OpenGPGPU
乘影

Ventus GPGPU Instruction Set Architecture

Reference Guide

v2025.3.21

Introduction

This document describes the design of the Ventus GPGPU from the perspective of the instruction set architecture and the hardware/software interface. The Ventus GPGPU instruction set is designed based on the RISC-V Vector Extension (hereafter referred to as RVV). Compared with RISC-V scalar instructions, RVV offers richer expressive capabilities, enabling features such as memory access characterization and differentiation between workgroup and thread operations. The core idea is to use the `v` instruction at the compiler level to describe thread behavior and to merge the common data of thread → warp/workgroup into scalar instructions. In hardware, one warp is essentially an RVV program, typically with a vector element length equal to `num_thread`, while common address computations, jumps, and other operations executed uniformly within a workgroup are performed as scalar instructions, forming a `Vector-Thread` architecture. The hardware maps warps onto the lanes of the RVV processor in a time-sharing manner.

Ventus SIMT architecture compute units adopt a SIMD (Vector) execution mode, executing per workgroup (or `warp_split` in branch conditions). The variable-length aspect of the RVV instruction set is manifested in three areas: changes in the hardware `vlen`; changes in the SEW (element width); and changes in the LMUL grouping. A notable feature of this architecture is that these three parameters are fixed at compile time, with the number of elements in most cases also fixed to num_thread.

Glossary

- `SM` : streaming multiprocessor
- `sGPR` : scalar general purpose register
- `vGPR` : vector general purpose register

Common concepts and definitions used in GPGPU are provided in the table below.

cuda	opencl	Explanation
globalmem	globalmem	Global memory, described by `__global`, accessible by all threads in the kernel.
constantmem	constantmem	Constant memory, described by `__constant`, part of the global address space.
localmem	privatemem	Private memory, variables and kernel parameters of each thread, part of global memory.
sharedmem	localmem	Local memory, described by `__local`, used for data exchange among threads in the same work-group.
grid	NDRange	Describes the number of thread blocks/work-groups within a kernel.
block/CTA	workgroup	Workgroup, the basic execution unit on an SM.
warp	wavefront (AMD)	32 threads form a warp, visible only to hardware.
thread	work-item	Thread/work-item, the smallest unit described in OpenCL C programming.

Programming Model and Driver Functionality

This section introduces the behaviors of OpenCL programs before and after execution by exploring the interaction between Ventus GPGPUas a device and the OpenCL programming framework. Programming model of Ventus is compatible with OpenCL, meaning that its hardware hierarchy corresponds one-to-one with the OpenCL execution model. Additionally, the OpenCL programming framework must offer a series of runtime implementations to accomplish functions such as task allocation, device memory management, command queue management, and more.

Task Execution Model

In OpenCL, the computing platform is divided into the host and the device. The device executes kernels, while the host is responsible for interactions, resource allocation, and device management. The device's computing resources can be partitioned into computing units (CUs), which can be further divided into processing elements (PEs), where all device computations are carried out.

In Ventus, one SM corresponds to one computing unit (CU) in OpenCL, and each vector lane corresponds to one processing element (PE).
When a kernel is executed on the device, multiple instances (which can be understood as threads in a multithreaded program) run concurrently; each instance is called a work-item, and work-items are organized into work-groups. Note that OpenCL only guarantees that work-items within a single work-group can execute concurrently and does not ensure concurrent execution between different work-groups, although in practice work-groups usually run in parallel.
In implementation of Ventus, one work-item corresponds to one thread, which occupies one vector lane during execution. Since hardware threads are organized into thread groups (warps) that execute in lockstep—and the number of threads in a warp is fixed—a work-group may map to multiple warps when implemented on hardware. However, these warps that form one work-group are guaranteed to execute on the same SM.

Functionality Provided by the Driver

The driver in Ventus is divided into two layers: one is the OpenCL runtime environment and the other is the hardware driver. The runtime environment is based on the open-source OpenCL implementation **pocl** and primarily manages the command queue, creates and manages buffers, and handles OpenCL events and synchronization. The hardware driver serves as an abstraction layer for the physical device, mainly providing low-level interfaces for the runtime environment and converting software data structures into hardware port signals. These low-level interfaces include operations for allocating, releasing, and reading/writing device memory; transferring buffers between the host and device; and controlling the device to start execution. The runtime environment interacts with the physical device by utilizing these hardware driver interfaces.

Each kernel requires certain information from the hardware. This information is created by the runtime and copied into the device memory in the form of buffers. Currently, the runtime environment creates the following buffers:

- Metadata buffer and kernel program of NDRange
- Argument buffer of kernel
- buffer explicitly referenced in kernel argument
- Space allocated for private mem, print buffer

When the kernel is started, the parameters of NDRange are passed through the buffer of metadata, and the
contents of the buffer are:

```
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
*/
```

Behavior During Kernel Launch

When the task starts, the signal directly transmitted by the hardware driver is:

Signal	Meaning
`PTBR`	page table base address
`CSR_KNL`	metadata buffer base address
`CSR_WGID`	The ID of the current workgroup in the SM, used solely for hardware identification
`LDS_SIZE`	`localmem_size` : the amount of local memory allocated for the workgroup by the compiler. `privatemem_size` is allocated by default as 1kB per thread.
`VGPR_SIZE`	`vGPR_usage` : the actual number of vGPRs used by the workgroup (aligned to 4)
`SGPR_SIZE`	`sGPR_usage` : the actual number of sGPRs used by the workgroup (aligned to 4)
`CSR_GIDX/Y/Z`	workgroup index in NDRange
`host_wf_size`	Number of threads in one warp
`host_num_wf`	Number of warps in one workgroup

