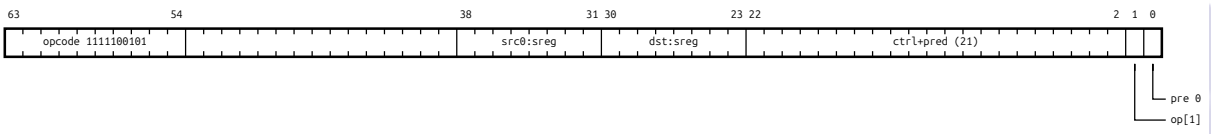
Leaf: umov__x_x

Format:

```
umov  dst, src0
```

Operands: dst: sreg; src0: sreg

Encoding Diagram: 64b, prefix 0, opcode 11111100101



Notes:

32-bit 数据复制 uniform 变体: dst = src0, 所有 operand 都是 sreg / imm, 整 warp 共享。mov 的 uniform 镜像。

无 64-bit 变体: 64-bit uniform move 走 2 条 umov 序列 (没有单条 umov_64)。

up2ur　**category**: Data move and selection　**predicate_mode**: uniform

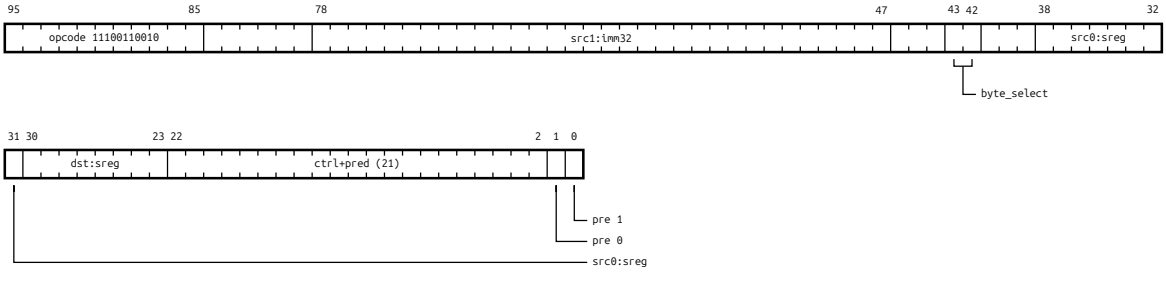
Leaf: up2ur__x_xi

Format:

```
up2ur{.byte_select}  dst, src0, src1
```

Operands: dst: sreg; src0: sreg; src1: imm32u

Encoding Diagram: 96b, prefix 10, opcode 11100110010



Notes:

把当前 warp 的 7 个 uniform predicate (UP0..UP6) 有选择地搬到 uniform register: dst = src0 但 byte_select

那个 byte 里逐 bit 替换 — SbMask[j]=1 取 UPR[j], SbMask[j]=0 保留 src0 该 byte 的 bit j。warp 级单次执行 (UPR 是 per-warp 不是 per-thread, 跟 simt p2r 的 PR 区别)。p2r 的 uniform 镜像。

operand 对应: asm “UP2UR{.insert} URd, UPR, URa, SbMask” → spec dst = URd, src0 = URa (原值 sreg), src1 = SbMask (uimm8 或 sreg)。UPR 是 architectural state (per-warp uniform predicate file 当 8-bit 整体读), 不占编码 bit、不放 operands 列表。

UPR 8-bit 布局: [reserved, UP6, UP5, UP4, UP3, UP2, UP1, UP0]。每 warp 一份。

byte_select=b0 / b1 / b2 / b3: 选 dst / src0 的哪个 byte 是合并目标。

伪代码 (warp 级): byte_b = (src0 » (8 * byte_select)) & 0xFF; merged = 0; for j in 0..7: merged |= (SbMask[j] ? UPR[j] : byte_b[j]) « j; dst = (src0 & ~(0xFF « (8 * byte_select))) | (merged « (8 * byte_select))。

典型用途: 跟 ur2up 配对做 uniform predicate state 的 spill / reload (up 寄存器不够用时把 7 个 up 暂存 UR 里)。应用代码不会直接生成。

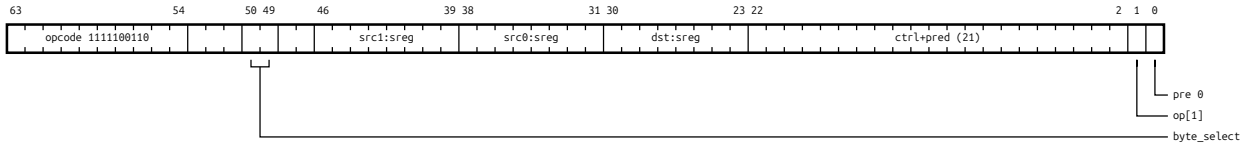
Leaf: up2ur__x_xx

Format:

```
up2ur{.byte_select}  dst, src0, src1
```

Operands: dst: sreg; src0: sreg; src1: sreg

Encoding Diagram: 64b, prefix 0, opcode 11111100110



Notes:

把当前 warp 的 7 个 uniform predicate (UP0..UP6) 有选择地搬到 uniform register: dst = src0 但 byte_select 那个 byte 里逐 bit 替换 — SbMask[j]=1 取 UPR[j], SbMask[j]=0 保留 src0 该 byte 的 bit j。warp 级单次执行 (UPR 是 per-warp 不是 per-thread, 跟 simt p2r 的 PR 区别)。p2r 的 uniform 镜像。

operand 对应: asm “UP2UR{.insert} URd, UPR, URa, SbMask” → spec dst = URd, src0 = URa (原值 sreg), src1 = SbMask (uimm8 或 sreg)。UPR 是 architectural state (per-warp uniform predicate file 当 8-bit 整体读), 不占编码 bit、不放 operands 列表。

UPR 8-bit 布局: [reserved, UP6, UP5, UP4, UP3, UP2, UP1, UP0]。每 warp 一份。

byte_select=b0 / b1 / b2 / b3: 选 dst / src0 的哪个 byte 是合并目标。

伪代码 (warp 级): byte_b = (src0 » (8 * byte_select)) & 0xFF; merged = 0; for j in 0..7: merged |= (SbMask[j] ? UPR[j] : byte_b[j]) « j; dst = (src0 & ~(0xFF « (8 * byte_select))) | (merged « (8 * byte_select))。

典型用途: 跟 ur2up 配对做 uniform predicate state 的 spill / reload (up 寄存器不够用时把 7 个 up 暂存 UR 里)。应用代码不会直接生成。

ur2up　**category**: Data move and selection　**predicate_mode**: uniform

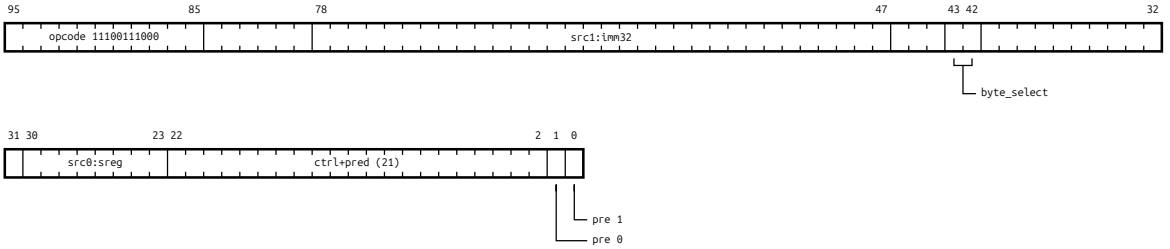
Leaf: ur2up___xi

Format:

```
ur2up{.byte_select}  src0, src1
```

Operands: src0: sreg; src1: imm32u

Encoding Diagram: 96b, prefix 10, opcode 11100111000



Notes:

从 uniform register 某个 byte 取 8 bit 选择性写当前 warp 的 7 个 uniform predicate: 对每 bit j, SbMask[j]=1 → UPR[j] = src0 的 byte_select byte 的 bit j; SbMask[j]=0 → UPR[j] 保持原值。warp 级单次执行 (跟 ur2up 是 up2ur 的反向, r2p 的 uniform 镜像)。

operand 对应: asm “UR2UP UPR, URa{.b0/.b1/.b2/.b3}, SbMask” → spec src0 = URa (输入 sreg), src1 = SbMask (uimm8 或 sreg)。UPR 是 architectural state (uniform predicate file 当 8-bit 整体写), 不占编码 bit、不放 operands 列表。

byte_select=b0 / b1 / b2 / b3: 选 src0 的哪个 byte 作为输入。

伪代码 (warp 级): byte_b = (src0 » (8 * byte_select)) & 0xFF; for j in 0..7: if SbMask[j]: UPR[j] = byte_b[j] (否则 UPR[j] 不变)。

典型用途: 跟 up2ur 配对做 uniform predicate state 的 reload。

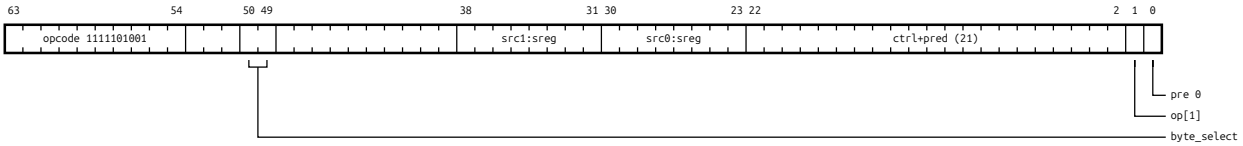
Leaf: ur2up___xx

Format:

```
ur2up{.byte_select}  src0, src1
```

Operands: src0: sreg; src1: sreg

Encoding Diagram: 64b, prefix 0, opcode 11111101001



Notes:

从 uniform register 某个 byte 取 8 bit 选择性写当前 warp 的 7 个 uniform predicate: 对每 bit j, SbMask[j]=1 → UPR[j] = src0 的 byte_select byte 的 bit j; SbMask[j]=0 → UPR[j] 保持原值。warp 级单次执行 (跟 ur2up 是 up2ur 的反向, r2p 的 uniform 镜像)。

operand 对应: asm “UR2UP UPR, URa{.b0/.b1/.b2/.b3}, SbMask” → spec src0 = URa (输入 sreg), src1 = SbMask (uimm8 或 sreg)。UPR 是 architectural state (uniform predicate file 当 8-bit 整体写), 不占编码 bit、不放 operands 列表。

byte_select=b0 / b1 / b2 / b3: 选 src0 的哪个 byte 作为输入。

伪代码 (warp 级): byte_b = (src0 » (8 * byte_select)) & 0xFF; for j in 0..7: if SbMask[j]: UPR[j] = byte_b[j] (否则 UPR[j] 不变)。

典型用途: 跟 up2ur 配对做 uniform predicate state 的 reload。

usel **category**: Data move and selection **predicate_mode**: uniform

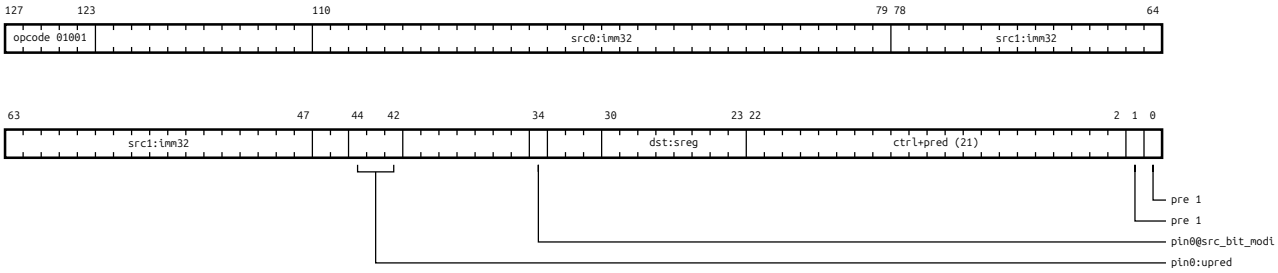
Leaf: usel__x_iip

Format:

```
usel  dst, src0, src1, pin0{.src_bit_modi}
```

Operands: dst: sreg; src0: imm32u; src1: imm32u; pin0: upred

Encoding Diagram: 128b, prefix 11, opcode 01001



Notes:

Predicate-controlled select uniform 变体: dst = pin0 ? src0 : src1, 所有 operand 都是 sreg / imm, pin0 是 upred (per-warp), 整 warp 共享。

src 类型组合: 4 指令形态 = 2×2 (sreg / imm × sreg / imm), 覆盖 dst = pred ? URa : URb / dst = pred ? imm_a : imm_b 等高频用例。

pin0 可挂 src_bit_modi: id / inv (取反)。

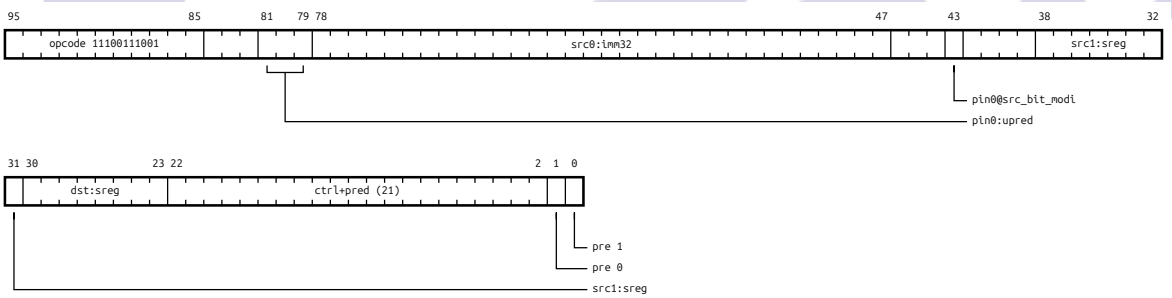
Leaf: usel__x_ixp

Format:

```
usel  dst, src0, src1, pin0{.src_bit_modi}
```

Operands: dst: sreg; src0: imm32u; src1: sreg; pin0: upred

Encoding Diagram: 96b, prefix 10, opcode 11100111001



Notes:

Predicate-controlled select uniform 变体: dst = pin0 ? src0 : src1, 所有 operand 都是 sreg / imm, pin0 是 upred (per-warp), 整 warp 共享。

src 类型组合: 4 指令形态 = 2×2 (sreg / imm × sreg / imm), 覆盖 dst = pred ? URa : URb / dst = pred ? imm_a : imm_b 等高频用例。

pin0 可挂 src_bit_modi: id / inv (取反)。

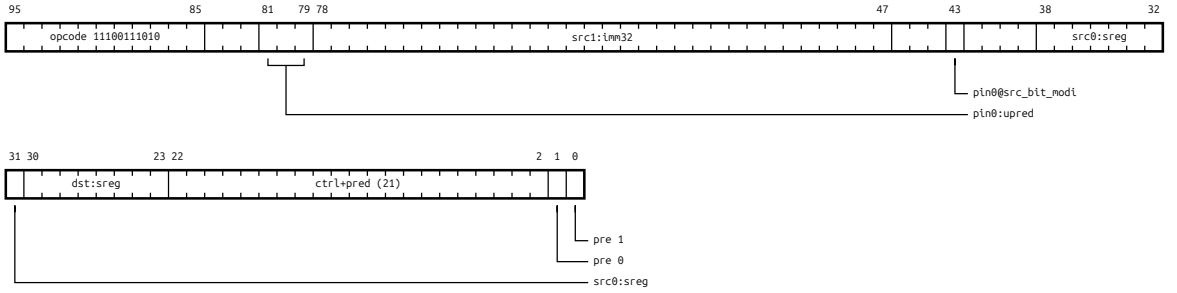
Leaf: usel__x_xip

Format:

```
usel  dst, src0, src1, pin0{.src_bit_modi}
```

Operands: dst: sreg; src0: sreg; src1: imm32u; pin0: upred

Encoding Diagram: 96b, prefix 10, opcode 11100111010



Notes:

Predicate-controlled select uniform 变体: dst = pin0 ? src0 : src1, 所有 operand 都是 sreg / imm, pin0 是 upred (per-warp), 整 warp 共享。

src 类型组合: 4 指令形态 = 2×2 (sreg / imm × sreg / imm), 覆盖 dst = pred ? URa : URb / dst = pred ? imm_a : imm_b 等高频用例。

pin0 可挂 src_bit_modi: id / inv (取反)。

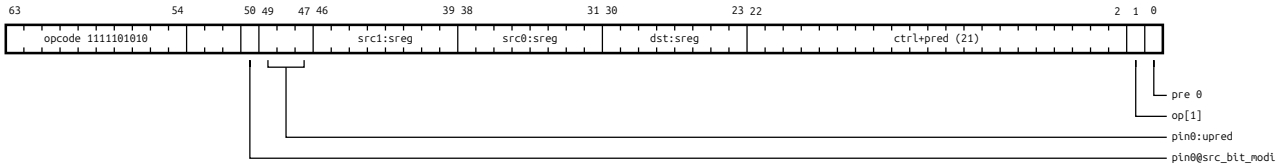
Leaf: usel__x_xxp

Format:

```
usel  dst, src0, src1, pin0{.src_bit_modi}
```

Operands: dst: sreg; src0: sreg; src1: sreg; pin0: upred

Encoding Diagram: 64b, prefix 0, opcode 11111101010



Notes:

Predicate-controlled select uniform 变体: dst = pin0 ? src0 : src1, 所有 operand 都是 sreg / imm, pin0 是 upred (per-warp), 整 warp 共享。

src 类型组合: 4 指令形态 = 2×2 (sreg / imm × sreg / imm), 覆盖 dst = pred ? URa : URb / dst = pred ? imm_a : imm_b 等高频用例。

pin0 可挂 src_bit_modi: id / inv (取反)。

14.7.8 Memory load/store

cctl **category**: Memory load/store **predicate_mode**: simt

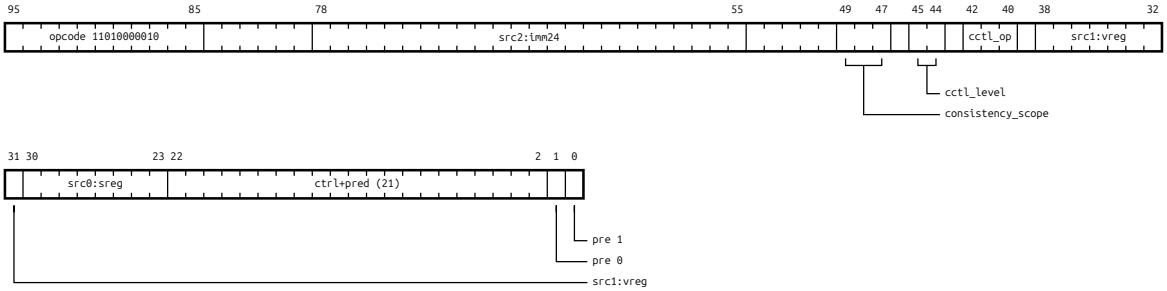
Leaf: cctl___xvi

Format:

```
cctl.cctl_op.cctl_level.consistency_scope  src0, src1, src2
```

Operands: src0: sreg; src1: vreg; src2: imm24s

Encoding Diagram: 96b, prefix 10, opcode 11010000010



Notes:

Cache line control (line-targeted): per-lane 提供 cache line 地址, 对该 line 做 cctl_op 指定的操作。整 warp 32 个 active lane 各自独立操作 per-lane 地址 (predicate_mode=simt)。

cctl_op 5 valid (line-targeted, simt 路径): pf1 (prefetch line into l1, 减后续 load latency) / pf2 (prefetch line into l2) / wb (writeback dirty line 到下一级, 保 durability before eviction) / iv (invalidate 单条 line, 强制下次访问 re-fetch) / rs (reserve line, hint for upcoming store 避免 read-for-ownership)。Broadcast 操作 (ivall / wball) 不在 simt cctl 路径 — 走 ucctl uniform u 指令形态。

cctl_level 4 valid: l1 (fastest, per-sm, pf1 / rs 主路径) / l2 (shared across SMs, pf2 主路径) / const (constant cache) / tex (texture cache, iv 主路径)。op × level 组合: pf1 仅 l1, pf2 仅 l2, wb 作用 l1 或 l2, iv 作用任意 level, rs l1 主路径。

consistency_scope: cta / sm / gpu / sys, 控操作可见范围。

operand 模型 (_xvi 指令形态): saddr (sreg base) + vaddr (vreg per-lane offset) + imm.24 — 跟 ldg / lds 地址模型同源。

典型用例: pf1 / pf2 prefetch hot tile to cache 减 latency / wb ensure durability before kernel exit / iv invalidate stale cache line / rs hint upcoming store。

ld **category**: Memory load/store **predicate_mode**: simt

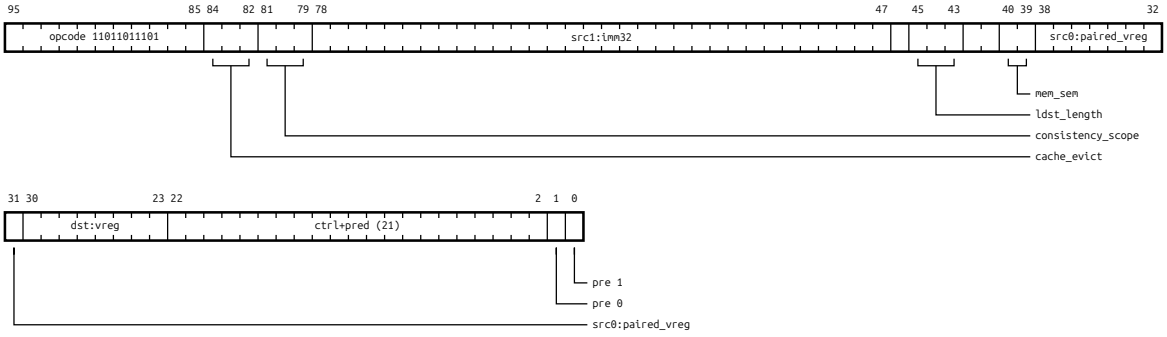
Leaf: ld__v_vi

Format:

```
ld.ldst_length{.mem_sem}.consistency_scope.cache_evict  dst, src0, src1
```

Operands: dst: vreg [notes: span=default=1; ldst_length=b64→2; ldst_length=b128→4, align=default=1; ldst_length=b64→2; ldst_length=b128→4]; src0: paired_vreg; src1: imm32s

Encoding Diagram: 96b, prefix 10, opcode 11011011101



Notes:

Generic flat address space load: dst = *(effective_address), 整 warp 32 个 active lane 各自独立读取 per-lane 地址。这条指令本身不写死 memory space, 硬件根据 addr[63:62] tag 自动路由到 global / local / shared / constant 四个空间之一。

地址高 2 bit 直接编码 memory space (硬件不需要窗口比较器): addr[63:62]=00 → global memory (设备级 DRAM); =01 → local memory (thread-private spill); =10 → shared memory (cta-shared SRAM); =11 → constant memory (只读, 独立 cache)。剩余 62 bit 是 space 内 byte offset。

memory family 通用地址约定: base reg 是 64-bit pair (r / r+1, 偶对齐), offset reg 32-bit signed (硬件 sign-extend 加到 base), imm 32-bit signed。tag 路线下 addr 必 64-bit (32-bit pointer 无法表达 [63:62] tag)。

有效地址计算: v_vi 指令形态 ea = vaddr + offset (vaddr 是 vreg pair 64-bit per-lane 完整地址); v_xvi 指令形态 ea = saddr + vaddr + offset (saddr 64-bit base + vaddr 32-bit per-lane offset + imm.32 signed)。

同 warp 同空间约束 (正式架构约束): 一次执行中, 门控后所有 active lane 的 addr[63:62] tag 必须相同, 否则硬件 trap (illegal address space mixing)。这是硬件上的有意简化 (省掉按空间分组重放的硬件); 它的直接后果是 generic 访存不是逐 lane 任意混合空间的透明通路, 而是 warp 一致空间的扁平访问。编译器契约: 能证明全 warp 同空间才可以直接发一条 ld; 证明不了时必须由编译器生成分组慢路径 — 读各 lane 的 tag、按空间分组改 EXEC、每组发一条 ld、最后汇合 (语义等价, 代价更高)。

自然对齐: 地址按 ldst_length 自然对齐 (u8/s8 任意, u16/s16 2-byte, b32 4-byte, b64 8-byte, b128 16-byte), 未对齐 trap。

ldst_length: generic ld 实际仅 u8 / s8 / u16 / s16 / b32 / b64 / b128 共 7 valid (b256 仅 ldg / stg 支持)。

mem_sem: constant / weak (默认) / strong / mmio, 控 acquire-release / strong / weak ordering。

consistency_scope: cta / sm / cluster / gpu / sys, 控可见范围。

cache_evict: ef / en / el / lu / eu / na, 控 cache 行为。

编译期已知 global 用 ldg (跳过 tag 检查 + 默认 mem_sem=weak), 已知 shared 用 lds, 已知 local 用 ldl, 已知 constant 用 ldc; 编译期不知空间、但能证明 warp 内空间一致时, 用 generic ld; 连一致性都证明不了时, 走上一条说的编译器分组慢路径。典型用例: 编译期未知空间的指针解引用 (同一个函数既被传入 global 指针又被传入 shared 指针) / 跨地址空间复用的公共代码路径。

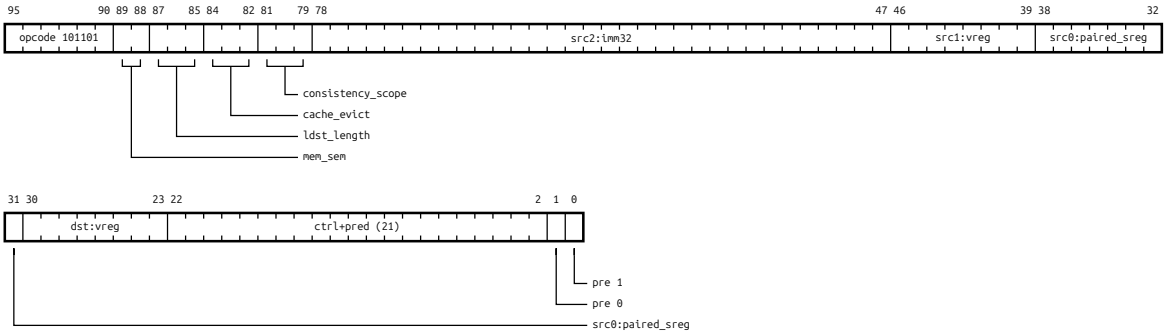
Leaf: ld__v_xvi

Format:

```
ld.ldst_length{.mem_sem}.consistency_scope.cache_evict  dst, src0, src1, src2
```

Operands: dst: vreg [notes: span=default=1; ldst_length=b64→2; ldst_length=b128→4, align=default=1; ldst_length=b64→2; ldst_length=b128→4]; src0: paired_sreg; src1: vreg; src2: imm32s

Encoding Diagram: 96b, prefix 10, opcode 101101



Notes:

Generic flat address space load: dst = *(effective_address), 整 warp 32 个 active lane 各自独立读取 per-lane 地址。这条指令本身不写死 memory space, 硬件根据 addr[63:62] tag 自动路由到 global / local / shared / constant 四个空间之一。

地址高 2 bit 直接编码 memory space (硬件不需要窗口比较器): addr[63:62]=00 → global memory (设备级 DRAM); =01 → local memory (thread-private spill); =10 → shared memory (cta-shared SRAM); =11 → constant

memory (只读, 独立 cache)。剩余 62 bit 是 space 内 byte offset。

memory family 通用地址约定: base reg 是 64-bit pair (r / r+1, 偶对齐), offset reg 32-bit signed (硬件 sign-extend 加到 base), imm 32-bit signed。tag 路线下 addr 必 64-bit (32-bit pointer 无法表达 [63:62] tag)。

有效地址计算: v_vi 指令形态 ea = vaddr + offset (vaddr 是 vreg pair 64-bit per-lane 完整地址); v_xvi 指令形态 ea = saddr + vaddr + offset (saddr 64-bit base + vaddr 32-bit per-lane offset + imm.32 signed)。

同 warp 同空间约束 (正式架构约束): 一次执行中, 门控后所有 active lane 的 addr[63:62] tag 必须相同, 否则硬件 trap (illegal address space mixing)。这是硬件上的有意简化 (省掉按空间分组重放的硬件); 它的直接后果是 generic 访存不是逐 lane 任意混合空间的透明通路, 而是 warp 一致空间的扁平访问。编译器契约: 能证明全 warp 同空间才可以直接发一条 ld; 证明不了时必须由编译器生成分组慢路径 — 读各 lane 的 tag、按空间分组改 EXEC、每组发一条 ld、最后汇合 (语义等价, 代价更高)。

自然对齐: 地址按 ldst_length 自然对齐 (u8/s8 任意, u16/s16 2-byte, b32 4-byte, b64 8-byte, b128 16-byte), 未对齐 trap。

ldst_length: generic ld 实际仅 u8 / s8 / u16 / s16 / b32 / b64 / b128 共 7 valid (b256 仅 ldg / stg 支持)。

mem_sem: constant / weak (默认) / strong / mmio, 控 acquire-release / strong / weak ordering。

consistency_scope: cta / sm / cluster / gpu / sys, 控可见范围。

cache_evict: ef / en / el / lu / eu / na, 控 cache 行为。

编译期已知 global 用 ldg (跳过 tag 检查 + 默认 mem_sem=weak), 已知 shared 用 lds, 已知 local 用 ldl, 已知 constant 用 ldc; 编译期不知空间、但能证明 warp 内空间一致时, 用 generic ld; 连一致性都证明不了时, 走上一条说的编译器分组慢路径。典型用例: 编译期未知空间的指针解引用 (同一个函数既被传入 global 指针又被传入 shared 指针) / 跨地址空间复用的公共代码路径。

ldc **category**: Memory load/store **predicate_mode**: simt

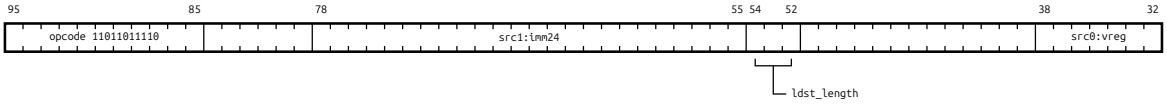
Leaf: ldc__v_vi

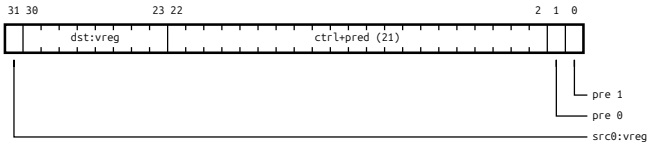
Format:

```
ldc.ldst_length  dst, src0, src1
```

Operands: dst: vreg [notes: span=default=1; ldst_length=b64→2; ldst_length=b128→4, align=default=1; ldst_length=b64→2; ldst_length=b128→4]; src0: vreg; src1: imm24s

Encoding Diagram: 96b, prefix 10, opcode 11011011110





Notes:

Explicit constant memory load: dst = *(constant_mem*)(addr + offset), read-only broadcast-optimized load。constant memory 是 gpu 只读数据区 (kernel 参数 / 编译期 immediate / resource descriptor / 实现定义的异步搬运描述信息等), 硬件有专用 constant cache (per-sm, 几 KB) 支持 warp broadcast — warp 内所有 lane 读同一地址只占一次 cache bandwidth。

地址模型: 走纯地址 (addr + offset, bank 管理交给 driver / runtime — driver 把 bank base 写到 sreg, ldc 指令看到 effective_address = bank_base + offset)。base reg 32-bit (跟 lds / ldl 一致, 不是 64-bit pair 跟 ldg 不同, 因 constant 容量小)。

对齐: 按 ldst_length 自然对齐 (b32 4-byte / b64 8-byte)。

modifier: 仅 ldst_length (constant 是只读 SRAM/cache 不挂 mem_sem / cache_evict / consistency_scope)。

典型用例: kernel parameter load (driver 设置 c[0x0] base ptr, ldc 直接 load) / compile-time immediate constant load (c[0x1]) / 用户 uniform buffer binding load (c[0x3..0x10]) / 异步搬运描述信息 load (descriptor 通常存 constant memory)。

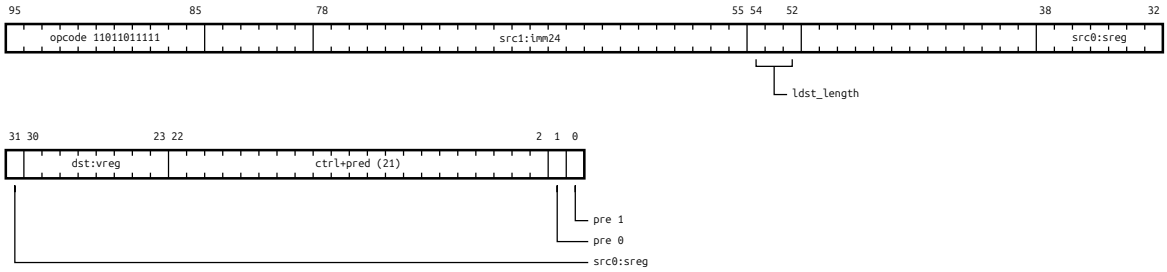
Leaf: ldc__v_xi

Format:

```
ldc.ldst_length  dst, src0, src1
```

Operands: dst: vreg [notes: span=default=1; ldst_length=b64→2; ldst_length=b128→4, align=default=1; ldst_length=b64→2; ldst_length=b128→4]; src0: sreg; src1: imm24s

Encoding Diagram: 96b, prefix 10, opcode 11011011111



Notes:

Explicit constant memory load: dst = *(constant_mem*)(addr + offset), read-only broadcast-optimized load。constant memory 是 gpu 只读数据区 (kernel 参数 / 编译期 immediate / resource descriptor / 实现定义的异步搬运描述信息等), 硬件有专用 constant cache (per-sm, 几 KB) 支持 warp broadcast — warp 内所有 lane 读同一地址只占一次 cache bandwidth。

地址模型: 走纯地址 (addr + offset, bank 管理交给 driver / runtime — driver 把 bank base 写到 sreg, ldc 指令看到 effective_address = bank_base + offset)。base reg 32-bit (跟 lds / ldl 一致, 不是 64-bit pair 跟 ldg 不同, 因 constant 容量小)。

对齐: 按 ldst_length 自然对齐 (b32 4-byte / b64 8-byte)。

modifier: 仅 ldst_length (constant 是只读 SRAM/cache 不挂 mem_sem / cache_evict / consistency_scope)。

典型用例: kernel parameter load (driver 设置 c[0x0] base ptr, ldc 直接 load) / compile-time immediate constant load (c[0x1]) / 用户 uniform buffer binding load (c[0x3..0x10]) / 异步搬运描述信息 load (descriptor 通常存 constant memory)。

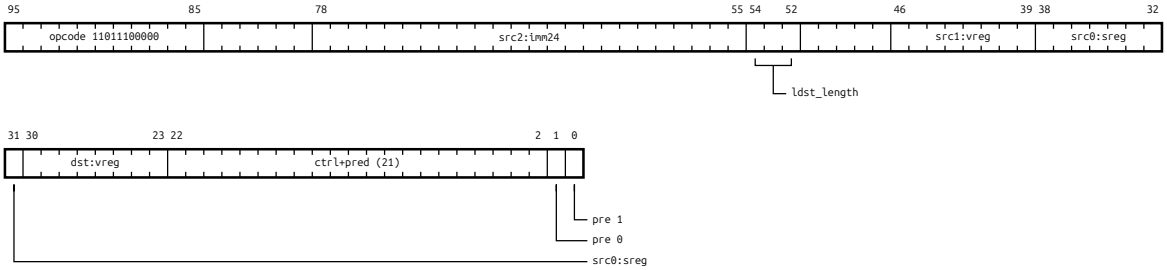
Leaf: ldc__v_xvi

Format:

```
ldc.ldst_length  dst, src0, src1, src2
```

Operands: dst: vreg [notes: span=default=1; ldst_length=b64→2; ldst_length=b128→4, align=default=1; ldst_length=b64→2; ldst_length=b128→4]; src0: sreg; src1: vreg; src2: imm24s

Encoding Diagram: 96b, prefix 10, opcode 11011100000



Notes:

Explicit constant memory load: dst = *(constant_mem*)(addr + offset), read-only broadcast-optimized load。constant memory 是 gpu 只读数据区 (kernel 参数 / 编译期 immediate / resource descriptor / 实现定义的异步搬运描述信息等), 硬件有专用 constant cache (per-sm, 几 KB) 支持 warp broadcast — warp 内所有 lane 读同一地址只占一次 cache bandwidth。

地址模型: 走纯地址 (addr + offset, bank 管理交给 driver / runtime — driver 把 bank base 写到 sreg, ldc 指令看到 effective_address = bank_base + offset)。base reg 32-bit (跟 lds / ldl 一致, 不是 64-bit pair 跟 ldg 不同, 因 constant 容量小)。

对齐: 按 ldst_length 自然对齐 (b32 4-byte / b64 8-byte)。

modifier: 仅 ldst_length (constant 是只读 SRAM/cache 不挂 mem_sem / cache_evict / consistency_scope)。

典型用例: kernel parameter load (driver 设置 c[0x0] base ptr, ldc 直接 load) / compile-time immediate constant load (c[0x1]) / 用户 uniform buffer binding load (c[0x3..0x10]) / 异步搬运描述信息 load (descriptor 通常存 constant memory)。

ldg **category**: Memory load/store **predicate_mode**: simt

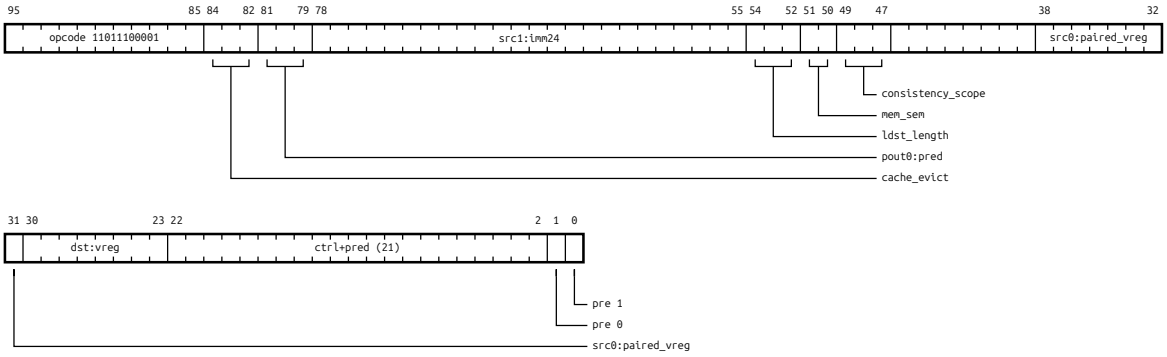
Leaf: ldg__vp_vi

Format:

```
ldg.ldst_length{.mem_sem}.consistency_scope.cache_evict  dst, pout0, src0, src1
```

Operands: dst: vreg [notes: span=default=1; ldst_length=b64→2; ldst_length=b128→4; ldst_length=b256→8, align=default=1; ldst_length=b64→2; ldst_length=b128→4; ldst_length=b256→8]; pout0: pred; src0: paired_vreg; src1: imm24s

Encoding Diagram: 96b, prefix 10, opcode 11011100001



Notes:

Explicit global memory load: dst = *(addr) if pout0, pout0 = (addr_in_bounds && pin_pred)。整 warp 32 个 active lane 各自独立读 per-lane 地址, 跳过 generic ld 的 [63:62] tag 检查 (编译期已知 global), 默认 mem_sem=weak 走更快路径。

pout0 (predicate output): 报告 ‘addr 是否落在合法 global range 且 pin_pred=true’; OOB lane 不写 dst, 后续指令用 @pout0 gate 处理 normal vs OOB 分支 (常用于 tile boundary handling)。

地址模型: base reg 64-bit pair (r / r+1 偶对齐), offset reg 32-bit signed, imm 24-bit signed (比 generic ld 的 32-bit imm 短一段, 因为 ldg 多 1 个 sreg base operand 占 encoding 空间)。

ldst_length: u8 / s8 / u16 / s16 / b32 / b64 / b128 / b256 共 8 valid。b256 (256-bit, 8 vreg) 是 ML kernel GEMM 主路径 (256-bit packed load 减半 instruction count), 写 dst..dst+7 共 8 个 vreg, 仅 ldg / stg 支持。

mem_sem: constant / weak (默认) / strong / mmio。

consistency_scope: cta / sm / cluster / gpu / sys。

cache_evict: 跟 ld 同源。

典型用例: ML kernel 主路径 (编译期确定地址在 global) / 编译期已知 global 时用 ldg 替代 generic ld 省 tag check。

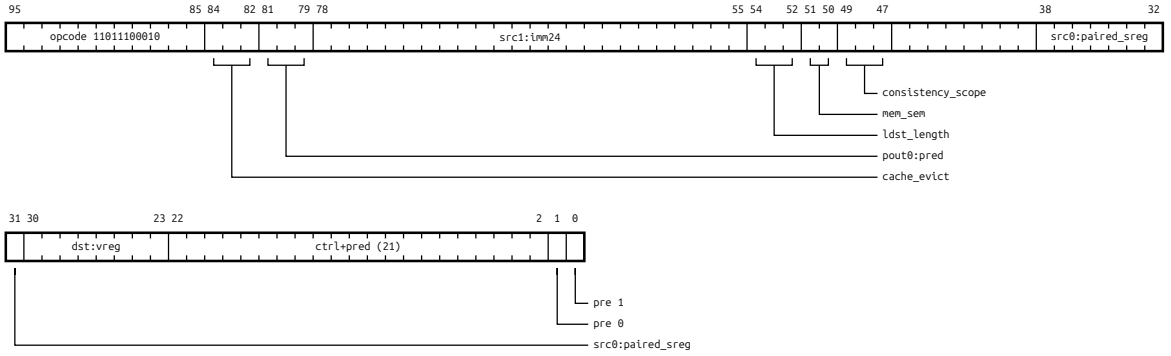
Leaf: ldg__vp_xi

Format:

```
ldg.ldst_length{.mem_sem}.consistency_scope.cache_evict  dst, pout0, src0, src1
```

Operands: dst: vreg [notes: span=default=1; ldst_length=b64→2; ldst_length=b128→4; ldst_length=b256→8, align=default=1; ldst_length=b64→2; ldst_length=b128→4; ldst_length=b256→8]; pout0: pred; src0: paired_sreg; src1: imm24s

Encoding Diagram: 96b, prefix 10, opcode 11011100010



Notes:

Explicit global memory load: dst = *(addr) if pout0, pout0 = (addr_in_bounds && pin_pred)。整 warp 32 个 active lane 各自独立读 per-lane 地址, 跳过 generic ld 的 [63:62] tag 检查 (编译期已知 global), 默认 mem_sem=weak 走更快路径。

pout0 (predicate output): 报告 ‘addr 是否落在合法 global range 且 pin_pred=true’; OOB lane 不写 dst, 后续指令用 @pout0 gate 处理 normal vs OOB 分支 (常用于 tile boundary handling)。

地址模型: base reg 64-bit pair (r / r+1 偶对齐), offset reg 32-bit signed, imm 24-bit signed (比 generic ld 的 32-bit imm 短一段, 因为 ldg 多 1 个 sreg base operand 占 encoding 空间)。

ldst_length: u8 / s8 / u16 / s16 / b32 / b64 / b128 / b256 共 8 valid。b256 (256-bit, 8 vreg) 是 ML kernel GEMM 主路径 (256-bit packed load 减半 instruction count), 写 dst..dst+7 共 8 个 vreg, 仅 ldg / stg 支持。

mem_sem: constant / weak (默认) / strong / mmio。

consistency_scope: cta / sm / cluster / gpu / sys。

cache_evict: 跟 ld 同源。

典型用例: ML kernel 主路径 (编译期确定地址在 global) / 编译期已知 global 时用 ldg 替代 generic ld 省 tag check。

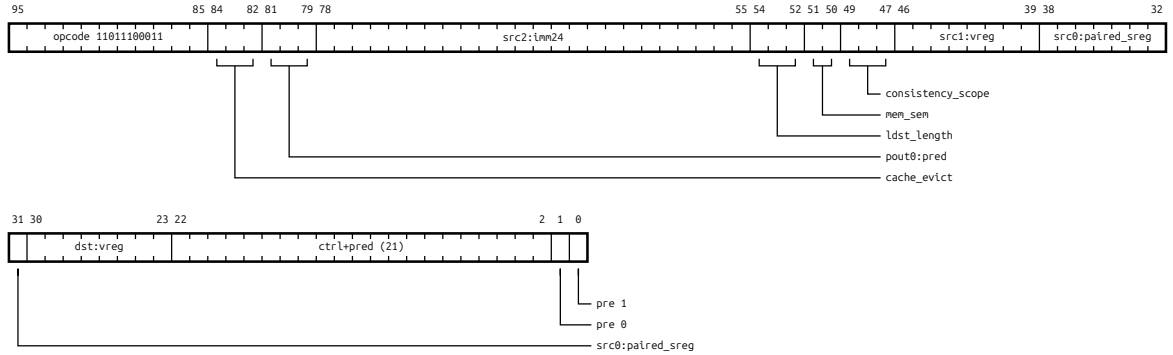
Leaf: ldg__vp_xvi

Format:

```
ldg.ldst_length{.mem_sem}.consistency_scope.cache_evict  dst, pout0, src0, src1, src2
```

Operands: dst: vreg [notes: span=default=1; ldst_length=b64→2; ldst_length=b128→4; ldst_length=b256→8, align=default=1; ldst_length=b64→2; ldst_length=b128→4; ldst_length=b256→8]; pout0: pred; src0: paired_sreg; src1: vreg; src2: imm24s

Encoding Diagram: 96b, prefix 10, opcode 11011100011



Notes:

Explicit global memory load: dst = *(addr) if pout0, pout0 = (addr_in_bounds && pin_pred)。整 warp 32 个 active lane 各自独立读 per-lane 地址, 跳过 generic ld 的 [63:62] tag 检查 (编译期已知 global), 默认 mem_sem=weak 走更快路径。

pout0 (predicate output): 报告 'addr 是否落在合法 global range 且 pin_pred=true'; OOB lane 不写 dst, 后续指令用 @pout0 gate 处理 normal vs OOB 分支 (常用于 tile boundary handling)。

