简介

本文档从指令集架构及软硬件接口角度描述了乘影GPGPU的设计内容。
乘影GPGPU指令集以RISC-V向量扩展（后文简称为RVV）为核心设计GPGPU，相比RISC-V标量指令，具有更丰富的表达含义，可以实现访存特性表征、区分workgroup和thread操作等功能。核心思想是在编译器层面以v指令作为thread的行为描述，并将thread->warp/workgroup的公共数据合并为标量指令。硬件上一个warp就是一个RVV程序，通常向量元素长度为num_thread，同时又将workgroup中统一执行的公共地址计算、跳转等作为标量指令执行，即Vector-Thread架构。硬件将warp分时映射到RVV处理器的lane上去执行。
乘影SIMT架构计算单元采用SIMD（Vector）执行方式，以workgroup（或分支状态下的warp_split）执行。RVV指令集在变长上有三个方面的体现：硬件vlen改变；SEW元素宽度改变；LMUL分组改变。本架构特点在于这三个参数在编译期都已固定，元素数目大部分情况也固定为num_thread。

术语表

- SM：streaming multiprocessor，流多处理器单元
- sGPR：scalar general purpose register，标量寄存器
- vGPR：vector general purpose register，向量寄存器

GPGPU中常用概念及释义如下表。

cuda	opencl	解释
globalmem	globalmem	全局内存，用__global描述，可以被kernel的所有线程访问到
constantmem	constantmem	常量内存，用__constant描述，是全局地址空间的一部分
localmem	privatemem	私有内存，各thread自己的变量，和内核参数，是全局内存的一部分
sharedmem	localmem	局部内存，用__local描述，供同一work-group间的线程进行数据交换
grid	NDRange	一个kernel由多个NDRange组成，一个NDRange由多个workgroup组成
block/CTA	workgroup	工作组，在SM上执行的基本单位
warp	wavefront(AMD)	32个thread组成一个warp，仅对硬件可见
thread	work-item	线程/工作项，是OpenCL C编程时描述的最小单位。

编程模型和驱动程序功能

这部分从乘影作为device和OpenCL编程框架的交互来介绍OpenCL程序在执行前后的行为。乘影的编程模型兼容OpenCL，即乘影的硬件层次与OpenCL的执行模型具有一一对应的关系。此外，OpenCL编程框架需要提供一系列运行时实现，完成包括任务分配，设备内存空间管理，命令队列管理等功能。

任务执行模型

在OpenCL中，计算平台分为主机端（host）和设备端（device），其中device端执行内核（kernel），host端负责交互，资源分配和设备管理等。Device上的计算资源可以划分为计算单元（computing units，CU），CU可以进一步划分为处理单元（processing elements，PE），设备上的计算都在PE中完成。
乘影的一个SM对应OpenCL的一个计算单元CU，每个向量车道（lane）对应OpenCL的一个处理单元PE。
Kernel在设备上执行时，会存在多个实例（可以理解为多线程程序中的每个线程），每个实例称为一个工作项（work-item），work-item会组织为工作组（work-group），work-group中的work-item可以并行执行，要说明的是，OpenCL只保证了单个work-group中的work-item可以并发执行，并没有保证work-group之间可以并发执行，尽管实际情况中通常work-group是并行的。
在乘影的实现中，一个work-item对应一个线程，在执行时将会占用一个向量车道，由于硬件线程会被组织成线程组（warp）锁步执行，而warp中的线程数量是固定的，因此一个work-group在映射到硬件时可能对应多个warp，但这些组成一个work-group的warp将会保证在同一个SM上执行。

驱动提供的功能

乘影的驱动分为两层，一层是OpenCL的运行时环境，一层是硬件驱动，运行时环境基于OpenCL的一个开源实现pocl实现，主要管理命令队列（command queue），创建和管理buffer，管理OpenCL事件（events）和同步。硬件驱动是对物理设备的一层封装，主要是为运行时环境提供一些底层接口，并将软件数据结构转换为硬件的端口信号，这些底层接口包括分配、释放、读写设备内存，在host和device端进行buffer的搬移，控制device开始执行等。运行时环境利用硬件驱动的接口实现和物理设备的交互。
每个kernel都有一些需要硬件获取的信息，这些信息由运行时创建，以buffer的形式拷贝到device端的内存空间。目前运行时环境创建的buffer包括：