The parameters of the kernel are passed by another piece of kernel_arg_buffer, which will prepare the arguments of the kernel in order, including specific parameter values or addresses of other buffers. Only the address knl_arg_base of kernel_arg_buffer is provided in the metadata of NDRange. Before the kernel function executes, the `start.S` script is run.

```
# start.S
_start:
  # set global pointer register
  .option push
  .option norelax
  la gp, __global_pointer$
  .option pop

  # allocate warp and per-thread level stack pointers, both
  # stacks grows upwards
  li t4,32
  vsetvli t4,t4,e32,m1,ta,ma
  li t4,0x2000
  csrrs t4, mstatus, t4
  li t4, 0
  csrr t1, CSR_WID
  csrr t2, CSR_LDS
  li t3, 1024           # 1M size for single warp
  mul t1, t1, t3        # sp wid * warp_size
  add sp, t1, t2        # sp points to baseaddr of local memory of each SM
  li tp, 0              # tp points to baseaddr for lower bound of private memory(1K) of each thread
  csrr t5, CSR_NUMW
  li t3, 1024
  mul t5, t5, t3
  add s0, t2, t5        # s0 points to local memory base addr in a workgroup

  # clear BSS segment
  la      a0, _edata
  la      a2, _end
  beq     a0, a2, 2f
1:
  sw      zero, (a0)
  addi    a0, a0, 4
  bltu    a0, a2, 1b

2:
  csrr t0, CSR_KNL                       # get addr of kernel metadata
  lw t1, KNL_ENTRY(t0)               # get kernel program address
  lw a0, KNL_ARG_BASE(t0)            # get kernel arg buffer base address
  lw t2, KNL_PRINT_ADDR(t0)          # get kernel print buffer address
  lw t3, KNL_PRINT_SIZE(t0)          # get kernel print buffer size
  la t4, BUFFER_ADDR
  la t5, BUFFER_SIZE
  sw t2, 0(t4)
  sw t3, 0(t5)
  la t6, spike_end                   # exception to stop spike
  csrw mtvec, t6
  jalr t1                            # call kernel program

  # call exit routine
  # tail    exit

  # End of warp execution
  endprg x0, x0, x0
  j spike_end
  .size  _start, .-_start


  .section ".tohost","aw",@progbits
  .align 6
  .globl tohost
  tohost: .dword 0
  .align 6
  .globl fromhost
  fromhost: .dword 0
```

```
    .text
    .global spike_end
    .type spike_end,function
  spike_end:
    li t1,1
    la t0,tohost
    sw t1,0(t0)
  # end.S
  end:
          endprg
```

It is agreed that the printing information of the kernel is transmitted to the host through the print buffer. The address and size of the print buffer are provided in the metadata_buffer. After the running thread finishes printing, it sets the CSR_PRINT of the warp it belongs to. When the host polls that there is unprocessed information, it will take out the print buffer from the device side and reset **CSR_PRINT**.

Stack Space Description

Since OpenCL does not allow the use of dynamic memory functions such as malloc in the Kernel, and there is no heap, the stack space can be increased upwards. tp is used to push the stack when the private registers of each thread are insufficient (that is, vGPR spill stack slots), sp is used to push public data on the stack, and the data that explicitly declares the local tag in programming (that is, sGPR spill stack slots and localmem access , in fact both sGPR spill stack slots and local data will be part of localmem). The compiler provides the overall usage of localmem data (according to sGPR spill 1kB, combined with the size of local data, collectively as localmem_size), for the hardware to complete the workgroup allocation.

Parameter Passing ABI

For the kernel function, a0 is the base address pointer of the parameter list, and the starting address of the memory set by the first clSetKernelArg is stored in a0 register, and the kernel loads parameters from this position by default. For non-kernel functions, use v0-v31 and stack pointer to pass parameters, and v0-v15 as the return value.

Registers

Register Configuration

The architecture defines 64 `sGPRs` (scalar) and 256 `vGPRs` (vector), with each element being 32 bits wide.
When 64-bit data is needed, it is stored as an even-aligned register pair (Register Pair), with the lower 32 bits stored in `GPR[n]` and the higher 32 bits stored in `GPR[n+1]` .
Currently, the physical register count in the hardware is 256 `sGPRs` (scalar) and 1024 `vGPRs` (vector), and the GPU hardware is responsible for mapping the architectural registers to the hardware registers.
The compiler provides the actual usage counts for `vGPRs` and `sGPRs` (in multiples of 4), and the hardware will allocate additional workgroups for concurrent scheduling based on the actual usage.

From the perspective of the RISC-V Vector register file, there are 256 `vGPRs` , with `vlen` fixed to the number of threads (`num_thread`) multiplied by 32 bits —equivalent to setting `SEW=32bit` , `ma` , `ta` , and `LMUL` to 1 via the `vsetvli` instruction. From the SIMT perspective, each thread can have up to 256 32-bit `vGPRs` . One simple way to understand this is to view the vector register file as a two-dimensional array with 256 rows and `num_thread` columns, where each row represents a `vGPR` and each column represents the registers available to one thread. Some vector types defined in OpenCL require grouped registers for representation; for example, `float16` is stored column-wise in the register file, occupying 16 32-bit `vGPRs` . This grouping is handled by the compiler.
A workgroup has 64 `sGPRs` ; operations that need to be performed only once for the entire workgroup—such as address calculations in the kernel—use `sGPRs` . In cases where branching occurs, `vGPRs` are used, as seen in parameter passing for non-kernel functions.