地址模型: base reg 64-bit pair (r / r+1 偶对齐), offset reg 32-bit signed, imm 24-bit signed (比 generic ld 的 32-bit imm 短一段, 因为 ldg 多 1 个 sreg base operand 占 encoding 空间)。

ldst_length: u8 / s8 / u16 / s16 / b32 / b64 / b128 / b256 共 8 valid。b256 (256-bit, 8 vreg) 是 ML kernel GEMM 主路径 (256-bit packed load 减半 instruction count), 写 dst..dst+7 共 8 个 vreg, 仅 ldg / stg 支持。

mem_sem: constant / weak (默认) / strong / mmio。

consistency_scope: cta / sm / cluster / gpu / sys。

cache_evict: 跟 ld 同源。

典型用例: ML kernel 主路径 (编译期确定地址在 global) / 编译期已知 global 时用 ldg 替代 generic ld 省 tag check。

ldl **category**: Memory load/store **predicate_mode**: simt

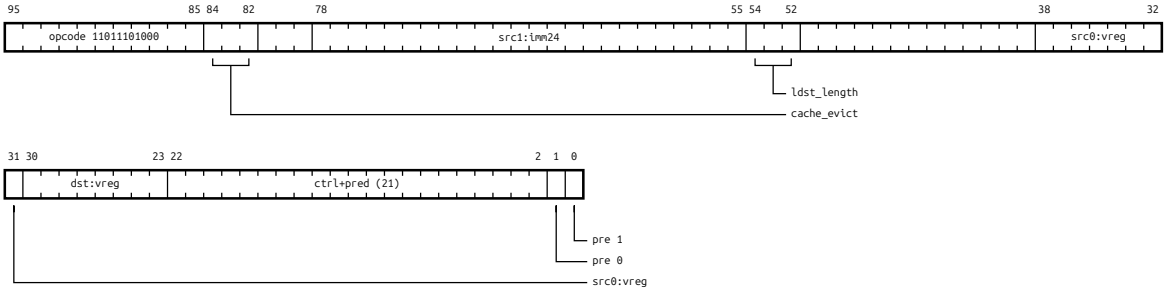
Leaf: ldl__v_vi

Format:

```
ldl.ldst_length.cache_evict  dst, src0, src1
```

Operands: dst: vreg [notes: span=default=1; ldst_length=b64→2; ldst_length=b128→4, align=default=1; ldst_length=b64→2; ldst_length=b128→4]; src0: vreg; src1: imm24s

Encoding Diagram: 96b, prefix 10, opcode 11011101000



Notes:

Explicit local memory load: dst = local[vaddr + offset], 整 warp 32 个 active lane 各自独立读 per-thread 私有 local memory。local memory 是 per-lane 私有的内存区 (lane 之间互不可见), 主要用于 register spill + 大型 per-thread 数组 (lane-private array)。

物理实现: lane i 的 local memory 落在 global memory + l2 cache 路径上, 硬件按 stride = lane_id 把同 warp 32 lane 的相同 local 地址散布到 32 个不同 global 物理位置 (保证 per-lane 私有 + coalesced 访问)。

地址模型: vaddr 32-bit per-thread 相对地址 (local memory per-thread 典型 KB 级 spill 区, 32-bit 足够), 不是 64-bit 全地址。effective_address = vaddr + offset, offset imm.24 signed; lane_id 由硬件隐式参与展开, ISA 层不暴露。

对齐: 按 ldst_length 自然对齐 (b32 4-byte / b64 8-byte / b128 16-byte), 未对齐 trap。

ldst_length: ≤ b128 (跟 lds 同源, 无 b256)。

modifier: ldst_length + cache_evict (走 l2 cache 路径所以挂 cache_evict, 但不挂 mem_sem / consistency_scope, local 是 per-thread 私有不需跨 thread ordering)。

典型用例: 寄存器 spill / reload (寄存器压力大时编译器把变量溢出到 local) / 大型 per-thread 数组 (declaration-time-known size 数组分配在 local 而不是 reg) / 函数调用 stack frame (调用约定中临时变量的存储)。

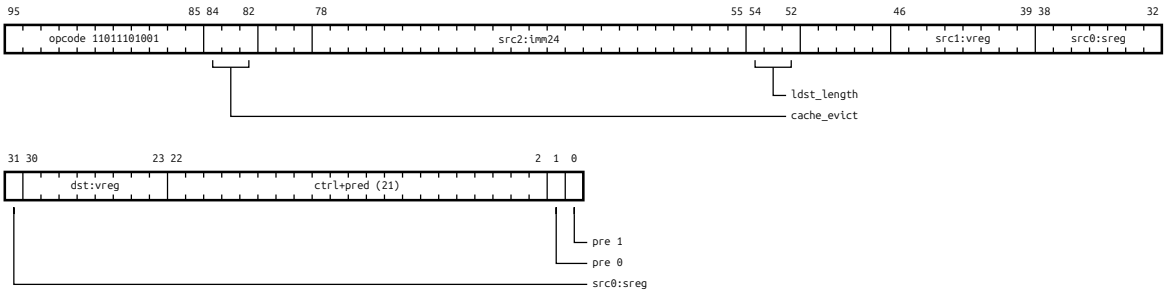
Leaf: ldl__v_xvi

Format:

```
ldl.ldst_length.cache_evict  dst, src0, src1, src2
```

Operands: dst: vreg [notes: span=default=1; ldst_length=b64→2; ldst_length=b128→4, align=default=1; ldst_length=b64→2; ldst_length=b128→4]; src0: sreg; src1: vreg; src2: imm24s

Encoding Diagram: 96b, prefix 10, opcode 11011101001



Notes:

Explicit local memory load: dst = local[vaddr + offset], 整 warp 32 个 active lane 各自独立读 per-thread 私有 local memory。local memory 是 per-lane 私有的内存区 (lane 之间互不可见), 主要用于 register spill + 大型 per-thread 数组 (lane-private array)。

物理实现: lane i 的 local memory 落在 global memory + l2 cache 路径上, 硬件按 stride = lane_id 把同 warp 32 lane 的相同 local 地址散布到 32 个不同 global 物理位置 (保证 per-lane 私有 + coalesced 访问)。

地址模型: vaddr 32-bit per-thread 相对地址 (local memory per-thread 典型 KB 级 spill 区, 32-bit 足够), 不是 64-bit 全地址。effective_address = vaddr + offset, offset imm.24 signed; lane_id 由硬件隐式参与展开, ISA 层不暴露。

对齐: 按 ldst_length 自然对齐 (b32 4-byte / b64 8-byte / b128 16-byte), 未对齐 trap。

ldst_length: ≤ b128 (跟 lds 同源, 无 b256)。

modifier: ldst_length + cache_evict (走 l2 cache 路径所以挂 cache_evict, 但不挂 mem_sem / consistency_scope, local 是 per-thread 私有不需跨 thread ordering)。

典型用例: 寄存器 spill / reload (寄存器压力大时编译器把变量溢出到 local) / 大型 per-thread 数组 (declaration-time-known size 数组分配在 local 而不是 reg) / 函数调用 stack frame (调用约定中临时变量的存储)。

lds **category**: Memory load/store **predicate_mode**: simt

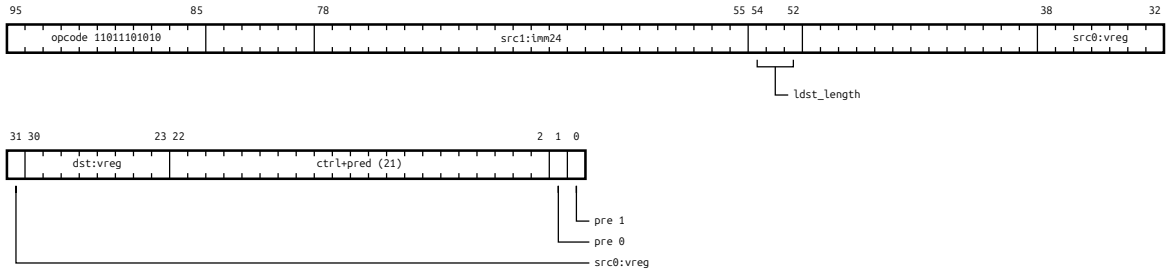
Leaf: lds__v_vi

Format:

```
lds.ldst_length  dst, src0, src1
```

Operands: dst: vreg [notes: span=default=1; ldst_length=b64→2; ldst_length=b128→4, align=default=1; ldst_length=b64→2; ldst_length=b128→4]; src0: vreg; src1: imm24s

Encoding Diagram: 96b, prefix 10, opcode 11011101010



Notes:

Explicit shared memory load: dst = shared[vaddr + offset], 整 warp 32 个 active lane 各自独立读。shared memory 是 sm 内部的 cta-local scratchpad SRAM, per-cta 分配, cta 内所有 warp 共享。跳过 generic ld 的 [63:62] tag 检查 (编译期已知 shared)。

地址模型: vaddr 是 32-bit cta-相对地址 (不是 64-bit 全地址, shared memory 容量小远不及 4GB)。effective_address = vaddr + offset, offset 是 imm.24 signed。base reg 跟 ldg 的 64-bit pair 不同, 仅占 1 个 sreg / vreg。

Banking 语义: shared memory 物理分 32 个 bank, 每 bank 4 byte 宽, 连续 32 个 4-byte word 散布到 32 bank (bank_id = (addr » 2) & 31)。同 warp 多 lane 同 bank 不同 word 触发 bank conflict (硬件序列化), 同 bank 同 word 触发 broadcast (硬件并行)。bank conflict 处理硬件透明, ISA 不暴露 modifier。

对齐: 按 ldst_length 自然对齐 (b32 4-byte / b64 8-byte / b128 16-byte), 未对齐 trap。

ldst_length: shared 路径仅 ≤ b128 (无 b256), 对应 sm scratchpad 单次访问宽度上限。

lds 仅 ldst_length modifier (不挂 mem_sem / consistency_scope / cache_evict, shared SRAM 不走 cache 路径也不需 cross-cta scope)。

典型用例: shared 数组读取 + tile reuse (GEMM 主路径用 shared 暂存 tile 反复读) / warp shuffle 序列代偿 / warp 间同步前的数据交换。

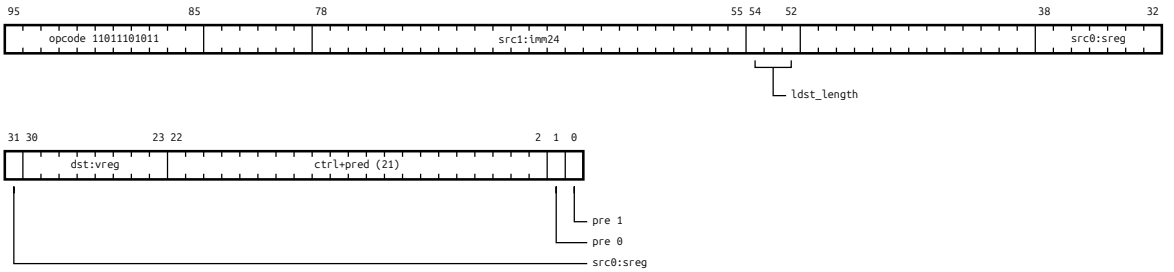
Leaf: lds__v_xi

Format:

```
lds.ldst_length  dst, src0, src1
```

Operands: dst: vreg [notes: span=default=1; ldst_length=b64→2; ldst_length=b128→4, align=default=1; ldst_length=b64→2; ldst_length=b128→4]; src0: sreg; src1: imm24s

Encoding Diagram: 96b, prefix 10, opcode 11011101011



Notes:

Explicit shared memory load: dst = shared[vaddr + offset], 整 warp 32 个 active lane 各自独立读。shared memory 是 sm 内部的 cta-local scratchpad SRAM, per-cta 分配, cta 内所有 warp 共享。跳过 generic ld 的 [63:62] tag 检查 (编译期已知 shared)。

地址模型: vaddr 是 32-bit cta-相对地址 (不是 64-bit 全地址, shared memory 容量小远不及 4GB)。effective_address = vaddr + offset, offset 是 imm.24 signed。base reg 跟 ldg 的 64-bit pair 不同, 仅占 1 个 sreg / vreg。

Banking 语义: shared memory 物理分 32 个 bank, 每 bank 4 byte 宽, 连续 32 个 4-byte word 散布到 32 bank (bank_id = (addr » 2) & 31)。同 warp 多 lane 同 bank 不同 word 触发 bank conflict (硬件序列化), 同 bank 同 word 触发 broadcast (硬件并行)。bank conflict 处理硬件透明, ISA 不暴露 modifier。

对齐: 按 ldst_length 自然对齐 (b32 4-byte / b64 8-byte / b128 16-byte), 未对齐 trap。

ldst_length: shared 路径仅 ≤ b128 (无 b256), 对应 sm scratchpad 单次访问宽度上限。

lds 仅 ldst_length modifier (不挂 mem_sem / consistency_scope / cache_evict, shared SRAM 不走 cache 路径也不需 cross-cta scope)。

典型用例: shared 数组读取 + tile reuse (GEMM 主路径用 shared 暂存 tile 反复读) / warp shuffle 序列代偿 / warp 间同步前的数据交换。

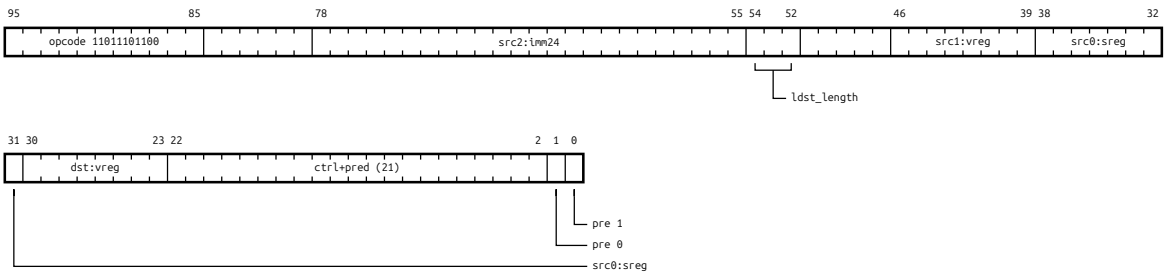
Leaf: lds__v_xvi

Format:

```
lds.ldst_length  dst, src0, src1, src2
```

Operands: dst: vreg [notes: span=default=1; ldst_length=b64→2; ldst_length=b128→4, align=default=1; ldst_length=b64→2; ldst_length=b128→4]; src0: sreg; src1: vreg; src2: imm24s

Encoding Diagram: 96b, prefix 10, opcode 11011101100



Notes:

Explicit shared memory load: dst = shared[vaddr + offset], 整 warp 32 个 active lane 各自独立读。shared memory 是 sm 内部的 cta-local scratchpad SRAM, per-cta 分配, cta 内所有 warp 共享。跳过 generic ld 的 [63:62]

tag 检查 (编译期已知 shared)。

地址模型: vaddr 是 32-bit cta-相对地址 (不是 64-bit 全地址, shared memory 容量小远不及 4GB)。effective_-address = vaddr + offset, offset 是 imm.24 signed。base reg 跟 ldg 的 64-bit pair 不同, 仅占 1 个 sreg / vreg。

Banking 语义: shared memory 物理分 32 个 bank, 每 bank 4 byte 宽, 连续 32 个 4-byte word 散布到 32 bank (bank_id = (addr » 2) & 31)。同 warp 多 lane 同 bank 不同 word 触发 bank conflict (硬件序列化), 同 bank 同 word 触发 broadcast (硬件并行)。bank conflict 处理硬件透明, ISA 不暴露 modifier。

对齐: 按 ldst_length 自然对齐 (b32 4-byte / b64 8-byte / b128 16-byte), 未对齐 trap。

ldst_length: shared 路径仅 ≤ b128 (无 b256), 对应 sm scratchpad 单次访问宽度上限。

lds 仅 ldst_length modifier (不挂 mem_sem / consistency_scope / cache_evict, shared SRAM 不走 cache 路径也不需 cross-cta scope)。

典型用例: shared 数组读取 + tile reuse (GEMM 主路径用 shared 暂存 tile 反复读) / warp shuffle 序列代偿 / warp 间同步前的数据交换。

ldsm **category**: Memory load/store **predicate_mode**: simt

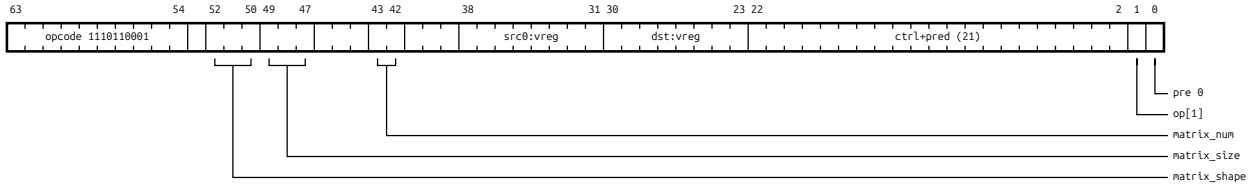
Leaf: ldsm__v_v

Format:

```
ldsm.matrix_shape.matrix_size.matrix_num  dst, src0
```

Operands: dst: vreg; src0: vreg

Encoding Diagram: 64b, prefix 0, opcode 11110110001



Notes:

Shared matrix load: 从 shared memory 协同加载矩阵 tile 到整 warp 32 个 lane 的 vreg(s) — 32 lane 各自带一个 row 起始地址, 硬件按 matrix_shape 指定的 tile 布局把数据分发到 dst, 每 lane 收到 32 / 64 / 96 / 128 bit 的 tile 切片。是 MMA tile load 的关键, 把 shared tile 一次摆成 hmma / imma / qmma 期望的 register layout, 不需要每 lane 单独 lds + permute。

matrix_shape 决定 tile 布局: m88 (8×8) / mt1616 (16×16 transposed) 等; 不同 shape 对应 hmma / imma 不同 K 路径。

matrix_size: 元素位宽 — b16 / b8 / u4to8 / u2to4 等; sub-byte 路径 (4-bit / 2-bit packed) 用于 FP4 / FP6 / INT4 / INT2 量化推理。

matrix_num 决定一次 load 几个矩阵 (1 / 2 / 4), 跟 matrix_shape 联合决定 dst 占用的 vreg 数。例: m88 × num=1 → 32-bit/lane (1 vreg), m88 × num=2 → 64-bit/lane (vreg pair), m88 × num=4 → 128-bit/lane (4 vreg), mt1616 × num=1 → 64-bit/lane, mt1616 × num=2 → 128-bit/lane。

典型用例: MMA kernel inner loop (attention / GEMM 在 inner loop 用 ldsm 把 K / V tile 从 shared 摆到 reg, 直接喂给 hmma / imma) / FP4 / FP6 / FP8 量化推理 (m816 × u4to8 / u2to4 路径精确匹配 INT4 / INT2 packed tile)。

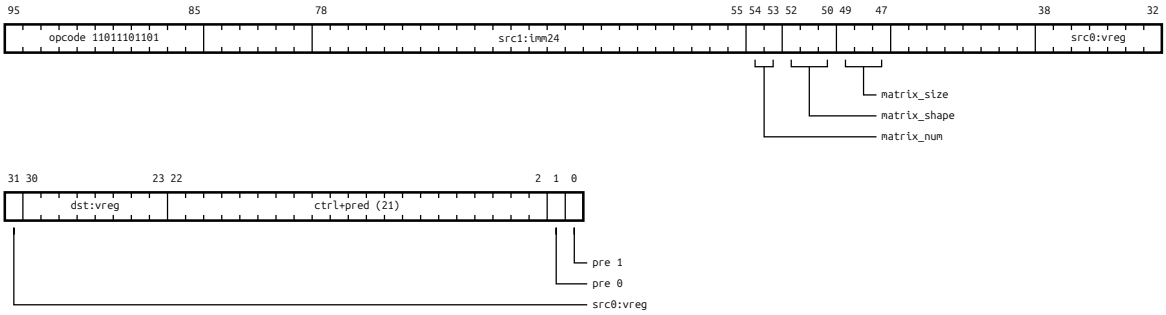
Leaf: ldsm__v_vi

Format:

```
ldsm.matrix_shape.matrix_size.matrix_num  dst, src0, src1
```

Operands: dst: vreg; src0: vreg; src1: imm24s

Encoding Diagram: 96b, prefix 10, opcode 11011101101



Notes:

Shared matrix load: 从 shared memory 协同加载矩阵 tile 到整 warp 32 个 lane 的 vreg(s) — 32 lane 各自带一个 row 起始地址, 硬件按 matrix_shape 指定的 tile 布局把数据分发到 dst, 每 lane 收到 32 / 64 / 96 / 128 bit 的 tile 切片。是 MMA tile load 的关键, 把 shared tile 一次摆成 hmma / imma / qmma 期望的 register layout, 不需要每 lane 单独 lds + permute。

matrix_shape 决定 tile 布局: m88 (8×8) / mt1616 (16×16 transposed) 等; 不同 shape 对应 hmma / imma 不同 K 路径。

matrix_size: 元素位宽 — b16 / b8 / u4to8 / u2to4 等; sub-byte 路径 (4-bit / 2-bit packed) 用于 FP4 / FP6 / INT4 / INT2 量化推理。

matrix_num 决定一次 load 几个矩阵 (1 / 2 / 4), 跟 matrix_shape 联合决定 dst 占用的 vreg 数。例: m88 × num=1 → 32-bit/lane (1 vreg), m88 × num=2 → 64-bit/lane (vreg pair), m88 × num=4 → 128-bit/lane (4 vreg), mt1616 × num=1 → 64-bit/lane, mt1616 × num=2 → 128-bit/lane。

典型用例: MMA kernel inner loop (attention / GEMM 在 inner loop 用 ldsm 把 K / V tile 从 shared 摆到 reg, 直接喂给 hmma / imma) / FP4 / FP6 / FP8 量化推理 (m816 × u4to8 / u2to4 路径精确匹配 INT4 / INT2 packed tile)。

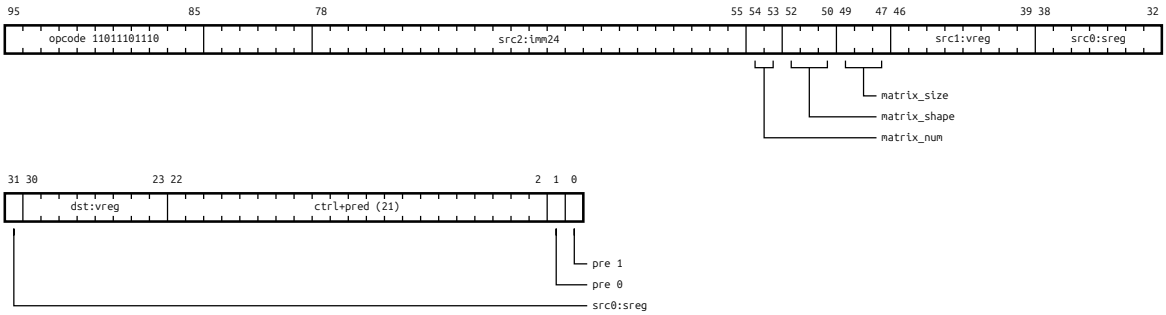
Leaf: ldsm__v_xvi

Format:

```
ldsm.matrix_shape.matrix_size.matrix_num  dst, src0, src1, src2
```

Operands: dst: vreg; src0: sreg; src1: vreg; src2: imm24s

Encoding Diagram: 96b, prefix 10, opcode 11011101110



Notes:

Shared matrix load: 从 shared memory 协同加载矩阵 tile 到整 warp 32 个 lane 的 vreg(s) — 32 lane 各自带一个 row 起始地址, 硬件按 matrix_shape 指定的 tile 布局把数据分发到 dst, 每 lane 收到 32 / 64 / 96 / 128 bit

的 tile 切片。是 MMA tile load 的关键, 把 shared tile 一次摆成 hmma / imma / qmma 期望的 register layout, 不需要每 lane 单独 lds + permute。

matrix_shape 决定 tile 布局: m88 (8×8) / mt1616 (16×16 transposed) 等; 不同 shape 对应 hmma / imma 不同 K 路径。

matrix_size: 元素位宽 — b16 / b8 / u4to8 / u2to4 等; sub-byte 路径 (4-bit / 2-bit packed) 用于 FP4 / FP6 / INT4 / INT2 量化推理。

matrix_num 决定一次 load 几个矩阵 (1 / 2 / 4), 跟 matrix_shape 联合决定 dst 占用的 vreg 数。例: m88 × num=1 → 32-bit/lane (1 vreg), m88 × num=2 → 64-bit/lane (vreg pair), m88 × num=4 → 128-bit/lane (4 vreg), mt1616 × num=1 → 64-bit/lane, mt1616 × num=2 → 128-bit/lane。

典型用例: MMA kernel inner loop (attention / GEMM 在 inner loop 用 ldsm 把 K / V tile 从 shared 摆到 reg, 直接喂给 hmma / imma) / FP4 / FP6 / FP8 量化推理 (m816 × u4to8 / u2to4 路径精确匹配 INT4 / INT2 packed tile)。

st **category**: Memory load/store **predicate_mode**: simt

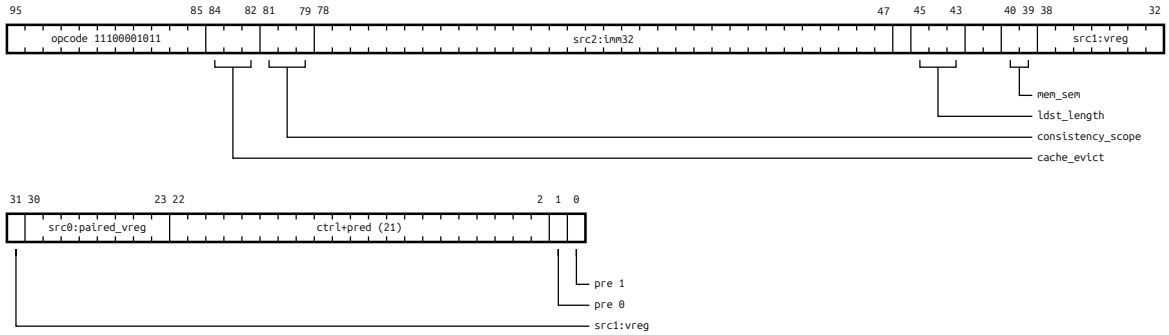
Leaf: st___vvi

Format:

```
st.ldst_length{.mem_sem}.consistency_scope.cache_evict  src0, src1, src2
```

Operands: src0: paired_vreg; src1: vreg [notes: span=default=1; ldst_length=b64→2; ldst_length=b128→4, align=default=1; ldst_length=b64→2; ldst_length=b128→4]; src2: imm32s

Encoding Diagram: 96b, prefix 10, opcode 11100001011



Notes:

Generic flat address space store: *(effective_address) = data, 整 warp 32 个 active lane 各自独立写到 per-lane 地址。跟 ld 完全镜像 (方向相反): ld 是 dst = *(addr), st 是 *(addr) = data; data 是 src input, 没有 reg dst slot。

地址路由 / 对齐跟 ld 同源: addr[63:62] tag 决定 global / local / shared / constant 四个空间之一, 自然对齐 trap。同 warp 同空间是正式架构约束, 违反 trap; 约束全文与编译器契约见 ld 条目。

ldst_length: u8 / s8 / u16 / s16 / b32 / b64 / b128 共 7 valid (b256 仅 stg 支持)。

mem_sem: weak (默认) / strong / mmio; constant memory 只读不支持 store, mem_sem=constant 配 st 是非法组合 (硬件检测 addr[63:62]=11 + st = trap)。

consistency_scope: cta / sm / cluster / gpu / sys。

cache_evict: 跟 ld 同源。

编译期已知 global 用 stg, 已知 shared 用 sts (省 tag + bank-aware 优化), 已知 local 用 stl; 编译期不知用 generic st。典型用例: C++ generic pointer 赋值 (*p = v 编译成 st 因 p 可能指向任何 space) / 跨地址空间 wrapper write-back。

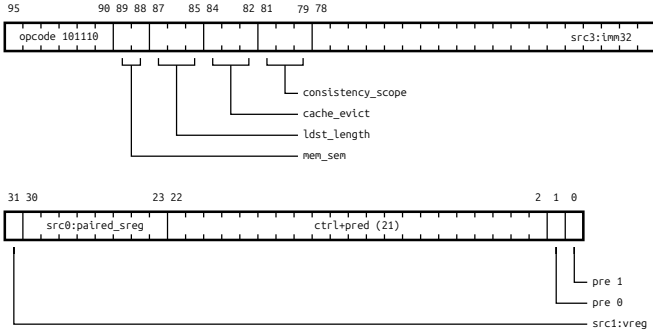
Leaf: st___xvvi

Format:

```
st.ldst_length{.mem_sem}.consistency_scope.cache_evict  src0, src1, src2, src3
```

Operands: src0: paired_sreg; src1: vreg; src2: vreg [notes: span=default=1; ldst_length=b64→2; ldst_length=b128→4, align=default=1; ldst_length=b64→2; ldst_length=b128→4]; src3: imm32s

Encoding Diagram: 96b, prefix 10, opcode 101110



Notes:

Generic flat address space store: *(effective_address) = data, 整 warp 32 个 active lane 各自独立写到 per-lane 地址。跟 ld 完全镜像 (方向相反): ld 是 dst = *(addr), st 是 *(addr) = data; data 是 src input, 没有 reg dst slot。

地址路由 / 对齐跟 ld 同源: addr[63:62] tag 决定 global / local / shared / constant 四个空间之一, 自然对齐 trap。同 warp 同空间是正式架构约束, 违反 trap; 约束全文与编译器契约见 ld 条目。

ldst_length: u8 / s8 / u16 / s16 / b32 / b64 / b128 共 7 valid (b256 仅 stg 支持)。

mem_sem: weak (默认) / strong / mmio; constant memory 只读不支持 store, mem_sem=constant 配 st 是非法组合 (硬件检测 addr[63:62]=11 + st = trap)。

consistency_scope: cta / sm / cluster / gpu / sys。

cache_evict: 跟 ld 同源。

编译期已知 global 用 stg, 已知 shared 用 sts (省 tag + bank-aware 优化), 已知 local 用 stl; 编译期不知用 generic st。典型用例: C++ generic pointer 赋值 (*p = v 编译成 st 因 p 可能指向任何 space) / 跨地址空间 wrapper write-back。

stg **category**: Memory load/store **predicate_mode**: simt

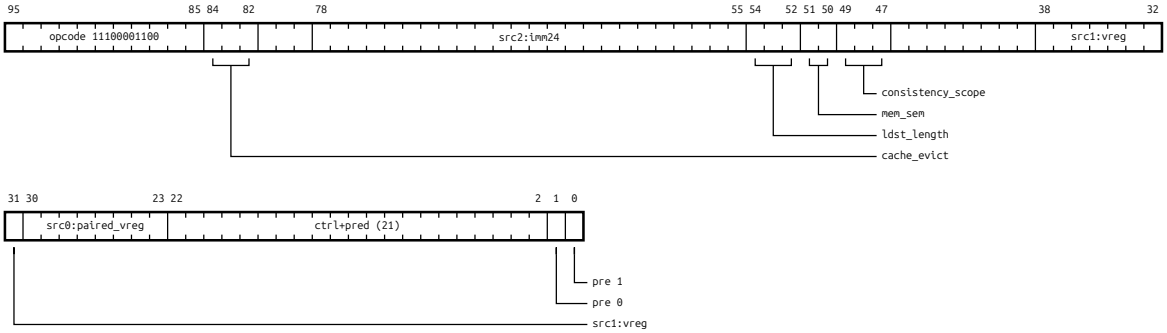
Leaf: stg___vvi

Format:

```
stg.ldst_length{.mem_sem}.consistency_scope.cache_evict  src0, src1, src2
```

Operands: src0: paired_vreg; src1: vreg [notes: span=default=1; ldst_length=b64→2; ldst_length=b128→4; ldst_length=b256→8, align=default=1; ldst_length=b64→2; ldst_length=b128→4; ldst_length=b256→8]; src2: imm24s

Encoding Diagram: 96b, prefix 10, opcode 11100001100



Notes:

Explicit global memory store: *(effective_address) = data, 整 warp 32 个 active lane 各自独立写 per-lane 地址。跳过 generic st 的 [63:62] tag 检查 (编译期已知 global), 性能比 generic st 好。

stg 没有 pout0 predicate output (跟 ldg 不对偶) — store 本身不需要报告 OOB 给后续 dataflow, 编译器在 stg 之前用 isetp + @P 控制 OOB lane 不写。

地址模型: base reg 64-bit pair, offset reg 32-bit signed, imm 24-bit signed (跟 ldg 一致)。

ldst_length: 8 valid 含 b256 (256-bit packed store, 主路径 ML kernel epilogue 写 tile)。

mem_sem: weak (默认) / strong / mmio; constant memory 只读 mem_sem=constant 配 stg 非法。

consistency_scope: cta / sm / cluster / gpu / sys。

cache_evict: 跟 stg 同源。

典型用例: ML kernel 主路径 store / output tile write / matrix store back to DRAM。

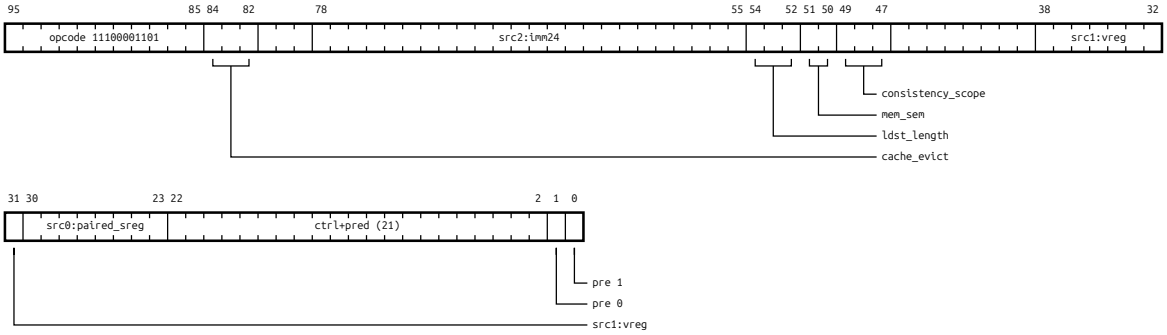
Leaf: stg___xvi

Format:

```
stg.ldst_length{.mem_sem}.consistency_scope.cache_evict  src0, src1, src2
```

Operands: src0: paired_sreg; src1: vreg [notes: span=default=1; ldst_length=b64→2; ldst_length=b128→4; ldst_length=b256→8, align=default=1; ldst_length=b64→2; ldst_length=b128→4; ldst_length=b256→8]; src2: imm24s

Encoding Diagram: 96b, prefix 10, opcode 11100001101



Notes:

Explicit global memory store: *(effective_address) = data, 整 warp 32 个 active lane 各自独立写 per-lane 地址。跳过 generic st 的 [63:62] tag 检查 (编译期已知 global), 性能比 generic st 好。

stg 没有 pout0 predicate output (跟 ldg 不对偶) — store 本身不需要报告 OOB 给后续 dataflow, 编译器在 stg 之前用 isetp + @P 控制 OOB lane 不写。

地址模型: base reg 64-bit pair, offset reg 32-bit signed, imm 24-bit signed (跟 ldg 一致)。

ldst_length: 8 valid 含 b256 (256-bit packed store, 主路径 ML kernel epilogue 写 tile)。

mem_sem: weak (默认) / strong / mmio; constant memory 只读 mem_sem=constant 配 stg 非法。

consistency_scope: cta / sm / cluster / gpu / sys。

cache_evict: 跟 stg 同源。

典型用例: ML kernel 主路径 store / output tile write / matrix store back to DRAM。

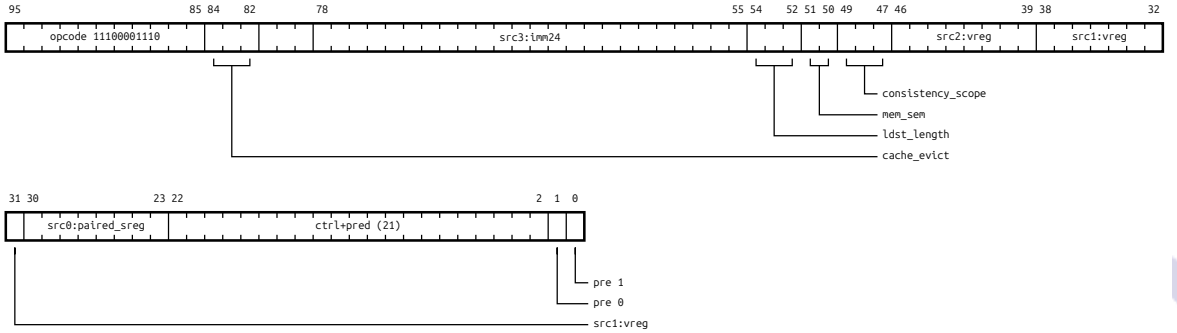
Leaf: stg___xvvi

Format:

```
stg.ldst_length{.mem_sem}.consistency_scope.cache_evict  src0, src1, src2, src3
```

Operands: src0: paired_sreg; src1: vreg; src2: vreg [notes: span=default=1; ldst_length=b64→2; ldst_length=b128→4; ldst_length=b256→8, align=default=1; ldst_length=b64→2; ldst_length=b128→4; ldst_length=b256→8]; src3: imm24s

Encoding Diagram: 96b, prefix 10, opcode 11100001110



Notes:

Explicit global memory store: *(effective_address) = data, 整 warp 32 个 active lane 各自独立写 per-lane 地址。跳过 generic st 的 [63:62] tag 检查 (编译期已知 global), 性能比 generic st 好。

stg 没有 pout0 predicate output (跟 ldg 不对偶) — store 本身不需要报告 OOB 给后续 dataflow, 编译器在 stg 之前用 isetp + @P 控制 OOB lane 不写。

地址模型: base reg 64-bit pair, offset reg 32-bit signed, imm 24-bit signed (跟 ldg 一致)。

ldst_length: 8 valid 含 b256 (256-bit packed store, 主路径 ML kernel epilogue 写 tile)。

mem_sem: weak (默认) / strong / mmio; constant memory 只读 mem_sem=constant 配 stg 非法。

consistency_scope: cta / sm / cluster / gpu / sys。

cache_evict: 跟 stg 同源。

典型用例: ML kernel 主路径 store / output tile write / matrix store back to DRAM。

stl **category**: Memory load/store **predicate_mode**: simt

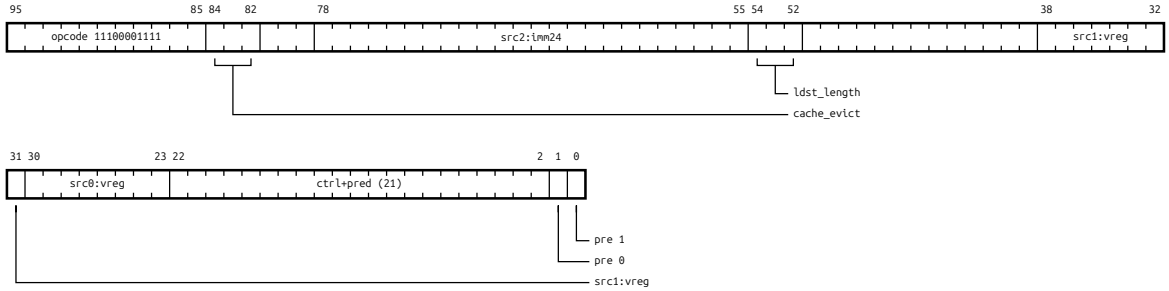
Leaf: stl___vvi

Format:

```
stl.ldst_length.cache_evict  src0, src1, src2
```

Operands: src0: vreg; src1: vreg [notes: span=default=1; ldst_length=b64→2; ldst_length=b128→4, align=default=1; ldst_length=b64→2; ldst_length=b128→4]; src2: imm24s

Encoding Diagram: 96b, prefix 10, opcode 11100001111



Notes:

Explicit local memory store: local[vaddr + offset] = data, ldl 的对偶 — per-thread 私有写入, lane i 的写不影响其他 lane 看到的 local memory。

地址模型 / 对齐 / lane_id 硬件隐式展开跟 ldl 同源 (vaddr 32-bit per-thread 相对地址, offset 24-bit signed imm)。

modifier: ldst_length (≤ b128) + cache_evict, 不挂 mem_sem / consistency_scope。

典型用例: 寄存器 spill 写回 / 大型 per-thread 数组写入 / 函数调用 stack frame 写入。

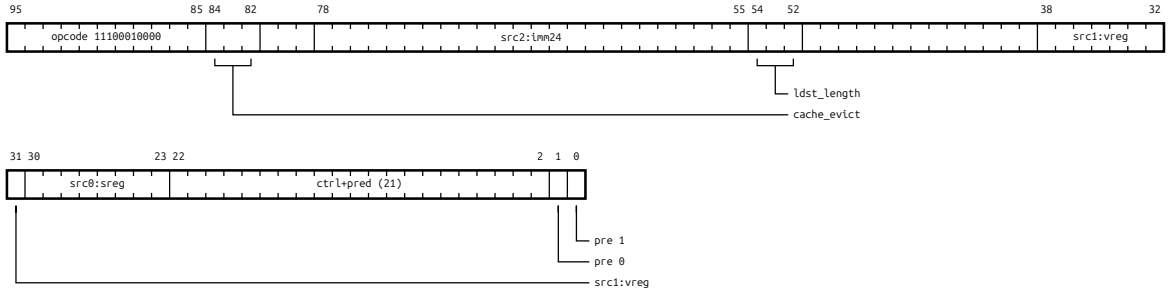
Leaf: stl___xvi

Format:

```
stl.ldst_length.cache_evict  src0, src1, src2
```

Operands: src0: sreg; src1: vreg [notes: span=default=1; ldst_length=b64→2; ldst_length=b128→4, align=default=1; ldst_length=b64→2; ldst_length=b128→4]; src2: imm24s

Encoding Diagram: 96b, prefix 10, opcode 11100010000



Notes:

Explicit local memory store: local[vaddr + offset] = data, ldl 的对偶 — per-thread 私有写入, lane i 的写不影响其他 lane 看到的 local memory。

地址模型 / 对齐 / lane_id 硬件隐式展开跟 ldl 同源 (vaddr 32-bit per-thread 相对地址, offset 24-bit signed imm)。

modifier: ldst_length (≤ b128) + cache_evict, 不挂 mem_sem / consistency_scope。

典型用例: 寄存器 spill 写回 / 大型 per-thread 数组写入 / 函数调用 stack frame 写入。

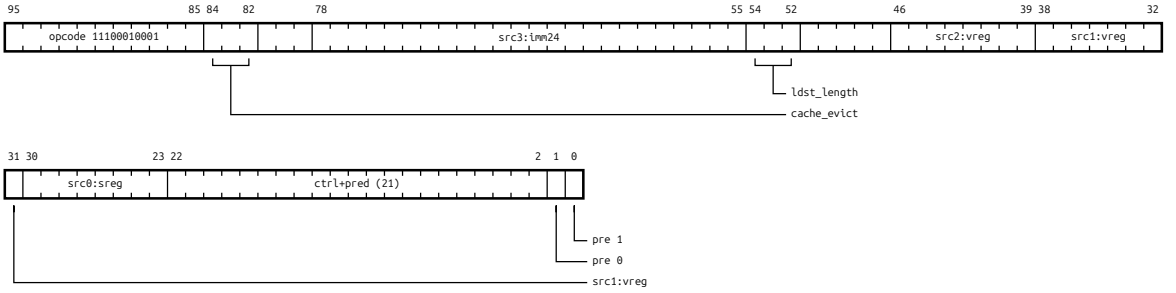
Leaf: stl___xvvi

Format:

```
stl.ldst_length.cache_evict  src0, src1, src2, src3
```

Operands: src0: sreg; src1: vreg; src2: vreg [notes: span=default=1; ldst_length=b64→2; ldst_length=b128→4, align=default=1; ldst_length=b64→2; ldst_length=b128→4]; src3: imm24s

Encoding Diagram: 96b, prefix 10, opcode 11100010001



Notes:

Explicit local memory store: local[vaddr + offset] = data, ldl 的对偶 — per-thread 私有写入, lane i 的写不影响其他 lane 看到的 local memory。

地址模型 / 对齐 / lane_id 硬件隐式展开跟 ldl 同源 (vaddr 32-bit per-thread 相对地址, offset 24-bit signed imm)。

modifier: ldst_length (≤ b128) + cache_evict, 不挂 mem_sem / consistency_scope。

典型用例: 寄存器 spill 写回 / 大型 per-thread 数组写入 / 函数调用 stack frame 写入。

sts **category**: Memory load/store **predicate_mode**: simt

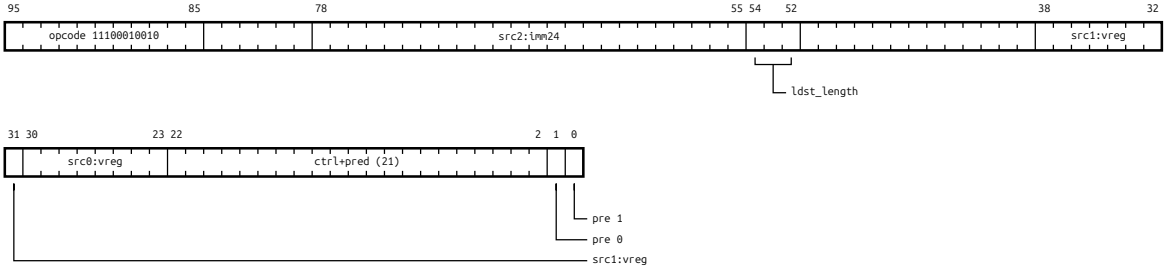
Leaf: sts___vvi

Format:

```
sts.ldst_length  src0, src1, src2
```

Operands: src0: vreg; src1: vreg [notes: span=default=1; ldst_length=b64→2; ldst_length=b128→4, align=default=1; ldst_length=b64→2; ldst_length=b128→4]; src2: imm24s

Encoding Diagram: 96b, prefix 10, opcode 11100010010



Notes:

Explicit shared memory store: shared[vaddr + offset] = data, 整 warp 32 个 active lane 各自独立写。lds 的对偶 — 编译期已知 shared 跳过 [63:62] tag 检查。

地址模型: vaddr 32-bit cta-相对地址, effective_address = vaddr + offset, offset 24-bit signed imm; banking + 对齐 + ldst_length 限制跟 lds 同源 (32 bank × 4 byte, 按 ldst_length 自然对齐 trap, ldst_length ≤ b128)。

modifier: 仅 ldst_length (跟 lds 同源)。

典型用例: shared 数组写入 / tile staging (从 global 读到 reg 经 sts 写到 shared 给同 cta 其他 warp 用) / warp shuffle 序列代偿。

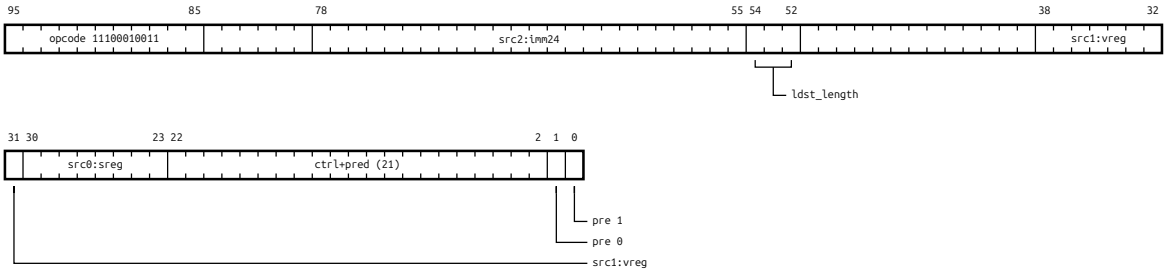
Leaf: sts___xvi

Format:

```
sts.ldst_length  src0, src1, src2
```

Operands: src0: sreg; src1: vreg [notes: span=default=1; ldst_length=b64→2; ldst_length=b128→4, align=default=1; ldst_length=b64→2; ldst_length=b128→4]; src2: imm24s

Encoding Diagram: 96b, prefix 10, opcode 11100010011



Notes:

Explicit shared memory store: shared[vaddr + offset] = data, 整 warp 32 个 active lane 各自独立写。lds 的对偶 — 编译期已知 shared 跳过 [63:62] tag 检查。