- NDRange的metadata buffer和kernel机器码
- kernel的argument data buffer，即每个kernel参数的具体输入
- kernel_arg_buffer，保存的内容为device端的指针，其值为kernel的argument data buffer在device端的地址。
- 为private mem、print buffer分配的空间

其中，metadata buffer保存kernel的一些属性，具体内容为：

```
cl_int clEnqueueNDRangeKernel(cl_command_queue command_queue,
                              cl_kernel kernel,                                  //kernel_entry_ptr & kernel_arg_ptr
                              cl_uint work_dim,                                  //work_dim
                              const size_t *global_work_offset, //global_work_offset_x/y/z
                              const size_t *global_work_size,   //global_work_size_x/y/z
                              const size_t *local_work_size,    //local_work_size_x/y/z
                              cl_uint num_events_in_wait_list,
                              const cl_event *event_wait_list,
                              cl_event *event)
/*
#define KNL_ENTRY 0
#define KNL_ARG_BASE 4
#define KNL_WORK_DIM 8
#define KNL_GL_SIZE_X 12
#define KNL_GL_SIZE_Y 16
#define KNL_GL_SIZE_Z 20
#define KNL_LC_SIZE_X 24
#define KNL_LC_SIZE_Y 28
#define KNL_LC_SIZE_Z 32
#define KNL_GL_OFFSET_X 36
#define KNL_GL_OFFSET_Y 40
#define KNL_GL_OFFSET_Z 44
#define KNL_PRINT_ADDR 48
#define KNL_PRINT_SIZE 52
*/
```

Kernel启动时的行为

任务启动时，硬件驱动需要传递给物理device一些信号，这些信号有：

信号	含义
PTBR	page table base addr
CSR_KNL	metadata buffer base addr
CSR_WGID	当前workgroup在SM中的id，仅供硬件辨识
CSR_WID	warp id，当前warp属于workgroup中的位置
LDS_SIZE	localmem_size，编译器提供workgroup需要占用的localmem空间。 privatemem_size默认按照每个线程1kB来分配。
VGPR_SIZE	vGPR_usage，编译器提供workgroup实际使用的vGPR数目（对齐4）
SGPR_SIZE	sGPR_usage，编译器提供workgroup实际使用的sGPR数目（对齐4）
CSR_GIDX/Y/Z	workgroup idx in NDRange
host_wf_size	一个warp中thread数目
host_num_wf	一个workgroup中warp数目

kernel的参数由前述的kernel_arg_buffer传递，该buffer中会按顺序准备好kernel的argument，包括具体参数值或其它buffer的地址。在NDRange的metadata中仅提供kernel_arg_buffer的地址knl_arg_base。 kernel函数执行前会先执行start.S：

```
# start.S
start:
        csrr sp, CSR_LDS # set localmemory pointer
        addi tp, x0, 0 # set privatememory pointer

        # clear BSS segment
        #
        # clear BSS complete

        csrr t0, CSR_KNL
        lw t1, KNL_ENTRY(t0)
        lw a0, KNL_ARG_BASE(t0)
        jalr t1
# end.S
end:
        endprg
```

约定kernel的打印信息通过print buffer向host传递。print buffer的地址和大小在metadata_buffer中提供，运行中的thread完成打印后，将所属warp的CSR_PRINT置位。host轮询到有未处理信息时，将print buffer从设备侧取出处理，并将CSR_PRINT复位。

栈空间说明

由于OpenCL不允许在Kernel中使用malloc等动态内存函数，也不存在堆，因此可以让栈空间向上增长。tp用于各thread私有寄存器不足时压栈（即vGPR spill stack slots），sp用于公共数据压栈，（即sGPR spill stack slots，实际上sGPR spill stack slots将作为localmem的一部分），在编程中显式声明了__local标签的数据也会存在localmem中。 编译器提供localmem的数据整体使用量（按照sGPR spill 1kB，结合local数据的大小，共同作为localmem_size），供硬件完成workgroup的分配。

参数传递ABI

对于kernel函数，a0是参数列表的基址指针，第一个clSetKernelArg设置的显存起始地址存入a0 register，kernel默认从该位置开始加载参数。 对于非kernel函数，使用v0-v31和stack pointer传递参数，v0-v31作为返回值。