Some special registers:

- `x0` : Zero register
- `x1/ra` : Return PC register
- `x2/sp` : Stack pointer / local mem base address
- `x4/tp` : Private mem base address

`Parameter Passing:`
For kernel functions, `a0` is the base pointer for the argument list. The first `clSetKernelArg` call stores the starting address of the device memory in the `a0` register, and the kernel by default begins loading its parameters from that position.

Custom CSR

description	name	addr
The smallest thread ID in this warp; its value equals CSR_WID * CSR_NUMT, which, together with vid.v, can be used to calculate the other thread IDs.	CSR_TID	0x800
Total number of warps in this workgroup	CSR_NUMW	0x801
Total number of threads in one warp	CSR_NUMT	0x802
Base address of the metadata buffer for this workgroup	CSR_KNL	0x803
The workgroup ID corresponding to this warp in the SM	CSR_WGID	0x804
The warp ID corresponding to this warp within the workgroup	CSR_WID	0x805
Base address of the local memory allocated for the workgroup, which is also the xGPR spill stack base address for this warp	CSR_LDS	0x806
Base address of the private memory allocated for this warp, which is also the VGPR spill stack base address for the thread	CSR_PDS	0x807
The x ID of this workgroup in the NDRange	CSR_GIDX	0x808
The y ID of this workgroup in the NDRange	CSR_GIDY	0x809
The z ID of this workgroup in the NDRange	CSR_GIDZ	0x80a
CSR used for interacting with the host when printing to the print buffer	CSR_PRINT	0x80b

description	name	addr
Re-convergence PC register	CSR_RPC	0x80c

Note: In the assembler, lowercase suffixes can be used to represent the corresponding CSR, for example, using "tid" instead of CSR_TID.

Instruction Set Architecture

Instruction Set Coverage

RV32V extension is chosen as the basic instruction set, supporting the following ranges: RV32I, M, A, zfinx, zve32f, RV64I, A.
The V extension instruction set mainly supports instructions for independent data paths and does not support the original RVV instructions such as shuffle, narrow, gather, and reduction.
The table below lists the currently supported standard instruction ranges, with any modified instructions indicated.

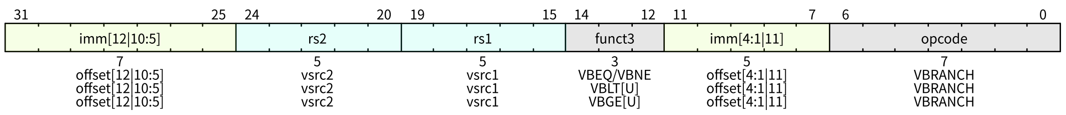
	Ventus Support	Instruction Changes
RV32I	Does not support ecall, ebreak	
RV32M F	Supports RV32M, zfinx, zve32f	
RV32A	Supported	
RV64I	Partially supported	Semantics changed, see later sections
RV64A	Partially supported	Semantics changed, see later sections
RV32V-Register State	Only supports LMUL = 1 and 2	
RV32V-ConfigureSetting	Supports vl computation; can configure support for different element widths via this option	
RV32V-LoadsAndStores	Supports vle32.v, vlse32.v, vluxei32.v memory access modes	The semantics of instructions like `vle8` have been changed to “each thread writes to its corresponding vector register element position” rather than continuous writing
RV32V-IntergerArithmetic	Supports most int32 arithmetic instructions	The semantics of `vmv.x.s` have been changed to “each thread writes to the scalar register” rather than always writing via vector register `idx_0`; simultaneous multi-thread writes are undefined (the correctness must be ensured by the programmer); `vmv.s.x` semantics have been modified to be consistent with `vmv.v.x`
RV32V-FixedPointArithmetic	Added int8 support; other types can be added as application demands	
RV32V-FloatingPointArithmetic	Supports most fp32 instructions, and adds fp64 and fp16 support	
RV32V-WideningIntergerArithmetic	Supports most arithmetic instructions	2*SEW-bit width elements are implemented via register pairs
RV32V-ReductionOperations	To be considered for addition as needed by applications and compiler demands, for example, to support work_group_reduce in OpenCL2.0	
RV32V-Mask	Supports instructions for computing and setting the mask for each lane independently	The semantics of instructions like `vmsle` have been changed to “each thread writes to its corresponding vector register element position” rather than continuous writing
RV32V-Permutation	Not supported; may be added depending on application and compiler needs	

	Ventus Support	Instruction Changes
RV32V-ExceptionHandling	Not supported; may be added depending on application and compiler needs	

Custom Instructions

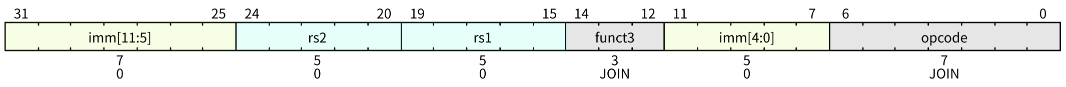
Branch Control Instructions

The VBRANCH branch instruction uses the B-type instruction format. A 12-bit signed immediate is sign-extended and added to the current PC to yield the starting PC of the else path; then the value from CSR_RPC is read (rpc), and based on the vector register comparison results, the SIMT thread branch management stack is operated.