地址模型: vaddr 32-bit cta-相对地址, effective_address = vaddr + offset, offset 24-bit signed imm; banking + 对齐 + ldst_length 限制跟 lds 同源 (32 bank × 4 byte, 按 ldst_length 自然对齐 trap, ldst_length ≤ b128)。

modifier: 仅 ldst_length (跟 lds 同源)。

典型用例: shared 数组写入 / tile staging (从 global 读到 reg 经 sts 写到 shared 给同 cta 其他 warp 用) / warp shuffle 序列代偿。

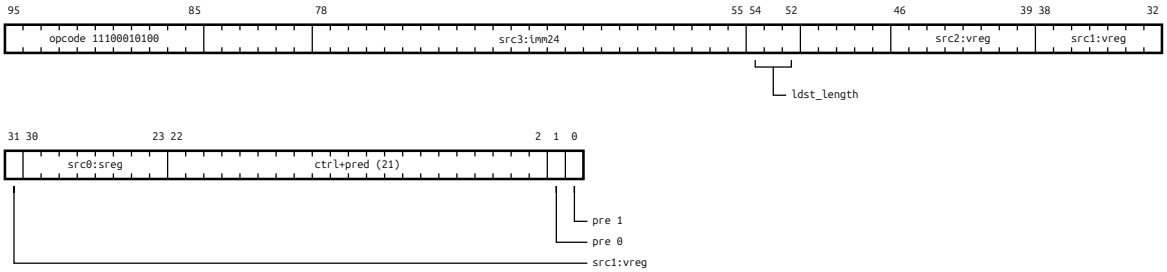
Leaf: sts___xvvi

Format:

```
sts.ldst_length  src0, src1, src2, src3
```

Operands: src0: sreg; src1: vreg; src2: vreg [notes: span=default=1; ldst_length=b64→2; ldst_length=b128→4, align=default=1; ldst_length=b64→2; ldst_length=b128→4]; src3: imm24s

Encoding Diagram: 96b, prefix 10, opcode 11100010100



Notes:

Explicit shared memory store: shared[vaddr + offset] = data, 整 warp 32 个 active lane 各自独立写。lds 的对偶 — 编译期已知 shared 跳过 [63:62] tag 检查。

地址模型: vaddr 32-bit cta-相对地址, effective_address = vaddr + offset, offset 24-bit signed imm; banking + 对齐 + ldst_length 限制跟 lds 同源 (32 bank × 4 byte, 按 ldst_length 自然对齐 trap, ldst_length ≤ b128)。

modifier: 仅 ldst_length (跟 lds 同源)。

典型用例: shared 数组写入 / tile staging (从 global 读到 reg 经 sts 写到 shared 给同 cta 其他 warp 用) / warp shuffle 序列代偿。

stsm **category**: Memory load/store **predicate_mode**: simt

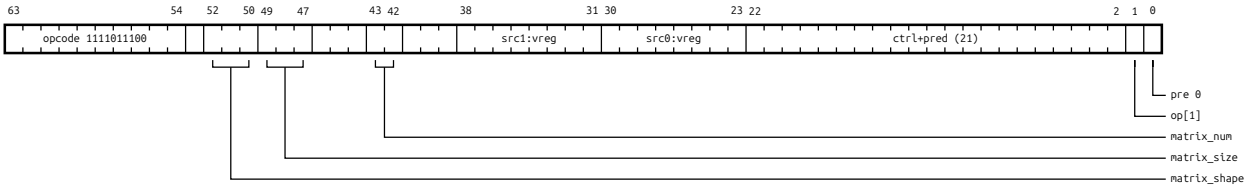
Leaf: stsm___vv

Format:

```
stsm.matrix_shape.matrix_size.matrix_num  src0, src1
```

Operands: src0: vreg; src1: vreg

Encoding Diagram: 64b, prefix 0, opcode 11111011100



Notes:

Shared matrix store: ldsm 的对偶 — warp 32 个 lane 协同把 vreg(s) 里的矩阵 tile 写回 shared memory, 按 matrix_shape 指定布局, matrix_size 指定元素位宽 / 子词 pack 模式, matrix_num 指定一次写几个矩阵。

matrix_shape / matrix_size / matrix_num 含义跟 ldsm 同源; 组合受限 (例: 某些 sz × num 组合硬件不支持), 走 prose 文档化, 越界硬件 trap。

典型用例: MMA epilogue (hmma / imma 算完 tile, stsm 写回 shared, 给下一轮 ldgsts / 异步大块 store 到 global) / 同 cta warp 间的矩阵传递。

src0 = shared memory 地址 (vreg), src1 = 被存矩阵数据 (vreg)。

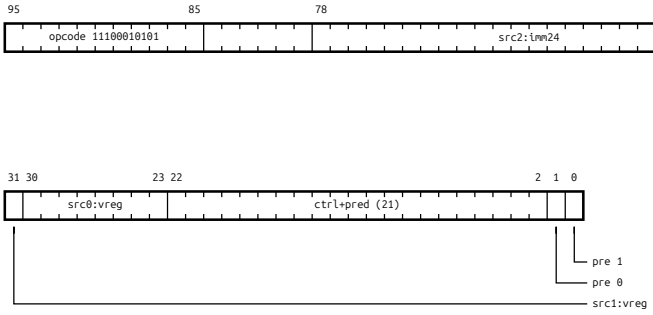
Leaf: stsm___vvi

Format:

```
stsm.matrix_shape.matrix_size.matrix_num  src0, src1, src2
```

Operands: src0: vreg; src1: vreg; src2: imm24s

Encoding Diagram: 96b, prefix 10, opcode 11100010101



Notes:

Shared matrix store: ldsm 的对偶 — warp 32 个 lane 协同把 vreg(s) 里的矩阵 tile 写回 shared memory, 按 matrix_shape 指定布局, matrix_size 指定元素位宽 / 子词 pack 模式, matrix_num 指定一次写几个矩阵。

matrix_shape / matrix_size / matrix_num 含义跟 ldsm 同源; 组合受限 (例: 某些 sz × num 组合硬件不支持), 走 prose 文档化, 越界硬件 trap。

典型用例: MMA epilogue (hmma / imma 算完 tile, stsm 写回 shared, 给下一轮 ldgsts / 异步大块 store 到 global) / 同 cta warp 间的矩阵传递。

src0 = shared memory 地址 (vreg), src1 = 被存矩阵数据 (vreg), src2 = 地址偏移 (imm24)。

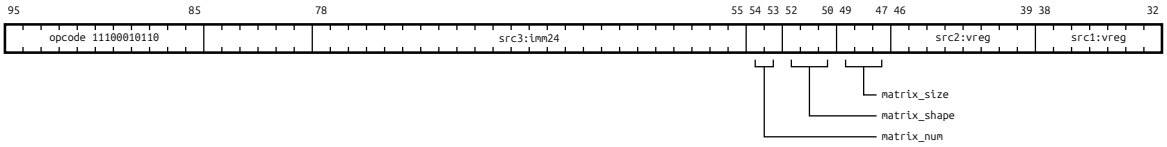
Leaf: stsm___xvvi

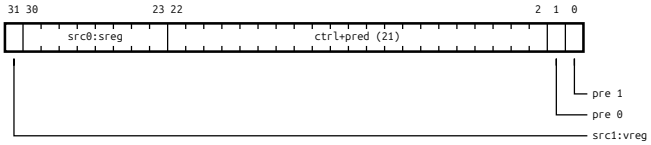
Format:

```
stsm.matrix_shape.matrix_size.matrix_num  src0, src1, src2, src3
```

Operands: src0: sreg; src1: vreg; src2: vreg; src3: imm24s

Encoding Diagram: 96b, prefix 10, opcode 11100010110





Notes:

Shared matrix store: ldsm 的对偶 — warp 32 个 lane 协同把 vreg(s) 里的矩阵 tile 写回 shared memory, 按 matrix_shape 指定布局, matrix_size 指定元素位宽 / 子词 pack 模式, matrix_num 指定一次写几个矩阵。

matrix_shape / matrix_size / matrix_num 含义跟 ldsm 同源; 组合受限 (例: 某些 sz × num 组合硬件不支持), 走 prose 文档化, 越界硬件 trap。

典型用例: MMA epilogue (hmma / imma 算完 tile, stsm 写回 shared, 给下一轮 ldgsts / 异步大块 store 到 global) / 同 cta warp 间的矩阵传递。

src0 = scalar 基地址 (sreg), src1 = shared memory 地址 (vreg), src2 = 被存矩阵数据 (vreg), src3 = 地址偏移 (imm24)。

ucctl　**category**: Memory load/store　**predicate_mode**: uniform

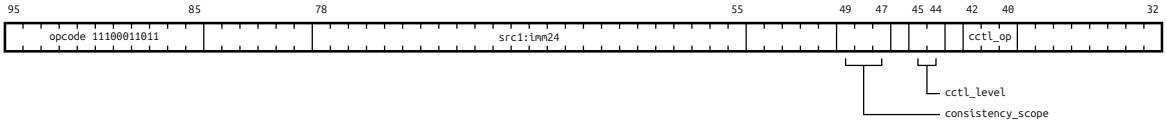
Leaf: ucctl___xi

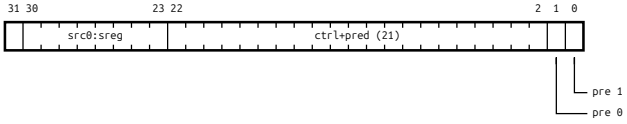
Format:

```
ucctl.cctl_op.cctl_level.consistency_scope  src0, src1
```

Operands: src0: sreg; src1: imm24s

Encoding Diagram: 96b, prefix 10, opcode 11100011011





Notes:

Uniform cache line control: cctl 的 uniform 镜像 — 两种指令形态路径。_xi 指令形态是 line-targeted uniform 路径 (saddr + imm.24), warp 集体对一条 cache line 操作; u 指令形态是 broadcast 路径 (无 operand), 整个 cache level 操作。

cctl_op 全集 7 valid: line-targeted (pf1 / pf2 / wb / iv / rs, 用 _xi 指令形态) + broadcast (ivall invalidate all entries / wball writeback all dirty entries, 用 u 指令形态)。simt cctl 不支持 broadcast — broadcast 路径只能走 ucctl。

cctl_level + consistency_scope 跟 cctl 同源。

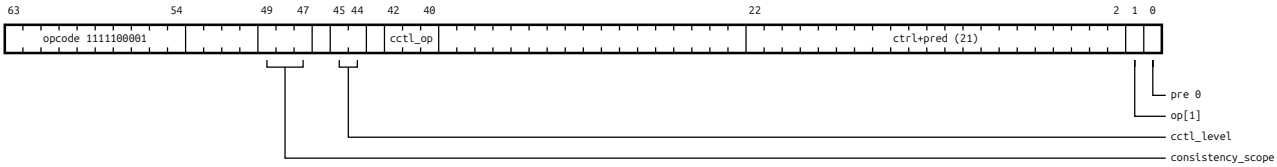
典型用例: kernel exit 前 ucctl wball 保 dirty line 落盘 (整 cache 一次回写) / 跨 kernel 边界 ucctl ivall 清 cache 避免读到旧数据 / warp-uniform 地址 prefetch 路径 (saddr + imm 地址全 warp 一致)。

Leaf: ucctl__u

Format:

```
ucctl.cctl_op.cctl_level.consistency_scope
```

Encoding Diagram: 64b, prefix 0, opcode 11111100001



Notes:

Uniform cache line control: cctl 的 uniform 镜像 — 两种指令形态路径。_xi 指令形态是 line-targeted uniform 路径 (saddr + imm.24), warp 集体对一条 cache line 操作; u 指令形态是 broadcast 路径 (无 operand), 整个 cache level 操作。

cctl_op 全集 7 valid: line-targeted (pf1 / pf2 / wb / iv / rs, 用 _xi 指令形态) + broadcast (ivall invalidate all entries / wball writeback all dirty entries, 用 u 指令形态)。simt cctl 不支持 broadcast — broadcast 路径只能走 ucctl。

cctl_level + consistency_scope 跟 cctl 同源。

典型用例: kernel exit 前 ucctl wball 保 dirty line 落盘 (整 cache 一次回写) / 跨 kernel 边界 ucctl ivall 清 cache 避免读到旧数据 / warp-uniform 地址 prefetch 路径 (saddr + imm 地址全 warp 一致)。

uld **category**: Memory load/store **predicate_mode**: uniform

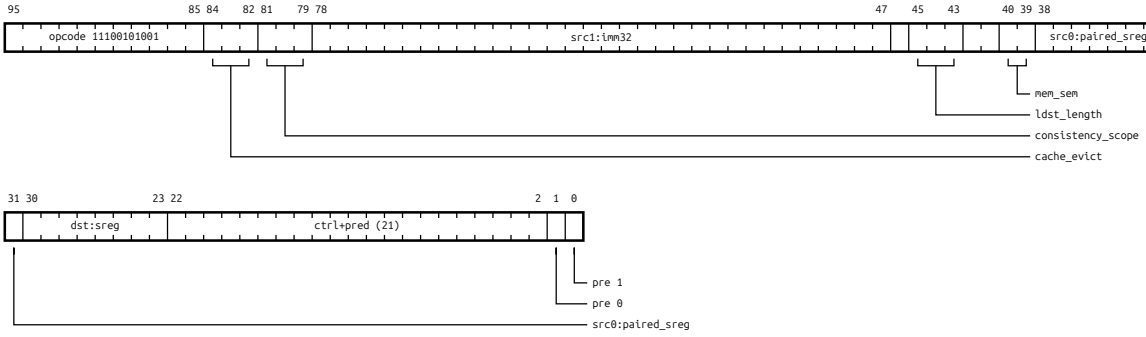
Leaf: uld__x_xi

Format:

```
uld.ldst_length{.mem_sem}.consistency_scope.cache_evict  dst, src0, src1
```

Operands: dst: sreg [notes: span=default=1; ldst_length=b64→2; ldst_length=b128→4, align=default=1; ldst_length=b64→2; ldst_length=b128→4]; src0: paired_sreg; src1: imm32s

Encoding Diagram: 96b, prefix 10, opcode 11100101001



Notes:

Uniform generic load (warp-broadcast): dst = *(effective_address), addr / offset / dst 都是 ureg, warp 集体执行一次拿同一份数据写到 ureg (1 个 32-bit slot 共享给 32 lane)。地址按 [63:62] tag 路由, 可落 global / shared / constant; local tag (addr[63:62]=01) 对 uld 非法, 硬件 trap — local 是 per-thread 私有空间, warp 一份地址定义

不出读哪个线程的 local (家族里也因此没有 uldl)。addr 必须是 uniform (warp-shared, 编译期保证全 warp 同地址)。

跟 simt ld 区别: simt ld 每 lane 独立 load 写到 vreg (32 slots), uld 整 warp 一次 load 写到 ureg (1 slot); 用 uld 替代 ld + r2u 两指令序列 (节省 vreg + 减少 issue)。

uniform memory family 4 元 (uld / uldg / ulds / uldc) 跟 simt ld / ldg / lds / ldc 对偶; 没有 uldl 因 local memory per-thread 私有跟 uniform 概念矛盾。

modifier 跟 ld 一致: ldst_length / mem_sem / consistency_scope / cache_evict。

典型用例: cta / kernel 共享的 64-bit pointer 直接 load 到 ureg / warp-broadcast scalar metadata / ML kernel host metadata 集体 broadcast load。

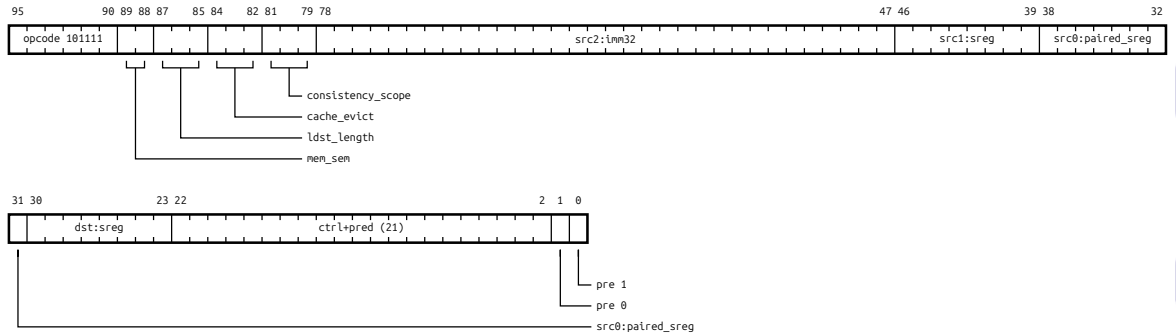
Leaf: uld__x_xxi

Format:

```
uld.ldst_length{.mem_sem}.consistency_scope.cache_evict  dst, src0, src1, src2
```

Operands: dst: sreg [notes: span=default=1; ldst_length=b64→2; ldst_length=b128→4, align=default=1; ldst_length=b64→2; ldst_length=b128→4]; src0: paired_sreg; src1: sreg; src2: imm32s

Encoding Diagram: 96b, prefix 10, opcode 101111



Notes:

Uniform generic load (warp-broadcast): dst = *(effective_address), addr / offset / dst 都是 ureg, warp 集体执行一次拿同一份数据写到 ureg (1 个 32-bit slot 共享给 32 lane)。地址按 [63:62] tag 路由, 可落 global / shared / constant; local tag (addr[63:62]=01) 对 uld 非法, 硬件 trap — local 是 per-thread 私有空间, warp 一份地址定义不出读哪个线程的 local (家族里也因此没有 uldl)。addr 必须是 uniform (warp-shared, 编译期保证全 warp 同地址)。

跟 simt ld 区别: simt ld 每 lane 独立 load 写到 vreg (32 slots), uld 整 warp 一次 load 写到 ureg (1 slot); 用 uld 替代 ld + r2u 两指令序列 (节省 vreg + 减少 issue)。

uniform memory family 4 元 (uld / uldg / ulds / uldc) 跟 simt ld / ldg / lds / ldc 对偶; 没有 uldl 因 local memory per-thread 私有跟 uniform 概念矛盾。

modifier 跟 ld 一致: ldst_length / mem_sem / consistency_scope / cache_evict。

典型用例: cta / kernel 共享的 64-bit pointer 直接 load 到 ureg / warp-broadcast scalar metadata / ML kernel host metadata 集体 broadcast load。

uldc **category**: Memory load/store **predicate_mode**: uniform

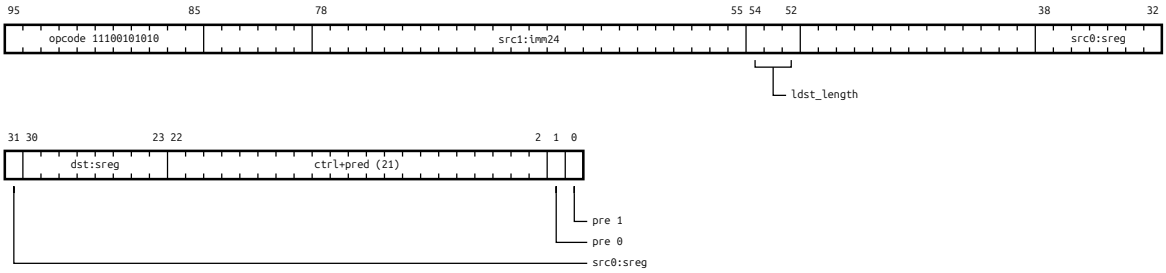
Leaf: uldc__x_xi

Format:

```
uldc.ldst_length  dst, src0, src1
```

Operands: dst: sreg [notes: span=default=1; ldst_length=b64→2; ldst_length=b128→4, align=default=1; ldst_length=b64→2; ldst_length=b128→4]; src0: sreg; src1: imm24s

Encoding Diagram: 96b, prefix 10, opcode 11100101010



Notes:

Uniform constant load (warp-broadcast): dst = *(constant_mem*)(addr + offset), addr / offset / dst 都是 ureg, warp 集体执行一次写到 ureg。把 constant 数据搬到 UR 通路的主要指令 (所有计算指令不再支持 c[] operand 后必须先 uldc 到 ureg 再参运算)。

modifier 跟 ldc 一致: 仅 ldst_length (≤ b128)。

地址模型: ubase 32-bit ureg (constant 容量小不需 64-bit pair), uoffset 32-bit ureg, imm 24-bit signed。

典型用例: kernel 启动时 driver 把 c[0x0] base ptr 设到 ureg, uldc 加载 kernel arg / descriptor 到 ureg / cta-shared 编译期 immediate broadcast load 到 ureg / 现代 ML kernel 在 prologue 用 uldc 把 weight / scale 等 metadata 一次性 broadcast 到 ureg 给 kernel 全程用。

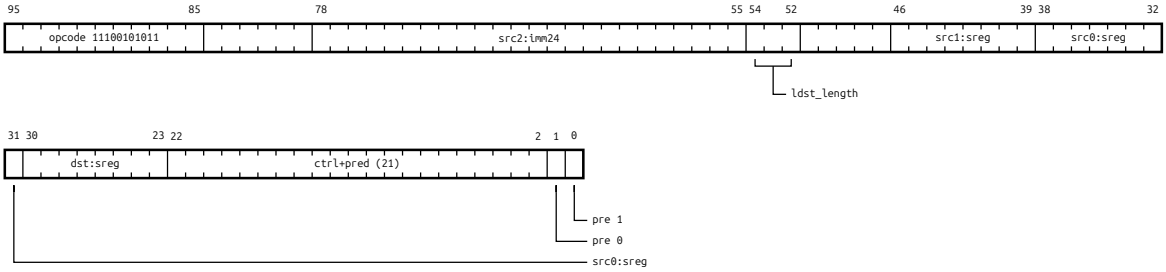
Leaf: uldc__x_xxi

Format:

```
uldc.ldst_length  dst, src0, src1, src2
```

Operands: dst: sreg [notes: span=default=1; ldst_length=b64→2; ldst_length=b128→4, align=default=1; ldst_length=b64→2; ldst_length=b128→4]; src0: sreg; src1: sreg; src2: imm24s

Encoding Diagram: 96b, prefix 10, opcode 11100101011



Notes:

Uniform constant load (warp-broadcast): dst = *(constant_mem*)(addr + offset), addr / offset / dst 都是 ureg, warp 集体执行一次写到 ureg。把 constant 数据搬到 UR 通路的主要指令 (所有计算指令不再支持 c[] operand 后必须先 uldc 到 ureg 再参运算)。

modifier 跟 ldc 一致: 仅 ldst_length (≤ b128)。

地址模型: ubase 32-bit ureg (constant 容量小不需 64-bit pair), uoffset 32-bit ureg, imm 24-bit signed。

典型用例: kernel 启动时 driver 把 c[0x0] base ptr 设到 ureg, uldc 加载 kernel arg / descriptor 到 ureg / cta-shared 编译期 immediate broadcast load 到 ureg / 现代 ML kernel 在 prologue 用 uldc 把 weight / scale 等 metadata 一次性 broadcast 到 ureg 给 kernel 全程用。

uldg　**category**: Memory load/store　**predicate_mode**: uniform

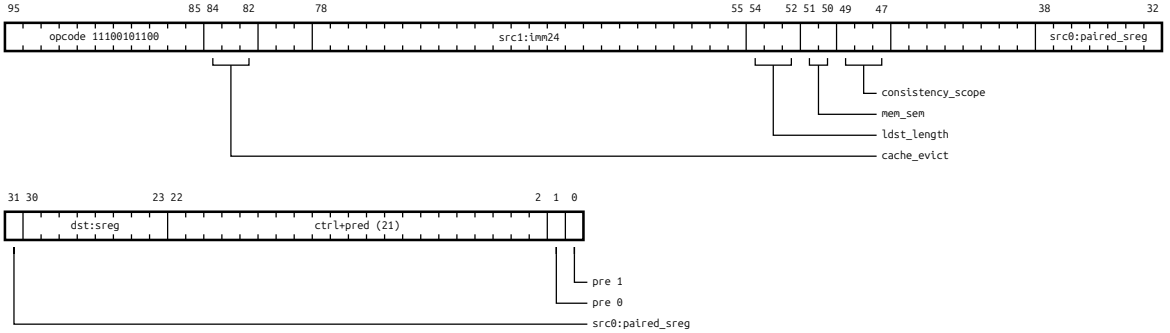
Leaf: uldg__x_xi

Format:

```
uldg.ldst_length{.mem_sem}.consistency_scope.cache_evict  dst, src0, src1
```

Operands: dst: sreg [notes: span=default=1; ldst_length=b64→2; ldst_length=b128→4; ldst_length=b256→8, align=default=1; ldst_length=b64→2; ldst_length=b128→4; ldst_length=b256→8]; src0: paired_sreg; src1: imm24s

Encoding Diagram: 96b, prefix 10, opcode 11100101100



Notes:

Uniform global load (warp-broadcast): dst = *(effective_address), addr / offset / dst 都是 ureg, warp 集体执行一次拿同一份数据写到 ureg。编译期已知 global 跳过 [63:62] tag 检查。

uldg 没有 pout0 predicate output (跟 ldg 不对偶) — uniform 路径下 warp 同地址同 OOB 状态, 整 warp 一致 trap 即可, 不需 per-lane OOB report。

地址模型: ubase 64-bit ureg pair, uoffset 32-bit ureg, imm 24-bit signed (跟 ldg 一致)。

ldst_length: 8 valid 含 b256 (256-bit packed broadcast load, 跟 ldg 同源)。

mem_sem / consistency_scope / cache_evict 跟 ldg 一致。

典型用例: cta-shared global pointer 直接 load 到 ureg / ML kernel host metadata broadcast (warp 共享 array length / tile size) / kernel parameter spill reload 到 ureg。

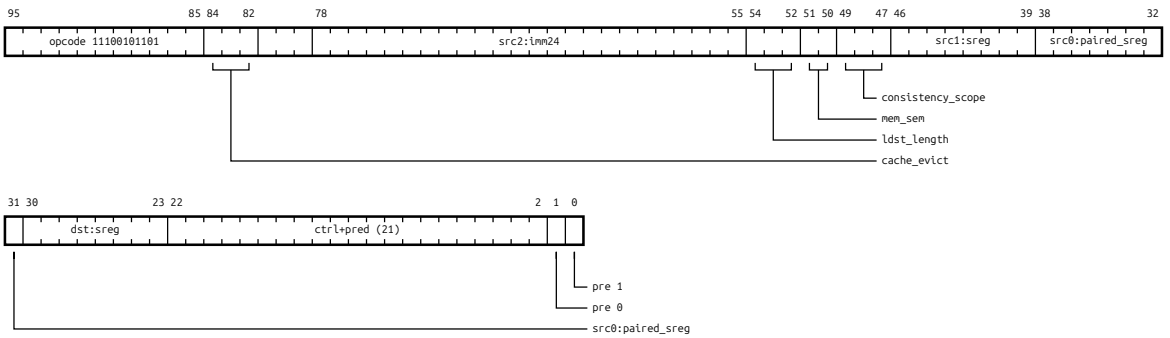
Leaf: uldg__x_xxi

Format:

```
uldg.ldst_length{.mem_sem}.consistency_scope.cache_evict  dst, src0, src1, src2
```

Operands: dst: sreg [notes: span=default=1; ldst_length=b64→2; ldst_length=b128→4; ldst_length=b256→8, align=default=1; ldst_length=b64→2; ldst_length=b128→4; ldst_length=b256→8]; src0: paired_sreg; src1: sreg; src2: imm24s

Encoding Diagram: 96b, prefix 10, opcode 11100101101



Notes:

Uniform global load (warp-broadcast): dst = *(effective_address), addr / offset / dst 都是 ureg, warp 集体执行一次拿同一份数据写到 ureg。编译期已知 global 跳过 [63:62] tag 检查。

uldg 没有 pout0 predicate output (跟 ldg 不对偶) — uniform 路径下 warp 同地址同 OOB 状态, 整 warp 一致 trap 即可, 不需 per-lane OOB report。

地址模型: ubase 64-bit ureg pair, uoffset 32-bit ureg, imm 24-bit signed (跟 ldg 一致)。

ldst_length: 8 valid 含 b256 (256-bit packed broadcast load, 跟 ldg 同源)。

mem_sem / consistency_scope / cache_evict 跟 ldg 一致。

典型用例: cta-shared global pointer 直接 load 到 ureg / ML kernel host metadata broadcast (warp 共享 array length / tile size) / kernel parameter spill reload 到 ureg。

ulds **category**: Memory load/store **predicate_mode**: uniform

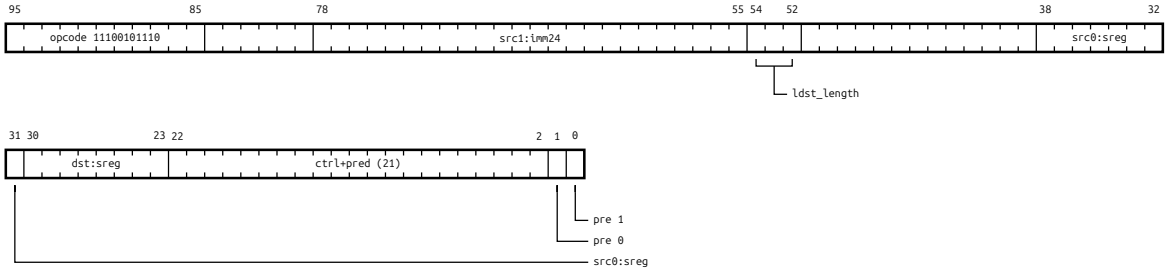
Leaf: ulds__x_xi

Format:

```
ulds.ldst_length  dst, src0, src1
```

Operands: dst: sreg [notes: span=default=1; ldst_length=b64→2; ldst_length=b128→4, align=default=1; ldst_length=b64→2; ldst_length=b128→4]; src0: sreg; src1: imm24s

Encoding Diagram: 96b, prefix 10, opcode 11100101110



Notes:

Uniform shared load (warp-broadcast): dst = shared[ubase + offset], addr / offset / dst 都是 ureg, warp 集体执行一次写到 ureg。编译期已知 shared 跳过 [63:62] tag 检查。

地址模型: ubase 是 32-bit ureg (shared memory 容量小不需 64-bit pair), uoffset 32-bit ureg, imm 24-bit signed。

modifier 跟 lds 一致: 仅 ldst_length (限 ≤ b128); 不挂 mem_sem / cache_evict / consistency_scope (shared SRAM 不走 cache, 也不需 cross-cta scope)。

典型用例: cta-shared **shared** scalar 数组的 broadcast load 到 ureg (warp 共享一份) / tile dimension / kernel parameter 从 shared memory 读到 ureg。

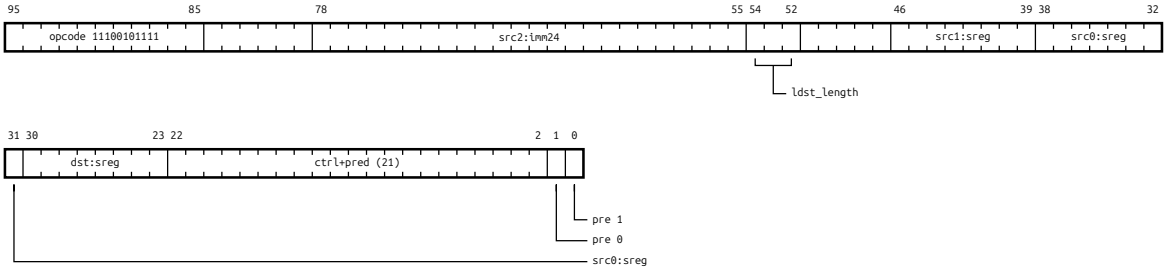
Leaf: ulds__x_xxi

Format:

```
ulds.ldst_length  dst, src0, src1, src2
```

Operands: dst: sreg [notes: span=default=1; ldst_length=b64→2; ldst_length=b128→4, align=default=1; ldst_length=b64→2; ldst_length=b128→4]; src0: sreg; src1: sreg; src2: imm24s

Encoding Diagram: 96b, prefix 10, opcode 11100101111



Notes:

Uniform shared load (warp-broadcast): dst = shared[ubase + offset], addr / offset / dst 都是 ureg, warp 集体执行一次写到 ureg。编译期已知 shared 跳过 [63:62] tag 检查。

地址模型: ubase 是 32-bit ureg (shared memory 容量小不需 64-bit pair), uoffset 32-bit ureg, imm 24-bit signed。

modifier 跟 lds 一致: 仅 ldst_length (限 ≤ b128); 不挂 mem_sem / cache_evict / consistency_scope (shared SRAM 不走 cache, 也不需 cross-cta scope)。

典型用例: cta-shared **shared** scalar 数组的 broadcast load 到 ureg (warp 共享一份) / tile dimension / kernel parameter 从 shared memory 读到 ureg。

ust **category**: Memory load/store **predicate_mode**: uniform

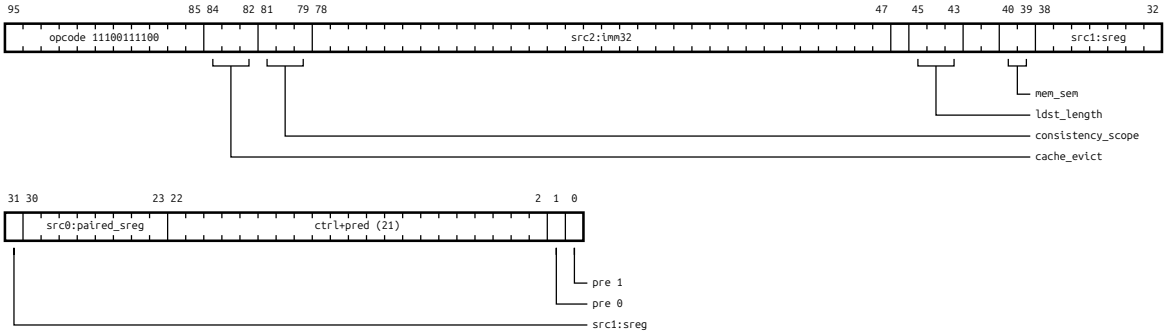
Leaf: ust___xxi

Format:

```
ust.ldst_length{.mem_sem}.consistency_scope.cache_evict  src0, src1, src2
```

Operands: src0: paired_sreg; src1: sreg [notes: span=default=1; ldst_length=b64→2; ldst_length=b128→4, align=default=1; ldst_length=b64→2; ldst_length=b128→4]; src2: imm32s

Encoding Diagram: 96b, prefix 10, opcode 11100111100



Notes:

Uniform generic store: *(effective_address) = data, addr / offset / data 都是 ureg, warp 集体执行一次写 (不是每 lane 写)。地址走 [63:62] tag 路由, 可落 global / shared; local tag (addr[63:62]=01) 非法, 硬件 trap (local 是 per-thread 私有空间, uniform 一份写定义不出写谁的 local); constant tag (=11) 同样非法, 硬件 trap (constant 只读)。

地址模型: ubase 是 64-bit ureg pair (r / r+1 偶对齐), offset 32-bit signed imm。

modifier 跟 st 一致: ldst_length / mem_sem / consistency_scope / cache_evict。

典型用例: cta-shared scalar metadata 写回 generic memory (kernel 公共状态写到编译期未知空间的指针) / warp-shared status flag 写回 / ML kernel epilogue 把 reduction 结果用 uniform 路径写回。

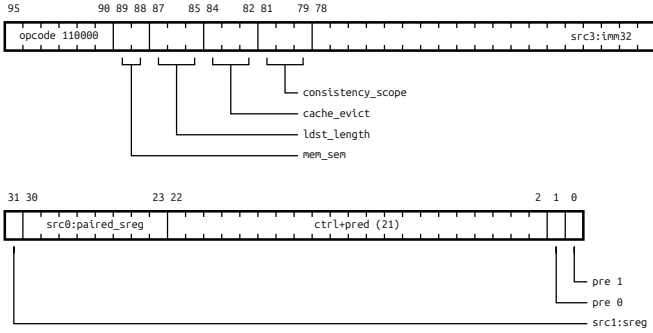
Leaf: ust___xxxi

Format:

```
ust.ldst_length{.mem_sem}.consistency_scope.cache_evict  src0, src1, src2, src3
```

Operands: src0: paired_sreg; src1: sreg; src2: sreg [notes: span=default=1; ldst_length=b64→2; ldst_length=b128→4, align=default=1; ldst_length=b64→2; ldst_length=b128→4]; src3: imm32s

Encoding Diagram: 96b, prefix 10, opcode 110000



Notes:

Uniform generic store: *(effective_address) = data, addr / offset / data 都是 ureg, warp 集体执行一次写 (不是每 lane 写)。地址走 [63:62] tag 路由, 可落 global / shared; local tag (addr[63:62]=01) 非法, 硬件 trap (local 是 per-thread 私有空间, uniform 一份写定义不出写谁的 local); constant tag (=11) 同样非法, 硬件 trap (constant 只读)。

地址模型: ubase 是 64-bit ureg pair (r / r+1 偶对齐), offset 32-bit signed imm。

modifier 跟 st 一致: ldst_length / mem_sem / consistency_scope / cache_evict。

典型用例: cta-shared scalar metadata 写回 generic memory (kernel 公共状态写到编译期未知空间的指针) / warp-shared status flag 写回 / ML kernel epilogue 把 reduction 结果用 uniform 路径写回。

ustg **category**: Memory load/store **predicate_mode**: uniform

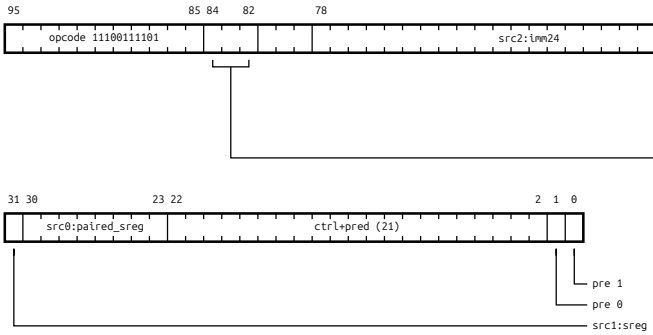
Leaf: ustg___xxi

Format:

```
ustg.ldst_length{.mem_sem}.consistency_scope.cache_evict  src0, src1, src2
```

Operands: src0: paired_sreg; src1: sreg [notes: span=default=1; ldst_length=b64→2; ldst_length=b128→4; ldst_length=b256→8, align=default=1; ldst_length=b64→2; ldst_length=b128→4; ldst_length=b256→8]; src2: imm24s

Encoding Diagram: 96b, prefix 10, opcode 11100111101



Notes:

Uniform global store: *(effective_address) = data, 所有 operand 都是 ureg / imm, warp 集体写一次相同数据(不是每 lane 写)。编译期已知 global 跳过 tag 检查。

地址模型: ubase 是 64-bit ureg pair (r / r+1 偶对齐), uoffset 32-bit ureg, imm 24-bit signed。

modifier 跟 stg 一致: ldst_length / mem_sem / consistency_scope / cache_evict。

典型用例: kernel parameter spill 写回 global memory / warp-shared scalar 状态写回 / cta-shared metadata 通过 uniform 路径写回 global。

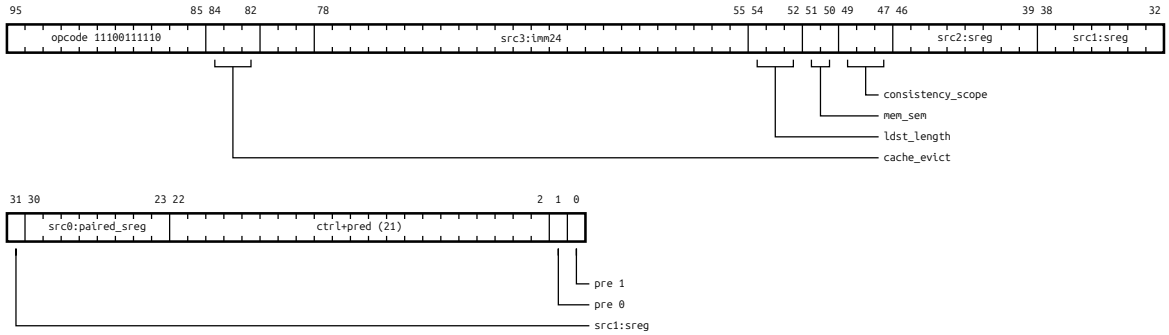
Leaf: ustg___xxxi

Format:

```
ustg.ldst_length{.mem_sem}.consistency_scope.cache_evict  src0, src1, src2, src3
```

Operands: src0: paired_sreg; src1: sreg; src2: sreg [notes: span=default=1; ldst_length=b64→2; ldst_length=b128→4; ldst_length=b256→8, align=default=1; ldst_length=b64→2; ldst_length=b128→4; ldst_length=b256→8]; src3: imm24s

Encoding Diagram: 96b, prefix 10, opcode 11100111110



Notes:

Uniform global store: *(effective_address) = data, 所有 operand 都是 ureg / imm, warp 集体写一次相同数据(不是每 lane 写)。编译期已知 global 跳过 tag 检查。

地址模型: ubase 是 64-bit ureg pair (r / r+1 偶对齐), uoffset 32-bit ureg, imm 24-bit signed。

modifier 跟 stg 一致: ldst_length / mem_sem / consistency_scope / cache_evict。

典型用例: kernel parameter spill 写回 global memory / warp-shared scalar 状态写回 / cta-shared metadata 通过 uniform 路径写回 global。

usts **category**: Memory load/store **predicate_mode**: uniform

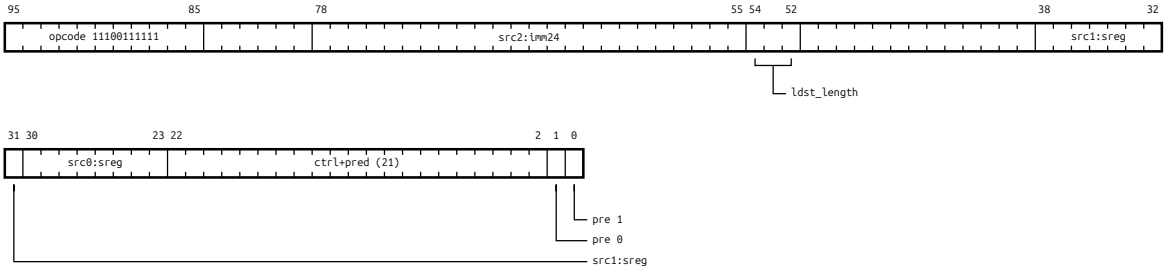
Leaf: usts___xxi

Format:

```
usts.ldst_length  src0, src1, src2
```

Operands: src0: sreg; src1: sreg [notes: span=default=1; ldst_length=b64→2; ldst_length=b128→4, align=default=1; ldst_length=b64→2; ldst_length=b128→4]; src2: imm24s

Encoding Diagram: 96b, prefix 10, opcode 11100111111



Notes:

Uniform shared store: shared[ubase + offset] = data, addr / offset / data 都是 ureg, warp 集体执行一次写。编译期已知 shared 跳过 [63:62] tag 检查。

地址模型: ubase 32-bit ureg (shared memory 容量小不需 64-bit pair), uoffset 32-bit ureg, imm 24-bit signed。

modifier 跟 sts 一致: 仅 ldst_length (≤ b128)。

典型用例: cta-shared **shared** scalar 写入 (warp 共享状态值 + 编译期可计算地址) / tile metadata 写到 shared memory 给其他 warp 读 / MMA kernel epilogue 用 uniform 路径写回 reduction 结果。

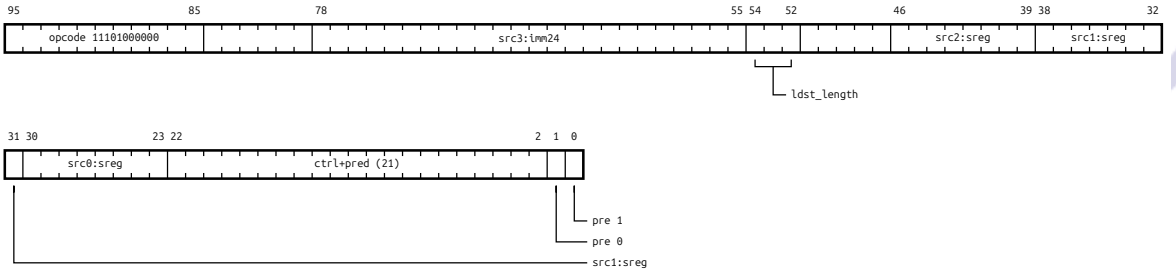
Leaf: usts___xxxi

Format:

```
usts.ldst_length  src0, src1, src2, src3
```

Operands: src0: sreg; src1: sreg; src2: sreg [notes: span=default=1; ldst_length=b64→2; ldst_length=b128→4, align=default=1; ldst_length=b64→2; ldst_length=b128→4]; src3: imm24s

Encoding Diagram: 96b, prefix 10, opcode 11101000000



Notes:

Uniform shared store: shared[ubase + offset] = data, addr / offset / data 都是 ureg, warp 集体执行一次写。编译期已知 shared 跳过 [63:62] tag 检查。

地址模型: ubase 32-bit ureg (shared memory 容量小不需 64-bit pair), uoffset 32-bit ureg, imm 24-bit signed。

modifier 跟 sts 一致: 仅 ldst_length (≤ b128)。

典型用例: cta-shared **shared** scalar 写入 (warp 共享状态值 + 编译期可计算地址) / tile metadata 写到 shared memory 给其他 warp 读 / MMA kernel epilogue 用 uniform 路径写回 reduction 结果。

14.7.9 Atomic memory

atom **category**: Atomic memory **predicate_mode**: simt

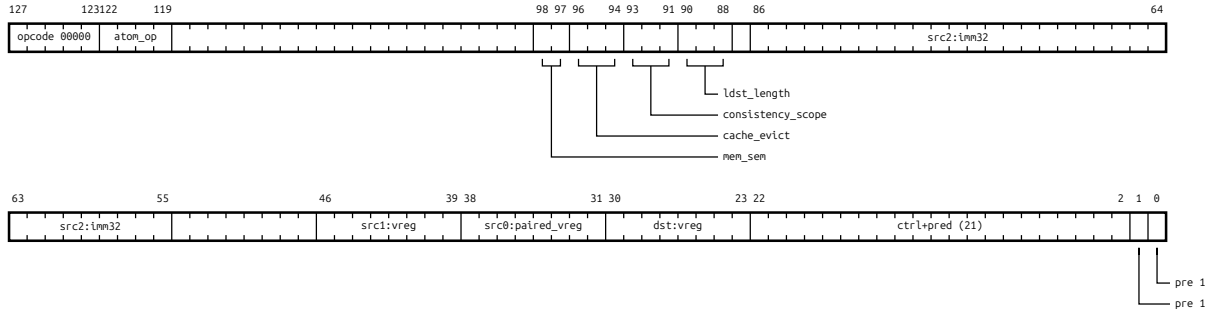
Leaf: atom__v_vvi

Format:

```
atom.ldst_length.atom_op{.mem_sem}.consistency_scope.cache_evict  dst, src0, src1, src2
```

Operands: dst: vreg [notes: span=default=1; atom_op=cast→1; ldst_length=b64→2; ldst_length=b128→4, align=default=1; atom_op=cast→1; ldst_length=b64→2; ldst_length=b128→4]; src0: paired_vreg; src1: vreg [notes: span=default=1; atom_op=cas & ldst_length=b32→2; atom_op=cast & ldst_length=b32→2; atom_op=cas & ldst_length=b64→4; atom_op=cast & ldst_length=b64→4; ldst_length=b64→2; ldst_length=b128→4, align=default=1; atom_op=cas & ldst_length=b32→2; atom_op=cast & ldst_length=b32→2; ldst_length=b64→2; ldst_length=b128→4]; src2: imm32s