寄存器

寄存器设置

架构寄存器数目为sGPR（标量）64个，vGPR（向量）256个，元素宽度均为32位。
当需要64位数据时，数据以偶对齐的寄存器对（Register Pair）的形式进行存储，数据低32位存储在GPR[n]中，高32位存储在GPR[n+1]中。
目前硬件中物理寄存器数目为sGPR（标量）256个，vGPR（向量）1024个，GPU硬件负责实现架构寄存器到硬件寄存器的映射。
编译器提供vGPR、sGPR的实际使用数目（4的倍数），硬件会根据实际使用情况分配更多的workgroup同时调度。
从RISC-V Vector的视角来看寄存器堆，vGPR共有256个，宽度vlen固定为线程数目num_thread乘以32位，即相当于通过vsetvli指令设置SEW=32bit，ma，ta，LMUL为1。而从SIMT的视角来看寄存器堆，每个thread至多拥有256个宽度32位的vGPR。一个简单的理解是，将向量寄存器堆视作一个256行、num_thread列的二维数组，数组的每行是一个vGPR，而每列是一个thread最多可用的寄存器。OpenCL中定义了一些向量类型，这些向量类型需要使用分组寄存器的形式表达，即float16在寄存器堆中以列存储，占用16个vGPR的各32位。这部分工作由编译器进行展开。
workgroup拥有64个sGPR，整个workgroup只需做一次的操作，如kernel中的地址计算，会使用sGPR；如果有发生分支的情况，则使用vGPR，例如非kernel函数的参数传递。

一些特殊寄存器：

- x0：0寄存器；
- x1/ra：返回PC寄存器；
- x2/sp：栈指针 / local mem基址；
- x4/tp：private mem基址。

参数传递：
对于kernel函数，a0是参数列表的基址指针，第一个clSetKernelArg设置的显存起始地址存入a0 register，kernel默认从该位置开始加载参数。

自定义CSR

description	name	addr
该warp中id最小的thread id，其值为CSR_WID*CSR_NUMT，配合vid.v可计算其它thread id。	CSR_TID	0x800
该workgroup中的warp总数	CSR_NUMW	0x801
一个warp中的thread总数	CSR_NUMT	0x802
该workgroup的metadata buffer的baseaddr	CSR_KNL	0x803
该SM中本warp对应的workgroup id	CSR_WGID	0x804
该workgroup中本warp对应的warp id	CSR_WID	0x805
该workgroup分配的local memory的baseaddr，同时也是该warp的xgpr spill stack基址	CSR_LDS	0x806
该workgroup分配的private memory的baseaddr，同时是该thread的vgpr spill stack基址	CSR_PDS	0x807

description	name	addr
该workgroup在NDRange中的x id	CSR_GIDX	0x808
该workgroup在NDRange中的y id	CSR_GIDY	0x809
该workgroup在NDRange中的z id	CSR_GIDZ	0x80a
向print buffer打印时用于与host交互的CSR	CSR_PRINT	0x80b
重汇聚pc寄存器	CSR_RPC	0x80c

注：在汇编器中可以使用小写后缀来表示对应的CSR，例如用tid代替CSR_TID。

指令集架构

指令集范围

选用RV32V扩展作为基本指令集，支持指令集范围为：RV32I, M, A, zfinx, zve32f, RV64I, A.
V扩展指令集主要支持独立数据通路的指令，不支持RVV原有的shuffle, narrow, gather, reduction指令。
下表列出目前支持的标准指令范围，有变化指令已声明。

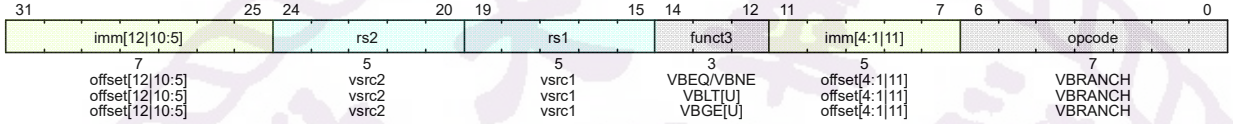
	乘影支持情况	指令变化
RV32I	不支持ecall ebreak	
RV32M F	支持RV32M zfinx zve32f	
RV32A	支持	
RV64I	部分支持	语义有变化，见后续章节
RV64A	部分支持	语义有变化，见后续章节
RV32V-Register State	仅支持LMUL=1和2	
RV32V-ConfigureSetting	支持计算vl，可通过该选项配置支持不同宽度元素	
RV32V-LoadsAndStores	支持vle32.v vlse32.v vluxei32.v访存模式	vle8等指令语义改为 “各thread向向量寄存器元素位置写入“， 而非连续写入
RV32V-IntergerArithmetic	支持绝大多数int32计算指令	vmv.x.s语义改为 “各thread均向标量寄存器写入”， 而非总由向量寄存器idx_0写入， 多线程同时写入是未定义行为， 正确性由程序员保证； vmv.s.x语义改为与vmv.v.x一致
RV32V-FixedPointArithmetic	添加int8支持，视应用需求再添加其它类型	
RV32V-FloatingPointArithmetic	支持绝大多数fp32指令，添加fp64 fp16支持	
RV32V-WideningIntergerArithmetic	支持绝大多数计算指令	2*SEW位宽元素通过寄存器对实现
RV32V-ReductionOperations	视应用和编译器需求再考虑添加， 例如需要支持OpenCL2.0中的work_group_reduce时	
RV32V-Mask	支持各lane独立计算和设置mask的指令	vmsle等指令语义改为 “各thread向向量寄存器元素位置写入”， 而非连续写入
RV32V-Permutation	不支持，视应用和编译器需求再考虑添加	