31 25 24 20 19 15 14 12 11 7 6 0
imm[12|10:5] rs2 rs1 funct3 imm[4:1|11] opcode
7 5 5 3 5 7
offset[12|10:5] vsrc2 vsrc1 VBEQ/VBNE offset[4:1|11] VBRANCH
offset[12|10:5] vsrc2 vsrc1 VBLT[U] offset[4:1|11] VBRANCH
offset[12|10:5] vsrc2 vsrc1 VBGE[U] offset[4:1|11] VBRANCH

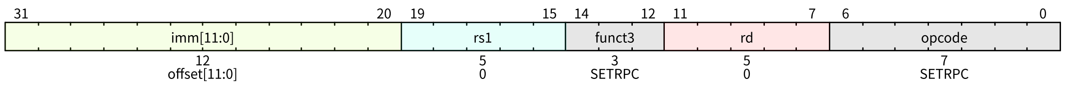
The thread branch instructions compare two vector registers. For threads whose corresponding elements are equal, the VBEQ instruction causes those threads to jump to the else PC; threads with unequal elements continue execution at PC+4. The jump, convergence, and mask control for both paths are managed by the SIMT stack, which pushes rpc, else PC, and the comparison results onto the stack. When all active threads have elements that are all equal or all unequal, no stack operation is triggered; all threads either jump to the else PC or continue at PC+4. When the operands in vs1 and vs2 differ, VBNE determines whether the corresponding thread jumps to PC else or PC+4. VBLT and VBLTU use signed and unsigned comparisons, respectively, for vs1 and vs2; if for any vector element vs1 is less than vs2, the corresponding thread jumps to PC else, with the remaining active threads continuing at PC+4. Similarly, VBGE and VBGEU use signed and unsigned comparisons for vs1 and vs2; if for any vector element vs1 is greater than or equal to vs2, the corresponding thread jumps to PC else, while the remaining active threads continue at PC+4.

The thread branch convergence (join) instruction uses the S-type instruction format, with default source operand and immediate values of 0.



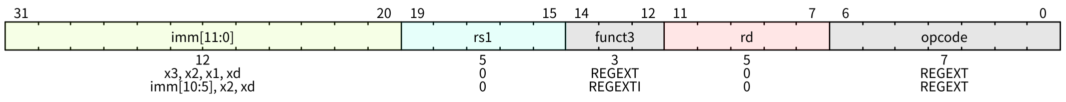
The JOIN instruction compares the current instruction PC with the re-convergence PC at the top of the SIMT stack. If they are equal, it sets the active thread mask to the mask at the top of the stack and jumps to the else PC at the top of the stack; if not, no action is taken.

The re-convergence PC setting instruction, SETRPC, sign-extends a 12-bit immediate and adds it to the source operand, then writes the result into the CSR_RPC csr register and the destination register.


31 20 19 15 14 12 11 7 6 0
imm[11:0] rs1 funct3 rd opcode
12 5 3 5 7
offset[11:0] 0 SETRPC 0 SETRPC

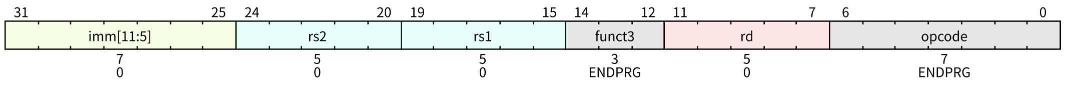
Register Extension Instructions

The REGEXT and REGEXTI instructions are used to extend the register encoding and immediate encoding of the instruction that follows. In the REGEXT instruction, the 12-bit immediate is split into four 3-bit segments, which are then concatenated to the high bits of the register encodings (rs3/vs3, rs2/vs2, rs1/vs1, rd/vd) in the following instruction. The REGEXTI instruction targets instructions that use a 5-bit immediate; its prefix includes a 6-bit high immediate along with the high bits for rs2/vs2 and rd/vd. The 11-bit immediate is obtained by directly concatenating the high immediate from the prefix instruction with the 5-bit immediate.
In addition, when no extension is used, the vs3 field in vector floating-point operations corresponds to bits [11:7] of vd; however, when vector extension is applied, the high 3 bits of vs3 and vd are stored separately.

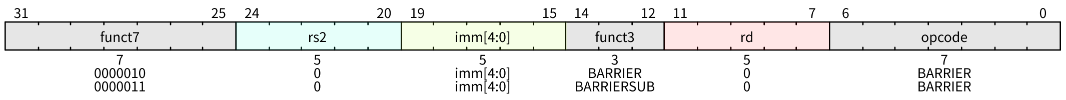

31 20 19 15 14 12 11 7 6 0
imm[11:0] rs1 funct3 rd opcode
12 5 3 5 7
x3, x2, x1, xd 0 REGEXT 0 REGEXT
imm[10:5], x2, xd 0 REGEXTI 0 REGEXT

Synchronization and Task Control Instructions

The ENDPRG instruction must be explicitly inserted at the end of a kernel to indicate the end of execution for the current warp. It can only be used in conditions without branches.



The BARRIER instruction corresponds to OpenCL's barrier(cl_mem_fence_flags flags) and work_group_barrier(cl_mem_fence_flags flags, [memory_scope scope]) functions, providing data synchronization among threads within the same workgroup. The default value for memory_scope is memory_scope_work_group.

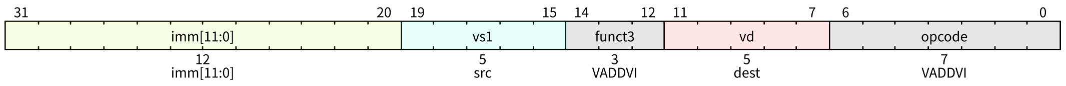

31 25 24 20 19 15 14 12 11 7 6 0
funct7 rs2 imm[4:0] funct3 rd opcode
7 5 5 3 5 7
0000010 0 imm[4:0] BARRIER 0 BARRIER
0000011 0 imm[4:0] BARRIERSUB 0 BARRIER

The encoding for the 5-bit immediate field is as follows:

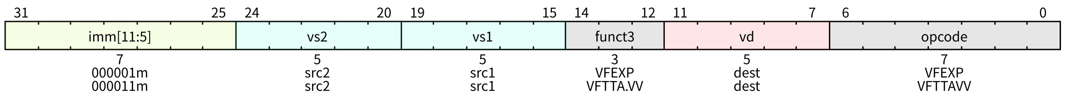
imm[4:3]	**00**	**01**	**10**	**11**
memory_scope	work_group (default)	work_item	device	all_svm_devices
imm[2:0]	**imm[2]=1**	**imm[1]=1**	**imm[0]=1**	**imm[2:0]=000**
CLK_X_MEM_FENCE	IMAGE	GLOBAL	LOCAL	USE_CNT

When the opencl_c_subgroups feature is enabled, the instruction changes to barriersub, corresponding to the memory_scope=subgroup case; in this case, imm[4:3] is fixed to 0, and cl_mem_fence_flags equals imm[2:0], which is identical to the barrier instruction.