Encoding Diagram: 128b, prefix 11, opcode 00000



Notes:

Generic atomic read-modify-write: dst = *(ea); *(ea) = atom_op(*(ea), data); 整 warp 32 个 active lane 各自独立做 RMW (旧值写 dst, 新值 atom_op(old, data) 写回内存)。dst=RZ 时退化为 reduce 语义 (无返回, 仅写回)。地址走 [63:62] tag 路由; 同 warp 同空间是正式架构约束 (约束全文与编译器契约见 ld 条目), 违反 trap; constant tag (=11) 对 atom 非法, 硬件 trap (constant 只读不可 RMW)。

atom_op 14 valid (sign 编进 op enum, 跟 isetp 风格): add / exch (atomic exchange) / cas / cast (compare-and-swap, paired register) / imin_s / imin_u / imax_s / imax_u (signed / unsigned int min / max) / fadd (FP add) / and / or / xor / inc (count-up wrap) / dec (count-down wrap)。

cas / cast 时 data operand 实际占 reg pair (r / r+1, 偶对齐): r 是 compare value, r+1 是 swap value。

地址模型: 跟 ld / st 一致 — v_vvi 指令形态用 vaddr=64-bit vreg pair (per-lane 完整地址), v_xvvi 指令形态用 saddr=64-bit ureg pair (uniform base) + vaddr (per-lane offset) + imm.32 signed。

mem_sem 4 valid (atomic 跨线程同步主路径必需): constant / weak / strong / mmio, 控 acquire / release / seq_cst 模型。

consistency_scope: cta / sm / cluster / gpu / sys, 控 atomic 可见范围。

cache_evict + ldst_length 跟 ld 同源。

典型用例: generic pointer atomic (编译期不知道地址空间) / 跨地址空间 lock-free 数据结构 / polymorphic 函数参数 atomic update。

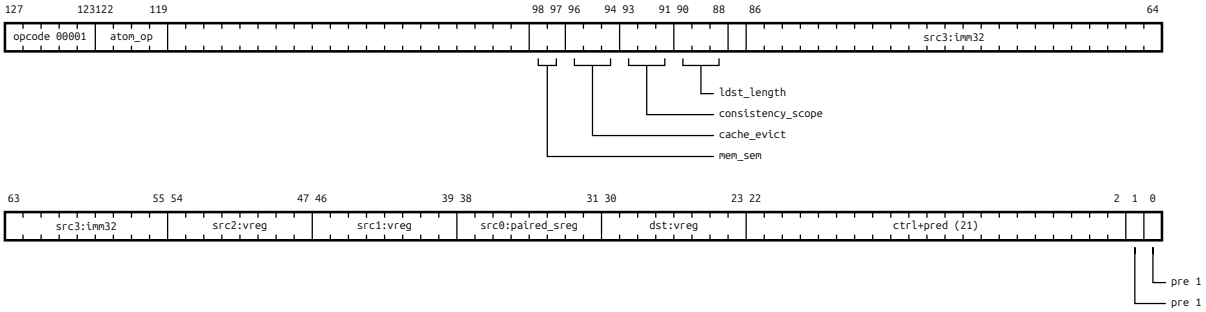
Leaf: atom__v_xvvi

Format:

```
atom.ldst_length.atom_op{.mem_sem}.consistency_scope.cache_evict  dst, src0, src1, src2,
↪  src3
```

Operands: dst: vreg [notes: span=default=1; atom_op=cast→1; ldst_length=b64→2; ldst_length=b128→4, align=default=1; atom_op=cast→1; ldst_length=b64→2; ldst_length=b128→4]; src0: paired_sreg; src1: vreg; src2: vreg [notes: span=default=1; atom_op=cas & ldst_length=b32→2; atom_op=cast & ldst_length=b32→2; atom_op=cas & ldst_length=b64→4; atom_op=cast & ldst_length=b64→4; ldst_length=b64→2; ldst_length=b128→4, align=default=1; atom_op=cas & ldst_length=b32→2; atom_op=cast & ldst_length=b32→2; ldst_length=b64→2; ldst_length=b128→4]; src3: imm32s

Encoding Diagram: 128b, prefix 11, opcode 00001



Notes:

Generic atomic read-modify-write: dst = *(ea); *(ea) = atom_op(*(ea), data); 整 warp 32 个 active lane 各自独立做 RMW (旧值写 dst, 新值 atom_op(old, data) 写回内存)。dst=RZ 时退化为 reduce 语义 (无返回, 仅写回)。地址走 [63:62] tag 路由; 同 warp 同空间是正式架构约束 (约束全文与编译器契约见 ld 条目), 违反 trap; constant tag (=11) 对 atom 非法, 硬件 trap (constant 只读不可 RMW)。

atom_op 14 valid (sign 编进 op enum, 跟 isetp 风格): add / exch (atomic exchange) / cas / cast (compare-and-swap, paired register) / imin_s / imin_u / imax_s / imax_u (signed / unsigned int min / max) / fadd (FP add) / and / or / xor / inc (count-up wrap) / dec (count-down wrap)。

cas / cast 时 data operand 实际占 reg pair (r / r+1, 偶对齐): r 是 compare value, r+1 是 swap value。

地址模型: 跟 ld / st 一致 — v_vvi 指令形态用 vaddr=64-bit vreg pair (per-lane 完整地址), v_xvvi 指令形态用 saddr=64-bit ureg pair (uniform base) + vaddr (per-lane offset) + imm.32 signed。

mem_sem 4 valid (atomic 跨线程同步主路径必需): constant / weak / strong / mmio, 控 acquire / release / seq_cst 模型。

consistency_scope: cta / sm / cluster / gpu / sys, 控 atomic 可见范围。

cache_evict + ldst_length 跟 ld 同源。

典型用例: generic pointer atomic (编译期不知道地址空间) / 跨地址空间 lock-free 数据结构 / polymorphic 函数参数 atomic update。

atomg **category**: Atomic memory **predicate_mode**: simt

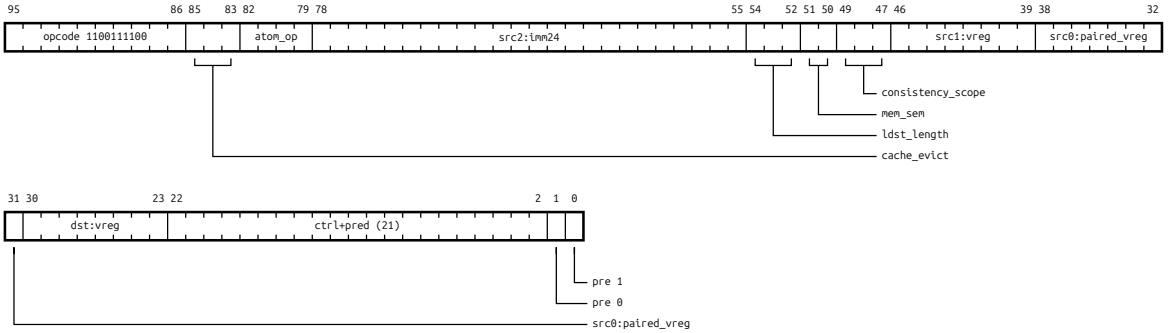
Leaf: atomg__v_vvi

Format:

```
atomg.ldst_length.atom_op{.mem_sem}.consistency_scope.cache_evict  dst, src0, src1, src2
```

Operands: dst: vreg [notes: span=default=1; atom_op=cast→1; ldst_length=b64→2; ldst_length=b128→4, align=default=1; atom_op=cast→1; ldst_length=b64→2; ldst_length=b128→4]; src0: paired_vreg; src1: vreg [notes: span=default=1; atom_op=cas & ldst_length=b32→2; atom_op=cast & ldst_length=b32→2; atom_op=cas & ldst_length=b64→4; atom_op=cast & ldst_length=b64→4; ldst_length=b64→2; ldst_length=b128→4, align=default=1; atom_op=cas & ldst_length=b32→2; atom_op=cast & ldst_length=b32→2; ldst_length=b64→2; ldst_length=b128→4]; src2: imm24s

Encoding Diagram: 96b, prefix 10, opcode 1100111100



Notes:

Explicit global atomic: dst = *(ea); *(ea) = atom_op(*(ea), data), 整 warp 32 个 active lane 各自独立 RMW global memory。编译期已知 global, 跳过 [63:62] tag 检查。dst=RZ 时退化为 reduce 语义 (无返回, 替代旧 red 独立 mnemonic)。

atom_op 14 valid 跟 atom 同源 (add / exch / cas / cast / imin_s / imin_u / imax_s / imax_u / fadd / and / or / xor / inc / dec)。cas / cast 时 data 是 vreg pair。

地址模型: v_vvi 指令形态用 vaddr=64-bit vreg pair (per-lane 完整地址, 跟 ldg.v_vi 同源), v_xvvi 指令形态用 saddr=64-bit ureg pair + vaddr per-lane offset + imm.24 signed (跟 ldg.v_xvi 同源)。

mem_sem / consistency_scope / cache_evict / ldst_length 跟 atom 同源。

典型用例: global memory 计数器 (warp-divergent atomic counter) / histogram bin 累加 (fadd reduce dst=RZ) / lock-free 数据结构 (cas / cast) / kernel-global state update。

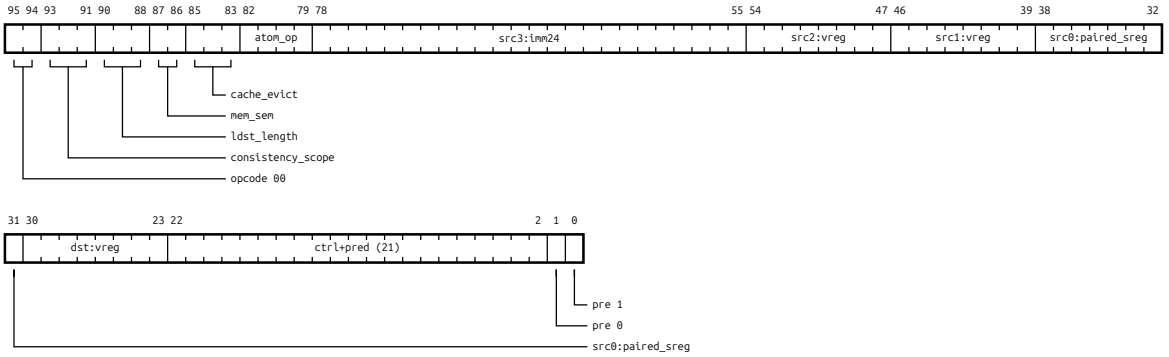
Leaf: atomg__v_xvvi

Format:

```
atomg.ldst_length.atom_op{.mem_sem}.consistency_scope.cache_evict  dst, src0, src1, src2,
↪  src3
```

Operands: dst: vreg [notes: span=default=1; atom_op=cast→1; ldst_length=b64→2; ldst_length=b128→4, align=default=1; atom_op=cast→1; ldst_length=b64→2; ldst_length=b128→4]; src0: paired_sreg; src1: vreg; src2: vreg [notes: span=default=1; atom_op=cas & ldst_length=b32→2; atom_op=cast & ldst_length=b32→2; atom_op=cas & ldst_length=b64→4; atom_op=cast & ldst_length=b64→4; ldst_length=b64→2; ldst_length=b128→4, align=default=1; atom_op=cas & ldst_length=b32→2; atom_op=cast & ldst_length=b32→2; ldst_length=b64→2; ldst_length=b128→4]; src3: imm24s

Encoding Diagram: 96b, prefix 10, opcode 00



Notes:

Explicit global atomic: dst = *(ea); *(ea) = atom_op(*(ea), data), 整 warp 32 个 active lane 各自独立 RMW global memory。编译期已知 global, 跳过 [63:62] tag 检查。dst=RZ 时退化为 reduce 语义 (无返回, 替代旧 red 独立 mnemonic)。

atom_op 14 valid 跟 atom 同源 (add / exch / cas / cast / imin_s / imin_u / imax_s / imax_u / fadd / and / or / xor / inc / dec)。cas / cast 时 data 是 vreg pair。

地址模型: v_vvi 指令形态用 vaddr=64-bit vreg pair (per-lane 完整地址, 跟 ldg.v_vi 同源), v_xvvi 指令形态用 saddr=64-bit ureg pair + vaddr per-lane offset + imm.24 signed (跟 ldg.v_xvi 同源)。

mem_sem / consistency_scope / cache_evict / ldst_length 跟 atom 同源。

典型用例: global memory 计数器 (warp-divergent atomic counter) / histogram bin 累加 (fadd reduce dst=RZ) / lock-free 数据结构 (cas / cast) / kernel-global state update。

atoms **category**: Atomic memory **predicate_mode**: simt

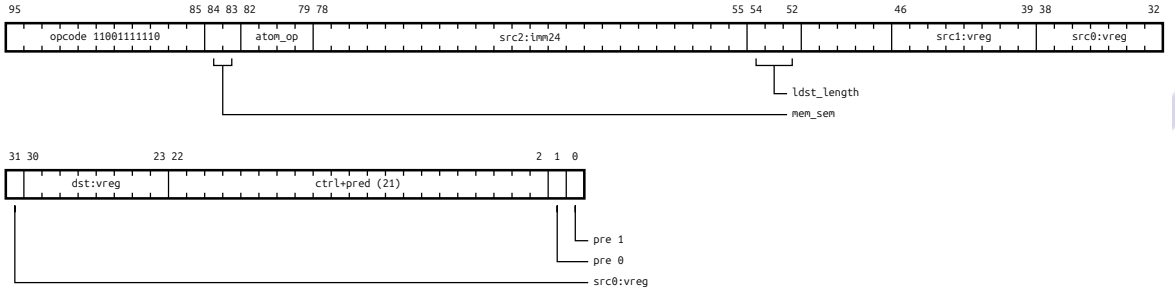
Leaf: atoms__v_vvi

Format:

```
atoms.ldst_length.atom_op{.mem_sem}  dst, src0, src1, src2
```

Operands: dst: vreg [notes: span=default=1; atom_op=cast→1; ldst_length=b64→2; ldst_length=b128→4, align=default=1; atom_op=cast→1; ldst_length=b64→2; ldst_length=b128→4]; src0: vreg; src1: vreg [notes: span=default=1; atom_op=cas & ldst_length=b32→2; atom_op=cast & ldst_length=b32→2; atom_op=cas & ldst_length=b64→4; atom_op=cast & ldst_length=b64→4; ldst_length=b64→2; ldst_length=b128→4, align=default=1; atom_op=cas & ldst_length=b32→2; atom_op=cast & ldst_length=b32→2; ldst_length=b64→2; ldst_length=b128→4]; src2: imm24s

Encoding Diagram: 96b, prefix 10, opcode 11001111110



Notes:

Explicit shared atomic: dst = shared[vaddr+offset]; shared[vaddr+offset] = atom_op(*(ea), data), 整 warp 32 个 active lane 各自独立 RMW shared memory。cta 内同步主路径 — warp 间 / lane 间共享 SRAM 数据时用 atomic 保证一致性。dst=RZ 时退化为 reduce。

atom_op 14 valid 跟 atom 同源。cas / cast 时 data 是 vreg pair (r / r+1, 偶对齐)。

地址模型: vaddr 32-bit cta-相对地址 (跟 lds 同源, 不是 64-bit pair, shared 容量小)。

modifier: ldst_length + atom_op + mem_sem (即使 shared 限 cta, mem_sem 仍有用 — acquire / release 模型在 cta 内多 warp 同步用), 不挂 consistency_scope (shared 限于 cta 内 SRAM 不需跨 sm scope)。

典型用例: cta-shared atomic counter (per-cta tile reduction) / cta 内 lock-free 数据结构 (workstealing queue) / warp-divergent shared memory 计数。

uatom **category**: Atomic memory **predicate_mode**: uniform

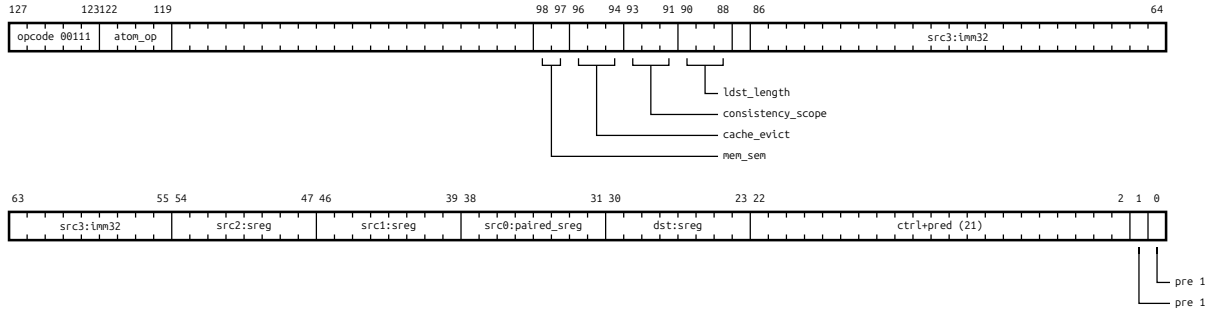
Leaf: uatom__x_xxxi

Format:

```
uatom.ldst_length.atom_op{.mem_sem}.consistency_scope.cache_evict  dst, src0, src1, src2,
↪  src3
```

Operands: dst: sreg [notes: span=default=1; atom_op=cast→1; ldst_length=b64→2; ldst_length=b128→4, align=default=1; atom_op=cast→1; ldst_length=b64→2; ldst_length=b128→4]; src0: paired_sreg; src1: sreg; src2: sreg [notes: span=default=1; atom_op=cas & ldst_length=b32→2; atom_op=cast & ldst_length=b32→2; atom_op=cas & ldst_length=b64→4; atom_op=cast & ldst_length=b64→4; ldst_length=b64→2; ldst_length=b128→4, align=default=1; atom_op=cas & ldst_length=b32→2; atom_op=cast & ldst_length=b32→2; ldst_length=b64→2; ldst_length=b128→4]; src3: imm32s

Encoding Diagram: 128b, prefix 11, opcode 00111



Notes:

Uniform generic atomic: addr / offset / data / dst 都是 ureg, warp 集体发一次原子操作 (不是每 lane 发)。dst=URZ 时退化为 uniform reduce (无返回)。地址走 [63:62] tag 路由跟 atom 同源。

modifier 跟 atom 一致: atom_op (14 valid 含 cas / cast) + mem_sem + consistency_scope + cache_evict + ldst_length。cas / cast 时 udata 是 ureg pair (UR / UR+1)。

典型用例: warp 一致更新全局计数器 / 锁 (1 个 uatom 替代 32 个 simt atom 减原子冲突) / warp-shared spinlock。

uatomg **category**: Atomic memory **predicate_mode**: uniform

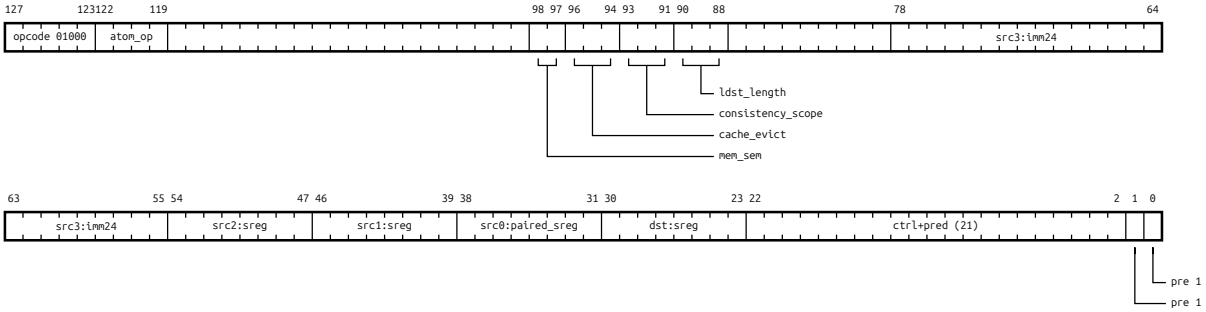
Leaf: uatomg__x_xxxi

Format:

```
uatomg.ldst_length.atom_op{.mem_sem}.consistency_scope.cache_evict  dst, src0, src1,
↪  src2, src3
```

Operands: dst: sreg [notes: span=default=1; atom_op=cast→1; ldst_length=b64→2; ldst_length=b128→4, align=default=1; atom_op=cast→1; ldst_length=b64→2; ldst_length=b128→4]; src0: paired_sreg; src1: sreg; src2: sreg [notes: span=default=1; atom_op=cas & ldst_length=b32→2; atom_op=cast & ldst_length=b32→2; atom_op=cas & ldst_length=b64→4; atom_op=cast & ldst_length=b64→4; ldst_length=b64→2; ldst_length=b128→4, align=default=1; atom_op=cas & ldst_length=b32→2; atom_op=cast & ldst_length=b32→2; ldst_length=b64→2; ldst_length=b128→4]; src3: imm24s

Encoding Diagram: 128b, prefix 11, opcode 01000



Notes:

Uniform global atomic: 全 ureg 路径, warp 集体发一次原子操作到 global memory。dst=URZ 时退化为 uniform reduce (无返回)。编译期已知 global 跳过 tag 检查。

modifier / atom_op / 地址模型跟 atomg + uatom 同源。cas / cast 时 udata 是 ureg pair。

典型用例: warp 一致 global 计数器 / kernel-level shared status flag update / global spinlock。

uatoms **category**: Atomic memory **predicate_mode**: uniform

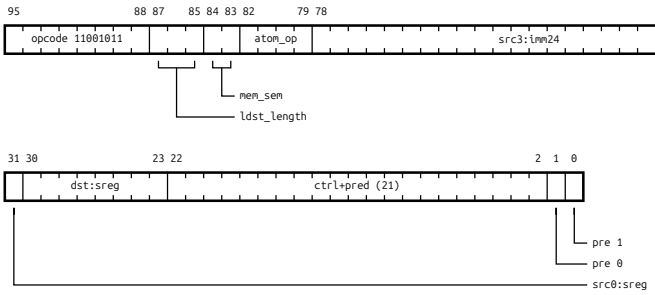
Leaf: uatoms__x_xxxi

Format:

```
uatoms.ldst_length.atom_op{.mem_sem}  dst, src0, src1, src2, src3
```

Operands: dst: sreg [notes: span=default=1; atom_op=cast→1; ldst_length=b64→2; ldst_length=b128→4, align=default=1; atom_op=cast→1; ldst_length=b64→2; ldst_length=b128→4]; src0: sreg; src1: sreg; src2: sreg [notes: span=default=1; atom_op=cas & ldst_length=b32→2; atom_op=cast & ldst_length=b32→2; atom_op=cas & ldst_length=b64→4; atom_op=cast & ldst_length=b64→4; ldst_length=b64→2; ldst_length=b128→4, align=default=1; atom_op=cas & ldst_length=b32→2; atom_op=cast & ldst_length=b32→2; ldst_length=b64→2; ldst_length=b128→4]; src3: imm24s

Encoding Diagram: 96b, prefix 10, opcode 11001011



Notes:

Uniform shared atomic: 全 ureg 路径, warp 集体发一次 shared memory 原子操作。dst=URZ 时退化为 uniform reduce。

modifier / atom_op / 地址模型跟 atoms 同源。cas / cast 时 udata 是 ureg pair。

典型用例: warp 一致 cta-shared 计数器 / cta-shared lock 操作 / warp 集体 update cta 状态。

14.7.10 Async and bulk memory

ldgsts **category**: Async and bulk memory **predicate_mode**: simt

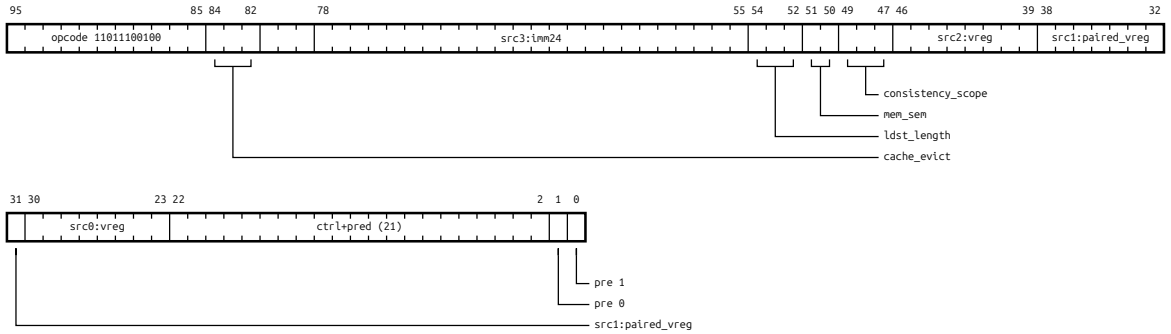
Leaf: ldgsts___vvvi

Format:

```
ldgsts.ldst_length{.mem_sem}.consistency_scope.cache_evict  src0, src1, src2, src3
```

Operands: src0: vreg; src1: paired_vreg; src2: vreg; src3: imm24s

Encoding Diagram: 96b, prefix 10, opcode 11011100100



Notes:

Async global → shared copy: 整 warp 32 lane 各自独立异步从 global memory 把数据搬到 shared memory, 数据不进 reg file (硬件直接走 sm 内 datapath, 不占 vreg)。warp 不阻塞继续 issue 后续指令; 完成追踪走 async-group — 用 async_commit 把发出去的这一批归成一组, 再用 depbar 等未完成的组数降到阈值以内 (见 async_-commit / depbar 条目)。

地址模型: shm 端 = shm_addr (vreg, 32-bit cta-相对) + simm24; glb 端有两种指令形态 — _vxvi 指令形态 (uniform addr 主路径) 用 glb_saddr (sreg pair 64-bit base) + glb_vaddr (vreg, 32-bit offset) + simm24, _vvvi 指令形态 (per-lane full address) 用 glb_vaddr (vreg pair 64-bit base) + glb_offset (vreg, 32-bit offset) + simm24。simm24 同时用于 shm 跟 glb 两端 offset。

_vxvip / _vvvip 指令形态含 pin0 (per-lane predicate input): pin0=true 时该 lane 走 normal copy (从 glb 读 sz byte 写到 shm); pin0=false 时该 lane 不读 glb, shm 对应位置写 0。专用于 tile boundary handling (动态 boundary 不规整 tile 的 padding, 越界 lane pin0=false 自动写 0)。

ldst_length 限 b32 / b64 / b128 / b256 共 4 valid (复用 ldst_length 全集的子集, 跟 ldg / lds 风格一致)。

modifier: ldst_length + mem_sem + consistency_scope + cache_evict (跟 ldg 同源)。

典型用例: ML kernel 主路径 (attention / GEMM) — kernel inner loop 用 ldgsts 把 K / V tile 异步 load 到 shared, MMA 在另一 stage 同时算前一 tile, global latency 隐藏在 MMA 计算之中, kernel throughput 不被 memory-bound 拖慢。

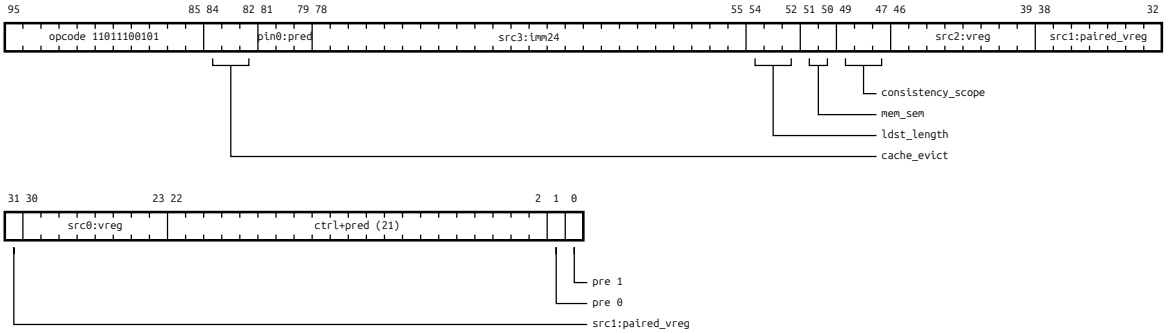
Leaf: ldgsts___vvvip

Format:

```
ldgsts.ldst_length{.mem_sem}.consistency_scope.cache_evict  src0, src1, src2, src3, pin0
```

Operands: src0: vreg; src1: paired_vreg; src2: vreg; src3: imm24s; pin0: pred

Encoding Diagram: 96b, prefix 10, opcode 11011100101



Notes:

Async global → shared copy: 整 warp 32 lane 各自独立异步从 global memory 把数据搬到 shared memory, 数据不进 reg file (硬件直接走 sm 内 datapath, 不占 vreg)。warp 不阻塞继续 issue 后续指令; 完成追踪走 async-group — 用 async_commit 把发出去的这一批归成一组, 再用 depbar 等未完成的组数降到阈值以内 (见 async_-commit / depbar 条目)。

地址模型: shm 端 = shm_addr (vreg, 32-bit cta-相对) + simm24; glb 端有两种指令形态 — _vxvi 指令形态 (uniform addr 主路径) 用 glb_saddr (sreg pair 64-bit base) + glb_vaddr (vreg, 32-bit offset) + simm24, _vvvi 指令形态 (per-lane full address) 用 glb_vaddr (vreg pair 64-bit base) + glb_offset (vreg, 32-bit offset) + simm24。simm24 同时用于 shm 跟 glb 两端 offset。

_vxvip / _vvvip 指令形态含 pin0 (per-lane predicate input): pin0=true 时该 lane 走 normal copy (从 glb 读 sz byte 写到 shm); pin0=false 时该 lane 不读 glb, shm 对应位置写 0。专用于 tile boundary handling (动态 boundary 不规整 tile 的 padding, 越界 lane pin0=false 自动写 0)。

ldst_length 限 b32 / b64 / b128 / b256 共 4 valid (复用 ldst_length 全集的子集, 跟 ldg / lds 风格一致)。

modifier: ldst_length + mem_sem + consistency_scope + cache_evict (跟 ldg 同源)。

典型用例: ML kernel 主路径 (attention / GEMM) — kernel inner loop 用 ldgsts 把 K / V tile 异步 load 到 shared, MMA 在另一 stage 同时算前一 tile, global latency 隐藏在 MMA 计算之中, kernel throughput 不被 memory-bound 拖慢。

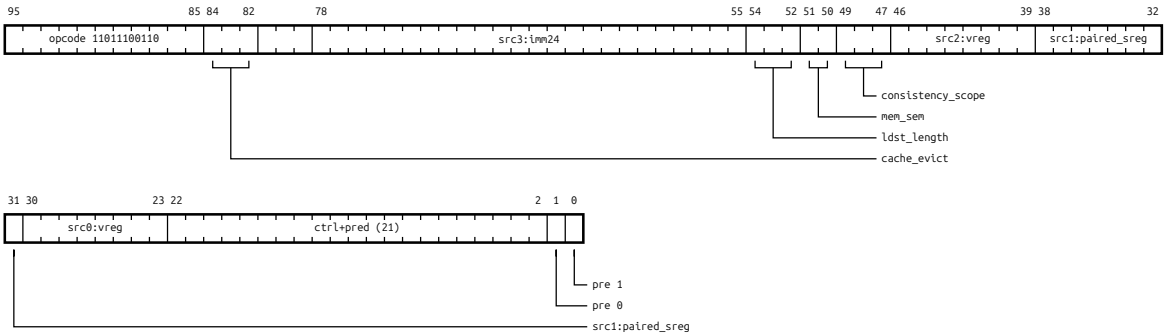
Leaf: ldgsts___vxvi

Format:

```
ldgsts.ldst_length{.mem_sem}.consistency_scope.cache_evict  src0, src1, src2, src3
```

Operands: src0: vreg; src1: paired_sreg; src2: vreg; src3: imm24s

Encoding Diagram: 96b, prefix 10, opcode 11011100110



Notes:

Async global → shared copy: 整 warp 32 lane 各自独立异步从 global memory 把数据搬到 shared memory, 数据不进 reg file (硬件直接走 sm 内 datapath, 不占 vreg)。warp 不阻塞继续 issue 后续指令; 完成追踪走 async-group — 用 async_commit 把发出去的这一批归成一组, 再用 depbar 等未完成的组数降到阈值以内 (见 async_-

commit / depbar 条目)。

地址模型: shm 端 = shm_addr (vreg, 32-bit cta-相对) + simm24; glb 端有两种指令形态 — _vxvi 指令形态 (uniform addr 主路径) 用 glb_saddr (sreg pair 64-bit base) + glb_vaddr (vreg, 32-bit offset) + simm24, _vvvi 指令形态 (per-lane full address) 用 glb_vaddr (vreg pair 64-bit base) + glb_offset (vreg, 32-bit offset) + simm24。simm24 同时用于 shm 跟 glb 两端 offset。

_vxvip / _vvvip 指令形态含 pin0 (per-lane predicate input): pin0=true 时该 lane 走 normal copy (从 glb 读 sz byte 写到 shm); pin0=false 时该 lane 不读 glb, shm 对应位置写 0。专用于 tile boundary handling (动态 boundary 不规整 tile 的 padding, 越界 lane pin0=false 自动写 0)。

ldst_length 限 b32 / b64 / b128 / b256 共 4 valid (复用 ldst_length 全集的子集, 跟 ldg / lds 风格一致)。

modifier: ldst_length + mem_sem + consistency_scope + cache_evict (跟 ldg 同源)。

典型用例: ML kernel 主路径 (attention / GEMM) — kernel inner loop 用 ldgsts 把 K / V tile 异步 load 到 shared, MMA 在另一 stage 同时算前一 tile, global latency 隐藏在 MMA 计算之中, kernel throughput 不被 memory-bound 拖慢。

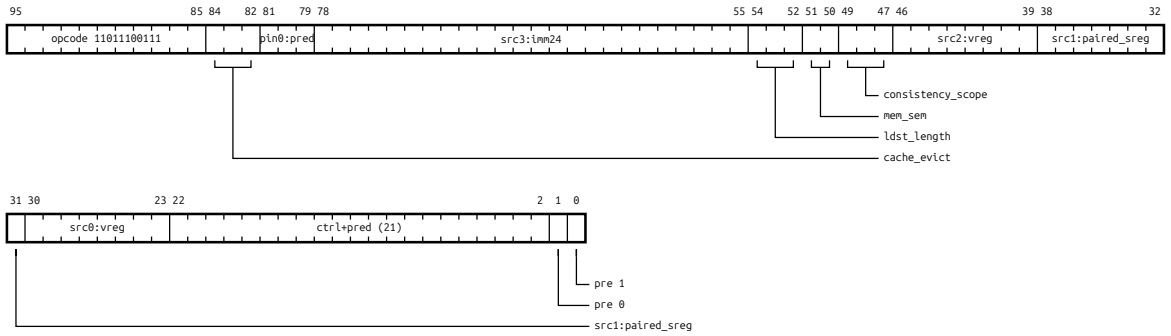
Leaf: ldgsts___vxvip

Format:

```
ldgsts.ldst_length{.mem_sem}.consistency_scope.cache_evict  src0, src1, src2, src3, pin0
```

Operands: src0: vreg; src1: paired_sreg; src2: vreg; src3: imm24s; pin0: pred

Encoding Diagram: 96b, prefix 10, opcode 11011100111



Notes:

Async global → shared copy: 整 warp 32 lane 各自独立异步从 global memory 把数据搬到 shared memory, 数据不进 reg file (硬件直接走 sm 内 datapath, 不占 vreg)。warp 不阻塞继续 issue 后续指令; 完成追踪走 async-group — 用 async_commit 把发出去的这一批归成一组, 再用 depbar 等未完成的组数降到阈值以内 (见 async_-commit / depbar 条目)。

地址模型: shm 端 = shm_addr (vreg, 32-bit cta-相对) + simm24; glb 端有两种指令形态 — _vxvi 指令形态 (uniform addr 主路径) 用 glb_saddr (sreg pair 64-bit base) + glb_vaddr (vreg, 32-bit offset) + simm24, _vvvi 指令形态 (per-lane full address) 用 glb_vaddr (vreg pair 64-bit base) + glb_offset (vreg, 32-bit offset) + simm24。simm24 同时用于 shm 跟 glb 两端 offset。

_vxvip / _vvvip 指令形态含 pin0 (per-lane predicate input): pin0=true 时该 lane 走 normal copy (从 glb 读 sz byte 写到 shm); pin0=false 时该 lane 不读 glb, shm 对应位置写 0。专用于 tile boundary handling (动态 boundary 不规整 tile 的 padding, 越界 lane pin0=false 自动写 0)。

ldst_length 限 b32 / b64 / b128 / b256 共 4 valid (复用 ldst_length 全集的子集, 跟 ldg / lds 风格一致)。

modifier: ldst_length + mem_sem + consistency_scope + cache_evict (跟 ldg 同源)。

典型用例: ML kernel 主路径 (attention / GEMM) — kernel inner loop 用 ldgsts 把 K / V tile 异步 load 到 shared, MMA 在另一 stage 同时算前一 tile, global latency 隐藏在 MMA 计算之中, kernel throughput 不被

memory-bound 拖慢。

ldsstg **category**: Async and bulk memory **predicate_mode**: simt

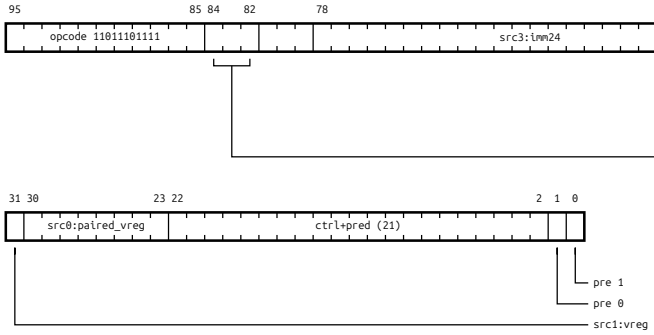
Leaf: ldsstg___vvvi

Format:

```
ldsstg.ldst_length{.mem_sem}.consistency_scope.cache_evict  src0, src1, src2, src3
```

Operands: src0: paired_vreg; src1: vreg; src2: vreg; src3: imm24s

Encoding Diagram: 96b, prefix 10, opcode 11011101111



Notes:

Async shared → global copy: ldgsts 的反方向, 整 warp 32 lane 各自独立异步从 shared memory 把数据搬到 global memory, 数据不进 reg file。warp 不阻塞继续 issue 后续指令; 完成追踪走 async-group — 用 async_commit 归组、depbar 等待, 跟 ldgsts 走同一条 group 队列 (见 async_commit / depbar 条目)。

地址模型: glb 端有两种指令形态 — _xvvi 指令形态 (uniform addr) 用 glb_saddr (sreg pair 64-bit base) + glb_vaddr (vreg offset) + shm_addr (vreg) + simm24, _vvvi 指令形态 (per-lane full address) 用 glb_vaddr (vreg pair 64-bit base) + glb_offset (vreg) + shm_addr (vreg) + simm24。

ldsstg 没有 pin0 指令形态 (不像 ldgsts 有 _vxvip / _vvvip), 因为 store 端本来就不需要 OOB padding (越界 lane 用 isetp + @P gate 控制即可)。

ldst_length 限 b32 / b64 / b128 / b256 共 4 valid。

modifier: ldst_length + mem_sem + consistency_scope + cache_evict (跟 ldgsts 同源)。

典型用例: ML kernel epilogue 写回 (MMA 算完 tile 用 stsm 写到 shared, 再用 ldsstg 异步写到 global, 计算跟写回 overlap) / cta-shared 数据写回 global 不阻塞后续算。

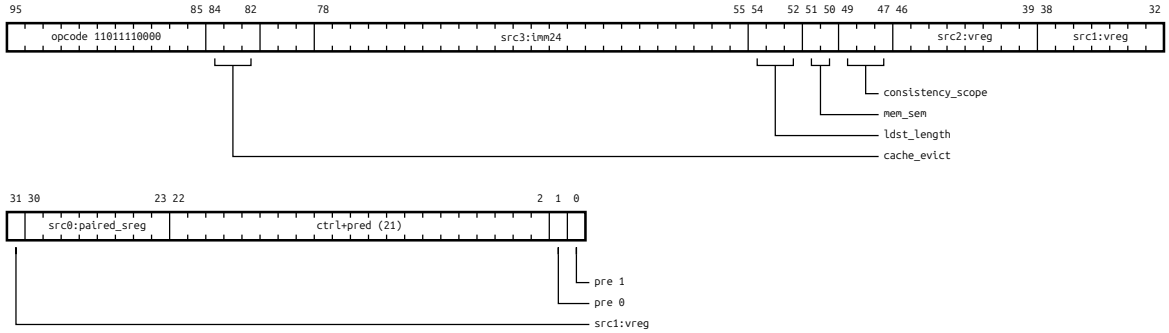
Leaf: ldsstg___xvvi

Format:

```
ldsstg.ldst_length{.mem_sem}.consistency_scope.cache_evict  src0, src1, src2, src3
```

Operands: src0: paired_sreg; src1: vreg; src2: vreg; src3: imm24s

Encoding Diagram: 96b, prefix 10, opcode 11011110000



Notes:

Async shared → global copy: ldgsts 的反方向, 整 warp 32 lane 各自独立异步从 shared memory 把数据搬到 global memory, 数据不进 reg file。warp 不阻塞继续 issue 后续指令; 完成追踪走 async-group — 用 async_commit 归组、depbar 等待, 跟 ldgsts 走同一条 group 队列 (见 async_commit / depbar 条目)。

地址模型: glb 端有两种指令形态 — _xvvi 指令形态 (uniform addr) 用 glb_saddr (sreg pair 64-bit base) + glb_vaddr (vreg offset) + shm_addr (vreg) + simm24, _vvvi 指令形态 (per-lane full address) 用 glb_vaddr (vreg pair 64-bit base) + glb_offset (vreg) + shm_addr (vreg) + simm24。

ldsstg 没有 pin0 指令形态 (不像 ldgsts 有 _vxvip / _vvvip), 因为 store 端本来就不需要 OOB padding (越界 lane 用 isetp + @P gate 控制即可)。

ldst_length 限 b32 / b64 / b128 / b256 共 4 valid。

modifier: ldst_length + mem_sem + consistency_scope + cache_evict (跟 ldgsts 同源)。

典型用例: ML kernel epilogue 写回 (MMA 算完 tile 用 stsm 写到 shared, 再用 ldsstg 异步写到 global, 计算跟写回 overlap) / cta-shared 数据写回 global 不阻塞后续算。

14.7.11 Control flow

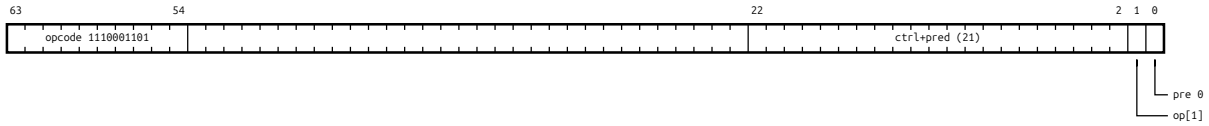
bpt **category**: Control flow **predicate_mode**: uniform

Leaf: bpt__u

Format:

```
bpt
```

Encoding Diagram: 64b, prefix 0, opcode 11110001101



Notes:

Software breakpoint: 程序执行到该位置暂停, 通过 debugger / driver host-side 触发继续执行 (跳到下一断点或直接 exit)。属于 debug 路径不影响 kernel 正确性。

无 operand。

典型用例: kernel 调试时插入断点 / 异常路径 trap (panic 等价物) / driver 强制 kernel pause 检查状态。

bra **category**: Control flow **predicate_mode**: uniform

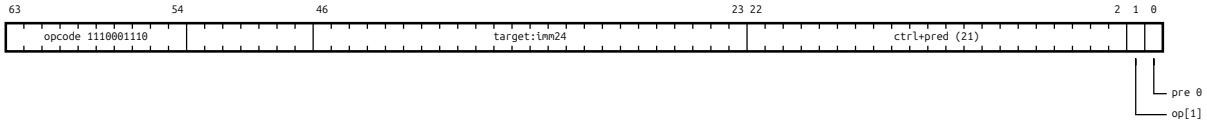
Leaf: bra__i

Format:

```
bra  target
```

Operands: target: imm24s×4 [notes: pcrel@next_pc]

Encoding Diagram: 64b, prefix 0, opcode 11110001110



Notes:

Uniform PC-relative branch: PC = next_pc + (target « 2), 其中 next_pc = 本条指令的地址 + 本条指令的长度。指令编码是 8 / 12 / 16 字节变长的, 硬件解码时已知自身长度; 偏移以 4 字节为单位 (4 是三种指令长度的最大公约数, 任何合法指令起点都能表达), target = signed 24-bit, 跳转范围 ±32 MB。整个 warp 一致跳到目标 PC。

SIMT 分支模式: 简单 if-then 分支走 isetp + @P inst predicate gate (不动 EXEC, 不用 bra); 复杂分支走 isetp + mset 序列管 EXEC mask + 走 bra 跨 basic block; loop back-edge 走 mset + mbnz 检测 EXEC + bra 跳; bra 主要用在 wave-uniform 跳转 (loop / goto / function call 入口) 场景。

brx　**category**: Control flow　**predicate_mode**: uniform

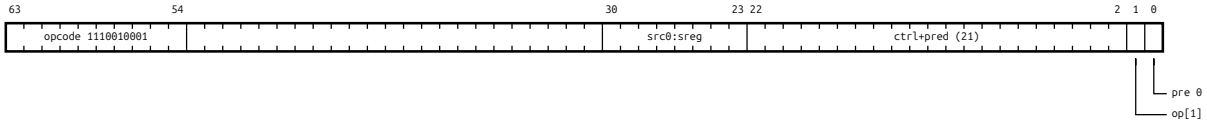
Leaf: brx__x

Format:

```
brx  src0
```

Operands: src0: sreg

Encoding Diagram: 64b, prefix 0, opcode 11110010001



Notes:

Indirect uniform PC-relative branch: PC = next_pc + (src0 « 2), next_pc = 本条指令地址 + 本条指令长度; target offset 在 sreg 里 (warp-uniform), 以 4 字节为单位。bra 的 indirect 版本 (bra 用 imm24, brx 用 sreg)。

典型用例: switch / jump-table dispatch (table base + index 算出 target offset 写到 sreg, brx 跳过去) / dynamic branch target / function pointer table dispatch (PC-relative variant)。

call　**category**: Control flow　**predicate_mode**: uniform

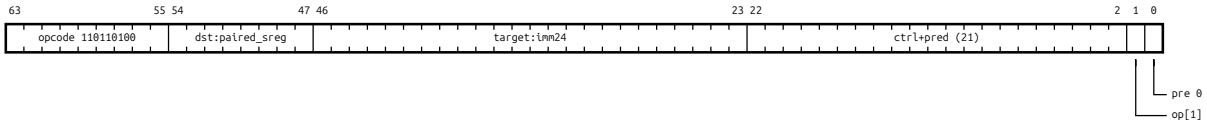
Leaf: call__x_i

Format:

```
call  dst, target
```

Operands: dst: paired_sreg; target: imm24s×4 [notes: pcrel@next_pc]

Encoding Diagram: 64b, prefix 0, opcode 1110110100



Notes:

Uniform PC-relative function call: dst = next_pc (64-bit 返回地址, 写到偶对齐 sreg pair {UR, UR+1}); PC = next_pc + (target « 2)。next_pc = 本条指令地址 + 本条指令长度 (指令编码 8 / 12 / 16 字节变长)。整 warp 一致跳到目标。target = signed 24-bit, 以 4 字节为单位, 调用范围 ±32 MB。

返回地址就是一个普通的 64-bit 值: 放在寄存器对里, 可以随意搬移或 spill 到栈内存, 调用深度只受软件栈限制。没有任何隐藏的硬件返回栈状态; 被调函数用 ret (经寄存器对间接跳转) 返回, 深递归 / coroutine 不需要特殊路径。dst 写 URZ = 丢弃返回地址 (纯尾调用场景)。

典型用例: 函数调用 (静态链接)。哪个 sreg pair 充当 link register 是编译器 ABI 约定, ISA 不指定。

callx **category**: Control flow **predicate_mode**: uniform

Leaf: callx__x_x

Format:

```
callx  dst, src0
```

Operands: dst: paired_sreg; src0: paired_sreg

Encoding Diagram: 64b, prefix 0, opcode 11110010011

Notes:

Indirect uniform function call (absolute): dst = next_pc (64-bit 返回地址, 写到偶对齐 sreg pair {UR, UR+1}); PC = src0 (64-bit 绝对地址, 偶对齐 sreg pair)。next_pc = 本条指令地址 + 本条指令长度。整 warp 一致跳到 src0 指向的入口。call 的 indirect 版本 (call 用 imm24 相对偏移, callx 用寄存器对里的绝对地址)。

返回地址就是一个普通的 64-bit 值: 放在寄存器对里, 可以随意搬移或 spill 到栈内存, 调用深度只受软件栈限制。没有任何隐藏的硬件返回栈状态。dst 写 URZ = 丢弃返回地址 (间接尾调用: 等价于 jmp)。

典型用例: virtual function call (vtable 查出绝对入口) / 函数指针 dispatch / 动态链接 indirect call / coroutine 切换 (保存对方返回地址、跳对方入口)。

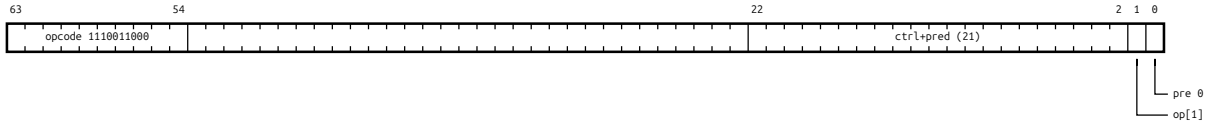
exit **category**: Control flow **predicate_mode**: none

Leaf: exit__u

Format:

```
exit
```

Encoding Diagram: 64b, prefix 0, opcode 11110011000



Notes:

Warp terminate: 整个 warp 的所有 lane 退出, 后续指令不再执行。kernel-level 退出指令, 必须放在 kernel 末尾。

predicate_mode=none (无条件): 不挂 @P / @up gate; 条件退出 (early exit when condition) 走 bra 跳过尾部 + exit 在尾部 (而不是在循环中条件 exit, 因为 exit 是 unconditional terminator)。

kill 是单 lane 退出 (per-lane EXEC mask 清), exit 是整 warp 退出 — 两者不同。

getpc **category**: Control flow **predicate_mode**: simt

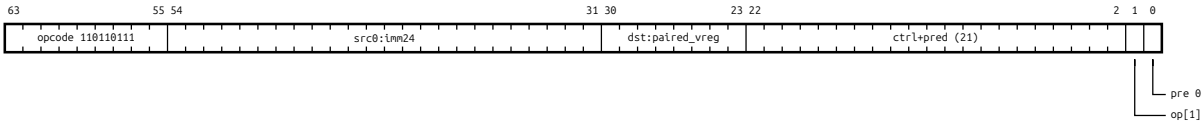
Leaf: getpc__v_i

Format:

```
getpc  dst, src0
```

Operands: dst: paired_vreg; src0: imm24s×4 [notes: pcrel@current_pc]

Encoding Diagram: 64b, prefix 0, opcode 1110110111



Notes:

Read PC + offset (per-lane): dst = pc + (src0 « 2), pc = 本条 getpc 指令自身的起始地址, offset 以 4 字节为单位。PC 是 64-bit 地址, dst 是偶对齐 vreg pair {R, R+1} (paired_vreg); 每个 active lane 写自己的 pair, warp 内 PC 一致 (warp-wide 同步执行) 所以各 lane 拿到同值。

典型用例: position-independent code (PIC, 取当前 PC 算 GOT / PLT 基址) / 跳表基址 (load 地址相对自身代码) / 指针自身定位 (self-pointer 用于 trampoline / closure)。

jmp **category**: Control flow **predicate_mode**: uniform

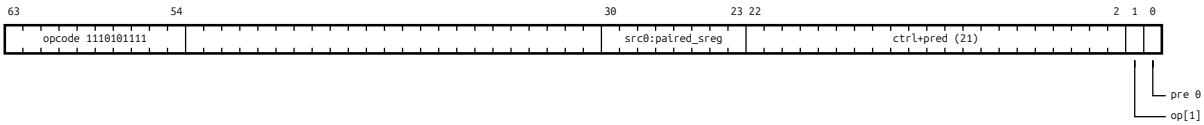
Leaf: jmp__x

Format:

```
jmp  src0
```

Operands: src0: paired_sreg

Encoding Diagram: 64b, prefix 0, opcode 11110101111



Notes:

Uniform absolute jump: PC = src0, 整 wave 一致跳到 src0 指向的 64-bit 绝对地址。跟 brx 区别: brx 是 PC-relative offset (跟 PC 相关), jmp 是 absolute 寻址 (跟 PC 无关)。

operand 模型: src0 是偶对齐 sreg pair {UR, UR+1} (paired_sreg kind), 装 64-bit absolute address。

典型用例: device 代码段内的尾跳转 (跳到与当前 kernel 使用相同 ABI、寄存器分配与资源配置的另一段 device 代码, 如 driver 预链接的公共例程; jmp 只改 PC, 不建立新的 kernel 执行上下文 — grid/CTA 维度、shared memory、寄存器分配、kernel 参数全部保持不变, 所以它不能替代 kernel dispatch) / function pointer table dispatch (table base 是 absolute address) / virtual function call (vtable lookup 后走 jmp)。

ret **category**: Control flow **predicate_mode**: uniform

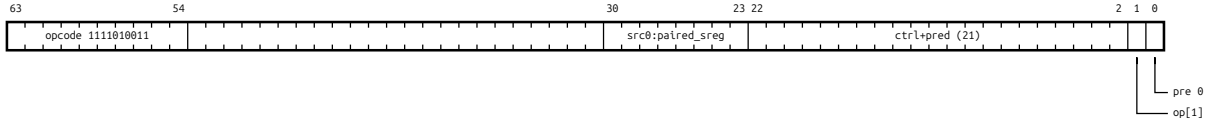
Leaf: ret__x

Format:

```
ret  src0
```

Operands: src0: paired_sreg

Encoding Diagram: 64b, prefix 0, opcode 11111010011



Notes:

Function return: PC = src0, src0 = 偶对齐 sreg pair {UR, UR+1} 里的 64-bit 返回地址 (call / callx 写入的 next_pc, 或软件从栈内存恢复的值)。整 warp 一致跳回。

语义上等价于 jmp 经同一个寄存器对跳转; 单列一条 ret 是给硬件返回地址预测和代码可读性留的显式标记。没有硬件返回栈: 返回地址全部经寄存器对流转, 调用深度没有硬件上限, 上下文切换也没有隐藏状态要保存。

ugetpc　**category**: Control flow　**predicate_mode**: uniform

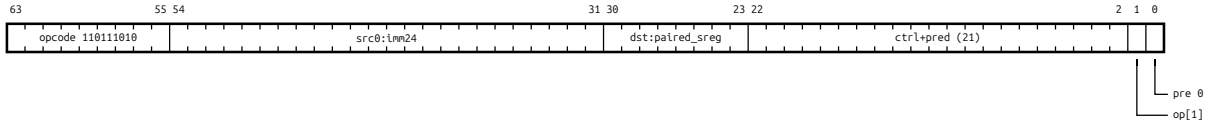
Leaf: ugetpc__x_i

Format:

```
ugetpc  dst, src0
```

Operands: dst: paired_sreg; src0: imm24s×4 [notes: pcrel@current_pc]

Encoding Diagram: 64b, prefix 0, opcode 1110111010



Notes:

Read PC + offset (warp-uniform): dst = pc + (src0 « 2), pc = 本条指令自身的起始地址, offset 以 4 字节为单位。PC 是 64-bit 地址, dst 是偶对齐 sreg pair {UR, UR+1} (paired_sreg), warp 内 1 份共享给 32 lane。getpc 的 uniform 镜像 — warp PC 本来就一致, 写到 sreg 省 vreg。

典型用例: position-independent code (PIC) / 跳表基址 / 指针自身定位 (PIC kernel 主路径用 ugetpc 比 getpc 更省 vreg)。

14.7.12　Ray tracing

rt_release　**category**: Ray tracing　**predicate_mode**: simt

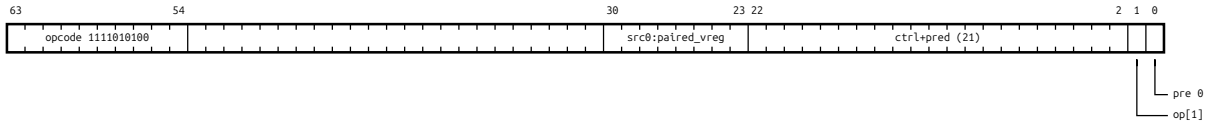
Leaf: rt_release___v

Format:

```
rt_release  src0
```

Operands: src0: paired_vreg

Encoding Diagram: 64b, prefix 0, opcode 11111010100



Notes:

Software-scheduled ray tracing terminal release hard operation. Effective active mask = EXEC 与本指令 predicate gate 的逐 lane 交集；每个 active lane 独立释放 src0 指向的 matching public/private resident authority。Inactive lane 不读取 handoff，也不改变任何 lifecycle state。

src0 (handoff_addr) 是 paired_vreg：每个 active lane 自己的 64-bit public handoff 地址。RTCore/scheduler 从 handoff 与 resident binding 中读取并验证 context_address、lane/owner/transaction、generation 和 terminal lifecycle；stale、wrong-owner 或 still-resumable context 必须拒绝，不能释放另一 lane 或另一 transaction 的资源。

rt_release 只在不再存在合法 traversal resume 或未完成 caller continuation，terminal status/output 已发布，required return/checkpoint 与 consumer capture 已完成后合法。terminateRay、accept-and-end-search、shader ordinary return 和 child/callable return 都不等于 rt_release。

本指令没有 architectural destination。完成效果包括使 matching public authority 和仍存在的 private resident authority 失效、推进 reuse protection，并允许生命周期管理者在 ordering 条件满足后复用 allocation；具体物理回收时机可以晚于指令完成，但不能重新暴露已 retire 的旧 authority。需要等待 release side effect 的后续操作使用 GPIDL 公共 dependence-counter/ordering 机制。

rt_traverse **category**: Ray tracing **predicate_mode**: simt

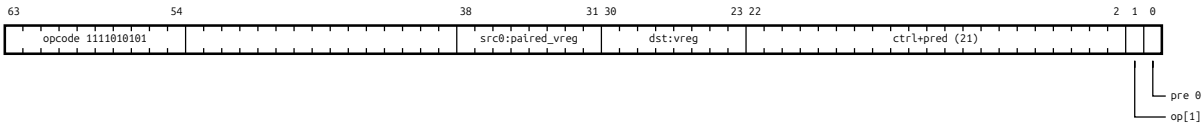
Leaf: rt_traverse__v_v

Format:

```
rt_traverse  dst, src0
```

Operands: dst: vreg; src0: paired_vreg

Encoding Diagram: 64b, prefix 0, opcode 11111010101



Notes:

Software-scheduled ray tracing traverse/resume hard operation. Effective active mask = EXEC 与本指令 predicate gate 的逐 lane 交集；每个 active lane 独立解释自己的 dst 与 src0。Inactive lane 不读取 handoff、不创建 request、不写 dst，也不改变 RTCore state。

dst (result_out) 是每个 active lane 的 32-bit completion-control destination。具体 completion class/readiness 编码由所选 RT ABI profile 定义；RTCore 必须先发布该 completion class 依赖的 public handoff/formal facts，再完成 dst writeback 和 scheduler/scoreboard dependency。软件先等待该 dependency，再按同一 transaction acquire-validate result 与 dependent facts。

src0 (handoff_addr) 是 paired_vreg：每个 active lane 自己的 64-bit public handoff 地址，低 32 位在偶数号基寄存器、高 32 位在下一寄存器。Handoff 提供 context_address、new launch 的 immutable trace_input 或 resume 所需 traversal-return fields，以及 owner/transaction/profile/readiness proof。Context、AS、descriptor、SBT 与 dispatch state 都不是额外 hard operands。

New launch 对当前 non-empty active mask 做原子 admission，并为每个 active lane 创建独立 request。Resume 必须验证当前 mask 是 matching resident previous mask 的非空子集：selected lane 在原 private state 上继续，omitted previous lane 通过 mask-shrink reconciliation 结束 request ownership。继续集合为空时不得发空 rt_traverse，backend 应进入 terminal rt_release 路径。

本指令是 convergent、side-effecting、可变长延迟 suspension boundary。它使用 GPIDL 公共 dependence-counter 控制位承载 long-latency dependency，不新增专用 RT wait opcode；handoff publish -> traverse、completion

publish -> result writeback -> dependent acquire 的顺序不得被编译器或硬件跨越。

14.7.13 Mask and predicate

kill **category**: Mask and predicate **predicate_mode**: simt

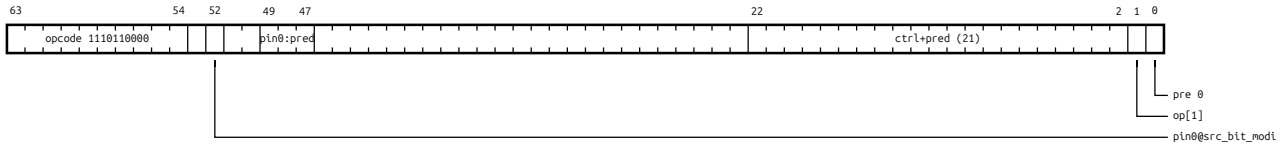
Leaf: kill___p

Format:

```
kill  pin0{.src_bit_modi}
```

Operands: pin0: pred

Encoding Diagram: 64b, prefix 0, opcode 11110110000



Notes:

Per-lane self-kill: pin0=true 的 lane 把自己从 EXEC 清除 (EXEC &= ~current_lane_bit), 该 lane 在 kernel 退出前不再执行任何指令。pin0=PT 表示无条件 kill 当前所有 active lane。

kill vs exit: kill 是 per-lane 自杀 (单 lane 退出, 其他 lane 继续), exit 是整 warp 退出。被 kill 的 lane 从此不在 EXEC 中, 因此也不再计入 cta barrier 的到达数 (bar_sync / bar_arrive / bar_red 按门控后 active lane 逐 lane 计数): 程序若按含该 lane 的 thread_count 建 barrier, 死锁属于程序错误。

kill vs mset.andn2: kill 等价于 mset.andn2 + one-hot mask (仅当前 lane bit), 但 kill 单条指令更直接 + 不需准备 one-hot mask。

典型用例: 发散区的 early exit (一个 lane 完成工作时退出, 让其它 lane 继续) / 异常 lane 终止 (例如 fragment shader 中 discard 的 lane)。

mall **category**: Mask and predicate **predicate_mode**: uniform

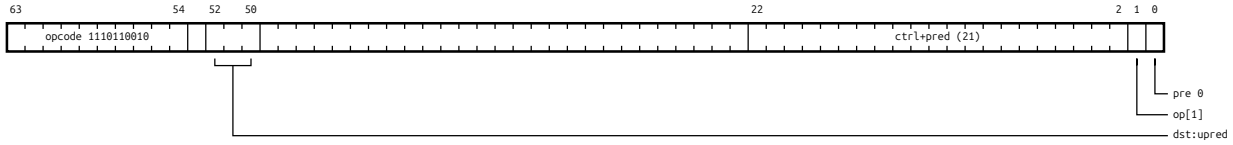
Leaf: mall__p_

Format:

```
mall  dst
```

Operands: dst: upred

Encoding Diagram: 64b, prefix 0, opcode 11110110010



Notes:

EXEC all-active query: dst (uniform predicate) = (EXEC has all 32 lanes set) ? true : false。查询当前 warp 是否所有 lane 都 active (完全收敛, EXEC 全 1)。

用法: 检测 fully-convergent execution state — 编译器优化 (如果 mall=true, 后续序列可走 SIMT-friendly fast path 跳过 divergent path) / kernel 进入 matrix multiply 内 loop 等高性能区段前的 sanity check。

many **category**: Mask and predicate **predicate_mode**: uniform

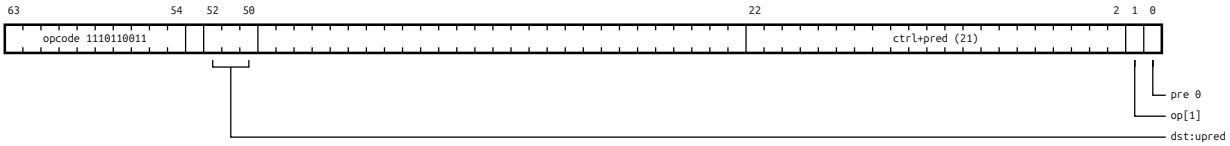
Leaf: many__p_

Format:

```
many  dst
```

Operands: dst: upred

Encoding Diagram: 64b, prefix 0, opcode 11110110011



Notes:

EXEC any-active query: dst (uniform predicate) = (EXEC != 0) ? true : false。查询当前 warp 是否还有任何 lane active, 结果写到 upred (broadcast 给所有 lane)。

用法: 配合 @up bra / @!up bra (uniform branch) 检测 dead path 或 close loop。也可走 mbnz 路径 (mbnz 是 EXEC≠0 检测 + 跳转 fused, 比 many + @!up bra 序列省 1 条)。

mbnz **category**: Mask and predicate **predicate_mode**: uniform

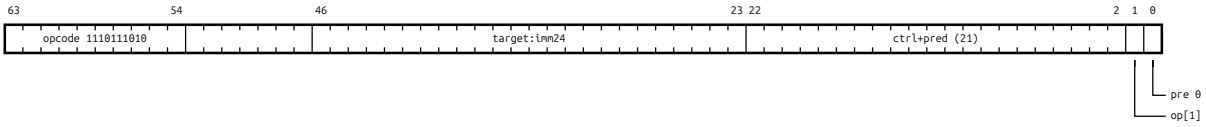
Leaf: mbnz__i

Format:

```
mbnz  target
```

Operands: target: imm24s×4 [notes: pcrel@next_pc]

Encoding Diagram: 64b, prefix 0, opcode 11110111010



Notes:

Branch if EXEC != 0: 任何 lane 还 active 时跳转 (PC = next_pc + (target « 2), next_pc = 本条指令地址 + 本条指令长度), 否则 fall through。跟 mbz 对偶 (n=non, z=zero)。target = signed 24-bit PC-relative offset, 以 4 字节为单位。

典型用例: loop back-edge on per-lane condition (循环条件: 任何 lane 还有 work to do 即继续) — 配合 mset 修改 EXEC + mbnz 检测构成 SIMT loop 主路径。

mbz **category**: Mask and predicate **predicate_mode**: uniform

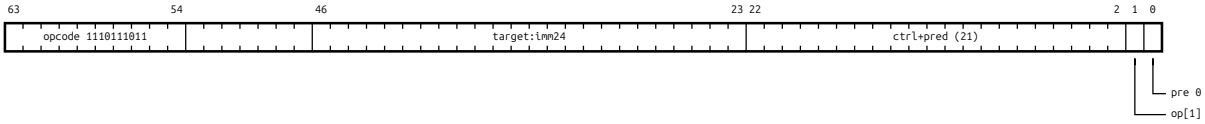
Leaf: mbz__i

Format:

```
mbz  target
```

Operands: target: imm24s×4 [notes: pcrel@next_pc]

Encoding Diagram: 64b, prefix 0, opcode 11110111011



Notes:

Branch if EXEC == 0: 整个 warp 没有 active lane 时跳转 (PC = next_pc + (target « 2), next_pc = 本条指令地址 + 本条指令长度), 否则 fall through。fused 1 条 = many + @!up bra 序列 2 条等价。target = signed 24-bit PC-relative offset, 以 4 字节为单位。

典型用例: 跳过 dead path (所有 lane 都被 mask 掉, 不需执行后续 body) / divergent loop 终止条件 (没 lane 满足继续条件即跳出循环)。

mpopc **category**: Mask and predicate **predicate_mode**: uniform

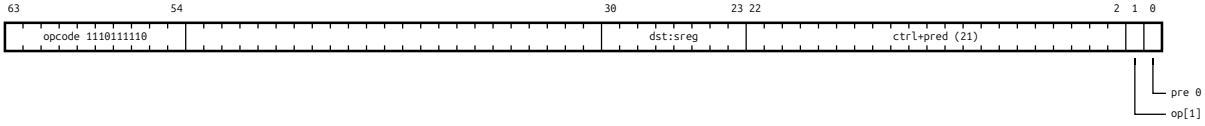
Leaf: mpopc__x_

Format:

```
mpopc  dst
```

Operands: dst: sreg

Encoding Diagram: 64b, prefix 0, opcode 11110111110



Notes:

EXEC popcount: dst = popcount(EXEC), 数当前 warp 里有多少 lane active (EXEC bit=1), 结果写到 sreg (warp 内 1 个 32-bit slot, broadcast 给所有 lane)。结果范围 0-32 (warp 32 lane)。

跟 many / mall 区别: many / mall 是布尔二态 (EXEC != 0 / EXEC 全 1), mpopc 是精确 count。

典型用例: dynamic load balance (per-warp work distribution) / parallel reduction inner-most (prefix sum of EXEC bits 估算 thread cooperation level) / convergent control flow detection (active lane count == warp size 表示完全收敛)。

mset **category**: Mask and predicate **predicate_mode**: uniform

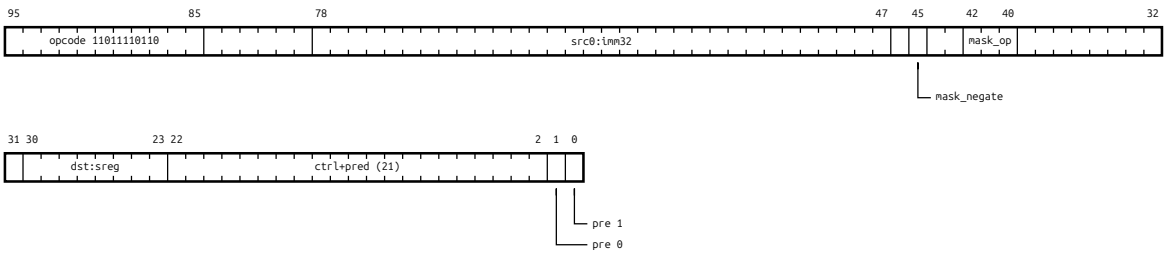
Leaf: mset___x_i

Format:

```
mset{.mask_op}{.mask_negate}  dst, src0
```

Operands: dst: sreg; src0: imm32u

Encoding Diagram: 96b, prefix 10, opcode 11011110110



Notes:

Set EXEC mask + save 旧 mask 一条搞定: dst = old_EXEC (RZ= 不 save); EXEC = (mask_op==nop) ? src : (src op old_EXEC); 如果 mask_negate=on 再对 EXEC 取反。SIMT 分支主路径 (复杂分支需要保存 / 恢复 EXEC mask 时使用)。

mask_op 6 valid: nop (直接赋值 EXEC = src) / and (EXEC = src ∧ old_EXEC, 缩小 active set) / or (EXEC = src ∨ old_EXEC, 扩大 active set) / xor (异或) / andn2 (EXEC = src ∧ ¬old_EXEC, ELSE 路径主路径) / NOR (EXEC = ¬(src ∨ old_EXEC))。

mask_negate 2 valid: off (默认) / on (mask_op 完成后再对 EXEC 全位取反)。

SIMT 分支模式: 简单 if-then 走 predicate gate (isetp + @P inst, 不动 EXEC); 复杂分支序列: isetp.lt P1, R0, t → mset.and saved, P1 (save EXEC + apply P1, THEN 路径) → mset.andn2 _, saved (ELSE 路径) → mset.nop _, saved (恢复 EXEC)。

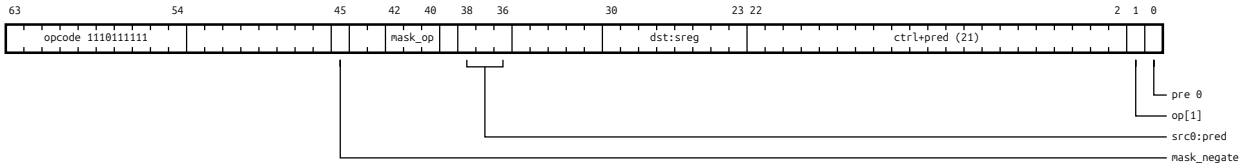
Leaf: mset___x_p

Format:

```
mset{.mask_op}{.mask_negate}  dst, src0
```

Operands: dst: sreg; src0: pred

Encoding Diagram: 64b, prefix 0, opcode 11110111111



Notes:

Set EXEC mask + save 旧 mask 一条搞定: dst = old_EXEC (RZ= 不 save); EXEC = (mask_op==nop) ? src : (src op old_EXEC); 如果 mask_negate=on 再对 EXEC 取反。SIMT 分支主路径 (复杂分支需要保存 / 恢复 EXEC mask 时使用)。

mask_op 6 valid: nop (直接赋值 EXEC = src) / and (EXEC = src ∧ old_EXEC, 缩小 active set) / or (EXEC = src ∨ old_EXEC, 扩大 active set) / xor (异或) / andn2 (EXEC = src ∧ ¬old_EXEC, ELSE 路径主路径) / NOR (EXEC = ¬(src ∨ old_EXEC))。

mask_negate 2 valid: off (默认) / on (mask_op 完成后再对 EXEC 全位取反)。

SIMT 分支模式: 简单 if-then 走 predicate gate (isetp + @P inst, 不动 EXEC); 复杂分支序列: isetp.lt P1, R0, t → mset.and saved, P1 (save EXEC + apply P1, THEN 路径) → mset.andn2 _, saved (ELSE 路径) → mset.nop _, saved (恢复 EXEC)。

predicate→mask 转换语义: src_mask = old_EXEC & lanes_where(src0_pred=true), inactive lane (old_EXEC bit=false) 的 src_mask bit 强制清零, 防止 predicate register 在 inactive lane 上的残留值让 mset.or 等操作错误激活已 kill 的 lane (predicate 寄存器在 inactive lane 上是 don’t care)。

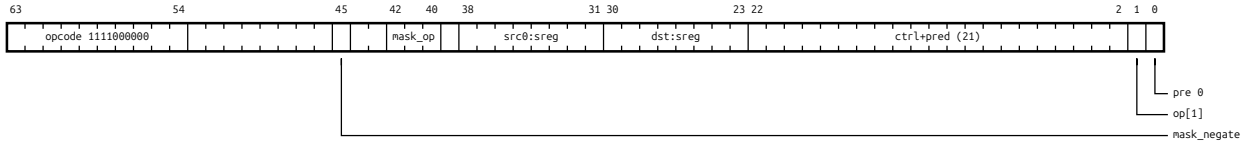
Leaf: mset___x_x

Format:

```
mset{.mask_op}{.mask_negate}  dst, src0
```

Operands: dst: sreg; src0: sreg

Encoding Diagram: 64b, prefix 0, opcode 11111000000



Notes:

Set EXEC mask + save 旧 mask 一条搞定: dst = old_EXEC (RZ= 不 save); EXEC = (mask_op==nop) ? src : (src op old_EXEC); 如果 mask_negate=on 再对 EXEC 取反。SIMT 分支主路径 (复杂分支需要保存 / 恢复 EXEC mask 时使用)。

mask_op 6 valid: nop (直接赋值 EXEC = src) / and (EXEC = src ∧ old_EXEC, 缩小 active set) / or (EXEC = src ∨ old_EXEC, 扩大 active set) / xor (异或) / andn2 (EXEC = src ∧ ¬old_EXEC, ELSE 路径主路径) / NOR (EXEC = ¬(src ∨ old_EXEC))。

mask_negate 2 valid: off (默认) / on (mask_op 完成后再对 EXEC 全位取反)。

SIMT 分支模式: 简单 if-then 走 predicate gate (isetp + @P inst, 不动 EXEC); 复杂分支序列: isetp.lt P1, R0, t → mset.and saved, P1 (save EXEC + apply P1, THEN 路径) → mset.andn2 _, saved (ELSE 路径) → mset.nop _, saved (恢复 EXEC)。

plop3 **category**: Mask and predicate **predicate_mode**: simt

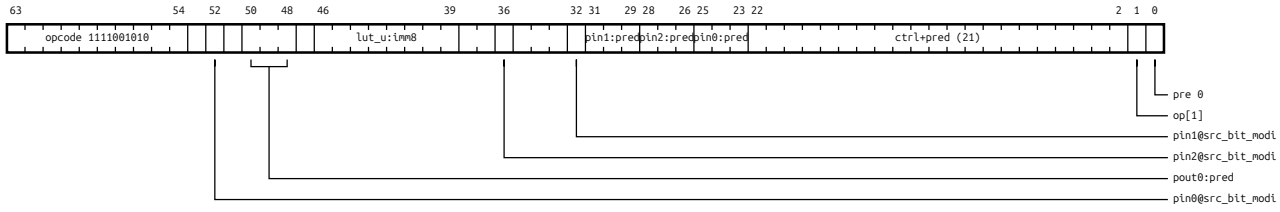
Leaf: plop3__p_pppi

Format:

```
plop3  pout0, pin0{.src_bit_modi}, pin1{.src_bit_modi}, pin2{.src_bit_modi}, lut_u
```

Operands: pout0: pred; pin0: pred; pin1: pred; pin2: pred; lut_u: imm8u

Encoding Diagram: 64b, prefix 0, opcode 11111001010



Notes:

3-input predicate 真值表查表: 每个 input pred (pin0 / pin1 / pin2) 是 1-bit, 3-bit 共 2^3=8 种组合; 8-bit imm LUT 完整描述任意 3-input boolean function。整 warp per-lane 各自计算。

pout0 计算: bit_idx = (pin0 « 2) | (pin1 « 1) | pin2 (pin0 是 msb), pout0 = (lut_u » bit_idx) & 1。

双输出指令形态的 pout1 用独立的 lut_v (跟 pout0 的 lut_u 完全独立, 同一组 pin0..pin2 输入), 两个 boolean function 一条指令算两个。

常用 LUT 值: AND3 = 0x80 (a∧b∧c), OR3 = 0xFE (a∨b∨c), XOR3 = 0x96 (a⊕b⊕c), majority(a,b,c) = 0xE8 (3 个里至少 2 个为真)。

pin0 / pin1 / pin2 各可挂 src_bit_modi: id / inv (取反), 配合 LUT 进一步压缩复合谓词逻辑表达。

典型用例: 复合谓词逻辑 (if (a && b) || c)、predicate-driven 控制流优化、bit logic on flags。

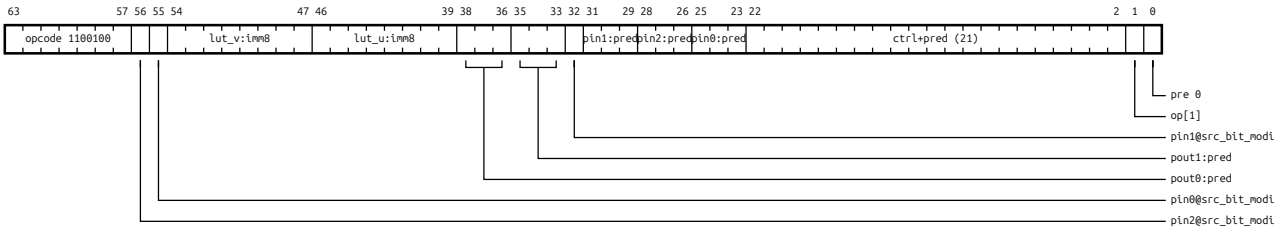
Leaf: plop3__pp_pppii

Format:

```
plop3  pout0, pout1, pin0{.src_bit_modi}, pin1{.src_bit_modi}, pin2{.src_bit_modi},
↪  lut_u, lut_v
```

Operands: pout0: pred; pout1: pred; pin0: pred; pin1: pred; pin2: pred; lut_u: imm8u; lut_v: imm8u

Encoding Diagram: 64b, prefix 0, opcode 11100100



Notes:

3-input predicate 真值表查表: 每个 input pred (pin0 / pin1 / pin2) 是 1-bit, 3-bit 共 2^3=8 种组合; 8-bit imm LUT 完整描述任意 3-input boolean function。整 warp per-lane 各自计算。

pout0 计算: bit_idx = (pin0 « 2) | (pin1 « 1) | pin2 (pin0 是 msb), pout0 = (lut_u » bit_idx) & 1。

双输出指令形态的 pout1 用独立的 lut_v (跟 pout0 的 lut_u 完全独立, 同一组 pin0..pin2 输入), 两个 boolean function 一条指令算两个。

常用 LUT 值: AND3 = 0x80 (a∧b∧c), OR3 = 0xFE (a∨b∨c), XOR3 = 0x96 (a⊕b⊕c), majority(a,b,c) = 0xE8 (3 个里至少 2 个为真)。

pin0 / pin1 / pin2 各可挂 src_bit_modi: id / inv (取反), 配合 LUT 进一步压缩复合谓词逻辑表达。

典型用例: 复合谓词逻辑 (if (a && b) || c)、predicate-driven 控制流优化、bit logic on flags。

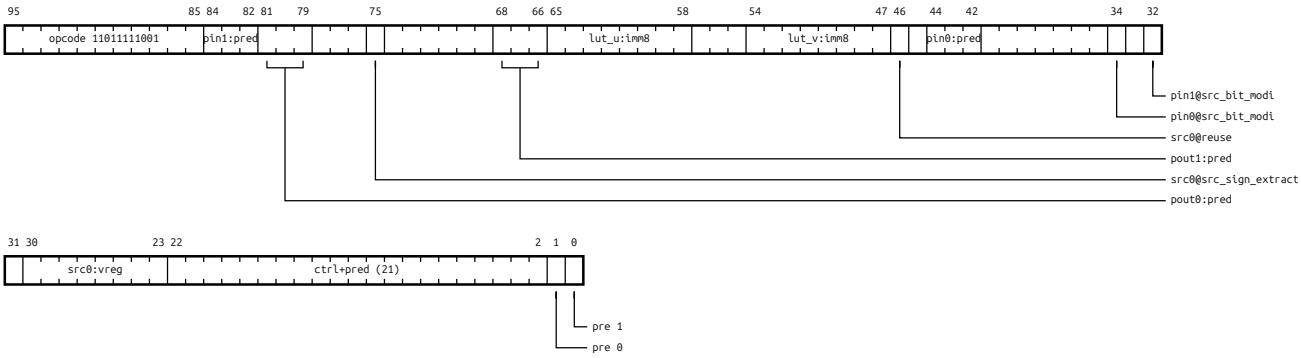
Leaf: plop3__pp_ppvii

Format:

```
plop3  pout0, pout1, pin0{.src_bit_modi}, pin1{.src_bit_modi},
↪  src0{.src_sign_extract}{.reuse}, lut_u, lut_v
```

Operands: pout0: pred; pout1: pred; pin0: pred; pin1: pred; src0: vreg; lut_u: imm8u; lut_v: imm8u

Encoding Diagram: 96b, prefix 10, opcode 11011111001



Notes:

3-input predicate 真值表查表: 每个 input pred (pin0 / pin1 / pin2) 是 1-bit, 3-bit 共 2^3=8 种组合; 8-bit imm LUT 完整描述任意 3-input boolean function。整 warp per-lane 各自计算。

pout0 计算: bit_idx = (pin0 « 2) | (pin1 « 1) | pin2 (pin0 是 msb), pout0 = (lut_u » bit_idx) & 1。

双输出指令形态的 pout1 用独立的 lut_v (跟 pout0 的 lut_u 完全独立, 同一组 pin0..pin2 输入), 两个 boolean function 一条指令算两个。

常用 LUT 值: AND3 = 0x80 (a∧b∧c), OR3 = 0xFE (a∨b∨c), XOR3 = 0x96 (a⊕b⊕c), majority(a,b,c) = 0xE8 (3 个里至少 2 个为真)。

pin0 / pin1 / pin2 各可挂 src_bit_modi: id / inv (取反), 配合 LUT 进一步压缩复合谓词逻辑表达。

典型用例: 复合谓词逻辑 (if (a && b) || c)、predicate-driven 控制流优化、bit logic on flags。

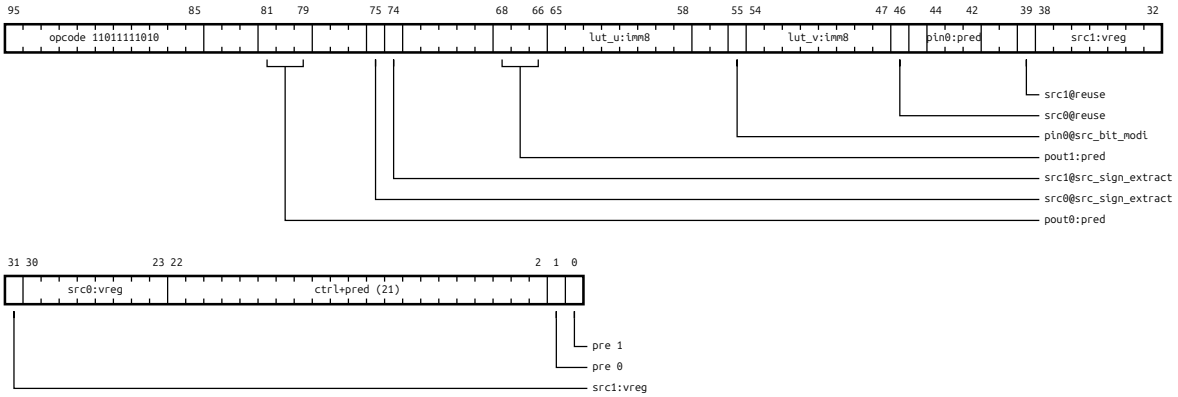
Leaf: plop3__pp_pvvii

Format:

```
plop3  pout0, pout1, pin0{.src_bit_modi}, src0{.src_sign_extract}{.reuse},
↪  src1{.src_sign_extract}{.reuse}, lut_u, lut_v
```

Operands: pout0: pred; pout1: pred; pin0: pred; src0: vreg; src1: vreg; lut_u: imm8u; lut_v: imm8u

Encoding Diagram: 96b, prefix 10, opcode 11011111010



Notes:

3-input predicate 真值表查表: 每个 input pred (pin0 / pin1 / pin2) 是 1-bit, 3-bit 共 2^3=8 种组合; 8-bit imm LUT 完整描述任意 3-input boolean function。整 warp per-lane 各自计算。

pout0 计算: bit_idx = (pin0 « 2) | (pin1 « 1) | pin2 (pin0 是 msb), pout0 = (lut_u » bit_idx) & 1。

双输出指令形态的 pout1 用独立的 lut_v (跟 pout0 的 lut_u 完全独立, 同一组 pin0..pin2 输入), 两个 boolean function 一条指令算两个。

常用 LUT 值: AND3 = 0x80 (a∧b∧c), OR3 = 0xFE (a∨b∨c), XOR3 = 0x96 (a⊕b⊕c), majority(a,b,c) = 0xE8 (3 个里至少 2 个为真)。

pin0 / pin1 / pin2 各可挂 src_bit_modi: id / inv (取反), 配合 LUT 进一步压缩复合谓词逻辑表达。

典型用例: 复合谓词逻辑 (if (a && b) || c)、predicate-driven 控制流优化、bit logic on flags。

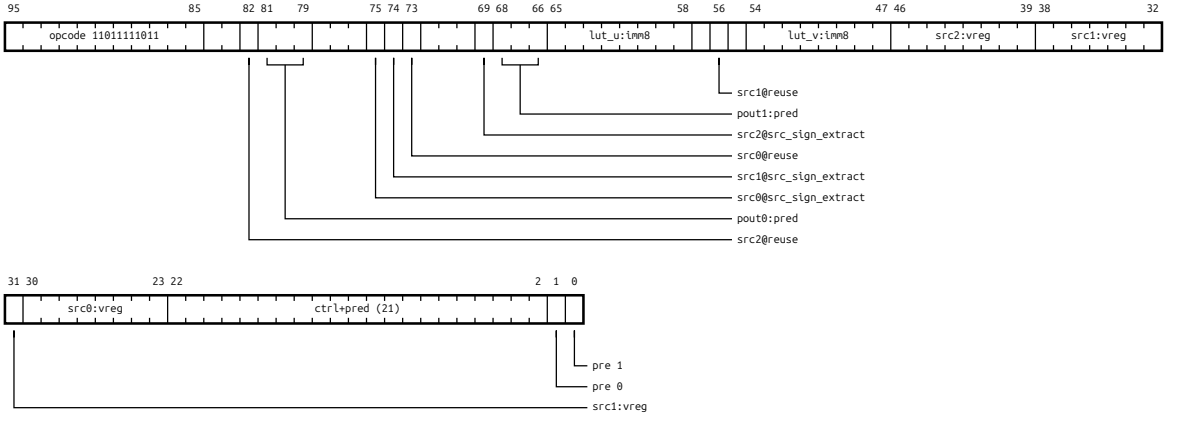
Leaf: plop3__pp_vvvii

Format:

```
plop3  pout0, pout1, src0{.src_sign_extract}{.reuse}, src1{.src_sign_extract}{.reuse},
↪  src2{.src_sign_extract}{.reuse}, lut_u, lut_v
```

Operands: pout0: pred; pout1: pred; src0: vreg; src1: vreg; src2: vreg; lut_u: imm8u; lut_v: imm8u

Encoding Diagram: 96b, prefix 10, opcode 11011111011



Notes:

3-input predicate 真值表查表: 每个 input pred (pin0 / pin1 / pin2) 是 1-bit, 3-bit 共 2^3=8 种组合; 8-bit imm LUT 完整描述任意 3-input boolean function。整 warp per-lane 各自计算。

pout0 计算: bit_idx = (pin0 « 2) | (pin1 « 1) | pin2 (pin0 是 msb), pout0 = (lut_u » bit_idx) & 1。

双输出指令形态的 pout1 用独立的 lut_v (跟 pout0 的 lut_u 完全独立, 同一组 pin0..pin2 输入), 两个 boolean function 一条指令算两个。

常用 LUT 值: AND3 = 0x80 (a∧b∧c), OR3 = 0xFE (a∨b∨c), XOR3 = 0x96 (a⊕b⊕c), majority(a,b,c) = 0xE8 (3 个里至少 2 个为真)。

pin0 / pin1 / pin2 各可挂 src_bit_modi: id / inv (取反), 配合 LUT 进一步压缩复合谓词逻辑表达。

典型用例: 复合谓词逻辑 (if (a && b) || c)、predicate-driven 控制流优化、bit logic on flags。

uplop3 **category**: Mask and predicate **predicate_mode**: uniform

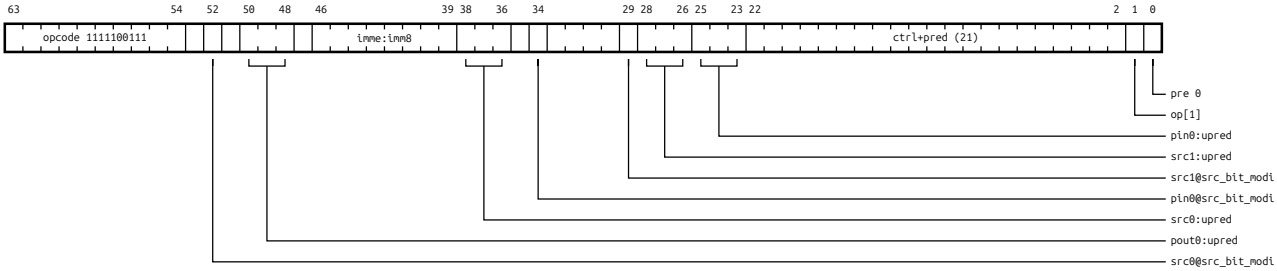
Leaf: uplop3__p_pppi

Format:

```
uplop3  pout0, pin0{.src_bit_modi}, src0{.src_bit_modi}, src1{.src_bit_modi}, imme
```

Operands: pout0: upred; pin0: upred; src0: upred; src1: upred; imme: imm8u

Encoding Diagram: 64b, prefix 0, opcode 11111100111



Notes:

3-input predicate 真值表查表 uniform 变体: 所有 input/output predicate 都是 upred (per-warp), 整 warp 共享一份结果。

LUT 语义跟 plop3 一致: bit_idx = (pin0 « 2) | (pin1 « 1) | pin2, pout0 = lut_u[bit_idx]; 双输出指令形态的 pout1 用独立 lut_v。

src 也可挂 src_bit_modi: id / inv; 部分指令形态的某些 src 是 sreg (走 sign_extract 把 sreg 的某 bit 当 boolean)。

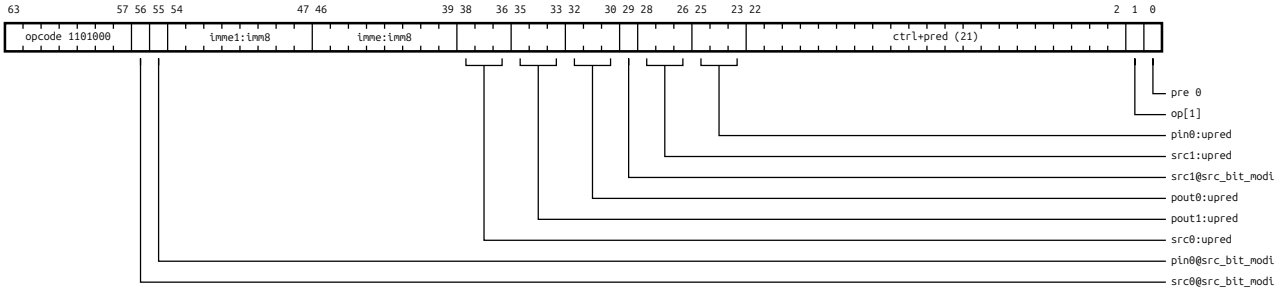
Leaf: uplop3__pp_pppii

Format:

```
uplop3  pout0, pout1, pin0{.src_bit_modi}, src0{.src_bit_modi}, src1{.src_bit_modi},
↪  imme, imme1
```

Operands: pout0: upred; pout1: upred; pin0: upred; src0: upred; src1: upred; imme: imm8u; imme1: imm8u

Encoding Diagram: 64b, prefix 0, opcode 11101000



Notes:

3-input predicate 真值表查表 uniform 变体: 所有 input/output predicate 都是 upred (per-warp), 整 warp 共享一份结果。

LUT 语义跟 plop3 一致: bit_idx = (pin0 « 2) | (pin1 « 1) | pin2, pout0 = lut_u[bit_idx]; 双输出指令形态的 pout1 用独立 lut_v。

src 也可挂 src_bit_modi: id / inv; 部分指令形态的某些 src 是 sreg (走 sign_extract 把 sreg 的某 bit 当 boolean)。