	乘影支持情况	指令变化
RV32V-ExceptionHandling	不支持，视应用和编译器需求再考虑添加	

自定义指令

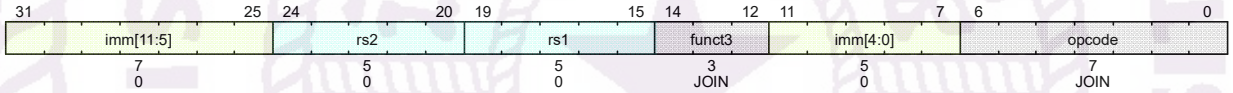
分支控制指令

VBRANCH分支指令使用B型指令格式。12位B型有符号立即数进行符号位扩展后加上当前的PC值给出else路径的起始位置PC，读取CSR_RPC的值rpc，根据向量寄存器计算比较结果操作SIMT线程分支管理栈。



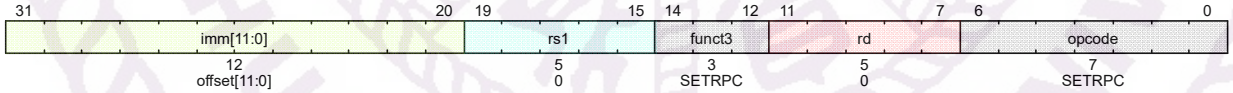
线程分支指令将比较两个向量寄存器，如果线程对应的元素相等，VBEQ的这些线程将跳转到else PC处执行，其余元素不等的线程将继续执行PC+4，分支两条路径的跳转、聚合和掩码控制由SIMT栈控制，将rpc，else PC和元素比较结果压入栈中，当全部活跃线程对应元素均相等或不等时，不触发栈操作，全部跳转到else PC或继续执行PC+4。当vs1和vs2中的操作数不等时，VBNE将判断对应线程跳转执行PC else还是PC+4。VBLT和VBLTU将分别使用有符号数和无符号数比较vs1和vs2，当对应的向量元素有vs1小于vs2时，对应线程跳转至PC else，剩余活跃线程继续执行PC+4。VBGE和VBGEU将分别使用有符号数和无符号数比较vs1和vs2，当对应的向量元素有vs1大于等于vs2时，对应线程跳转至PC else，剩余活跃线程继续执行PC+4。

线程分支汇聚join指令采用S型指令格式，默认源操作数、立即数均为0。



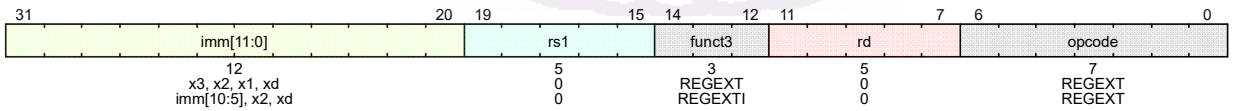
JOIN指令将对比当前指令PC与SIMT栈顶重汇聚PC是否相等，若相等，则设置活跃线程掩码为当前栈顶掩码项，跳转至栈顶else PC处执行指令，若不等，则不做任何操作。