Custom Computational Instructions


31 20 19 15 14 12 11 7 6 0
imm[11:0] vs1 funct3 vd opcode
12 5 3 5 7
imm[11:0] src VADDVI dest VADDVI

VADD12.VI adds the unsigned immediate (imm) to the vector register vs1 and writes the result into the destination vector register vd.


31 25 24 20 19 15 14 12 11 7 6 0
imm[11:5] vs2 vs1 funct3 vd opcode
7 5 5 3 5 7
000001m src2 src1 VFEXP dest VFEXP
000011m src2 src1 VFTTA.VV dest VFTTAVV

VFEXP calculates the exponential (exp) of vs2 and assigns the result to the destination vector register vd. VFTTAVV computes the convolution of vectors vs1 and vs2, adds the result to vd, and writes the final result back to vd.

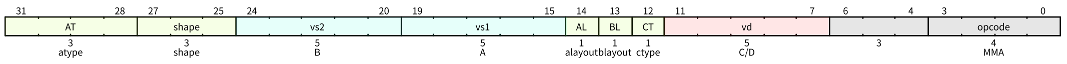
Custom Matrix Multiply-Accumulate Instruction

The MMA instruction is used for computing a matrix multiply-accumulate operation:
D = A×B + C.
The precision and dimensions of the four matrices A, B, C, and D are configurable. Specifically, A is of size M×K, B is of size K×N, and both C and D are of size M×N. The supported configurations are listed in the table below:

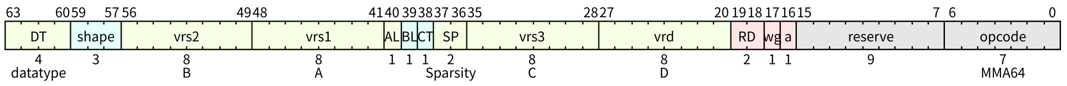
A/B type	shape	C/D type
FP32	m8n8k8	FP32
	m16n8k8	
	m16n8k16	
FP16	m8n8k8	FP16
	m16n8k8	
	m16n8k16	
	m8n8k8	FP32
	m16n8k8	
	m16n8k16	
BF16	m8n8k8	FP32
	m16n8k8	

	m16n8k16	
FP8(E4M3)	m16n8k32	INT32
FP8(E5M2)	m16n8k32	
INT8	m8n8k16	
	m16n16k16	
INT4	m8n8k32	
	m16n8k32	
	m16n8k64	
Binary	m8n8k128	

In the instruction, the **vd** field simultaneously serves as the index for source register **C** and destination register **D**.



64-Bit Instruction Word Length Matrix Multiply-Accumulate



The MMA64 instruction is an extension of the MMA instruction. Functionally, it aligns with the MMA instruction, with additional encoding fields used to extend register encoding to 8 bits and support three source operand encodings. In MMA64, the matrices **C** and **D** are stored in separate registers. This instruction supports precisions including FP16, BF16, INT8, INT4, Binary, FP8, FP4, etc.; it also supports specifying different rounding modes for floating-point operations and whether integer operations output with saturation. Reserved fields are provided to support sparse computation.

The field names in the MMA64 instruction include:

- `opcode` (7 bits, indicating the instruction type as MMA64)
- `reserve` (9 bits, reserved)
- `a` (1 bit, indicating the source of operand A)
- `wg` (1 bit, warp group flag)
- `RD` (2 bits, for rounding mode/saturation overflow flag)
- `vrd` (8 bits, encoding for destination register D)
- `vrs3` (8 bits, encoding for source register C)
- `SP` (2 bits, sparsity flag)
- `CT` (1 bit, C/D matrix data type flag)
- `BL` (1 bit, B matrix layout flag)
- `AL` (1 bit, A matrix layout flag)
- `vrs1` (8 bits, encoding for source register A)
- `vrs2` (8 bits, encoding for source register B)
- `shape` (3 bits, matrix dimension flag)
- `DT` (4 bits, data type flag)

In MMA64, the `DT` and `CT` fields indicate the element data types of matrices A, B, C, and D. The 4-bit `DT` specifies the data precision for matrices A and B, while the 1-bit `CT` indicates the precision for matrices C and D. For example, when accumulating FP16 data, matrices A and B use FP16 precision; when accumulating INT32 data, matrices A and B are in INT8, INT4, or Binary precision; and when accumulating FP32 data, matrices A and B can be in FP16, BF16, FP8 (E4M3/E5M2), FP6 (E3M2/E2M3), FP4 (E2M1/E3M0), etc. The 3-bit scale instruction varies with data precision.