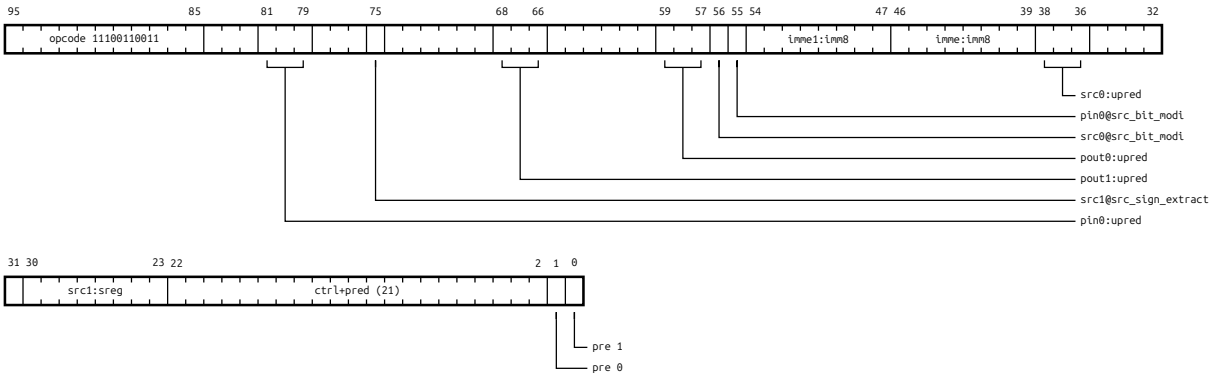
Leaf: uplop3__pp_ppxii

Format:

```
uplop3  pout0, pout1, pin0{.src_bit_modi}, src0{.src_bit_modi}, src1{.src_sign_extract},
↪  imme, imme1
```

Operands: pout0: upred; pout1: upred; pin0: upred; src0: upred; src1: sreg; imme: imm8u; imme1: imm8u

Encoding Diagram: 96b, prefix 10, opcode 11100110011



Notes:

3-input predicate 真值表查表 uniform 变体: 所有 input/output predicate 都是 upred (per-warp), 整 warp 共享一份结果。

LUT 语义跟 plop3 一致: bit_idx = (pin0 « 2) | (pin1 « 1) | pin2, pout0 = lut_u[bit_idx]; 双输出指令形态的 pout1 用独立 lut_v。

src 也可挂 src_bit_modi: id / inv; 部分指令形态的某些 src 是 sreg (走 sign_extract 把 sreg 的某 bit 当 boolean)。

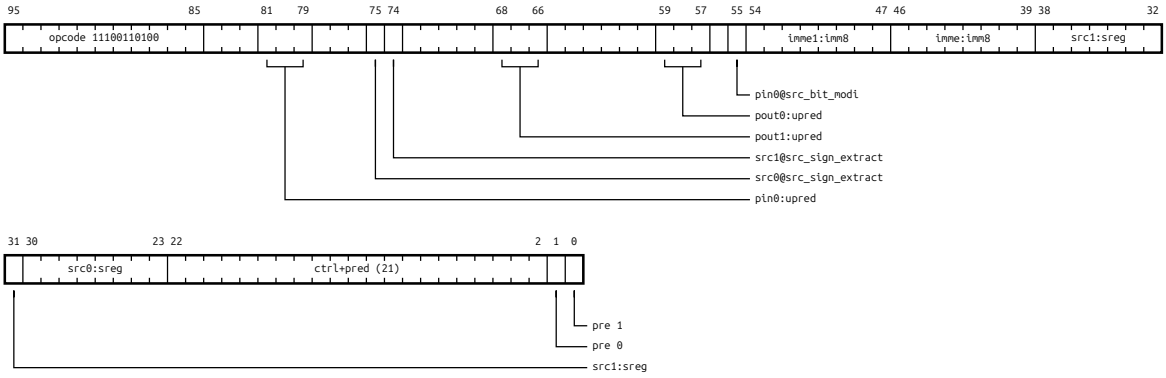
Leaf: uplop3__pp_pxxii

Format:

```
uplop3  pout0, pout1, pin0{.src_bit_modi}, src0{.src_sign_extract},
↪  src1{.src_sign_extract}, imme, imme1
```

Operands: pout0: upred; pout1: upred; pin0: upred; src0: sreg; src1: sreg; imme: imm8u; imme1: imm8u

Encoding Diagram: 96b, prefix 10, opcode 11100110100



Notes:

3-input predicate 真值表查表 uniform 变体: 所有 input/output predicate 都是 upred (per-warp), 整 warp 共享一份结果。

LUT 语义跟 plop3 一致: bit_idx = (pin0 « 2) | (pin1 « 1) | pin2, pout0 = lut_u[bit_idx]; 双输出指令形态的 pout1 用独立 lut_v。

src 也可挂 src_bit_modi: id / inv; 部分指令形态的某些 src 是 sreg (走 sign_extract 把 sreg 的某 bit 当 boolean)。

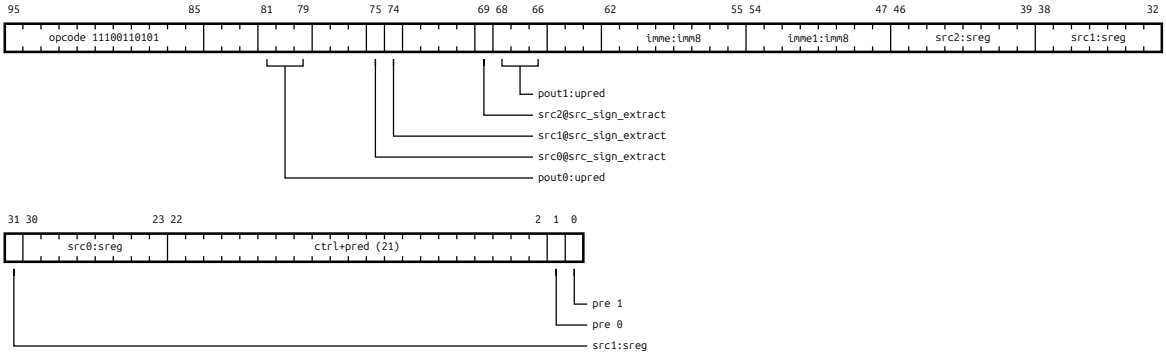
Leaf: uplop3__pp_xxxii

Format:

```
uplop3  pout0, pout1, src0{.src_sign_extract}, src1{.src_sign_extract},
↪  src2{.src_sign_extract}, imme, imme1
```

Operands: pout0: upred; pout1: upred; src0: sreg; src1: sreg; src2: sreg; imme: imm8u; imme1: imm8u

Encoding Diagram: 96b, prefix 10, opcode 11100110101



Notes:

3-input predicate 真值表查表 uniform 变体: 所有 input/output predicate 都是 upred (per-warp), 整 warp 共享一份结果。

LUT 语义跟 plop3 一致: bit_idx = (pin0 « 2) | (pin1 « 1) | pin2, pout0 = lut_u[bit_idx]; 双输出指令形态的 pout1 用独立 lut_v。

src 也可挂 src_bit_modi: id / inv; 部分指令形态的某些 src 是 sreg (走 sign_extract 把 sreg 的某 bit 当 boolean)。

14.7.14 Cross-lane and collective

elect **category**: Cross-lane and collective **predicate_mode**: simt

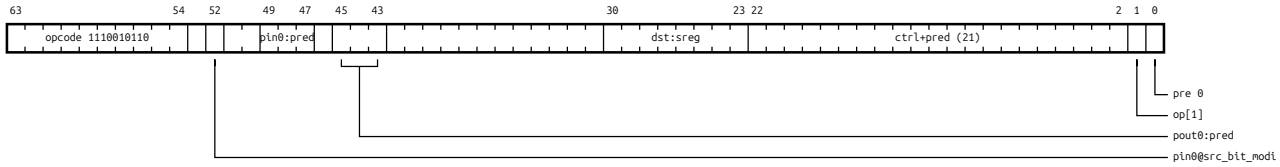
Leaf: elect__xp_p

Format:

```
elect  dst, pout0, pin0{.src_bit_modi}
```

Operands: dst: sreg; pout0: pred; pin0: pred

Encoding Diagram: 64b, prefix 0, opcode 11110010110



Notes:

Warp leader election: warp 内 32 lane 集体选举 — 挑一个候选 (EXEC ∩ candidate) 集合里最低 index 的 lane 当 leader, 所有 lane 同时拿到两个回答: dst 写 leader lane id (warp-uniform), pout0 在 leader lane 写 true / 其他 lane 写 false (per-lane)。

指令形态区分 (candidate 来源): xp_x (src0 = ureg member_mask, 32-bit uniform mask, bit i=1 表示 lane i 参与选举, candidate = EXEC ∩ src0; mask 风格主路径) / xp_p (src0 = per-lane pin0 predicate, 每个 lane 自己的 predicate 决定参不参与, candidate = EXEC ∩ {lanes where pin0=true}; 用 pred 直接表达条件, 省 isetp + r2u 准备 mask 步骤)。

operand 语义: dst (URd, sreg) = leader lane id (0..31, warp 内所有 lane 看到同一个值); pout0 (pred per-lane) = '当前 lane 是不是 leader' (leader=true / 其他 =false, 用作 @P0 do_leader_work 单 leader 序列 gate); src0 = ureg member_mask (xp_x, 挂 src_bit_modi 支持 [~]URa 位取反表达' 除了这些 lane 之外') 或 per-lane pin0 predicate (xp_p, 挂 src_bit_modi 支持 [!]pin0 取反)。

leader 选举确定性: 永远挑最低 index 的 active lane 作 leader, 保证多次执行同 mask 选同 leader 可重现。

典型用例: atomic per-warp aggregation (elect xp_x UR_lid, P0, UR_mask; @P0 atomicAdd(addr, agg_val); 让 leader lane 单独做 atomic 减少冲突) / per-warp print / log (@P0 printf(…) 单 leader 输出避免 32 倍重复) / cooperative kernel 入口 (@P0 setup_warp_state(…) 单 lane init)。

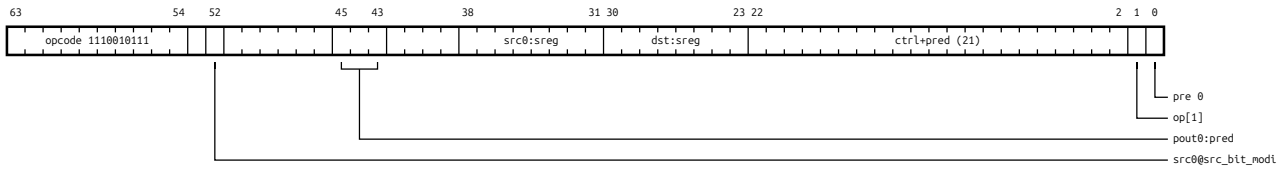
Leaf: elect__xp_x

Format:

```
elect  dst, pout0, src0{.src_bit_modi}
```

Operands: dst: sreg; pout0: pred; src0: sreg

Encoding Diagram: 64b, prefix 0, opcode 11110010111



Notes:

Warp leader election: warp 内 32 lane 集体选举 — 挑一个候选 (EXEC ∩ candidate) 集合里最低 index 的 lane 当 leader, 所有 lane 同时拿到两个回答: dst 写 leader lane id (warp-uniform), pout0 在 leader lane 写 true / 其他 lane 写 false (per-lane)。

指令形态区分 (candidate 来源): xp_x (src0 = ureg member_mask, 32-bit uniform mask, bit i=1 表示 lane i 参与选举, candidate = EXEC ∩ src0; mask 风格主路径) / xp_p (src0 = per-lane pin0 predicate, 每个 lane 自己的

predicate 决定参不参与, candidate = EXEC ∩ {lanes where pin0=true}; 用 pred 直接表达条件, 省 isetp + r2u 准备 mask 步骤)。

operand 语义: dst (URd, sreg) = leader lane id (0..31, warp 内所有 lane 看到同一个值); pout0 (pred per-lane) = '当前 lane 是不是 leader' (leader=true / 其他 =false, 用作 @P0 do_leader_work 单 leader 序列 gate); src0 = ureg member_mask (xp_x, 挂 src_bit_modi 支持 [~]URa 位取反表达' 除了这些 lane 之外') 或 per-lane pin0 predicate (xp_p, 挂 src_bit_modi 支持 [!]pin0 取反)。

leader 选举确定性: 永远挑最低 index 的 active lane 作 leader, 保证多次执行同 mask 选同 leader 可重现。

典型用例: atomic per-warp aggregation (elect xp_x UR_lid, P0, UR_mask; @P0 atomicAdd(addr, agg_val); 让 leader lane 单独做 atomic 减少冲突) / per-warp print / log (@P0 printf(…) 单 leader 输出避免 32 倍重复) / cooperative kernel 入口 (@P0 setup_warp_state(…) 单 lane init)。

match **category**: Cross-lane and collective **predicate_mode**: simt

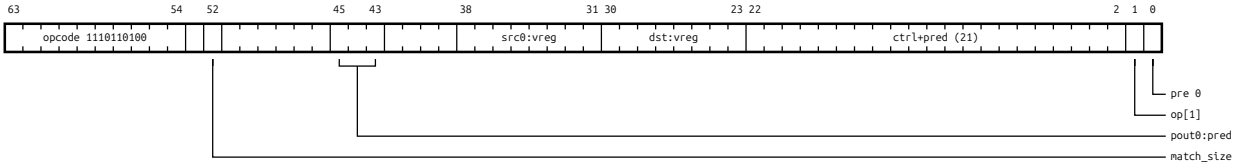
Leaf: match__all.vp_v

Format:

```
match{.match_size}  dst, pout0, src0
```

Operands: dst: vreg; pout0: pred; src0: vreg

Encoding Diagram: 64b, prefix 0, opcode 11110110100



Notes:

Warp-wide equality search: warp 内每个 active lane 自带一个值 src0, match 让每个 lane 同时回答 '我跟谁的 src0 相等?' — dst[lane] = bitmask (active lane 中 src0[l'] == src0[lane] 的 lane 集合, 一定含自己)。

指令形态区分 (op 选择信息蕴含在 operand 形态里, 不用单独 modifier): any.v_v (仅 dst, 'who matches me' 路径) / all.vp_v (dst + pout0, all 路径 — pout0[lane] = active lane 内 src0 是否全相同, 不全相同时 dst 写 0)。物理上 all / any 共享同一 opcode, 仅 1 bit 内部区分。

match_size: b32 (默认, src0 占 1 vreg) / b64 (src0 占 vreg pair r / r+1 偶对齐, 比较 64-bit 值 pointer / uint64 match 主路径)。

operand 语义: dst = vreg per-lane mask (32-bit); pout0 (all 路径) = pred per-lane (true 仅当全 active lane 一致); src0 = vreg 比较值 (per-lane 不同)。

典型用例: hash table cooperative (32 lane 各算一个 hash, match.any 找出哪些 lane 命中相同 bucket) / collision detection (match.all + pout0 判断全 warp 是否一致, 例如索引数组的去重判定) / histogram bin (match.any 找出哪些 lane 落在同一 bin)。

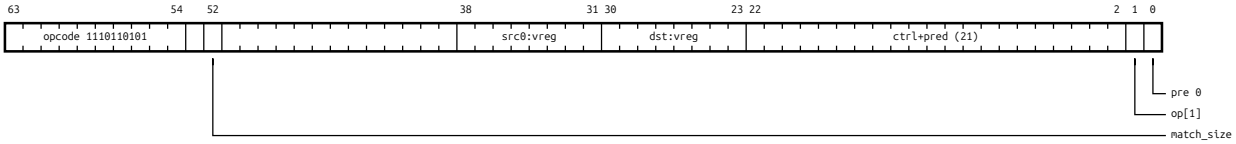
Leaf: match__any.v_v

Format:

```
match{.match_size}  dst, src0
```

Operands: dst: vreg; src0: vreg

Encoding Diagram: 64b, prefix 0, opcode 11110110101



Notes:

Warp-wide equality search: warp 内每个 active lane 自带一个值 src0, match 让每个 lane 同时回答 '我跟谁的 src0 相等?' — dst[lane] = bitmask (active lane 中 src0[l'] == src0[lane] 的 lane 集合, 一定含自己)。

指令形态区分 (op 选择信息蕴含在 operand 形态里, 不用单独 modifier): any.v_v (仅 dst, 'who matches me' 路径) / all.vp_v (dst + pout0, all 路径 — pout0[lane] = active lane 内 src0 是否全相同, 不全相同时 dst 写 0)。物理上 all / any 共享同一 opcode, 仅 1 bit 内部区分。

match_size: b32 (默认, src0 占 1 vreg) / b64 (src0 占 vreg pair r / r+1 偶对齐, 比较 64-bit 值 pointer / uint64 match 主路径)。

operand 语义: dst = vreg per-lane mask (32-bit); pout0 (all 路径) = pred per-lane (true 仅当全 active lane 一致); src0 = vreg 比较值 (per-lane 不同)。

典型用例: hash table cooperative (32 lane 各算一个 hash, match.any 找出哪些 lane 命中相同 bucket) / collision detection (match.all + pout0 判断全 warp 是否一致, 例如索引数组的去重判定) / histogram bin (match.any 找出哪些 lane 落在同一 bin)。

readlane **category**: Cross-lane and collective **predicate_mode**: simt

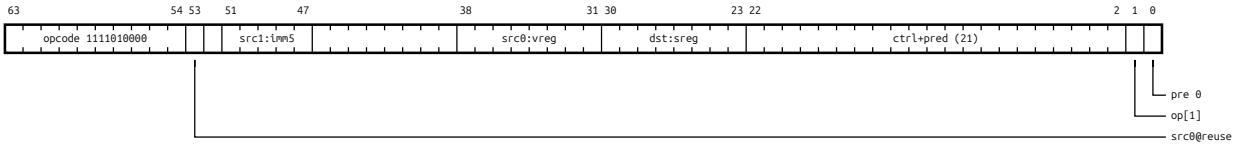
Leaf: readlane__x_vi

Format:

```
readlane  dst, src0{.reuse}, src1
```

Operands: dst: sreg; src0: vreg; src1: imm5u

Encoding Diagram: 64b, prefix 0, opcode 11111010000



Notes:

Read specific lane's vreg: dst (sreg) = src0[lane=src1]。从 warp 内指定的某个 lane 读它的 vreg 值, 结果广播给整 warp (写到 sreg)。

指令形态: x_vi (src1 = imm.5 静态 lane index, 编译期确定 — 主路径, 例如 readlane UR0, R5, 17 拿 lane17 的 R5) / x_vx (src1 = sreg 动态 lane index, runtime 决定)。

operand 语义: dst = sreg (warp-uniform 写一份); src0 = vreg per-lane 数据源 (挂 reuse 跟 cross-lane permute network 共享); src1 = lane index (0..31, imm.5 或 sreg)。

inactive source lane 语义: src1 指定的 lane 如果 inactive (EXEC[src1]=0), dst 值 undefined。程序员 / 编译器负责保证 src1 指向 active lane (典型场景 readlane 通常配合 elect 拿 leader lane id, leader 一定 active 所以安全; 但如果用 isetp + r2u 自己算 lane index 必须保证算出的 lane active)。

典型用例: 选 leader 拿值 — elect xp_x UR_lid, P_leader, UR_mask; readlane UR_val, R_data, UR_lid (让 leader lane 的 R_data 广播给整 warp); reduce 收尾 (但 redux 已经写 ureg 不需要 readlane)。

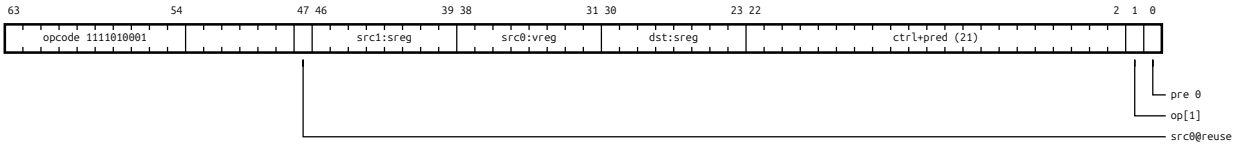
Leaf: readlane__x_vx

Format:

```
readlane  dst, src0{.reuse}, src1
```

Operands: dst: sreg; src0: vreg; src1: sreg

Encoding Diagram: 64b, prefix 0, opcode 11111010001



Notes:

Read specific lane's vreg: dst (sreg) = src0[lane=src1]。从 warp 内指定的某个 lane 读它的 vreg 值, 结果广播给整 warp (写到 sreg)。

指令形态: x_vi (src1 = imm.5 静态 lane index, 编译期确定 — 主路径, 例如 readlane UR0, R5, 17 拿 lane17 的 R5) / x_vx (src1 = sreg 动态 lane index, runtime 决定)。

operand 语义: dst = sreg (warp-uniform 写一份); src0 = vreg per-lane 数据源 (挂 reuse 跟 cross-lane permute network 共享); src1 = lane index (0..31, imm.5 或 sreg)。

inactive source lane 语义: src1 指定的 lane 如果 inactive (EXEC[src1]=0), dst 值 undefined。程序员 / 编译器负责保证 src1 指向 active lane (典型场景 readlane 通常配合 elect 拿 leader lane id, leader 一定 active 所以安全; 但如果用 isetp + r2u 自己算 lane index 必须保证算出的 lane active)。

典型用例: 选 leader 拿值 — elect xp_x UR_lid, P_leader, UR_mask; readlane UR_val, R_data, UR_lid (让 leader lane 的 R_data 广播给整 warp); reduce 收尾 (但 redux 已经写 ureg 不需要 readlane)。

redux　**category**: Cross-lane and collective　**predicate_mode**: simt

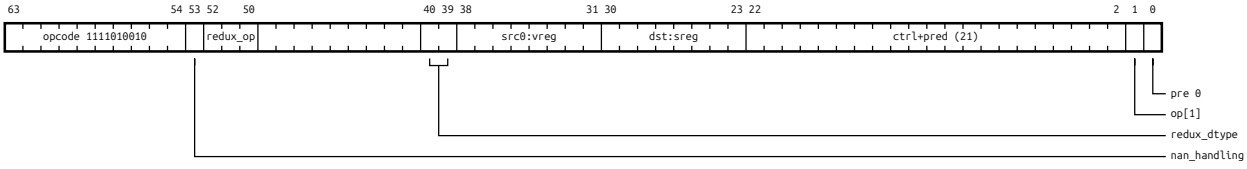
Leaf: redux__x_v

Format:

```
redux.redux_op.redux_dtype{.nan_handling}  dst, src0
```

Operands: dst: sreg; src0: vreg

Encoding Diagram: 64b, prefix 0, opcode 11111010010



Notes:

Warp-wide reduction: warp 内每个 active lane 各贡献一个 src0 值, 整个 warp 一起归约成一个值给所有 lane (写到 sreg, warp-uniform)。

redux_op 8 valid: add (整数加法 reduce, 32-bit 截断, 仅 u32 / s32) / min (整数 / f32 最小值) / max (整数 / f32 最大值) / and (按位 and, 仅 u32 / s32) / or (按位 or, 仅 u32 / s32) / xor (按位 xor, 仅 u32 / s32) / maxabs (max(|src0|), 仅 f32) / minabs (min(|src0|), 仅 f32)。

redux_dtype 3 valid: u32 / s32 / f32 — sign 影响 min / max 路径 (s32 走 signed compare)。

redux_op × redux_dtype valid_combo: u32 / s32 (整数) 路径 op ∈ {add, min, max, and, or, xor}; f32 路径 op ∈ {min, max, maxabs, minabs} (浮点 add 不结合律 cross-lane 顺序敏感, 硬件不支持 — 用户走 redux.min / max 或 shfl.bfly 序列代偿)。

nan_handling (仅 f32 + min / max / maxabs / minabs valid): nonan (默认, non-NaN 优先, reduce 时跳过 NaN 输入, 仅全 NaN 时 dst = canonical NaN) / nan (NaN 传播, 任一 lane NaN 即 dst = canonical NaN)。其他 valid_combo nan_handling 必须 nonan。

operand 语义: dst = sreg (warp-uniform 一份); src0 = vreg per-lane 数据。

典型用例: redux.add.u32 UR0, R5 / redux.max.s32 UR0, R5 / redux.or.u32 UR0, R5 / redux.maxabs.f32.nan UR0, R5。

shfl **category**: Cross-lane and collective **predicate_mode**: simt

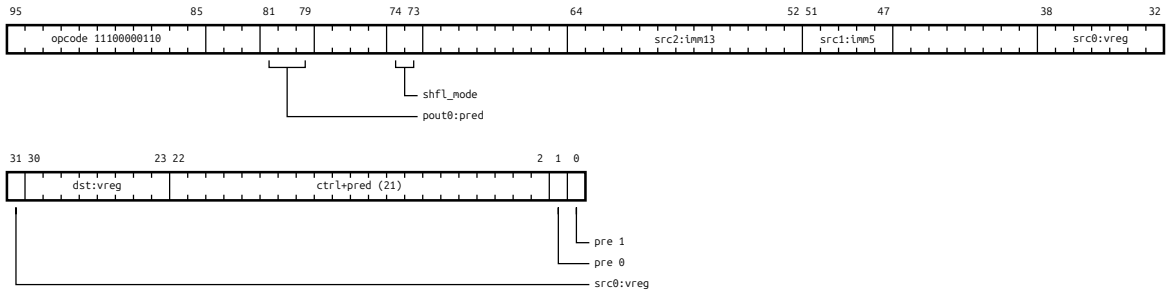
Leaf: shfl__vp_vii

Format:

```
shfl{.shfl_mode}  pout0, dst, src0, src1, src2
```

Operands: pout0: pred; dst: vreg; src0: vreg; src1: imm5u; src2: imm13u

Encoding Diagram: 96b, prefix 10, opcode 11100000110



Notes:

Warp shuffle (lane-to-lane vreg permute): warp 内 lane 之间交换 vreg 数据 — 每个 lane 计算一个 '源 lane index j' (基于自己的 lane_id + shfl_mode + src1 / src2), 然后从 lane j 拿 src0 写到自己的 dst。如果 j 越界 (跨过 segment mask 或超 32), pout=false 且 dst 退化为拷贝自己 (src0[lane_id])。

shfl_mode 4 valid: idx (默认, j = (minLane | (src1[4:0] & ~segmask)), '我去 lane src1 拿值', 绝对 index, readlane 序列代偿基础) / up (j = lane - src1, '我从下面第 src1 个 lane 拿值', prefix sum 左 → 右传播) / down (j = lane + src1, '我从上面第 src1 个 lane 拿值', 右 → 左传播或 reduce) / bfly (j = lane xor src1, '我跟 src1 异或得到的 lane 互换值', butterfly reduction, src1=1 / 2 / 4 / 8 / 16 做对数级 fold)。

operand 语义: src0 = 要被 shuffle 的 vreg (per-lane 不同, 真正跨 lane 搬运的数据); src1 = imm.5 或 vreg, idx 时是 srcLane 绝对值, up / down 时是 delta, bfly 时是 xor mask; src2 = imm.13 或 vreg, 打包 clamp[4:0] + segmask[12:8] (5-bit clamp 设上界 + 5-bit segmask 把 warp 切成子段限制 shuffle 跨段); dst = shuffle 结果 vreg (per-lane); pout = 'j 在范围内' boolean (越界 false + dst 退化)。

inactive source lane 语义: 如果计算出的源 lane j 是 inactive (EXEC[j]=0), dst 值 undefined。程序员 / 编译器负责保证 j active (divergent flow 后必须用 sync membermask 保证); pout 反映 j 是否 in-range (clamp + segmask 边界), 不反映 j 是否 active。

指令形态: vp_vvi / vp_viv / vp_vii / vp_vvv (4 指令形态对应 src1 / src2 是 vreg 还是 imm 的笛卡尔积)。

典型用例: shfl.idx vp_vii pout0, R5, R3, 17, 0x1f (拿 lane17 的值, readlane 等价) / shfl.down vp_vii pout0, R5, R3, 1, 0x1f (拿下一个 lane, 配合 reduce) / shfl.bfly vp_vii pout0, R5, R3, 16, 0x1f (butterfly reduction 第一步)。

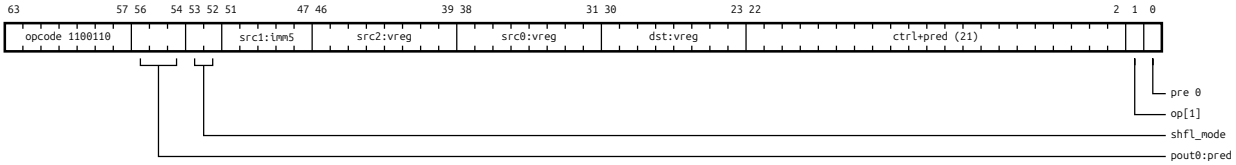
Leaf: shfl__vp_viv

Format:

```
shfl{.shfl_mode}  pout0, dst, src0, src1, src2
```

Operands: pout0: pred; dst: vreg; src0: vreg; src1: imm5u; src2: vreg

Encoding Diagram: 64b, prefix 0, opcode 11100110



Notes:

Warp shuffle (lane-to-lane vreg permute): warp 内 lane 之间交换 vreg 数据 — 每个 lane 计算一个 '源 lane index j' (基于自己的 lane_id + shfl_mode + src1 / src2), 然后从 lane j 拿 src0 写到自己的 dst。如果 j 越界 (跨过 segment mask 或超 32), pout=false 且 dst 退化为拷贝自己 (src0[lane_id])。

shfl_mode 4 valid: idx (默认, j = (minLane | (src1[4:0] & ~segmask)), '我去 lane src1 拿值', 绝对 index, readlane 序列代偿基础) / up (j = lane - src1, '我从下面第 src1 个 lane 拿值', prefix sum 左 → 右传播) / down (j = lane + src1, '我从上面第 src1 个 lane 拿值', 右 → 左传播或 reduce) / bfly (j = lane xor src1, '我跟 src1 异或得到的 lane 互换值', butterfly reduction, src1=1 / 2 / 4 / 8 / 16 做对数级 fold)。

operand 语义: src0 = 要被 shuffle 的 vreg (per-lane 不同, 真正跨 lane 搬运的数据); src1 = imm.5 或 vreg, idx 时是 srcLane 绝对值, up / down 时是 delta, bfly 时是 xor mask; src2 = imm.13 或 vreg, 打包 clamp[4:0] + segmask[12:8] (5-bit clamp 设上界 + 5-bit segmask 把 warp 切成子段限制 shuffle 跨段); dst = shuffle 结果 vreg (per-lane); pout = 'j 在范围内' boolean (越界 false + dst 退化)。

inactive source lane 语义: 如果计算出的源 lane j 是 inactive (EXEC[j]=0), dst 值 undefined。程序员 / 编译器负责保证 j active (divergent flow 后必须用 sync membermask 保证); pout 反映 j 是否 in-range (clamp + segmask 边界), 不反映 j 是否 active。

指令形态: vp_vvi / vp_viv / vp_vii / vp_vvv (4 指令形态对应 src1 / src2 是 vreg 还是 imm 的笛卡尔积)。

典型用例: shfl.idx vp_vii pout0, R5, R3, 17, 0x1f (拿 lane17 的值, readlane 等价) / shfl.down vp_vii pout0, R5, R3, 1, 0x1f (拿下一个 lane, 配合 reduce) / shfl.bfly vp_vii pout0, R5, R3, 16, 0x1f (butterfly reduction 第一步)。

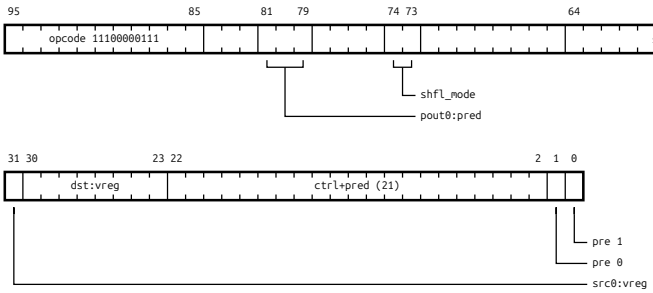
Leaf: shfl__vp_vvi

Format:

```
shfl{.shfl_mode}  pout0, dst, src0, src1, src2
```

Operands: pout0: pred; dst: vreg; src0: vreg; src1: vreg; src2: imm13u

Encoding Diagram: 96b, prefix 10, opcode 11100000111



Notes:

Warp shuffle (lane-to-lane vreg permute): warp 内 lane 之间交换 vreg 数据 — 每个 lane 计算一个 '源 lane index j' (基于自己的 lane_id + shfl_mode + src1 / src2), 然后从 lane j 拿 src0 写到自己的 dst。如果 j 越界 (跨过 segment mask 或超 32), pout=false 且 dst 退化为拷贝自己 (src0[lane_id])。

shfl_mode 4 valid: idx (默认, j = (minLane | (src1[4:0] & ~segmask)), '我去 lane src1 拿值', 绝对 index, readlane 序列代偿基础) / up (j = lane - src1, '我从下面第 src1 个 lane 拿值', prefix sum 左 → 右传播) / down (j

= lane + src1, ‘我从上面第 src1 个 lane 拿值’, 右 → 左传播或 reduce) / bfly (j = lane xor src1, ‘我跟 src1 异或得到的 lane 互换值’, butterfly reduction, src1=1 / 2 / 4 / 8 / 16 做对数级 fold)。

operand 语义: src0 = 要被 shuffle 的 vreg (per-lane 不同, 真正跨 lane 搬运的数据); src1 = imm.5 或 vreg, idx 时是 srcLane 绝对值, up / down 时是 delta, bfly 时是 xor mask; src2 = imm.13 或 vreg, 打包 clamp[4:0] + segmask[12:8] (5-bit clamp 设上界 + 5-bit segmask 把 warp 切成子段限制 shuffle 跨段); dst = shuffle 结果 vreg (per-lane); pout = ‘j 在范围内’ boolean (越界 false + dst 退化)。

inactive source lane 语义: 如果计算出的源 lane j 是 inactive (EXEC[j]=0), dst 值 undefined。程序员 / 编译器负责保证 j active (divergent flow 后必须用 sync membermask 保证); pout 反映 j 是否 in-range (clamp + segmask 边界), 不反映 j 是否 active。

指令形态: vp_vvi / vp_viv / vp_vii / vp_vvv (4 指令形态对应 src1 / src2 是 vreg 还是 imm 的笛卡尔积)。

典型用例: shfl.idx vp_vii pout0, R5, R3, 17, 0x1f (拿 lane17 的值, readlane 等价) / shfl.down vp_vii pout0, R5, R3, 1, 0x1f (拿下一个 lane, 配合 reduce) / shfl.bfly vp_vii pout0, R5, R3, 16, 0x1f (butterfly reduction 第一步)。

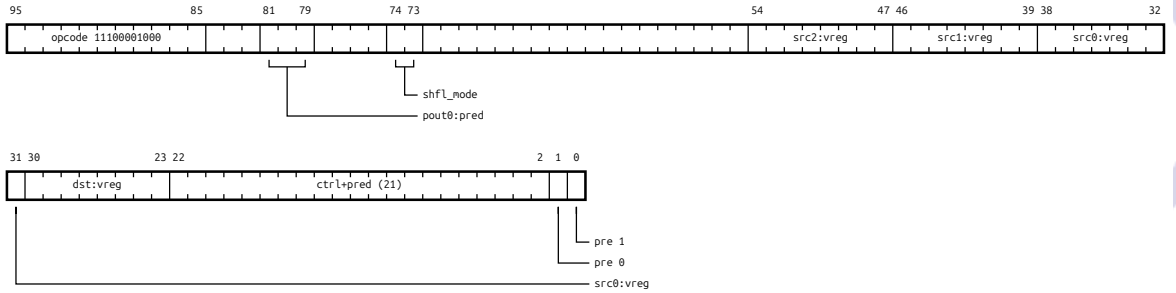
Leaf: shfl__vp_vvv

Format:

```
shfl{.shfl_mode}  pout0, dst, src0, src1, src2
```

Operands: pout0: pred; dst: vreg; src0: vreg; src1: vreg; src2: vreg

Encoding Diagram: 96b, prefix 10, opcode 11100001000



Notes:

Warp shuffle (lane-to-lane vreg permute): warp 内 lane 之间交换 vreg 数据 — 每个 lane 计算一个 ‘源 lane index j’ (基于自己的 lane_id + shfl_mode + src1 / src2), 然后从 lane j 拿 src0 写到自己的 dst。如果 j 越界 (跨过 segment mask 或超 32), pout=false 且 dst 退化为拷贝自己 (src0[lane_id])。

shfl_mode 4 valid: idx (默认, j = (minLane | (src1[4:0] & ~segmask)), ‘我去 lane src1 拿值’, 绝对 index, readlane 序列代偿基础) / up (j = lane - src1, ‘我从下面第 src1 个 lane 拿值’, prefix sum 左 → 右传播) / down (j = lane + src1, ‘我从上面第 src1 个 lane 拿值’, 右 → 左传播或 reduce) / bfly (j = lane xor src1, ‘我跟 src1 异或得到的 lane 互换值’, butterfly reduction, src1=1 / 2 / 4 / 8 / 16 做对数级 fold)。

operand 语义: src0 = 要被 shuffle 的 vreg (per-lane 不同, 真正跨 lane 搬运的数据); src1 = imm.5 或 vreg, idx 时是 srcLane 绝对值, up / down 时是 delta, bfly 时是 xor mask; src2 = imm.13 或 vreg, 打包 clamp[4:0] + segmask[12:8] (5-bit clamp 设上界 + 5-bit segmask 把 warp 切成子段限制 shuffle 跨段); dst = shuffle 结果 vreg (per-lane); pout = ‘j 在范围内’ boolean (越界 false + dst 退化)。

inactive source lane 语义: 如果计算出的源 lane j 是 inactive (EXEC[j]=0), dst 值 undefined。程序员 / 编译器负责保证 j active (divergent flow 后必须用 sync membermask 保证); pout 反映 j 是否 in-range (clamp + segmask 边界), 不反映 j 是否 active。

指令形态: vp_vvi / vp_viv / vp_vii / vp_vvv (4 指令形态对应 src1 / src2 是 vreg 还是 imm 的笛卡尔积)。

典型用例: shfl.idx vp_vii pout0, R5, R3, 17, 0x1f (拿 lane17 的值, readlane 等价) / shfl.down vp_vii pout0, R5, R3, 1, 0x1f (拿下一个 lane, 配合 reduce) / shfl.bfly vp_vii pout0, R5, R3, 16, 0x1f (butterfly reduction 第一步)。

uvote **category**: Cross-lane and collective **predicate_mode**: simt

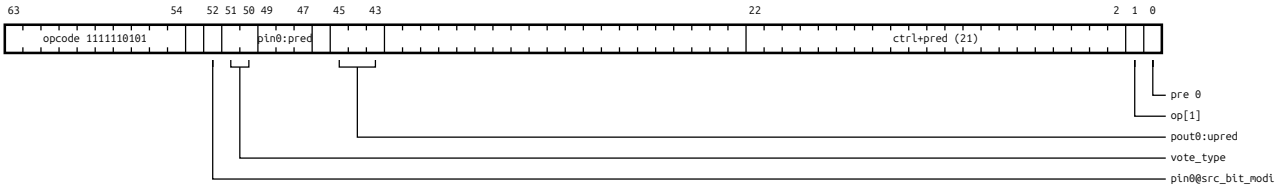
Leaf: uvote__p_p

Format:

```
uvote{.vote_type}  pout0, pin0{.src_bit_modi}
```

Operands: pout0: upred; pin0: pred

Encoding Diagram: 64b, prefix 0, opcode 11111110101



Notes:

Uniform-output warp vote: vote 的 uniform 输出版本 — warp 32 lane 集体归约 per-thread pin0, 但结果输出到 uniform 寄存器 (sreg + upred) 而不是 simt 寄存器 (vreg + pred)。pout0 = all / any / eq 归约结果, dst (xp_p 指令形态) = ballot mask 32-bit ureg。

vote_type 3 valid: all / any / eq, 含义跟 vote 同源。

指令形态: p_p (仅 pout0 upred 输出) / xp_p (dst sreg ballot mask + pout0 upred 同时输出)。

operand 语义 (uniform 输出 + per-thread 输入特例): dst (URd, sreg) 输出 ballot mask warp 写一份; pout0 (upred) 输出归约结果; pin0 (per-thread predicate) 输入 — 每个 lane 自己的 predicate 参与归约。Pg gate 是 per-thread @P 不是 @up, 因为 vote 算法内禀需要 per-thread 粒度的参与信息 (每个 lane 自己决定参不参与, @P gate 关掉的 lane 不计入 ballot)。

uvote 是 uniform-pipe 特例 — inputs (Pg gate 跟 pin0) 是 per-thread, outputs (URd 跟 pout0) 是 uniform; spec 把 predicate_mode 标 simt 反映 gate 类型 (跟 vote 同 gate 模型)。

典型用例: warp-级 ballot / all / any / eq 归约且结果直接进 uniform 寄存器使用 (例如 control flow 判断 / scalar 路径计算), 省去 r + P 资源 vs vote + r2u 序列。

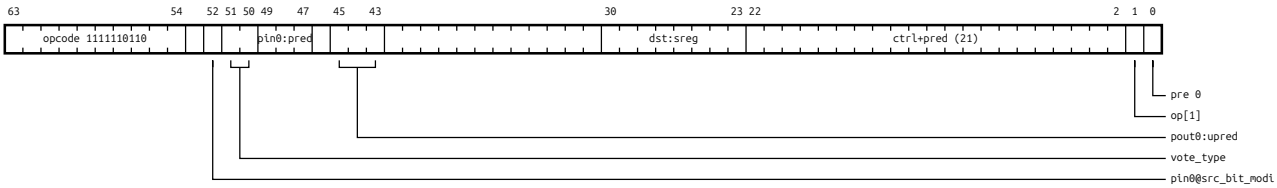
Leaf: uvote__xp_p

Format:

```
uvote{.vote_type}  dst, pout0, pin0{.src_bit_modi}
```

Operands: dst: sreg; pout0: upred; pin0: pred

Encoding Diagram: 64b, prefix 0, opcode 11111110110



Notes:

Uniform-output warp vote: vote 的 uniform 输出版本 — warp 32 lane 集体归约 per-thread pin0, 但结果输出到 uniform 寄存器 (sreg + upred) 而不是 simt 寄存器 (vreg + pred)。pout0 = all / any / eq 归约结果, dst (xp_p 指令形态) = ballot mask 32-bit ureg。

vote_type 3 valid: all / any / eq, 含义跟 vote 同源。

指令形态: p_p (仅 pout0 upred 输出) / xp_p (dst sreg ballot mask + pout0 upred 同时输出)。

operand 语义 (uniform 输出 + per-thread 输入特例): dst (URd, sreg) 输出 ballot mask warp 写一份; pout0 (upred) 输出归约结果; pin0 (per-thread predicate) 输入 — 每个 lane 自己的 predicate 参与归约。Pg gate 是 per-thread @P 不是 @up, 因为 vote 算法内禀需要 per-thread 粒度的参与信息 (每个 lane 自己决定参不参与, @P gate 关掉的 lane 不计入 ballot)。

uvote 是 uniform-pipe 特例 — inputs (Pg gate 跟 pin0) 是 per-thread, outputs (URd 跟 pout0) 是 uniform; spec 把 predicate_mode 标 simt 反映 gate 类型 (跟 vote 同 gate 模型)。

典型用例: warp-级 ballot / all / any / eq 归约且结果直接进 uniform 寄存器使用 (例如 control flow 判断 / scalar 路径计算), 省去 r + P 资源 vs vote + r2u 序列。

vote **category**: Cross-lane and collective **predicate_mode**: simt

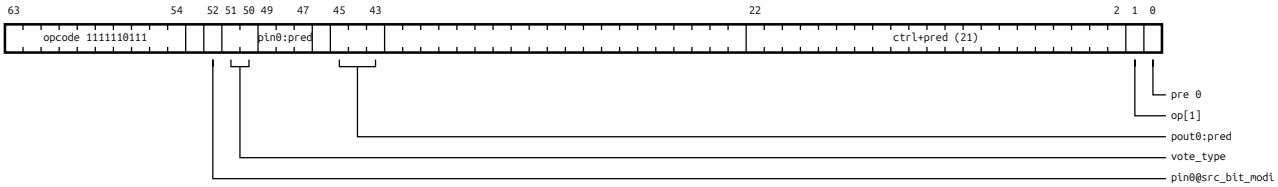
Leaf: vote__p_p

Format:

```
vote{.vote_type}  pout0, pin0{.src_bit_modi}
```

Operands: pout0: pred; pin0: pred

Encoding Diagram: 64b, prefix 0, opcode 11111110111



Notes:

Warp-wide predicate vote: warp 内 32 个 active lane 集体投票 — 每个 lane 自带一个 predicate (pin0), 整个 warp 一起归约成一个 boolean 答案 (pout0) 给所有 lane (warp-uniform); 顺手输出 ballot mask (32-bit, 每个 bit 对应一 lane 的 pin0 值, inactive lane 贡献 0)。

vote_type 3 valid: all (pout0 = 所有 active lane 的 pin0 都是 true) / any (pout0 = 至少一个 active lane 的 pin0 是 true) / eq (pout0 = 所有 active lane 的 pin0 值相同, 全 true 或全 false)。

指令形态: p_p (仅 pout0 输出, 不要 ballot) / vp_p (pout0 + dst 同时输出, dst 是 ballot mask 32-bit vreg)。

operand 语义: pin0 = pin0 per-lane predicate; pout0 = pout0 warp-uniform pred output (所有 lane 同值); dst (vp_p only) = vreg ballot mask。

典型用例: vote.all p_p pout0, P0 (全 lane 是否都满足条件) / vote.any p_p pout0, P0 (至少一个 lane 满足) / vote.eq p_p pout0, P0 (lane 间值是否一致) / vote.any vp_p R5, pout0, P0 (同时拿 reduce + mask)。

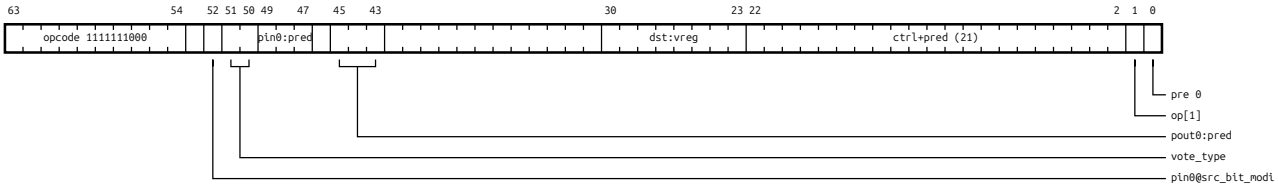
Leaf: vote__vp_p

Format:

```
vote{.vote_type}  dst, pout0, pin0{.src_bit_modi}
```

Operands: dst: vreg; pout0: pred; pin0: pred

Encoding Diagram: 64b, prefix 0, opcode 11111111000



Notes:

Warp-wide predicate vote: warp 内 32 个 active lane 集体投票 — 每个 lane 自带一个 predicate (pin0), 整个 warp 一起归约成一个 boolean 答案 (pout0) 给所有 lane (warp-uniform); 顺手输出 ballot mask (32-bit, 每个 bit 对应一 lane 的 pin0 值, inactive lane 贡献 0)。

vote_type 3 valid: all (pout0 = 所有 active lane 的 pin0 都是 true) / any (pout0 = 至少一个 active lane 的 pin0 是 true) / eq (pout0 = 所有 active lane 的 pin0 值相同, 全 true 或全 false)。

指令形态: p_p (仅 pout0 输出, 不要 ballot) / vp_p (pout0 + dst 同时输出, dst 是 ballot mask 32-bit vreg)。

operand 语义: pin0 = pin0 per-lane predicate; pout0 = pout0 warp-uniform pred output (所有 lane 同值); dst (vp_p only) = vreg ballot mask。

典型用例: vote.all p_p pout0, P0 (全 lane 是否都满足条件) / vote.any p_p pout0, P0 (至少一个 lane 满足) / vote.eq p_p pout0, P0 (lane 间值是否一致) / vote.any vp_p R5, pout0, P0 (同时拿 reduce + mask)。

14.7.15 Synchronization and memory ordering

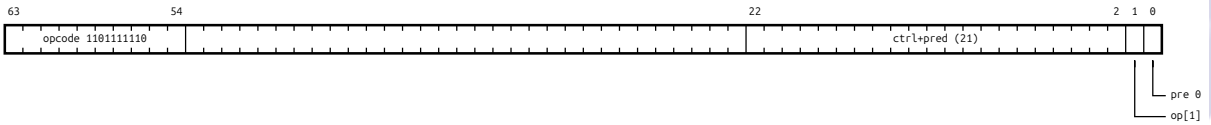
async_commit **category**: Synchronization and memory ordering **predicate_mode**: uniform

Leaf: async_commit__u

Format:

```
async_commit
```

Encoding Diagram: 64b, prefix 0, opcode 11101111110



Notes:

Close current async-group: 把本 warp 之前所有尚未提交的 ldgsts / ldsstg 归入当前 group 并关闭它, 之后再发的 ldgsts / ldsstg 归入下一个 group。典型用法是双缓冲 / 多缓冲软件流水线 — 每准备好一块数据就提交一次, 形成一个滑动窗口。

async_commit 自己就是一条变延迟指令, 用每条指令都带的公共控制字段记账, 不需要任何专用的 group 计数器: issue 时给 wr_sb / rd_sb 选中的 dependence counter +1, 在对应的完成事件发生时由硬件 -1。于是那个 counter 的当前值就等于本 warp 还未完成的 async-group 数量。

两个完成事件分别对应两件不同的事。wr_sb 对应 '本组内所有拷贝的数据都已写入目的内存' (ldgsts 写进 shared memory, ldsstg 写进 global memory) — 消费者读这批数据之前必须等这个。rd_sb 对应 '本组内所有拷贝的源数据都已读完' (ldgsts 读完 global memory 侧, ldsstg 读完 shared memory 侧) — 要复写源缓冲之前必须等这个, 它比数据写到目的地早得多, 软件流水线复用共享内存缓冲只需要等它。两个事件各自选一个 counter, 互不干扰; 只关心其中一件事时另一个填 SB_NONE 即可。

完成 ≠ 可见: 这两个事件说的都是 '搬运硬件的活干完了', 不是 '别的线程 / 别的观察者看得见了'。跨线程可见性由内存序机制定义 — 访存指令自己的 mem_sem 加上 fence, 见 fence 条目。ldsstg 把数据搬到 global memory 之后, 要让别的 cta 或 host 可靠地看到, 必须在 wr_sb 等到之后再按需要的 consistency_scope 发 fence; 只等 wr_sb 不足以保证可见性。

没有专门的 async-group 等待指令: 等待走 depbar, 它的 'SB[x] ≤ threshold' 语义正是滑动窗口需要的 — threshold=0 等本组之前的全部完成, threshold=1 允许最近 1 个 group 还在飞 (双缓冲主路径), threshold=2 三缓冲, 以此类推。

无 operand。

bar_arrive　**category**: Synchronization and memory ordering　**predicate_mode**: simt

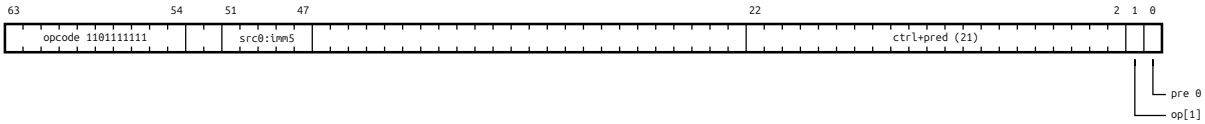
Leaf: bar_arrive___i

Format:

```
bar_arrive  src0
```

Operands: src0: imm5u

Encoding Diagram: 64b, prefix 0, opcode 11101111111



Notes:

Split-phase barrier arrive (non-blocking): 通知 barrier ‘i arrived’ 但不等其他 thread, 继续执行后续。配合 bar_sync 实现 split-phase 模式 (此 thread 不参与等待, 其他 thread 用 bar_sync 等齐)。

到达计数以线程为单位, 由谓词门控与 EXEC 掩码逐 lane 决定: 一个 warp 执行一次本指令, 向 barrier 贡献的到达数 = 门控后 active lane 的数量 (popcount(EXEC 与谓词门的按位与))。被 kill 的 lane 已从 EXEC 清除, 自然不计入; 门控后没有任何 active lane 时执行本指令是合法空操作 (贡献 0、不阻塞)。软件契约: 每个计入 thread_count 的线程必须最终以预期的掩码执行到 barrier, 否则死锁属于程序错误 — barrier 应在参与线程收敛的位置执行。

代际规则: barrier 释放 (到达数达到参与总数) 后计数自动复位、进入下一代, 之后的到达计入新一代。thread_count = 0 或超过 cta 线程总数是非法参数, 硬件 trap; 同一代内不同 warp 提供的 thread_count 必须一致, 不一致时行为等同程序错误。

指令形态跟 bar_sync 同源 (4 种 id × count 组合)。

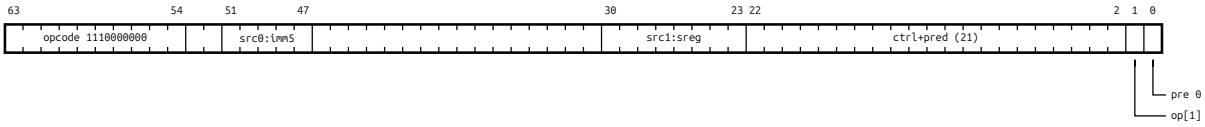
Leaf: bar_arrive___ix

Format:

```
bar_arrive  src0, src1
```

Operands: src0: imm5u; src1: sreg

Encoding Diagram: 64b, prefix 0, opcode 11110000000



Notes:

Split-phase barrier arrive (non-blocking): 通知 barrier ‘i arrived’ 但不等其他 thread, 继续执行后续。配合 bar_sync 实现 split-phase 模式 (此 thread 不参与等待, 其他 thread 用 bar_sync 等齐)。

到达计数以线程为单位, 由谓词门控与 EXEC 掩码逐 lane 决定: 一个 warp 执行一次本指令, 向 barrier 贡献的到达数 = 门控后 active lane 的数量 (popcount(EXEC 与谓词门的按位与))。被 kill 的 lane 已从 EXEC 清除, 自然不计入; 门控后没有任何 active lane 时执行本指令是合法空操作 (贡献 0、不阻塞)。软件契约: 每个计入 thread_count 的线程必须最终以预期的掩码执行到 barrier, 否则死锁属于程序错误 — barrier 应在参与线程收敛的位置执行。

代际规则: barrier 释放 (到达数达到参与总数) 后计数自动复位、进入下一代, 之后的到达计入新一代。thread_count = 0 或超过 cta 线程总数是非法参数, 硬件 trap; 同一代内不同 warp 提供的 thread_count 必须一致, 不一致时行为等同程序错误。

指令形态跟 bar_sync 同源 (4 种 id × count 组合)。

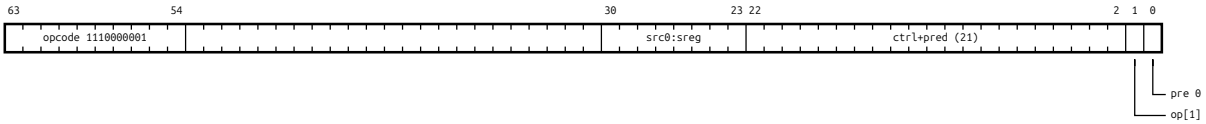
Leaf: bar_arrive___x

Format:

```
bar_arrive  src0
```

Operands: src0: sreg

Encoding Diagram: 64b, prefix 0, opcode 11110000001



Notes:

Split-phase barrier arrive (non-blocking): 通知 barrier ‘i arrived’ 但不等其他 thread, 继续执行后续。配合 bar_sync 实现 split-phase 模式 (此 thread 不参与等待, 其他 thread 用 bar_sync 等齐)。

到达计数以线程为单位, 由谓词门控与 EXEC 掩码逐 lane 决定: 一个 warp 执行一次本指令, 向 barrier 贡献的到达数 = 门控后 active lane 的数量 (popcount(EXEC 与谓词门的按位与))。被 kill 的 lane 已从 EXEC 清除, 自然不计入; 门控后没有任何 active lane 时执行本指令是合法空操作 (贡献 0、不阻塞)。软件契约: 每个计入 thread_count 的线程必须最终以预期的掩码执行到 barrier, 否则死锁属于程序错误 — barrier 应在参与线程收敛的位置执行。

代际规则: barrier 释放 (到达数达到参与总数) 后计数自动复位、进入下一代, 之后的到达计入新一代。thread_count = 0 或超过 cta 线程总数是非法参数, 硬件 trap; 同一代内不同 warp 提供的 thread_count 必须一致, 不一致时行为等同程序错误。

指令形态跟 bar_sync 同源 (4 种 id × count 组合)。

Leaf: bar_arrive___xx

Format:

```
bar_arrive  src0, src1
```

Operands: src0: sreg; src1: sreg

Encoding Diagram: 64b, prefix 0, opcode 11110000010

Notes:

Split-phase barrier arrive (non-blocking): 通知 barrier ‘i arrived’ 但不等其他 thread, 继续执行后续。配合 bar_sync 实现 split-phase 模式 (此 thread 不参与等待, 其他 thread 用 bar_sync 等齐)。

到达计数以线程为单位, 由谓词门控与 EXEC 掩码逐 lane 决定: 一个 warp 执行一次本指令, 向 barrier 贡献的到达数 = 门控后 active lane 的数量 (popcount(EXEC 与谓词门的按位与))。被 kill 的 lane 已从 EXEC 清除, 自然不计入; 门控后没有任何 active lane 时执行本指令是合法空操作 (贡献 0、不阻塞)。软件契约: 每个计入 thread_count 的线程必须最终以预期的掩码执行到 barrier, 否则死锁属于程序错误 — barrier 应在参与线程收敛的位置执行。

代际规则: barrier 释放 (到达数达到参与总数) 后计数自动复位、进入下一代, 之后的到达计入新一代。thread_count = 0 或超过 cta 线程总数是非法参数, 硬件 trap; 同一代内不同 warp 提供的 thread_count 必须一致, 不一致时行为等同程序错误。

指令形态跟 bar_sync 同源 (4 种 id × count 组合)。

bar_red **category**: Synchronization and memory ordering **predicate_mode**: simt

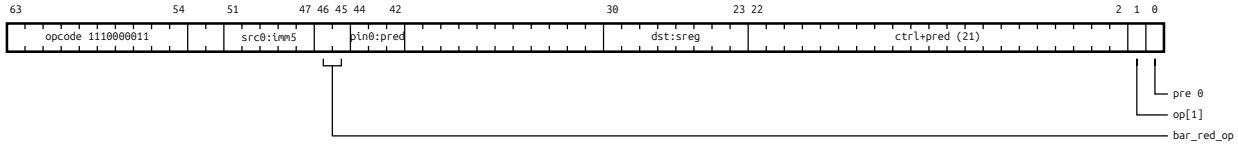
Leaf: bar_red___xip

Format:

```
bar_red{.bar_red_op}  dst, src0, pin0
```

Operands: dst: sreg; src0: imm5u; pin0: pred

Encoding Diagram: 64b, prefix 0, opcode 11110000011



Notes:

cta barrier sync + cross-thread predicate reduction: bar_sync + 跨 thread predicate 归约一条搞定 — 所有 thread 到达 barrier 时, 用 pin0 (per-thread 1-bit predicate) 做归约, 结果 broadcast 到所有 thread 的 dst sreg。到达计数与阻塞规则同 bar_sync (逐 lane 计数 = 门控后 active lane 数, 代际自动复位); 归约同样只覆盖各 warp 门控后 active 的 lane, 不 active 的 lane 不贡献 pin0。

bar_red_op 3 valid: popc (result = count of threads where pin0=true, 等价 cta-level vote + popc) / and (所有 thread pin0 都 true 时 result=true, 等价 cta-level vote.all) / or (任一 thread pin0 true 时 result=true, 等价 cta-level vote.any)。

指令形态: _xip (imm barrier_id) / _xxp (sreg barrier_id), 两种都带 pin0 predicate。

operand 模型: dst (sreg, broadcast 归约结果) + barrier_id (imm.5 或 sreg) + pin0 (per-thread predicate input)。隐式调用 barrier sync (整 cta 同步, 跟 bar_sync 同源)。

典型用例: convergence check (loop 终止: bar_red.or 检测全 cta 是否还要继续) / dynamic load balance (bar_red.popc 数 active thread) / cta-level all / any vote。

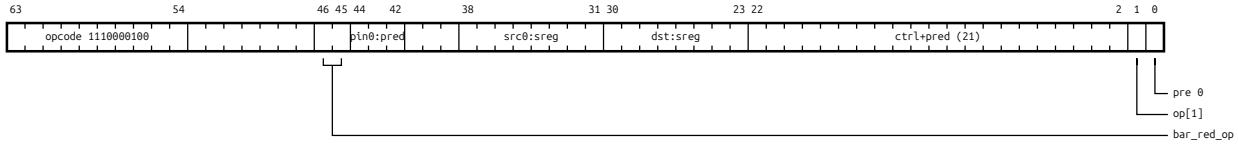
Leaf: bar_red___xxp

Format:

```
bar_red{.bar_red_op}  dst, src0, pin0
```

Operands: dst: sreg; src0: sreg; pin0: pred

Encoding Diagram: 64b, prefix 0, opcode 11110000100



Notes:

cta barrier sync + cross-thread predicate reduction: bar_sync + 跨 thread predicate 归约一条搞定 — 所有 thread 到达 barrier 时, 用 pin0 (per-thread 1-bit predicate) 做归约, 结果 broadcast 到所有 thread 的 dst sreg。到达计数与阻塞规则同 bar_sync (逐 lane 计数 = 门控后 active lane 数, 代际自动复位); 归约同样只覆盖各 warp 门控后 active 的 lane, 不 active 的 lane 不贡献 pin0。

bar_red_op 3 valid: popc (result = count of threads where pin0=true, 等价 cta-level vote + popc) / and (所有

thread pin0 都 true 时 result=true, 等价 cta-level vote.all) / or (任一 thread pin0 true 时 result=true, 等价 cta-level vote.any)。

指令形态: _xip (imm barrier_id) / _xxp (sreg barrier_id), 两种都带 pin0 predicate。

operand 模型: dst (sreg, broadcast 归约结果) + barrier_id (imm.5 或 sreg) + pin0 (per-thread predicate input)。隐式调用 barrier sync (整 cta 同步, 跟 bar_sync 同源)。

典型用例: convergence check (loop 终止: bar_red.or 检测全 cta 是否还要继续) / dynamic load balance (bar_red.popc 数 active thread) / cta-level all / any vote。

bar_sync **category**: Synchronization and memory ordering **predicate_mode**: simt

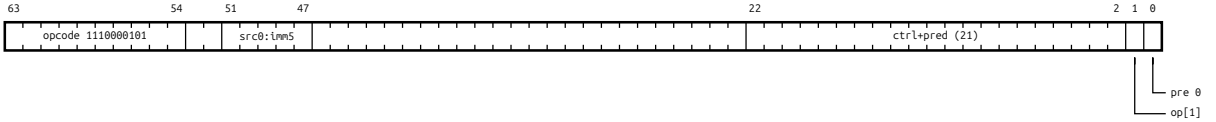
Leaf: bar_sync___i

Format:

```
bar_sync  src0
```

Operands: src0: imm5u

Encoding Diagram: 64b, prefix 0, opcode 11110000101



Notes:

cta named barrier sync: 所有参与 thread 必须都到达这条指令后才能继续; 只要 warp 内还有 ≥1 个门控后 active 的 lane, warp 就停在这条指令直到 barrier 释放。最多 32 个 named barrier per cta (barrier_id 用 imm.5 即 0..31)。

到达计数以线程为单位, 由谓词门控与 EXEC 掩码逐 lane 决定: 一个 warp 执行一次本指令, 向 barrier 贡献的到达数 = 门控后 active lane 的数量 (popcount(EXEC 与谓词门的按位与))。被 kill 的 lane 已从 EXEC 清除, 自然不计入; 门控后没有任何 active lane 时执行本指令是合法空操作 (贡献 0、不阻塞)。软件契约: 每个计入 thread_count 的线程必须最终以预期的掩码执行到 barrier, 否则死锁属于程序错误 — barrier 应在参与线程收敛的位置执行。

代际规则: barrier 释放 (到达数达到参与总数) 后计数自动复位、进入下一代, 之后的到达计入新一代。thread_count = 0 或超过 cta 线程总数是非法参数, 硬件 trap; 同一代内不同 warp 提供的 thread_count 必须一致, 不一致时行为等同程序错误。

thread_count 可选: 不带 thread_count 时全 cta 参与; 带 thread_count 时只有指定数量的 thread 参与 (counted barrier, 用于 producer / consumer warpgroup 模型)。

指令形态 4 种 (id × count 笛卡尔积): _i (imm id, 全 cta) / _ix (imm id + sreg count) / _x (sreg id, 全 cta, 用于动态选 barrier slot) / _xx (sreg id + sreg count)。

典型用例: cta 内 warp 间同步 (例如 GEMM 主循环每轮迭代结束 sync) / producer / consumer warpgroup 协作 (counted barrier 让 producer 不参与同步直接 bar_arrive 通知, consumer bar_sync 等齐)。

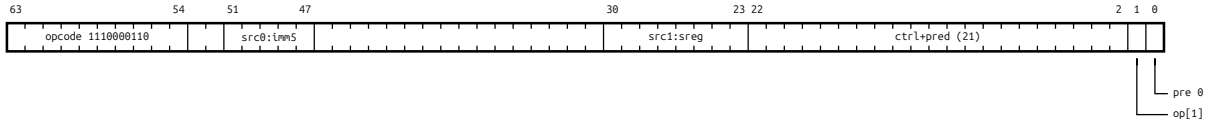
Leaf: bar_sync___ix

Format:

```
bar_sync  src0, src1
```

Operands: src0: imm5u; src1: sreg

Encoding Diagram: 64b, prefix 0, opcode 11110000110



Notes:

cta named barrier sync: 所有参与 thread 必须都到达这条指令后才能继续; 只要 warp 内还有 ≥1 个门控后 active 的 lane, warp 就停在这条指令直到 barrier 释放。最多 32 个 named barrier per cta (barrier_id 用 imm.5 即 0..31)。

到达计数以线程为单位, 由谓词门控与 EXEC 掩码逐 lane 决定: 一个 warp 执行一次本指令, 向 barrier 贡献的到达数 = 门控后 active lane 的数量 (popcount(EXEC 与谓词门的按位与))。被 kill 的 lane 已从 EXEC 清除, 自然不计入; 门控后没有任何 active lane 时执行本指令是合法空操作 (贡献 0、不阻塞)。软件契约: 每个计入 thread_count 的线程必须最终以预期的掩码执行到 barrier, 否则死锁属于程序错误 — barrier 应在参与线程收敛的位置执行。

代际规则: barrier 释放 (到达数达到参与总数) 后计数自动复位、进入下一代, 之后的到达计入新一代。thread_count = 0 或超过 cta 线程总数是非法参数, 硬件 trap; 同一代内不同 warp 提供的 thread_count 必须一致, 不一致时行为等同程序错误。

thread_count 可选: 不带 thread_count 时全 cta 参与; 带 thread_count 时只有指定数量的 thread 参与 (counted barrier, 用于 producer / consumer warpgroup 模型)。

指令形态 4 种 (id × count 笛卡尔积): _i (imm id, 全 cta) / _ix (imm id + sreg count) / _x (sreg id, 全 cta, 用于动态选 barrier slot) / _xx (sreg id + sreg count)。

典型用例: cta 内 warp 间同步 (例如 GEMM 主循环每轮迭代结束 sync) / producer / consumer warpgroup 协作 (counted barrier 让 producer 不参与同步直接 bar_arrive 通知, consumer bar_sync 等齐)。

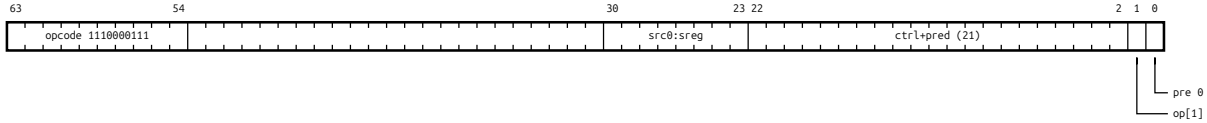
Leaf: bar_sync___x

Format:

```
bar_sync  src0
```

Operands: src0: sreg

Encoding Diagram: 64b, prefix 0, opcode 11110000111



Notes:

cta named barrier sync: 所有参与 thread 必须都到达这条指令后才能继续; 只要 warp 内还有 ≥1 个门控后 active 的 lane, warp 就停在这条指令直到 barrier 释放。最多 32 个 named barrier per cta (barrier_id 用 imm.5 即 0..31)。

到达计数以线程为单位, 由谓词门控与 EXEC 掩码逐 lane 决定: 一个 warp 执行一次本指令, 向 barrier 贡献的到达数 = 门控后 active lane 的数量 (popcount(EXEC 与谓词门的按位与))。被 kill 的 lane 已从 EXEC 清除, 自然不计入; 门控后没有任何 active lane 时执行本指令是合法空操作 (贡献 0、不阻塞)。软件契约: 每个计入 thread_count 的线程必须最终以预期的掩码执行到 barrier, 否则死锁属于程序错误 — barrier 应在参与线程收敛的位置执行。

代际规则: barrier 释放 (到达数达到参与总数) 后计数自动复位、进入下一代, 之后的到达计入新一代。thread_count = 0 或超过 cta 线程总数是非法参数, 硬件 trap; 同一代内不同 warp 提供的 thread_count 必须一致, 不一致时行为等同程序错误。

thread_count 可选: 不带 thread_count 时全 cta 参与; 带 thread_count 时只有指定数量的 thread 参与 (counted barrier, 用于 producer / consumer warpgroup 模型)。

指令形态 4 种 (id × count 笛卡尔积): _i (imm id, 全 cta) / _ix (imm id + sreg count) / _x (sreg id, 全 cta, 用于动态选 barrier slot) / _xx (sreg id + sreg count)。

典型用例: cta 内 warp 间同步 (例如 GEMM 主循环每轮迭代结束 sync) / producer / consumer warpgroup 协作 (counted barrier 让 producer 不参与同步直接 bar_arrive 通知, consumer bar_sync 等齐)。

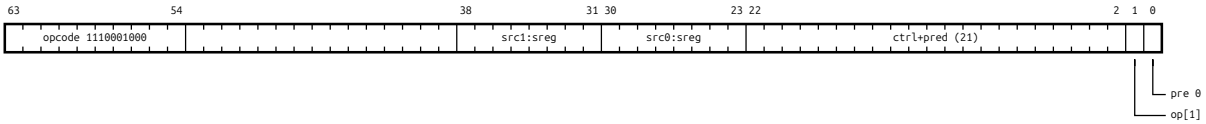
Leaf: bar_sync___xx

Format:

```
bar_sync  src0, src1
```

Operands: src0: sreg; src1: sreg

Encoding Diagram: 64b, prefix 0, opcode 11110001000



Notes:

cta named barrier sync: 所有参与 thread 必须都到达这条指令后才能继续; 只要 warp 内还有 ≥1 个门控后 active 的 lane, warp 就停在这条指令直到 barrier 释放。最多 32 个 named barrier per cta (barrier_id 用 imm.5 即 0..31)。

到达计数以线程为单位, 由谓词门控与 EXEC 掩码逐 lane 决定: 一个 warp 执行一次本指令, 向 barrier 贡献的到达数 = 门控后 active lane 的数量 (popcount(EXEC 与谓词门的按位与))。被 kill 的 lane 已从 EXEC 清除, 自然不计入; 门控后没有任何 active lane 时执行本指令是合法空操作 (贡献 0、不阻塞)。软件契约: 每个计入 thread_count 的线程必须最终以预期的掩码执行到 barrier, 否则死锁属于程序错误 — barrier 应在参与线程收敛的位置执行。

代际规则: barrier 释放 (到达数达到参与总数) 后计数自动复位、进入下一代, 之后的到达计入新一代。thread_count = 0 或超过 cta 线程总数是非法参数, 硬件 trap; 同一代内不同 warp 提供的 thread_count 必须一致, 不一致时行为等同程序错误。

thread_count 可选: 不带 thread_count 时全 cta 参与; 带 thread_count 时只有指定数量的 thread 参与 (counted barrier, 用于 producer / consumer warpgroup 模型)。

指令形态 4 种 (id × count 笛卡尔积): _i (imm id, 全 cta) / _ix (imm id + sreg count) / _x (sreg id, 全 cta, 用于动态选 barrier slot) / _xx (sreg id + sreg count)。

典型用例: cta 内 warp 间同步 (例如 GEMM 主循环每轮迭代结束 sync) / producer / consumer warpgroup 协作 (counted barrier 让 producer 不参与同步直接 bar_arrive 通知, consumer bar_sync 等齐)。

depbar **category**: Synchronization and memory ordering **predicate_mode**: uniform

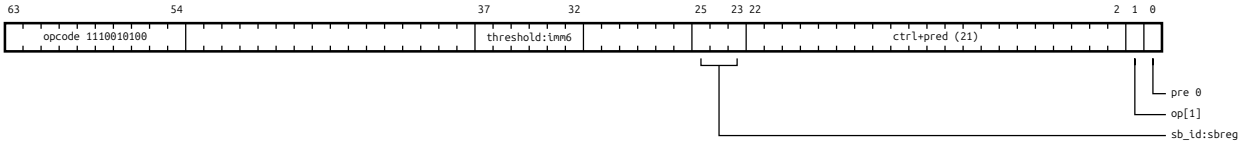
Leaf: depbar___xi

Format:

```
depbar  sb_id, threshold
```

Operands: sb_id: sbreg; threshold: imm6u

Encoding Diagram: 64b, prefix 0, opcode 11110010100



Notes:

Explicit dependency wait (le 指令形态): stall issue until SB[sbidx] <= threshold。卡住后续指令 issue 直到 dependence counter SBx 降到 ≤ threshold (le = Less or Equal)。

SBx 模型 (software-managed scoreboard): 硬件 6 个 per-warp dependence counter (SB0..SB5, 6-bit 即 0..63)。每条指令通过 rd_sb (read-release, WAR) + wr_sb (write-release, RAW / WAW) 公共控制字段在 issue 时 SBx +1, 完成时 SBx -1; 后续指令通过 inline req_sb 6-bit mask 或 depbar 显式指令等待。

vs inline req_sb: req_sb (6-bit inline mask) 只能表达 'wait until SBx == 0', 适合紧耦合等待 (ML kernel 90%+ 路径); depbar 补足三件 inline 做不到的事 — (1) threshold 比较 (SB ≤ N) 实现双缓冲 pipeline (允许 N 个 outstanding op 让 prefetch 跑); (2) 跨 basic block (depbar 是 issue slot 形态可放任意位置, 跟 dataflow 解耦); (3) 多 SBx 联合等待 (_xii 指令形态在 sbidx threshold 之上叠加 sb_set bitmask, 单条等多个)。

_xi 指令形态 (主路径): src0 = SBx selector (reg.sbreg, 3-bit, SB0..SB5 + SB_NONE=7), src1 = threshold (imm.6, 0..63)。

_xii 指令形态 (完整路径): 在 _xi 基础上加 src2 = sb_set (imm.6 6-bit bitmask), 同时等 sb_set 内 SBx 全 == 0。约束: sb_set 不能包含 sbidx (主 SBx 跟 aux 集合互斥), 违反属于非法编码。

典型用法: depbar SB0, 0 (等 SB0 上所有 op 完成) / depbar SB0, 3 (双缓冲 pipeline 主路径, 保 3 个 outstanding) / depbar SB0, 0, 0b001110 (等 SB0==0 + SB1 / SB2 / SB3 全 0)。

SBx 分配 (compiler 决策): 编译器把 long-latency 指令归到哪个 SBx — 多条共享一个 SBx 可粗粒度合并等待, 分散用不同 SBx 可细粒度 pipelined 等待。

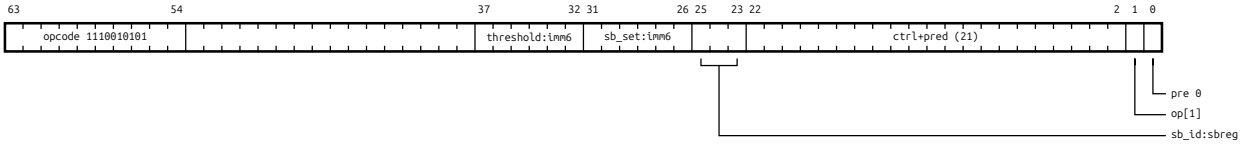
Leaf: depbar___xii

Format:

```
depbar  sb_id, threshold, sb_set
```

Operands: sb_id: sbreg; threshold: imm6u; sb_set: imm6u

Encoding Diagram: 64b, prefix 0, opcode 11110010101



Notes:

Explicit dependency wait (le 指令形态): stall issue until SB[sbidx] <= threshold。卡住后续指令 issue 直到 dependence counter SBx 降到 ≤ threshold (le = Less or Equal)。

SBx 模型 (software-managed scoreboard): 硬件 6 个 per-warp dependence counter (SB0..SB5, 6-bit 即 0..63)。每条指令通过 rd_sb (read-release, WAR) + wr_sb (write-release, RAW / WAW) 公共控制字段在 issue 时 SBx +1, 完成时 SBx -1; 后续指令通过 inline req_sb 6-bit mask 或 depbar 显式指令等待。

vs inline req_sb: req_sb (6-bit inline mask) 只能表达 'wait until SBx == 0', 适合紧耦合等待 (ML kernel 90%+ 路径); depbar 补足三件 inline 做不到的事 — (1) threshold 比较 (SB ≤ N) 实现双缓冲 pipeline (允许 N 个 outstanding op 让 prefetch 跑); (2) 跨 basic block (depbar 是 issue slot 形态可放任意位置, 跟 dataflow 解耦); (3) 多 SBx 联合等待 (_xii 指令形态在 sbidx threshold 之上叠加 sb_set bitmask, 单条等多个)。

_xi 指令形态 (主路径): src0 = SBx selector (reg.sbreg, 3-bit, SB0..SB5 + SB_NONE=7), src1 = threshold

(imm.6, 0..63)。

_xii 指令形态 (完整路径): 在 _xi 基础上加 src2 = sb_set (imm.6 6-bit bitmask), 同时等 sb_set 内 SBx 全 == 0。约束: sb_set 不能包含 sbidx (主 SBx 跟 aux 集合互斥), 违反属于非法编码。

典型用法: depbar SB0, 0 (等 SB0 上所有 op 完成) / depbar SB0, 3 (双缓冲 pipeline 主路径, 保 3 个 outstanding) / depbar SB0, 0, 0b001110 (等 SB0==0 + SB1 / SB2 / SB3 全 0)。

SBx 分配 (compiler 决策): 编译器把 long-latency 指令归到哪个 SBx — 多条共享一个 SBx 可粗粒度合并等待, 分散用不同 SBx 可细粒度 pipelined 等待。

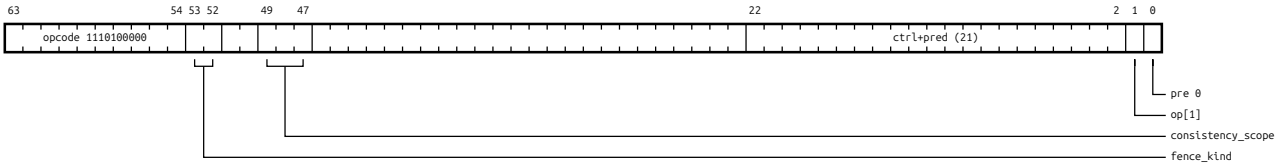
fence **category**: Synchronization and memory ordering **predicate_mode**: uniform

Leaf: fence__u

Format:

```
fence{.fence_kind}.consistency_scope
```

Encoding Diagram: 64b, prefix 0, opcode 11110100000



Notes:

Memory ordering fence: 在本指令处对内存操作施加一道顺序约束, 方向和强度由 fence_kind 选。fence 不读写数据、不搬任何东西, 它只规定 '本指令之前 / 之后的内存操作' 谁必须先于谁被别的观察者看到。

fence_kind 4 valid: acq (acquire, 定序后续读 — 本线程之后的任何 load 都能看到 fence 之前其他线程已 release 的 store) / rel (release, 定序前置写 — fence 之前本线程所有 store 在 fence 之后对其他线程可见) / sc (sequentially-consistent, 双向全序 — fence 之前的所有内存操作都在指定 scope 上可见之后, 才放出之后的操作, 对所有类型的内存操作都生效; 这是最强的一档) / async (async-proxy fence, 跨越 '普通访存通路' 与 '异步搬运引擎通路' 两个代理之间的可见性边界, 配合异步搬运使用 — 它管的是 '哪个代理看得见', 不是 '多远可见')。

acq / rel 通常成对: 生产者写完数据做一次 rel, 消费者读数据前做一次 acq, 两者配合才能保证消费者读到的是生产者写好的数据。需要双向全序的强 ordering 场景用 sc。

consistency_scope: cta / sm / cluster / gpu / sys, 控这道顺序约束覆盖到多远的观察者范围。async 方向不看距离 (它管代理不管 scope), 其余方向按 scope 生效。

无 operand (fence 是 control-only 指令)。predicate_mode=uniform: 整 warp 一致执行, 可被 uniform 谓词门控。

nanosleep **category**: Synchronization and memory ordering **predicate_mode**: uniform

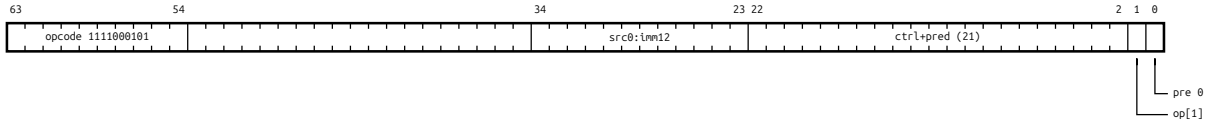
Leaf: nanosleep___i

Format:

```
nanosleep  src0
```

Operands: src0: imm12u

Encoding Diagram: 64b, prefix 0, opcode 11111000101



Notes:

Approximate nanosleep: 暂停约 delay_ns 纳秒。实际延迟近似 (硬件可能 round 到内部时基), 不保证精确。

指令形态: _i (imm.12 静态 delay) / _x (sreg 32-bit 动态 delay, 用于 exponential / random backoff)。

典型用例: spinlock retry pattern 做 backoff 减少 cache thrashing / busy-wait polling 节流减 power。

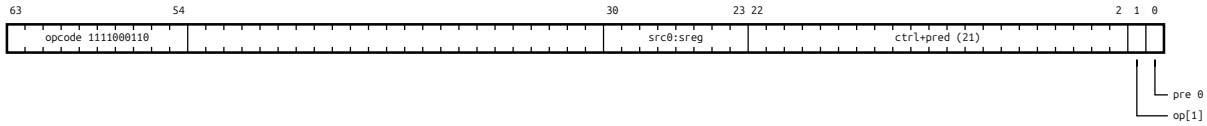
Leaf: nanosleep___x

Format:

```
nanosleep  src0
```

Operands: src0: sreg

Encoding Diagram: 64b, prefix 0, opcode 11111000110



Notes:

Approximate nanosleep: 暂停约 delay_ns 纳秒。实际延迟近似 (硬件可能 round 到内部时基), 不保证精确。

指令形态: _i (imm.12 静态 delay) / _x (sreg 32-bit 动态 delay, 用于 exponential / random backoff)。

典型用例: spinlock retry pattern 做 backoff 减少 cache thrashing / busy-wait polling 节流减 power。

14.7.16 Matrix and tensor

dmma **category**: Matrix and tensor **predicate_mode**: simt

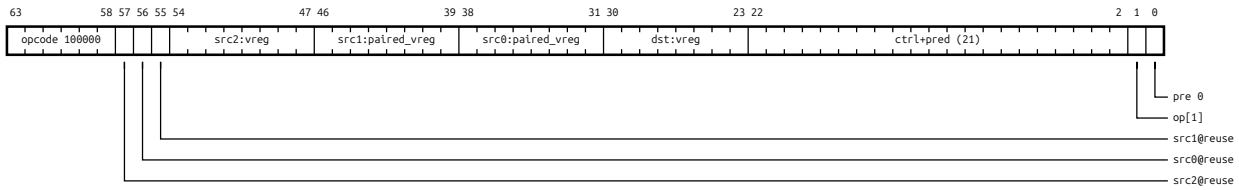
Leaf: dmma__v_vvv

Format:

```
dmma  dst, src0{.reuse}, src1{.reuse}, src2{.reuse}
```

Operands: dst: vreg [notes: span=4, align=2]; src0: paired_vreg; src1: paired_vreg; src2: vreg [notes: span=4, align=2]

Encoding Diagram: 64b, prefix 0, opcode 1100000



Notes:

FP64 matrix multiply-accumulate: D = A × B + C, 整 warp 32 lane 协作算一个 m8n8 FP64 矩阵块乘累加 — A (m8×k4 FP64) × B (k4×n8 FP64) + C (m8×n8 FP64) → D (m8×n8 FP64)。一条 dmma 一次累加 K=4 个 FP64 元素。datapath: tensor core (FP64 path)。

dmma 不暴露 mma_shape / accum_type / rounding modifier: FP64 路径仅 M8N8K4 + FP64 accum + rne 一种组合 (FP64 元素 64-bit 占 reg pair, 每行只装 8 个元素, 所以 m=8 而不是 hmma / imma / qmma 的 m=16; 累加只能 FP64 没更高精度可选)。

operand 语义: dst (D) = vreg multi (4 reg = 2 FP64 pair, 偶对齐), src0 (A) = vreg (2 reg), src1 (B) = vreg (2 reg), src2 (C) = vreg multi (4 reg 跟 D 同 size); 所有 vreg 偶对齐 (跟 dadd / dmul / dfma 同源)。

典型用例: 科学计算 / HPC 主路径 (FP64 高精度数值线性代数) / FP64 训练 (罕见路径, 多数 ML 训练已转 FP8 / bf16)。

hmma **category**: Matrix and tensor **predicate_mode**: simt

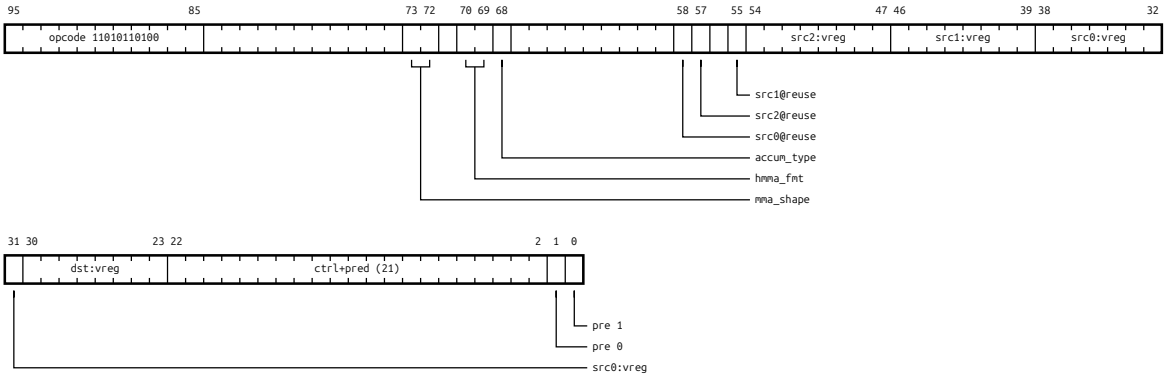
Leaf: hmma__v_vvv

Format:

```
hmma.mma_shape{.hmma_fmt}{.accum_type}  dst, src0{.reuse}, src1{.reuse}, src2{.reuse}
```

Operands: dst: vreg; src0: vreg; src1: vreg; src2: vreg

Encoding Diagram: 96b, prefix 10, opcode 11010110100



Notes:

Half-precision matrix multiply-accumulate: D = A × B + C, 整 warp 32 lane 协作算一个 m16n8 矩阵块乘累加 — A (m16×k 矩阵 bf16 / f16 / tf32) × B (k×n8 矩阵) + C (m16×n8 累加 input) → D (m16×n8 output)。一条指令累加 K 个元素 (K 跟 fmt 自然耦合)。datapath: tensor core。

hmma_fmt 3 valid: bf16 (1 sign + 8 exp + 7 mantissa, ML 训练主流, 大 dynamic range 跟 FP32 同精度低, K=16 主路径) / f16 (IEEE 754 半精度, 1 sign + 5 exp + 10 mantissa, 推理 + 训练 with loss-scaling, 精度比 bf16 高 dynamic range 小, K=16 主路径) / tf32 (TensorFloat-32, 1 sign + 8 exp + 10 mantissa 实际 19-bit 编码占 32-bit reg, bf16 dynamic range + f16 精度折衷, K=8 主路径因为 tf32 reg pair 装 K 元素只有 bf16 / f16 一半)。

accum_type 2 valid: f16 (累加紧凑精度低易溢出) / f32 (累加高精度主路径, ML 训练主流避免精度损失)。

mma_shape × hmma_fmt valid_combo: bf16 / f16 + m16n8k8 (小 K, ML kernel 极少用) / bf16 / f16 + m16n8k16 (主路径) / bf16 / f16 + m16n8k32 (大 K) / tf32 + m16n8k8 (主路径) / tf32 + m16n8k16 (大 K)。其他组合 (tf32 + K=32 / bf16 + K=64 等) invalid 硬件未实现 trap。

operand 语义: dst (D) / src0 (A) / src1 (B) / src2 (C) 都是 vreg multi-reg, 实际 reg 占用跟 (mma_shape, hmma_fmt, accum_type) 联合决定 (典型 bf16 / f16 主路径 m16n8k16 + f32 accum: D=4 reg, A=2 reg, B=1 reg, C=4 reg)。

imma **category**: Matrix and tensor **predicate_mode**: simt

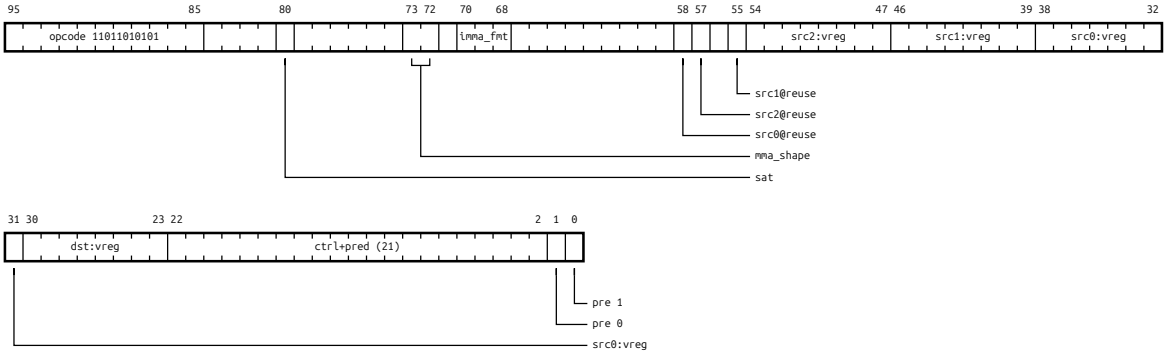
Leaf: imma__v_vvv

Format:

```
imma.mma_shape{.imma_fmt}{.sat}  dst, src0{.reuse}, src1{.reuse}, src2{.reuse}
```

Operands: dst: vreg; src0: vreg; src1: vreg; src2: vreg

Encoding Diagram: 96b, prefix 10, opcode 11011010101



Notes:

Integer matrix multiply-accumulate: D = A × B + C, 整 warp 32 lane 协作算一个 m16n8 整数矩阵块乘累加 — A (m16×k INT8 / INT4) × B (k×n8 INT8 / INT4) + C (m16×n8 int32) → D (m16×n8 int32)。K 跟 fmt 自然耦合 (8-bit 主路径 K=32, 4-bit 主路径 K=64)。累加器固定 int32。datapath: tensor core。

imma_fmt 8 valid (mixed-precision A × B 笛卡尔积): u8u8 (A 跟 B 都 unsigned 8-bit, 推理常用) / s8s8 (都 signed 8-bit, 训练 backward / 量化 fine-tune) / u8s8 (A unsigned + B signed, 主流量化 weight signed × activation unsigned) / s8u8 (反向 mixed) / u4u4 (都 unsigned 4-bit, 低精度推理) / s4s4 (都 signed 4-bit) / u4s4 / s4u4 (4-bit mixed)。

sat (1-bit, 默认 off): 累加饱和 — on 时 dst clamp to int32 [INT32_MIN, INT32_MAX], off 时 wrap-around (跟普通整数运算一致)。ML 量化主路径 off (饱和会破坏梯度), 推理路径有时用 on 实现安全 int8 量化。

mma_shape × imma_fmt valid_combo: 8-bit + m16n8k16 (小 K) / 8-bit + m16n8k32 (主路径) / 4-bit + m16n8k32 (小 K) / 4-bit + m16n8k64 (主路径)。

operand 语义: dst (D) / src0 (A) / src1 (B) / src2 (C) 都是 vreg multi-reg, reg 占用跟 (mma_shape, imma_fmt) 联合决定 (典型 8-bit + m16n8k32: A=4 reg, B=2 reg, D=C=4 reg)。

qmma **category**: Matrix and tensor **predicate_mode**: simt

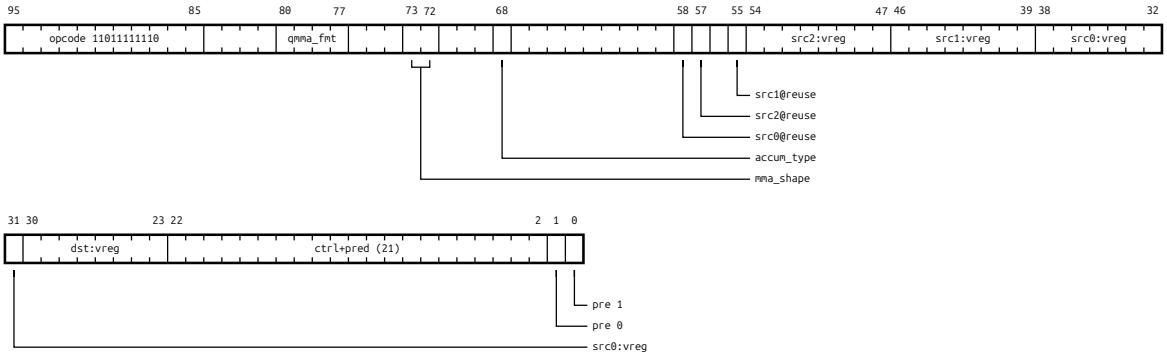
Leaf: qmma__v_vvv

Format:

```
qmma.mma_shape{.qmma_fmt}{.accum_type}  dst, src0{.reuse}, src1{.reuse}, src2{.reuse}
```