重汇聚PC设置指令SETRPC将12位立即数进行符号位扩展并与源操作数相加后，将结果写入CSR_RPC的csr寄存器和目的寄存器。



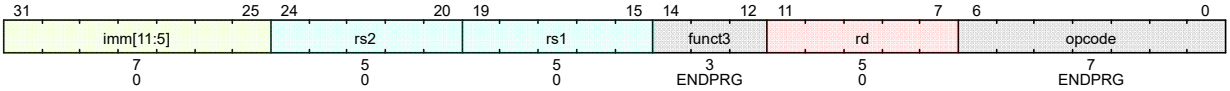
寄存器扩展指令

REGEXT和REGEXTI指令用于扩展它之后一条指令的寄存器编码及立即数编码。在REGEXT指令中，12位的立即数被分拆为四段3位数，分别拼接在下一条指令中寄存器rs3/vs3、rs2/vs2、rs1/vs1、rd/vd编码的高位上。REGEXTI指令则面向使用了5位立即数的指令，前缀包含6位的立即数高位，以及rs2/vs2、rd/vd编码的高位。11立即数由前缀指令中的立即数高位与5位立即数直接拼接得到。
此外，在不使用扩展的情况下，向量浮点运算的vs3就是vd的[11:7]段，但进行向量扩展时，vs3和vd的高3位分离存储。

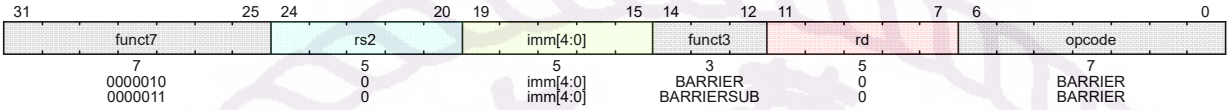


同步和任务控制指令

ENDPRG指令需要显式插入到Kernel末尾，以指示当前warp执行结束。只能在无分支的条件下使用。



BARRIER对应OpenCL的barrier(cl_mem_fence_flags flags)和work_group_barrier(cl_mem_fence_flags flags, [memory_scope scope])函数，实现同一workgroup内的thread间数据同步。memory_scope缺省值为 memory_scope_work_group。

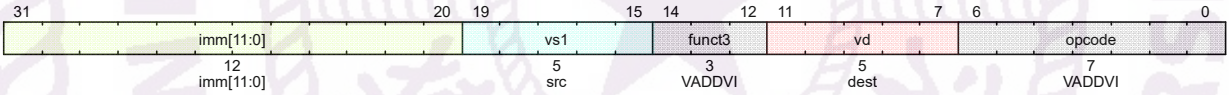


五位立即数字段编码如下：

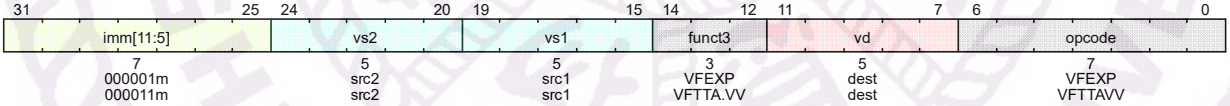
imm[4:3]	00	01	10	11
memory_scope	work_group(default)	work_item	device	all_svm_devices
imm[2:0]	**imm[2]=1**	**imm[1]=1**	**imm[0]=1**	**imm[2:0]=000**
CLK_X_MEM_FENCE	IMAGE	GLOBAL	LOCAL	USE_CNT

开启 opencl_c_subgroups 特性后，则改为barriersub指令，对应memory_scope=subgroup的情况，此时imm[4:3]固定为0，cl_mem_fence_flags为imm[2:0]，与barrier指令一致。