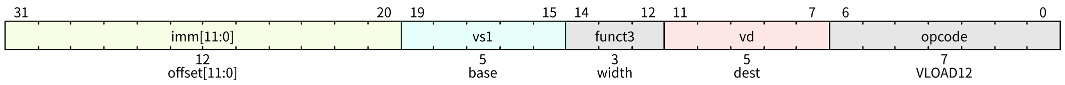
The `AL` and `BL` flags indicate the register layout of matrices A and B. For matrix A, AL = 0 denotes row-major order while AL = 1 denotes column-major order; for matrix B, BL = 1 denotes row-major order while BL = 0 denotes column-major order. The `RD` flag (rounding mode/saturation) is 2 bits: for floating-point C/D operands, these bits represent the rounding mode (00 = rn, 01 = rz, 10 = rm, 11 = rp); for integer C/D operands, the flag indicates saturation (00 = saturate on overflow; 01 = no saturation). When the `wg` flag is 1, the instruction is at the warp group level, extending the computation scale compared to warp-level instructions—with the specific dimensions varying according to precision. At the warp group level, an additional flag, `a (R/SMEM)`, is defined: when set to 1, it indicates that operand A is sourced from registers; when 0, operand A comes from SMEM; in both cases, operand

B is sourced from SMEM. In the SMEM case, the data register in registers stores the descriptor for matrix A. The computation precision, sparsity, matrix layout, and rounding mode remain consistent with those of the warp-level computation instructions.

Custom Immediate Memory Access Instructions

Custom immediate memory access instructions support per-thread accesses to the global, local, and private address spaces. Based on the memory address computation method, these instructions are divided into four series: `flat`, `global`, `local`, and `private`.

`Flat` access is a general-purpose memory access method that can access global, local, and private memory spaces using flat memory access instructions. The memory is addressed in a unified address space. The target address (Addr) is determined by performing a range check on the address contained in the source register (vs1) and then adding the immediate offset.



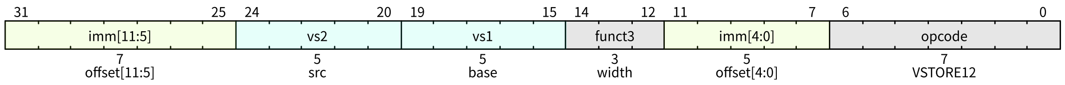
In line with standard RISC-V instructions, the custom flat access instructions use byte addressing in little-endian format. Each thread computes an effective byte address Addr, and the element at memory starting at Addr is copied into the vector register `vd`. The VLW12 instruction loads a vector of 32-bit values from memory into `vd`. Similarly, VLH12 loads 16-bit values, sign-extends them to 32 bits, and stores them in `vd`; VLHU12 loads 16-bit values, zero-extends them to 32 bits, and stores them in `vd`. The VLB12 and VLBU12 instructions have analogous definitions for 8-bit values.

The computation of Addr depends on the memory range check performed on the data in `vs1`. The relationship between Addr and the target space is as follows:

Target Space	Addr
global	Addr = vs1 + offset
local	Addr = vs1 + offset
private	Addr = ((vs1 + offset)[31:2] << 2) * num_thread_per_warp + (vs1 + offset)[1:0] + (thread_idx_in_warp << 2) + csr_pds[wid]

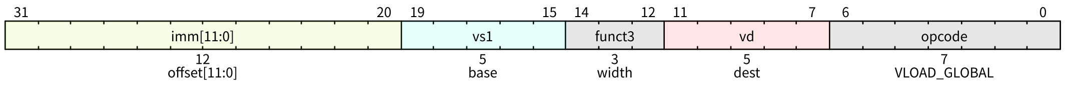
The method for performing the range check on **vs1** is as follows:

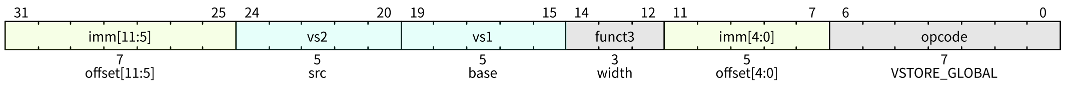
Range of vs1	Target Space
gds_base ≤ vs1 ≤ gds_limit	global
lds_base ≤ vs1 ≤ lds_limit	local
vs1[31:24] == 8'h00	private



Each thread computes Addr in the same manner as above. The VSW12, VSH12, and VSB12 instructions store a vector of data from the lower 32-bit, 16-bit, and 8-bit portions of vector register `vs2`, respectively, into the memory starting at Addr.

`Global` access is used for accessing global memory.
For custom global memory access instructions, the address space is byte-addressed and little-endian. Each thread's effective byte address is computed as:
`Addr = vs1 + offset + csr_gds.`
The VLW_GLOBAL instruction loads a vector of 32-bit values from global memory into `vd`. Similarly, VLH_GLOBAL loads 16-bit values (sign-extending them to 32 bits), VLHU_GLOBAL loads 16-bit values (zero-extending them), and VLB_GLOBAL/VLBU_GLOBAL are defined analogously for 8-bit values.

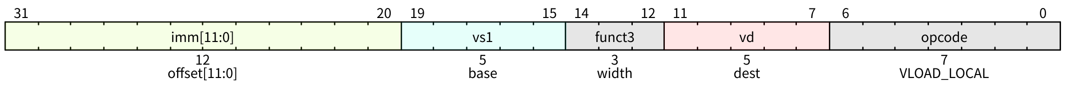


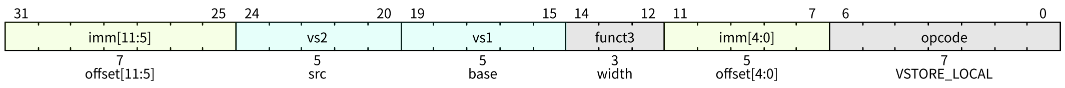


Each thread computes Addr as described above, and the VSW_GLOBAL, VSH_GLOBAL, and VSB_GLOBAL instructions store data from the lower portions of `vs2` into global memory starting at Addr.