Operands: dst: vreg; src0: vreg; src1: vreg; src2: vreg

Encoding Diagram: 96b, prefix 10, opcode 11011111110



Notes:

Quantized FP matrix multiply-accumulate: D = A × B + C, 整 warp 32 lane 协作算一个 m16n8 矩阵块乘累

加, 输入是 FP8 / FP6 / FP4 微缩浮点 (mxfp 标准的训练 / 推理低精度 fmt)。K 跟 fmt 自然耦合 (8-bit FP8 / FP6 K=32 主路径, 4-bit FP4 K=64 主路径)。FP6 实际占 8-bit reg (高位填 2 个 0, 跟 mxfp 标准一致)。datapath: tensor core。

qmma_fmt 9 valid: FP8 路径 (e4m3_e4m3 高精度 FP8 训练 forward 主流 / e5m2_e5m2 大 dynamic range 训练 backward gradient 主流 / e4m3_e5m2 BWD pass mixed weight × grad / e5m2_e4m3 反向 mixed) / FP6 路径 (e2m3_e2m3 FP6 高精度子型 / e3m2_e3m2 FP6 大范围子型 / e2m3_e3m2 / e3m2_e2m3 FP6 mixed) / FP4 路径 (e2m1_e2m1, NVF4 标准, MX-FP4 训练 / 推理)。

accum_type 2 valid (跟 hmma 共享): f32 (训练主流, 避免低精度累加误差) / f16 (推理紧凑路径)。

mma_shape × qmma_fmt valid_combo: FP8 + m16n8k16 (小 K) / FP8 + m16n8k32 (主路径) / FP6 + m16n8k32 (主路径, 跟 mxfp 标准) / FP4 (e2m1) + m16n8k64 (主路径)。

operand 语义: dst (D) / src0 (A) / src1 (B) / src2 (C) 都是 vreg multi-reg, reg 占用跟 (mma_shape, qmma_fmt, accum_type) 联合决定 (跟 hmma / imma 同源约定)。

utcalloc **category**: Matrix and tensor **predicate_mode**: uniform

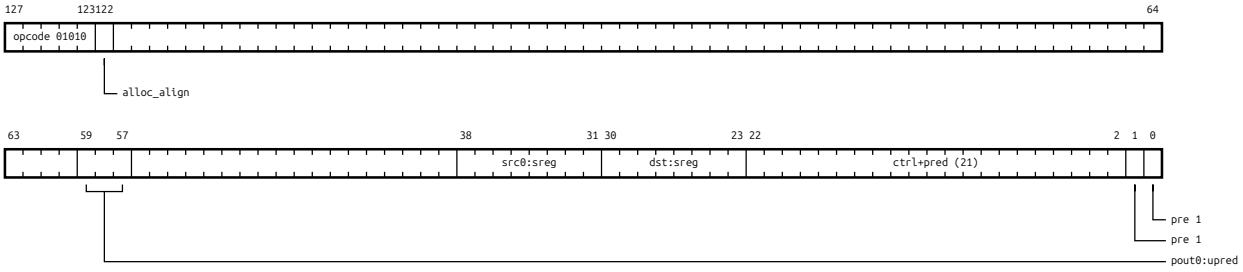
Leaf: utcalloc__sp_s

Format:

```
utcalloc{.alloc_align}  dst, pout0, src0
```

Operands: dst: sreg; pout0: upred; src0: sreg

Encoding Diagram: 128b, prefix 11, opcode 01010



Notes:

Tensor memory 分配: 在当前 CTA 的 tensor memory 分配位图中寻找一段连续的空闲块, 并把它原子地标记为已占用。分配位图是每个 CTA 一份的 32 位硬件状态字: 把 tensor memory 的列空间等分成 32 个块, 位 i 对应第 i 个块, 1 表示该块已被占用、0 表示空闲。utcalloc / utcfree / utcamap 三条指令操作的是同一张位图, 共同构成 tensor memory 的分配管理接口: utcalloc 与 utcfree 覆盖最常见的分配 / 释放路径, utcamap 提供位图的原始访问原语供软件自建策略。

语义: 硬件在位图中从低位向高位扫描, 寻找第一段长度恰好覆盖 src0 个块的连续 0 位; 找到则把这段位原子地置 1、把段的起始块编号写入 dst、把 pout0 置真; 整张位图中不存在满足条件的空闲段则分配失败, pout0 置假, dst 的内容未定义 — 软件必须以 pout0 判断成败, 不得在失败时使用 dst 的值。查找与置位在硬件里是单次原子动作, 多个 warp 并发执行 utcalloc 不会分到相互重叠的块。

src0 (block_count): 请求的连续块数量, 合法取值 1..32, 整 warp 一致 (uniform 通路)。dst (block_base): 分配成功时的起始块编号 (0..31)。软件把块编号乘以每块的列宽即得 tensor memory 列地址, 交给 utcmma / utccp / utcld / utcst 使用。

alloc_align=align 时, 硬件把 src0 向上取整到 2 的幂并要求起始块编号按该值自然对齐 (见 alloc_align modifier); noalign (默认) 只保证连续。

典型使用序列: utcalloc 分配 → 依赖计数器等到完成 → 检查 pout0、把 dst 换算成 tensor memory 地址

→ utccp / utcmma 在这段列上工作 → 计算完成后用 utcfree 释放同一段块。可变延迟指令, 发射时经 rd_sb / wr_sb 挂依赖计数器, 消费者用 req_sb 等待。

仅 CS 可用。128-bit 指令。

utcamap **category**: Matrix and tensor **predicate_mode**: uniform

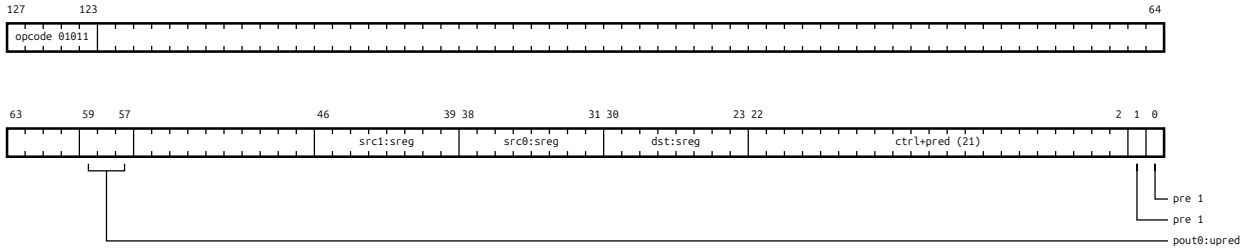
Leaf: utcamap__cas.sp_ss

Format:

```
utcamap  dst, pout0, src0, src1
```

Operands: dst: sreg; pout0: upred; src0: sreg; src1: sreg

Encoding Diagram: 128b, prefix 11, opcode 01011



Notes:

Tensor memory 分配位图的比较交换 (compare-and-swap): 如果位图当前值等于 src0, 就把位图整字替换成 src1 并把 pout0 置真; 不相等则位图保持不变、pout0 置假。无论成败, dst 都写入比较那一瞬间的位图旧值。比较与替换在硬件里是单次原子动作 (位图定义见 utcalloc)。

这是软件自建无锁分配策略的仲裁原语。典型模式: 先用 ld 指令形态读出旧位图、在旧值的基础上算出期望的新位图 (占用或释放若干位), 再用 cas 提交 — src0 填读到的旧值、src1 填算好的新值; 提交失败 (pout0 为假) 说明位图在读与提交之间已被其他 warp 并发修改, 软件拿 dst 里返回的最新值重新计算并重试。

操作数: dst (旧值, sreg), pout0 (成功标志, upred), src0 (期望的比较值, sreg), src1 (要写入的新值, sreg)。

可变延迟指令, 完成经依赖计数器追踪。仅 CS 可用。128-bit 指令。

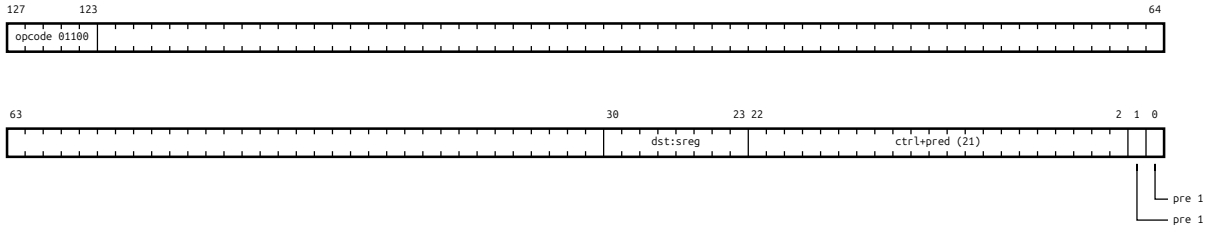
Leaf: utcamap__ld.s

Format:

```
utcamap  dst
```

Operands: dst: sreg

Encoding Diagram: 128b, prefix 11, opcode 01100



Notes:

Tensor memory 分配位图的原始读取: dst = 当前 CTA 分配位图的即时快照 (32 位, 每位对应列空间的一个等分块, 1= 占用 / 0= 空闲, 定义见 utcalloc)。

utcamap 一族共四个指令形态 (ld / st / or / cas), 是分配位图的原始访问原语, 服务于 utcalloc / utcfree 之外的软件自建策略: 例如在两个 warp 组之间移交一段已分配的块 (生产者分配、消费者释放)、保存并恢复整张

位图 (软件调度器切换工作集)、或用 cas 实现多方无锁仲裁。操作种类由指令形态区分, 不设单独的 modifier (风格同 match 的 any / all)。

快照是读取那一瞬间的位图值, 不阻止其他 warp 的并发修改; 要基于读到的值做决策并写回, 必须用 cas 指令形态保证原子性, 不要用 ld + st 两条指令的序列 (两条指令之间位图可能已被并发改动, 覆写会丢失那些改动)。

可变延迟指令, 完成经依赖计数器追踪。仅 CS 可用。128-bit 指令。

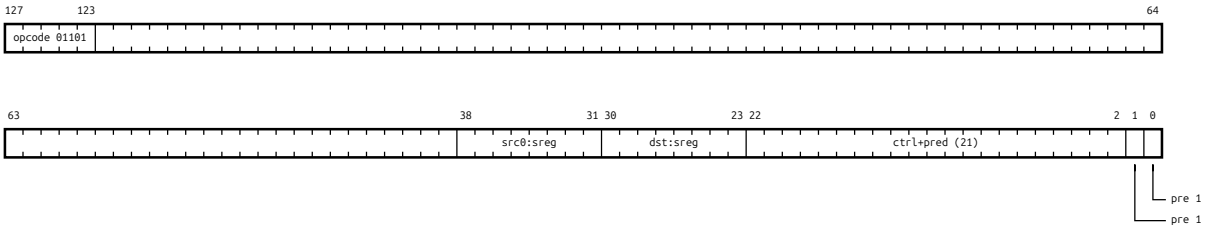
Leaf: utcamap__or.s_s

Format:

```
utcamap  dst, src0
```

Operands: dst: sreg; src0: sreg

Encoding Diagram: 128b, prefix 11, opcode 01101



Notes:

Tensor memory 分配位图的原子置位: dst = 位图旧值; 位图 = 位图 OR src0。把 src0 中置 1 的位标记为已占用, 读旧值与置位在硬件里是单次原子动作 (位图定义见 utcalloc)。

与 utcalloc 的分工: utcalloc 是由硬件寻找空闲段的分配路径; or 指令形态由软件自行指定要占用哪些位 — 用于软件自定分配策略 (先用 ld 观察位图、自行挑选块、再落位) 或把外部约定好的固定布局直接标记进位图。

注意 or 不检查目标位是否已经被占用, 对已占用的位置 1 不报错也不产生任何标记; 需要 '仅当这些位全部空闲才占用' 的语义时, 用 cas 指令形态做条件提交。

可变延迟指令, 完成经依赖计数器追踪。仅 CS 可用。128-bit 指令。

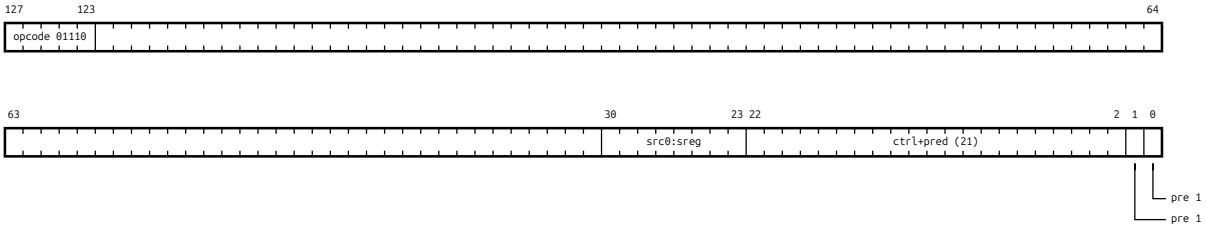
Leaf: utcamap__st.s

Format:

```
utcamap  src0
```

Operands: src0: sreg

Encoding Diagram: 128b, prefix 11, opcode 01110



Notes:

Tensor memory 分配位图的整字覆写: 位图 = src0, 一次写入整张 32 位位图, 无条件覆盖旧值 (位图定义见 utcalloc)。

用途限于两类场景: 初始化 (内核启动阶段把位图清零, 或写入一个预先约定的固定布局) 与恢复 (把先前用 ld 指令形态保存的位图快照写回, 恢复软件调度器的分配状态)。

覆写不与并发的原子操作协调, 会无条件丢弃其他 warp 同时刻做出的修改; 因此只能在确知没有并发分配活动的时刻使用, 例如 CTA 内先用屏障把所有 warp 等齐再执行。运行中的常规修改一律走 or / cas 指令形态或 utcalloc / utcfree。

可变延迟指令, 完成经依赖计数器追踪。仅 CS 可用。128-bit 指令。

utcbar **category**: Matrix and tensor **predicate_mode**: uniform

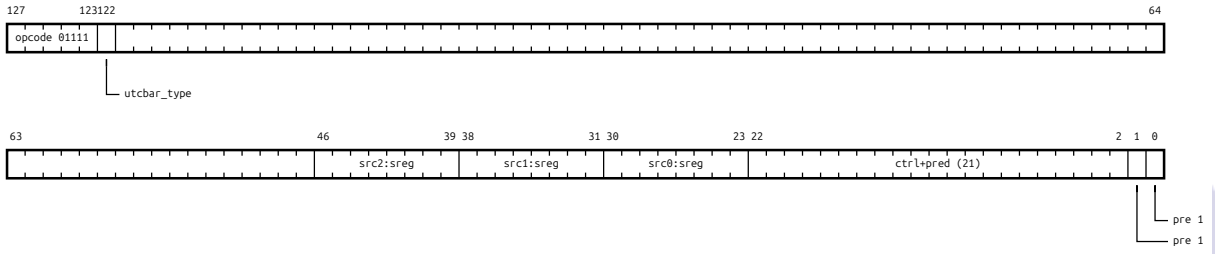
Leaf: utcbar__sss

Format:

```
utcbar{.utcbar_type}  src0, src1, src2
```

Operands: src0: sreg; src1: sreg; src2: sreg

Encoding Diagram: 128b, prefix 11, opcode 01111



Notes:

Tensor core barrier: 同步 TC 异步操作的完成。

src0: barrier descriptor, 1 个 UR。

src1: barrier parameter, 1 个 UR。

src2: multicast mask (URZ= 不做 multicast), 1 个 UR。

仅 CS 可用。128-bit 指令。

utccp **category**: Matrix and tensor **predicate_mode**: uniform

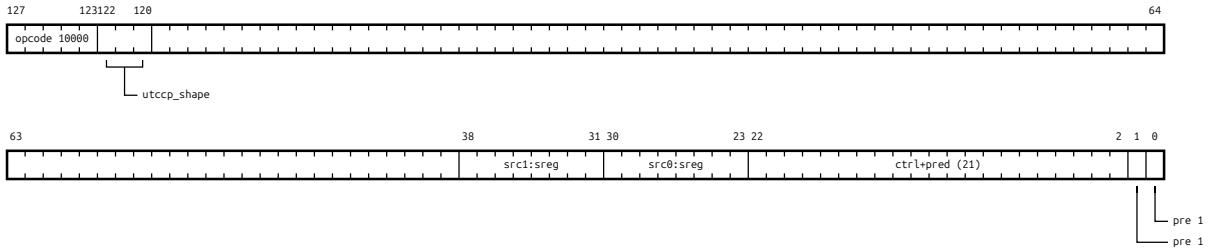
Leaf: utccp__smem_to_tmem

Format:

```
utccp{.utccp_shape}  src0, src1
```

Operands: src0: sreg; src1: sreg

Encoding Diagram: 128b, prefix 11, opcode 10000



Notes:

Shared memory → tensor memory copy: 从 shared memory 拷贝矩阵 tile 到 tensor memory。单线程 issue 即可。

src0 (tmem_addr): tensor memory 目标地址, 1 个 UR (32-bit)。

src1 (smem_desc): shared memory descriptor, 1 个 UR (32-bit), 编码 shared memory 起始地址 + leading dimension + stride + swizzle mode。

utccp_shape 选择 copy tile 大小: 128x256b (默认) / 128x128b / 64x128b / 32x128b。

异步指令。仅 CS 可用。128-bit 指令。

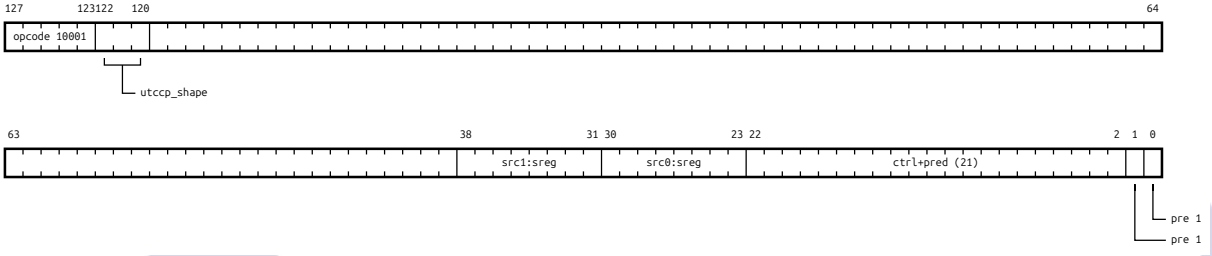
Leaf: utccp__tmem_to_smem

Format:

```
utccp{.utccp_shape}  src0, src1
```

Operands: src0: sreg; src1: sreg

Encoding Diagram: 128b, prefix 11, opcode 10001



Notes:

Tensor memory → shared memory copy: 从 tensor memory 拷贝数据到 shared memory。

src0 (smem_desc): shared memory descriptor, 1 个 UR。

src1 (tmem_addr): tensor memory 源地址, 1 个 UR。

异步指令。仅 CS 可用。128-bit 指令。

utcfree **category**: Matrix and tensor **predicate_mode**: uniform

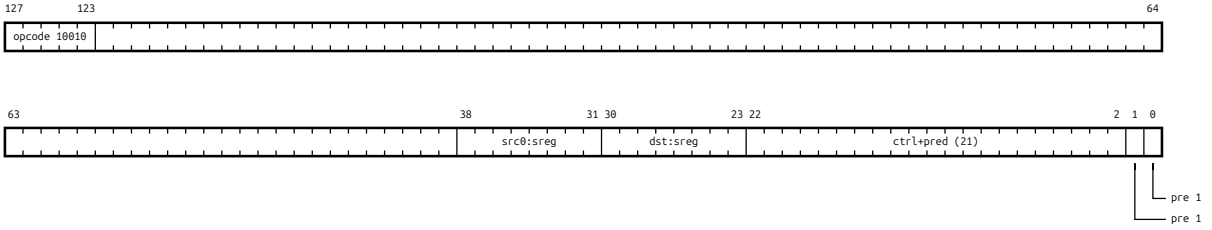
Leaf: utcfree__s_s

Format:

```
utcfree  dst, src0
```

Operands: dst: sreg; src0: sreg

Encoding Diagram: 128b, prefix 11, opcode 10010



Notes:

Tensor memory 释放: 把 src0 中置 1 的位在当前 CTA 的 tensor memory 分配位图中原子地清 0, 对应的块回到空闲状态, 可以被后续的 utcalloc 重新分配。分配位图的定义见 utcalloc (32 位状态字, 每位对应列空间的一个等分块, 1= 占用 / 0= 空闲)。

语义: dst = 位图旧值; 位图 = 位图 AND NOT src0。读出旧值与清位在硬件里是单次原子动作。注意 src0 是 ** 释放掩码 ** 而不是保留掩码 — 要释放从块 b 开始的 n 个块, 软件构造掩码 ((1«n)-1)«b 作为 src0 (可以用 bmsk 指令生成)。对本来就空闲的位执行释放不报错, 这些位保持 0 不变。

dst 返回旧位图, 便于软件做一致性校验 (例如断言被释放的位在旧值中确实为 1, 以捕捉重复释放这类分配器 bug); 不需要旧值时 dst 填 URZ 丢弃。

可变延迟指令, 完成经依赖计数器追踪 (同 utcalloc)。仅 CS 可用。128-bit 指令。

utcld **category**: Matrix and tensor **predicate_mode**: uniform

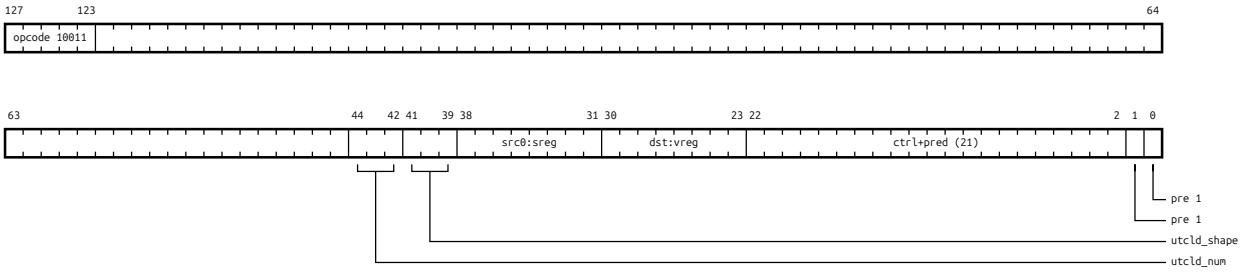
Leaf: utcld__v_s

Format:

```
utcld{.utcld_shape}{.utcld_num}  dst, src0
```

Operands: dst: vreg; src0: sreg

Encoding Diagram: 128b, prefix 11, opcode 10011



Notes:

Tensor memory load: 从 tensor memory 加载数据到 thread register。warp 内每个 lane 发出一个 load, 但一条 warp 的 load 只能访问当前 CTA tensor memory 的 1/4 — 需要整个 warpgroup (4 warp) 协作才能覆盖完整 tensor memory。

dst: thread register (vreg) 起始编号, 实际占用 reg 数量由 shape×num 联合决定 (1~128 个 vreg)。

src0: tensor memory 地址, 1 个 UR (32-bit), 编码 [31:16]=lane index, [15:0]=column index。

shape 选择 tile 布局 (16x64b / 16x128b / 16x256b / 32x32b / 16x32bx2); num 选择重复次数 (x1..x128)。

异步指令, 通过 depbar (req_sb) 等待完成。仅 CS 可用。128-bit 指令。

utcmma **category**: Matrix and tensor **predicate_mode**: uniform

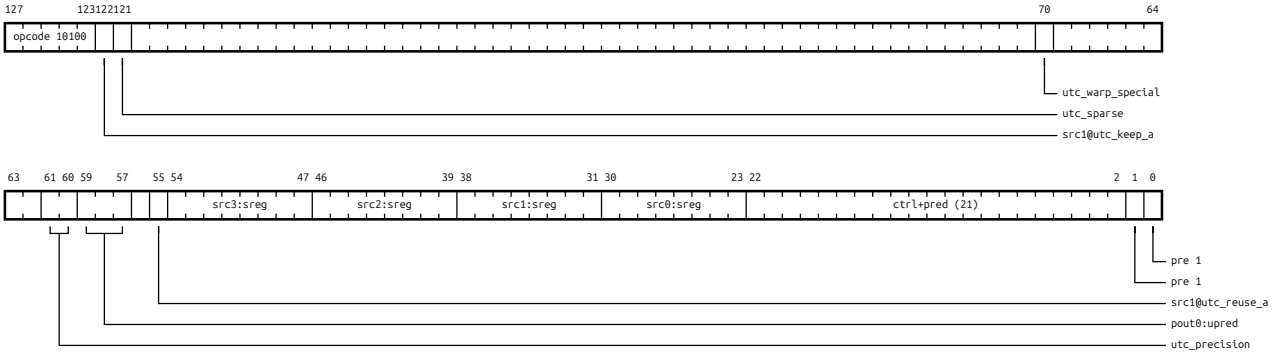
Leaf: utcmma__base

Format:

```
utcmma.utc_precision{.utc_warp_special}{.utc_sparse}  src0,
↪  src1{.utc_reuse_a}{.utc_keep_a}, src2, src3, pout0
```

Operands: src0: sreg; src1: sreg; src2: sreg; src3: sreg; pout0: upred

Encoding Diagram: 128b, prefix 11, opcode 10100



Notes:

Asynchronous warpgroup-level matrix multiply-accumulate through tensor core: D = A × B + C。单线程 issue 即可启动整个 tensor core 运算, 所有操作数通过 uniform register 传递 descriptor 或 tensor memory 地址, 不消耗 thread register。datapath: tensor core (decoupled, asynchronous)。

src0 (d_tmem): D 矩阵在 tensor memory 中的地址, 1 个 UR (32-bit)。tensor memory 是 SM 内部 on-chip scratchpad, 由 TC 直接读写, 不经过 register file。

src1 (ab_desc): A+B 矩阵描述符的 base UR, 占 4 个连续 UR (consecutive, 2-aligned)。UR[base] 和 UR[base+1] 编码 A 矩阵的数据位置 (tensor memory 地址或 shared memory descriptor); UR[base+2] 和 UR[base+3] 编码 B 矩阵位置。A 可来自 tensor memory 或 shared memory, B 可来自 shared memory (由 instruction descriptor 内 bit[0:1] 选择)。

src2 (idesc): instruction descriptor, 1 个 UR (32-bit)。编码 M/N 维度、A/B 数据类型 (atype/btype)、D 累加类型 (dtype)、transpose A/B、negate A/B。精度路径由 utc_precision modifier 在 opcode 级选择, 类型细分在 idesc 内编码。

pout0 (enable_d): uniform predicate output — 控制 D 矩阵是否参与累加 (true= 跳过 D 读取, 只写结果; false= 读 D 作为累加输入)。也用作异步完成信号。

src3 (e_tmem): 2:4 结构化稀疏 metadata 在 tensor memory 中的地址, 1 个 UR (32-bit)。utc_sparse=sp24 时由硬件稀疏选择器消费; utc_sparse=dense 时不参与执行, 填 URZ。

utc_precision 4 valid: f16 (bf16/f16/tf32 路径, idesc 区分) / int8 (INT8/INT4) / fp8 (E4M3/E5M2) / fp4 (E2M1)。各精度支持的 shape (M×N×K) 组合在 prose 里文档化, 不在 modifier 穷举。

utc_sparse 2 valid: dense (默认) / sp24 (2:4 结构化稀疏, A 压缩存储 + metadata 经 src3 供给, B 稠密, 等效 K 翻倍)。各精度与稀疏的合法组合在 prose 里文档化。

per-operand reuse/keep: utc_reuse_a 和 utc_keep_a 互斥 — reuse 用于 A 来自 descriptor 的模式, keep 用于 A 来自 tensor memory 的直接寻址模式。utc_reuse_b / utc_keep_b / utc_buffer 仅在 warp specialization (utc_warp_special=on) 下有效。

仅 compute shader (CS) 可用。128-bit 指令。

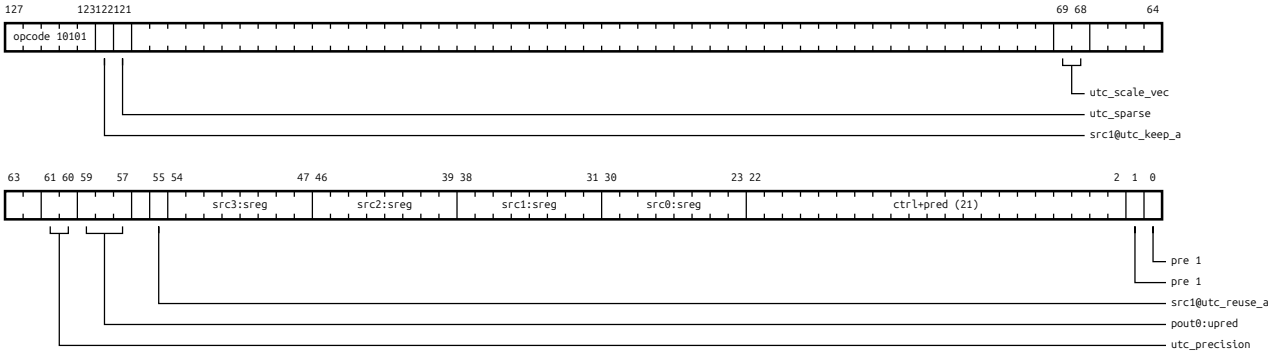
Leaf: utcmma__scale

Format:

```
utcmma.utc_precision{.utc_scale_vec}{.utc_sparse}  src0, src1{.utc_reuse_a}{.utc_keep_a},
↪  src2, src3, pout0
```

Operands: src0: sreg; src1: sreg; src2: sreg; src3: sreg; pout0: upred

Encoding Diagram: 128b, prefix 11, opcode 10101



Notes:

Asynchronous warpgroup-level MMA with per-block scale factors (块缩放, 覆盖 OCP MX 标准与 NVFP4 方案)。操作: D = scale(A, scaleA) × scale(B, scaleB) + C。与 utcmma__base 的区别: src1 从 4 个连续 UR 扩展到 6 个连续 UR, 额外 2 个 UR 编码 A-scale 和 B-scale 的 tensor memory 地址。

src0 (d_tmem): 同 utcmma__base, D 矩阵 tensor memory 地址, 1 UR。

src1 (ab_desc_scale): A+B 描述符 + scale 地址的 base UR, 占 6 个连续 UR (consecutive, 2-aligned)。UR[base:base+3] = A/B 矩阵描述符 (同 base 指令形态); UR[base+4:base+5] = A-scale 和 B-scale 的 tensor memory 地址。

src2 (idesc): 同 utcmma__base, instruction descriptor, 1 UR。

pout0 (enable_d): 同 utcmma__base。

src3 (e_tmem): 同 utcmma__base — 2:4 稀疏 metadata 的 tensor memory 地址; utc_sparse=dense 时填 URZ。稀疏与 scale 可组合 (稀疏 MX 路径), 合法组合在 prose 里文档化。

utc_scale_vec 选择块缩放方案, 一个取值同时定块粒度与因子格式, 三种合法组合: block32_ue8m0 (OCP MX 标准, MXFP8/6/4 通用) / block16_ue8m0 (更细的块, 因子仍为纯指数) / block16_ue4m3 (NVFP4 — 因子带尾数, 缩放量化误差更小)。完整语义见 utc_scale_vec modifier 定义。

utc_precision 在 scale 指令形态中通常为 fp8 或 fp4; f16 scale 指令形态用于 mixed-precision (A=bf16, B=FP8/FP4)。

无 warp specialization — scale 指令形态的 B 数据路径不经过 WS buffer。utc_reuse_a / utc_keep_a 仍有效。

仅 compute shader (CS) 可用。128-bit 指令。

utcst **category**: Matrix and tensor **predicate_mode**: uniform

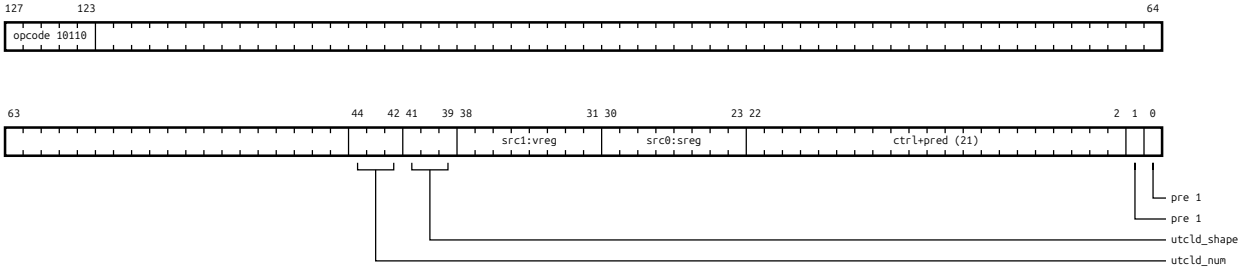
Leaf: utcst__s_v

Format:

```
utcst{.utcld_shape}{.utcld_num}  src0, src1
```

Operands: src0: sreg; src1: vreg

Encoding Diagram: 128b, prefix 11, opcode 10110



Notes:

Tensor memory store: 从 thread register 写入 tensor memory。warp 粒度和 tensor memory 访问范围同 utcld。

src0: tensor memory 地址, 1 个 UR (32-bit), 编码同 utcld。

src1: thread register (vreg) 起始编号, 实际 reg 占用由 shape×num 决定。

异步指令。仅 CS 可用。128-bit 指令。

14.7.17 System and resource control

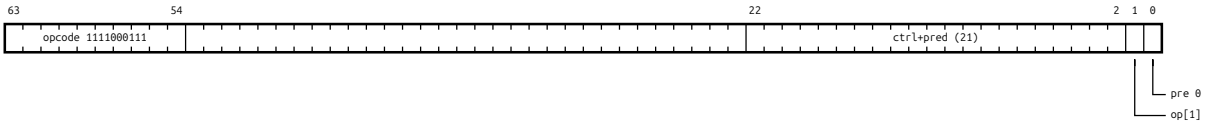
nop **category**: System and resource control **predicate_mode**: none

Leaf: nop___

Format:

```
nop
```

Encoding Diagram: 64b, prefix 0, opcode 11111000111



Notes:

No operation: 占一个指令 slot 让 PC 前进一步, 不做任何计算 / 内存 / 控制流副作用。

predicate_mode=none: nop 不参与 predicate gate (没有' 执行' 意义, 即使 gate=false 也是 nop)。无 modifier, 无 operand。

典型用例: loop align (跟 cache line / 编码 boundary 对齐) / 调度 padding (在 high-latency op 后插 nop 等待 latency) / reserved encoding (某些预留 slot, 写 nop 比 0 编码更安全避免 'invalid encoding' trap) / debug build 保留占位 (release 优化阶段消除)。

usetmaxreg **category**: System and resource control **predicate_mode**: uniform

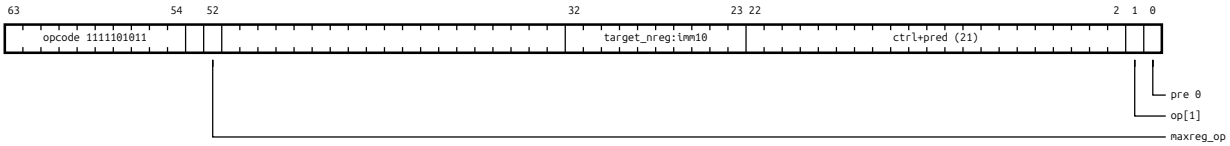
Leaf: usetmaxreg___i

Format:

```
usetmaxreg{.maxreg_op}  target_nreg
```

Operands: target_nreg: imm10u

Encoding Diagram: 64b, prefix 0, opcode 11111101011



Notes:

Warp specialization register reallocation: 申请 (try_alloc) / 释放 (dealloc) 当前 warpgroup 的目标物理寄存器数量。允许同一 cta 内不同 warpgroup 配置不同数量的物理寄存器 — Producer warpgroup (做异步搬运) 用 dealloc 释放寄存器, Consumer warpgroup (做 MMA 计算) 用 try_alloc 接收寄存器, occupancy 不变但 consumer 可拥有更多 reg。

target_nreg 语义: 表达目标 maximum register count (不是 delta) — try_alloc 256 表示 'try to set my warpgroup max to 256 registers' (不是 allocate 256 more), dealloc 24 表示 'set my warpgroup max to 24' (不是 release 24 more)。

value 限制: target_nreg 必须 4 的倍数; 范围由硬件实现决定 (典型 24-256, 走 prose 文档化, spec 不在编码层 enforce)。

maxreg_op: try_alloc (返 success predicate, 失败时由软件 retry; 用 p_i / p_x 指令形态) / dealloc (立即成功无 pout0; 用 _i / _x 指令形态)。

典型用例: 异步数据搬运流水线 kernel — kernel 启动时 producer warpgroup 立即 usetmaxreg.dealloc 24 (释放到 24 reg), consumer warpgroup 跑 usetmaxreg.try_alloc P0, 256; @!P0 bra retry; (try 256 直到成功) 再做 GEMM。

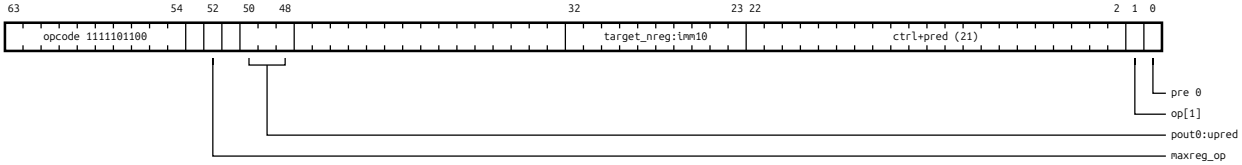
Leaf: usetmaxreg__p_i

Format:

```
usetmaxreg{.maxreg_op}  pout0, target_nreg
```

Operands: pout0: upred; target_nreg: imm10u

Encoding Diagram: 64b, prefix 0, opcode 11111101100



Notes:

Warp specialization register reallocation: 申请 (try_alloc) / 释放 (dealloc) 当前 warpgroup 的目标物理寄存器数量。允许同一 cta 内不同 warpgroup 配置不同数量的物理寄存器 — Producer warpgroup (做异步搬运) 用 dealloc 释放寄存器, Consumer warpgroup (做 MMA 计算) 用 try_alloc 接收寄存器, occupancy 不变但 consumer 可拥有更多 reg。

target_nreg 语义: 表达目标 maximum register count (不是 delta) — try_alloc 256 表示 'try to set my warpgroup max to 256 registers' (不是 allocate 256 more), dealloc 24 表示 'set my warpgroup max to 24' (不是 release 24 more)。

value 限制: target_nreg 必须 4 的倍数; 范围由硬件实现决定 (典型 24-256, 走 prose 文档化, spec 不在编码层 enforce)。

maxreg_op: try_alloc (返 success predicate, 失败时由软件 retry; 用 p_i / p_x 指令形态) / dealloc (立即成功无 pout0; 用 _i / _x 指令形态)。

典型用例: 异步数据搬运流水线 kernel — kernel 启动时 producer warpgroup 立即 usetmaxreg.dealloc 24 (释放到 24 reg), consumer warpgroup 跑 usetmaxreg.try_alloc P0, 256; @!P0 bra retry; (try 256 直到成功) 再做 GEMM。

usetshmsz **category**: System and resource control **predicate_mode**: uniform

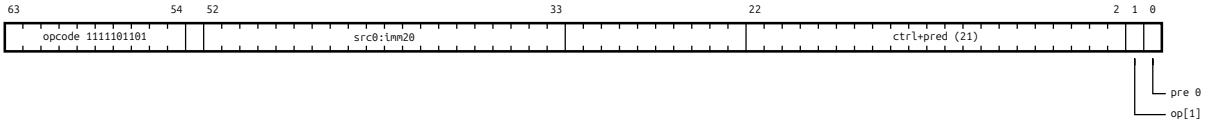
Leaf: usetshmsz___i

Format:

```
usetshmsz  src0
```

Operands: src0: imm20u

Encoding Diagram: 64b, prefix 0, opcode 11111101101



Notes:

Dynamic shared memory configuration: 调整当前 cta 的 shared memory 容量 (size_bytes 指令形态) 或失效全部 shared memory (u flush 指令形态)。shared memory 跟 l1 cache 共享同一物理 SRAM, 调整 shm 大小会改变 l1 cache 可用容量 (gpu configurable shared / l1 split)。

指令形态 3 种: _i (imm.20 size_bytes, 静态调 size, max ~1MB) / _x (sreg size_bytes, 动态调) / u (flush 模式, 失效所有 shared memory 内容, 不改变 size)。

调整后数据保留行为: 调整完成后, 之前 shared memory 数据不保证保留 (硬件可能重新分配 SRAM partition)。flush 指令形态显式失效内容跟 size 调整分开, 避免误用。

同步约束: 调整 SRAM partition / flush 前必须保证 pending shared / 异步搬运操作完成, 否则 behavior unknown。典型序列: depbar SB_shared, 0 → depbar SB_async, 0 → bar_sync 0 → usetshmsz → bar_sync 0。

典型用例: kernel 主循环切换阶段动态调 shared 容量 (例如 prologue 大 shared 给 weight tile, epilogue 小 shared 释放给 l1 cache 加速 reduce) / cluster 配置切换。

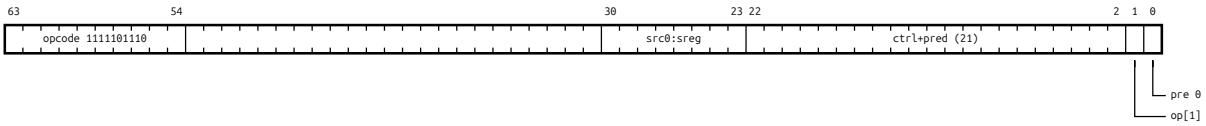
Leaf: usetshmsz___x

Format:

```
usetshmsz  src0
```

Operands: src0: sreg

Encoding Diagram: 64b, prefix 0, opcode 11111101110



Notes:

Dynamic shared memory configuration: 调整当前 cta 的 shared memory 容量 (size_bytes 指令形态) 或失效全部 shared memory (u flush 指令形态)。shared memory 跟 l1 cache 共享同一物理 SRAM, 调整 shm 大小会改变 l1 cache 可用容量 (gpu configurable shared / l1 split)。

指令形态 3 种: _i (imm.20 size_bytes, 静态调 size, max ~1MB) / _x (sreg size_bytes, 动态调) / u (flush 模式, 失效所有 shared memory 内容, 不改变 size)。

调整后数据保留行为: 调整完成后, 之前 shared memory 数据不保证保留 (硬件可能重新分配 SRAM partition)。flush 指令形态显式失效内容跟 size 调整分开, 避免误用。

同步约束: 调整 SRAM partition / flush 前必须保证 pending shared / 异步搬运操作完成, 否则 behavior unknown。典型序列: depbar SB_shared, 0 → depbar SB_async, 0 → bar_sync 0 → usetshmsz → bar_sync 0。

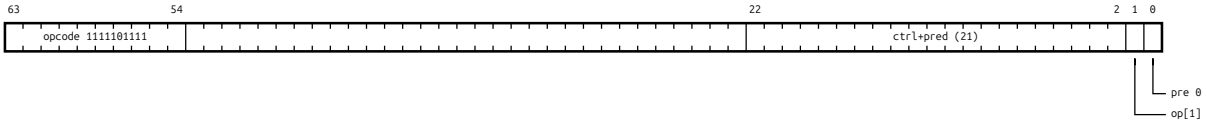
典型用例: kernel 主循环切换阶段动态调 shared 容量 (例如 prologue 大 shared 给 weight tile, epilogue 小 shared 释放给 l1 cache 加速 reduce) / cluster 配置切换。

Leaf: usetshmsz__u

Format:

```
usetshmsz
```

Encoding Diagram: 64b, prefix 0, opcode 11111101111



Notes:

Dynamic shared memory configuration: 调整当前 cta 的 shared memory 容量 (size_bytes 指令形态) 或失效全部 shared memory (u flush 指令形态)。shared memory 跟 l1 cache 共享同一物理 SRAM, 调整 shm 大小会改变 l1 cache 可用容量 (gpu configurable shared / l1 split)。

指令形态 3 种: _i (imm.20 size_bytes, 静态调 size, max ~1MB) / _x (sreg size_bytes, 动态调) / u (flush 模式, 失效所有 shared memory 内容, 不改变 size)。

调整后数据保留行为: 调整完成后, 之前 shared memory 数据不保证保留 (硬件可能重新分配 SRAM partition)。flush 指令形态显式失效内容跟 size 调整分开, 避免误用。

同步约束: 调整 SRAM partition / flush 前必须保证 pending shared / 异步搬运操作完成, 否则 behavior unknown。典型序列: depbar SB_shared, 0 → depbar SB_async, 0 → bar_sync 0 → usetshmsz → bar_sync 0。

典型用例: kernel 主循环切换阶段动态调 shared 容量 (例如 prologue 大 shared 给 weight tile, epilogue 小 shared 释放给 l1 cache 加速 reduce) / cluster 配置切换。

A　编程模型映射（资料性附录）

本附录给出本架构与 OpenCL 任务执行模型的概念映射，便于面向本架构生成可移植代码。映射不构成规范性条款；具体软件 ABI、运行时实现、调用约定由实现确定。

A.1　任务粒度对应

本架构概念	OpenCL 概念	说明
kernel	kernel	设备侧并行计算入口函数
grid	NDRange	kernel 的整体执行范围，含若干 work-group
CTA	work-group	共享存储与同步的协作单元，由若干 warp 组成
warp	sub-group	硬件锁步执行的 32 个 lane 组成的执行单元
lane / thread	work-item	单个执行车道，对应 OpenCL 单个工作项

A.2　地址空间对应

本架构地址空间	OpenCL 地址空间	address_space_qualifier
global	global memory	__global
shared	local memory	__local
local (per-thread private)	private memory	__private

本架构地址空间	OpenCL 地址空间	address_space_qualifier
constant	constant memory	__constant
generic	generic address space	(OpenCL 2.0+ generic pointers)

A.3 同步原语对应

本架构原语	OpenCL 原语	说明
bar_sync (CTA-wide)	barrier(CLK_LOCAL_MEM_-FENCE)	work-group 范围栅障
fence	mem_fence(...)	内存屏障
vote / match	sub_group_* 原语	warp / sub-group 范围 ballot / collective
atom*	atomic_*	原子读改写

A.4 寄存器与执行状态

OpenCL 不直接暴露寄存器和 EXEC 掩码，本架构在此层面更精细：

- vreg 对应每个 work-item 的私有变量在编译期分配的寄存器；
- sreg 对应 work-group / sub-group 范围的统一标量；
- EXEC 对应 OpenCL 中由 if / 控制流生成的发散收敛逻辑（编译器在底层映射）；
- special register (如 TID_X / CTAID_X / LANEID) 对应 OpenCL 内置函数 get_local_id / get_group_id / get_sub_group_local_id。