自定义计算指令

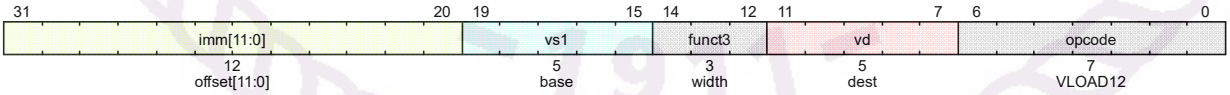


VADD12.VI将无符号数imm加到向量寄存器vs1赋予向量寄存器vd。

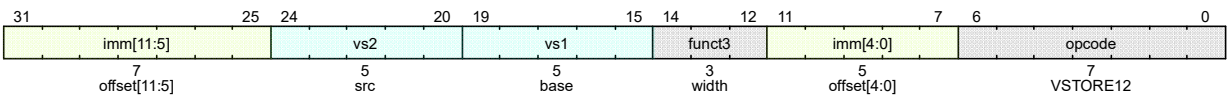


VFEXP计算exp(v2)赋予向量寄存器vd。VFTTAVV计算向量vs1和vs2的卷积，加上向量寄存器vd后赋予vd。

自定义立即数访存指令

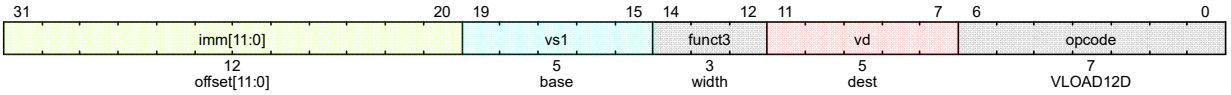


与标准RISC-V指令一致，自定义访存指令的地址空间按字节寻址且为小端格式。每个线程有效字节地址Addr=vs1+offset，将内存中以Addr为起始的元素复制到向量寄存器vd。VLW12指令从内存中加载32位值的向量到vd中。VLH12从内存中加载16位值，然后将其符号扩展到32位，再存储到vd中。VLHU12从内存中加载16位值，然后将其无符号扩展到32位，再存储到vd中。VLB12和VLBU12对8位值有类似定义。

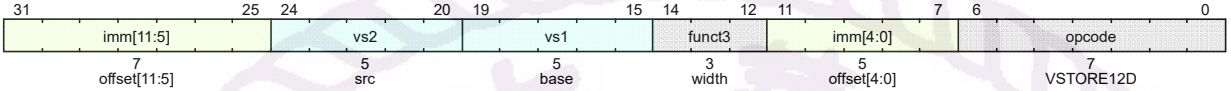


每个线程Addr=vs1+offset，VSW12、VSH12和VSB12指令分别将向量寄存器vs2的低位中的32位、16位和8位值的向量长度个数据存入以Addr起始的内存空间。

自定义64位地址空间立即数访存指令



自定义64位访存指令的地址空间按字节寻址且为小端格式，vs1表示偶对齐的寄存器对。每个线程有效字节地址addr=[vs1+1:vs1]+offset，将内存中以addr为起始的元素复制到向量寄存器vd。VLW12D指令从内存中加载32位值的向量到vd中。VLH12D从内存中加载16位值，然后将其符号扩展到32位，再存储到vd中。VLHU12D从内存中加载16位值，然后将其无符号扩展到32位，再存储到vd中。VLB12和VLBU12对8位值有类似定义。



每个线程addr=[vs1+1:vs1]+offset，VSW12D、VSH12D和VSB12D指令分别将向量寄存器vs2的低位中的32位、16位和8位值的向量长度个数据存入以addr起始的内存空间。

自定义访存前缀指令

PREFIX_MEMORY系列指令将作为前缀指令指示后续访存指令的访存空间、缓存策略、读取数据宽度，若pair字段为0，则地址位宽（32/64）将根据其接续访存指令所指定的地址位宽进行访存，若为1，则按照64位地址空间访存，若其后一条指令非访存指令，则当前指令失效。

高12位的立即数被分拆为四段3位数，分别拼接在下一条指令中寄存器rs3/vs3、rs2/vs2、rs1/vs1、rd/vd编码的高位上。

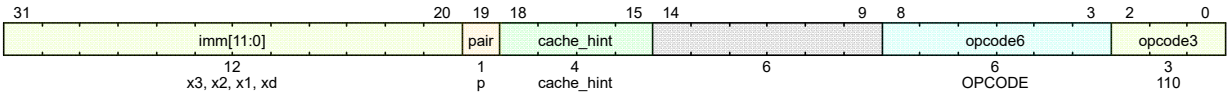
PREFIX_MEMORY指令指示的地址空间及映射关系如下：

地址空间	地址计算	编码
private	addr=(vs1+imm)*num_thread_in_wg+thread_idx+csr_pds 或者 addr=([vs1:vs1+1]+imm)*num_thread_in_wg+thread_idx+csr_pds	2
local	addr=(vs1+imm)+csr_lds 或者 addr=([vs1:vs1+1]+imm)+csr_lds	3
global	addr=(vs1+imm)+csr_gds 或者 addr=([vs1:vs1+1]+imm)+csr_gds	1
default	根据访存指令定义进行地址计算	0

OPCODE4字段高2bit将作为地址空间编码，低2bit作为数据宽度编码，若存取数据宽度大于32bit（1word），则从连续的地址空间读取数据存入相邻寄存器，或从相邻寄存器读出数据存入连续地址空间。