`Local` access is used for accessing shared memory.
In the custom local memory access instructions, the address space is byte-addressable and follows a little-endian format. Each thread has an effective byte address, denoted as `Addr`, which is used to copy elements from memory starting at `Addr` into the vector register `vd`.

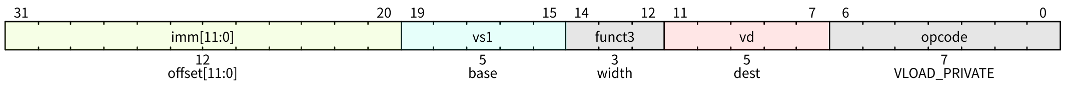
The `VLW_LOCAL` instruction loads 32-bit values from shared memory into `vd`. The `VLH_LOCAL` instruction loads 16-bit values from shared memory, sign-extends them to 32 bits, and then stores them into `vd`. Similarly, the `VLHU_LOCAL` instruction loads 16-bit values from shared memory, zero-extends them to 32 bits, and stores them into `vd`. The `VLB_LOCAL` and `VLBU_LOCAL` instructions operate on 8-bit values in a similar manner. Here, `Addr` is calculated as `vs1 + offset + csr_lds`.

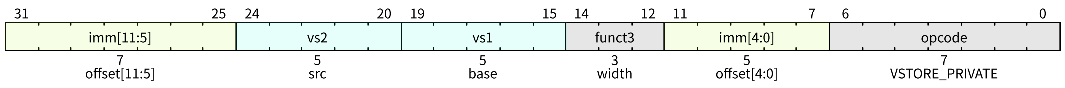




Each thread computes Addr as above, and the VSW_LOCAL, VSH_LOCAL, and VSB_LOCAL instructions store data from the lower portions of `vs2` into shared memory starting at Addr.

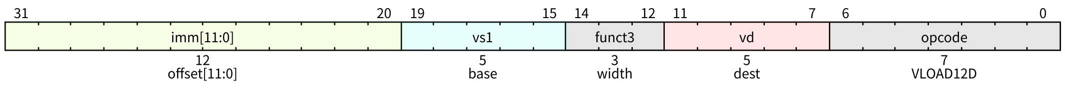
`Private` access is used for accessing private memory.
In custom `PRIVATE` memory access instructions, the address space is byte-addressable and follows a little-endian format. Each thread computes an effective byte address (`Addr`) to copy elements from memory starting at `Addr` into the vector register `vd`.
The `VLW_PRIVATE` instruction loads 32-bit values from private memory into `vd`. The `VLH_PRIVATE` instruction loads 16-bit values from private memory, sign-extends them to 32 bits, and stores them into `vd`. Similarly, the `VLHU_PRIVATE` instruction loads 16-bit values from private memory, zero-extends them to 32 bits, and stores them into `vd`. The `VLB_PRIVATE` and `VLBU_PRIVATE` instructions operate on 8-bit values in a similar manner.
The effective address (`Addr`) is calculated as:
`Addr = ((vs1 + offset)[31:2] << 2) * num_thread_per_warp + (vs1 + offset)[1:0] + (thread_idx_in_warp << 2) + csr_pds[wid]`.



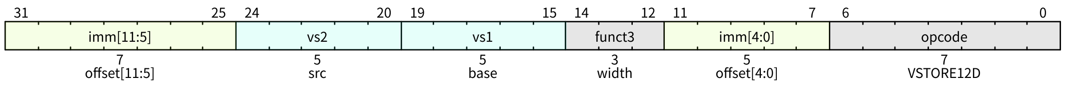


Each thread computes Addr as described above, and the VSW_PRIVATE, VSH_PRIVATE, and VSB_PRIVATE instructions store data from the lower portions of `vs2` into private memory starting at Addr.

Custom 64-bit Address Space Immediate Memory Access Instructions



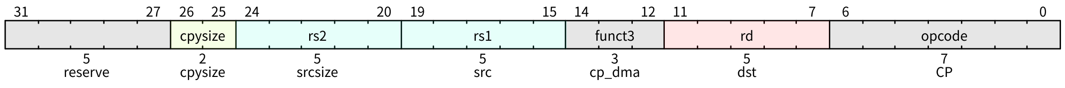
The custom 64-bit memory access instruction's address space is byte-addressed and in little-endian format. `vs1` represents an even-aligned register pair. The effective byte address for each thread is `addr = [vs1+1:vs1] + offset`, which copies the element starting at `addr` in memory to the vector register `vd`. The `VLW12D` instruction loads a vector of 32-bit values from memory into `vd`. `VLH12D` loads a 16-bit value from memory, sign-extends it to 32 bits, and stores it in `vd`. `VLHU12D` loads a 16-bit value, zero-extends it to 32 bits, and stores it in `vd`. `VLB12` and `VLBU12` have similar definitions for 8-bit values.



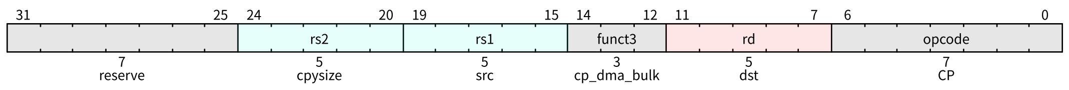
For each thread, `addr = [vs1+1:vs1] + offset`. The `VSW12D`, `VSH12D`, and `VSB12D` instructions store the vector of 32-bit, 16-bit, and 8-bit values from the lower bits of vector register `vs2` into the memory space starting at `addr`.