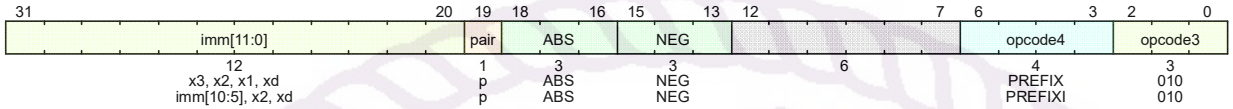
数据宽度（bit）	编码
32	0
64	1
96	2
128	3

指令的cachehint字段将作为硬件缓存策略指导，但不一定生效。



自定义计算前缀指令

PREFIX和PREFIXI指令用于扩展它之后一条指令的寄存器编码及立即数编码，并对下一条指令的计算结果进行modified。在PREFIX指令中，12位的立即数被分拆为四段3位数，分别拼接在下一条指令中寄存器rs3/vs3、rs2/vs2、rs1/vs1、rd/vd编码的高位上。PREFIXI指令则面向使用了5位立即数的指令，前缀包含6位的立即数高位，以及rs2/vs2、rd/vd编码的高位。11立即数由前缀指令中的立即数高位与5位立即数直接拼接得到。
此外，在不使用扩展的情况下，向量浮点运算的vs3就是vd的[11:7]段，但进行向量扩展时，vs3和vd的高3位分离存储。



字段	含义
pair	下一条指令的是否采用寄存器对拼接的64位数据，0-不采用，1-采用
ABS	下一条指令的输入数据是否取绝对值，比特0指示src1，比特1指示src2，比特2指示src3，0-不取绝对值，1-取绝对值
NEG	下一条指令的输入数据是否取反，比特0指示src1，比特1指示src2，比特2指示src3，0-不取反，1-取反

RV64I

RV64I相较于RV32I新增的指令及语义总结如下

指令	语义
SLLW	源操作数与目的操作数为相邻寄存器对拼接产生的64比特数据，寄存器对拼接偶对齐
SLLW	源操作数与目的操作数为相邻寄存器对拼接产生的64比特数据，寄存器对拼接偶对齐
SRLW	源操作数与目的操作数为相邻寄存器对拼接产生的64比特数据，寄存器对拼接偶对齐
SRAIW	源操作数与目的操作数为相邻寄存器对拼接产生的64比特数据，寄存器对拼接偶对齐
SRAW	源操作数与目的操作数为相邻寄存器对拼接产生的64比特数据，寄存器对拼接偶对齐
SRAIW	源操作数与目的操作数为相邻寄存器对拼接产生的64比特数据，寄存器对拼接偶对齐
ADDW	源操作数与目的操作数为相邻寄存器对拼接产生的64比特数据，寄存器对拼接偶对齐
ADDIW	源操作数与目的操作数为相邻寄存器对拼接产生的64比特数据，寄存器对拼接偶对齐
SUBW	源操作数与目的操作数为相邻寄存器对拼接产生的64比特数据，寄存器对拼接偶对齐
LD	源操作数为相邻寄存器对拼接产生的64比特数据，寄存器对拼接偶对齐， 从64位地址空间取出32位数据存入目的寄存器
SD	地址操作数rs1为相邻寄存器对拼接产生的64比特数据，寄存器对拼接偶对齐，将rs2中32位数据存入64位地址空间

RV64A

RV64A相较于RV32A新增的指令及语义总结如下：

指令集	语义
RV64A	地址操作数rs1为相邻寄存器对拼接产生的64比特数据，寄存器对拼接偶对齐，rs2及rd为32比特数据

RVV widening

指令范围	语义	是否与RVV定义一致
vwop.vv	源操作数为32比特向量元素，目的操作数为64比特向量元素，64 = 32 op 32	是
vwop.vx	源操作数分别为32比特向量元素和32比特标量元素，目的操作数为64比特向量元素，64 = 32 op 32	是
vwop.wv	源操作数均为64比特向量元素，目的操作数为64比特向量元素，64 = 64 op 64	否
vwop.wx	源操作数分别为64比特向量元素和32比特标量元素，目的操作数为64比特向量元素，64 = 64 op 64	否

总结

自定义扩展指令总结如下：

自定义扩展指令						
imm[12|10:5]	vs2	vs1	000	imm[4:1|11]	1011011	VBEQ
imm[12|10:5]	vs2	vs1	001	imm[4:1|11]	1011011	VBNE
imm[12|10:5]	vs2	vs1	v00	imm[4:1|11]	1011011	VBLT
imm[12|10:5]	vs2	vs1	101	imm[4:1|11]	1011011	VBGE
imm[12|10:5]	vs2	vs1	110	imm[4:1|11]	1011011	VBLTU
imm[12|10:5]	vs2	vs1	111	imm[4:1|11]	1011011	VBGEU
0000000	00000	00000	010	00000	1011011	JOIN
imm[11:0]		rs1	011	rd	1011011	SETRPC
imm[11:0](x3,x2,x1,xd)		00000	010	00000	0001011	REGEXT
imm[11:0](imm[10:5],x2,xd)		00000	011	00000	0001011	REGEXTI
0000000	rs2	rs1	100	rd	0001011	ENDPRG
0000010	rs2	imm[4:0]	100	rd	0001011	BARRIER

0000011	rs2	imm[4:0]	100	rd	0001011	BARRIERSUB
imm[11:0]		vs1	000	vd	1011011	VADD12
000010m	vs2	00000	110	vd	0001011	VFEXP
000010m	vs2	vs1	100	vd	0001011	VFTTA
imm[11:0]		vs1	000	vd	1111011	VLB12.V
imm[11:0]		vs1	001	vd	1111011	VLH12.V
imm[11:0]		vs1	010	vd	1111011	VLW12.V
imm[11:0]		vs1	100	vd	1111011	VLBU12.V
imm[11:0]		vs1	101	vd	1111011	VLHU12.V
imm[11:5]	vs2	vs1	000	imm[4:0]	1111011	VSB12.V
imm[11:5]	vs2	vs1	001	imm[4:0]	1111011	VSH12.V
imm[11:5]	vs2	vs1	010	imm[4:0]	1111011	VSW12.V
imm[11:0]		vs1	000	vd	0101011	VLB12D.V
imm[11:0]		vs1	001	vd	0101011	VLH12D.V
imm[11:0]		vs1	010	vd	0101011	VLW12D.V
imm[11:0]		vs1	100	vd	0101011	VLBU12D.V
imm[11:0]		vs1	101	vd	0101011	VLHU12D.V
imm[11:5]	vs2	vs1	000	imm[4:0]	0101011	VSB12D.V
imm[11:5]	vs2	vs1	001	imm[4:0]	0101011	VSH12D.V
imm[11:5]	vs2	vs1	010	imm[4:0]	0101011	VSW12D.V

imm[11:0](x3,x2,x1,xd)	pair	ch	000000	110	PRE_M_32
imm[11:0](x3,x2,x1,xd)	pair	ch	000001	110	PRE_M_64

imm[11:0](x3,x2,x1,xd)	pair	ch		000010	110	PRE_M_96
imm[11:0](x3,x2,x1,xd)	pair	ch		000011	110	PRE_M_128
imm[11:0](x3,x2,x1,xd)	pair	ch		000100	110	PRE_M_GLOBAL_32
imm[11:0](x3,x2,x1,xd)	pair	ch		000101	110	PRE_M_GLOBAL_64
imm[11:0](x3,x2,x1,xd)	pair	ch		000110	110	PRE_M_GLOBAL_96
imm[11:0](x3,x2,x1,xd)	pair	ch		000111	110	PRE_M_GLOBAL_128
imm[11:0](x3,x2,x1,xd)	pair	ch		001000	110	PRE_M_PRIVATE_32
imm[11:0](x3,x2,x1,xd)	pair	ch		001001	110	PRE_M_PRIVATE_64
imm[11:0](x3,x2,x1,xd)	pair	ch		001010	110	PRE_M_PRIVATE_96
imm[11:0](x3,x2,x1,xd)	pair	ch		001011	110	PRE_M_PRIVATE_128
imm[11:0](x3,x2,x1,xd)	pair	ch		001100	110	PRE_M_LOCAL_32
imm[11:0](x3,x2,x1,xd)	pair	ch		001101	110	PRE_M_LOCAL_64
imm[11:0](x3,x2,x1,xd)	pair	ch		001110	110	PRE_M_LOCAL_96
imm[11:0](x3,x2,x1,xd)	pair	ch		001111	110	PRE_M_LOCAL_128
imm[11:0](x3,x2,x1,xd)	pair	ABS	NEG	0000010		PREFIX
imm[11:0](imm[10:5],x2,xd)	pair	ABS	NEG	0001010		PREFIXI