Custom Asynchronous Data Copy Instructions

The `cp_dma` instruction specifies the source address in the L2 cache, the destination address in shared memory, the copy size (`copysize`), and the source size (`srcsize`). Data addresses in the instruction are aligned to 1 byte. The copy size specifies the total amount of data to be copied, which in this instruction can only be 4, 8, or 16 bytes. The source size specifies the number of valid data bytes to be copied, so the source size cannot exceed the copy size. Data beyond the source size is replaced with zeros.

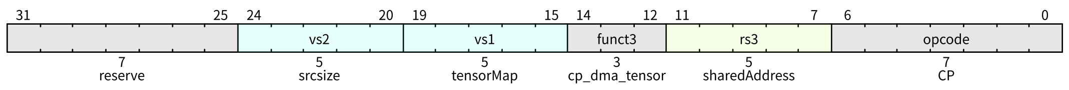


The `cp_dma_bulk` instruction is used for large-scale data transfers. The instruction specifies that the source address, destination address, and copy size are aligned to 16 bytes. It reads the copy size, source address, and destination address stored in three scalar registers and performs asynchronous large-scale data copying. The copy size ranges from 4 to $2^{32} - 4$ bytes.

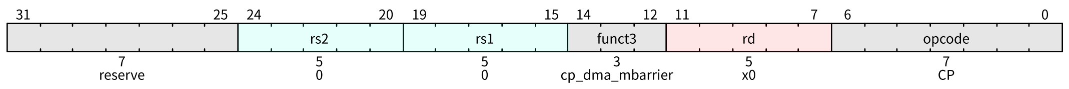


The `cp_dma_tensor` instruction is used to copy a tensor-type data from global memory to shared memory. The addressing box and tensor require many parameters, which are assumed to be stored in vector registers when the instruction is executed. Each position in the register is an xLen-bit number that can be directly read. The parameters are as follows:

Data Info	Function	Storage Location
`datatype`	Tensor data type and width	`vs1[0]`
`tensorRank`	Tensor dimension, not exceeding 5	`vs1[1]`
`globalAddress`	Starting address of the tensor in global memory	`vs1[2]`
`globalDim1-5`	Number of elements along dimensions 1-5, non-zero and ≤ 2^{32}	`vs1[3]-vs1[7]`
`globalStride1-5`	Stride along dimensions 1-5, must be a multiple of 16 and ≤ 2^{40}	`vs1[8]-vs1[12]`
`BoxAddress`	Starting address of the box	`vs2[0]`
`boxDim1-5`	Number of elements traversed along dimensions 1-5, non-zero and ≤ 256	`vs2[1]-vs2[5]`
`elementStrides1-5`	Iteration stride along dimensions 1-5, non-zero and ≤ 8	`vs2[6]-vs2[10]`
`interleave`	Data interleaving mode	`vs2[11]`
`swizzle`	Swizzling mode in shared memory	`vs2[12]`
`L2promotion`	L2 cache byte granularity	`vs2[13]`
`oobfill`	Specifies whether to fill out-of-bounds elements with 0 or NaN	`vs2[14]`
`sharedAddress`	Starting address in shared memory	`rs3`



The `cp_dma_mbarrier` instruction is used for synchronization of asynchronous copy instructions. Its function is to ensure that the warp executes the following instructions only after reaching this instruction and completing the DMA data transfer. Additionally, the warp is divided into groups of 4 threads each. Only when all 4 threads in a group have reached the barrier instruction and completed the DMA data transfer can the group proceed to execute the subsequent instructions.



Custom Memory Access Prefix Instructions

The `PREFIX_MEMORY` series of instructions serve as prefix instructions to indicate the memory space, cache policy, and data width for subsequent memory access instructions. If the `pair` field is 0, the address width (32/64) will be determined by the subsequent memory access instruction. If it is 1, the 64-bit address space is used. If the next instruction is not a memory access instruction, the current instruction becomes invalid.

The high 12 bits of the immediate value are split into four 3-bit segments, which are concatenated with the high bits of the `rs3/vs3`, `rs2/vs2`, `rs1/vs1`, and `rd/vd` encodings in the next instruction.

The address spaces and mappings indicated by the `PREFIX_MEMORY` instruction are as follows:

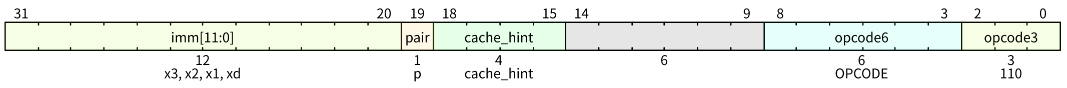
Address Space	Address Calculation	E
`private`	`Addr = ((vs1+offset)[31:2]<<2) * num_thread_per_warp + (vs1+offset)[1:0] + thread_idx_in_warp<<2 + csr_pds[wid]` or `Addr = (([vs1:vs1+1]+offset)[31:2]<<2) * num_thread_per_warp + ([vs1:vs1+1]+offset)[1:0] + thread_idx_in_warp<<2 + csr_pds[wid]`	2

Address Space	Address Calculation	E
`local`	`addr = (vs1+imm) + csr_lds` or `addr = ([vs1:vs1+1]+imm) + csr_lds`	3
`global`	`addr = (vs1+imm) + csr_gds` or `addr = ([vs1:vs1+1]+imm) + csr_gds`	1
`default`	Address calculation based on the memory access instruction definition	0

The high 2 bits of the `OPCODE4` field serve as the address space encoding, and the low 2 bits serve as the data width encoding. If the data width is greater than 32 bits (1 word), data is read from consecutive address spaces into adjacent registers, or data is read from adjacent registers into consecutive address spaces.

Data Width (bits)	Encoding
32	0
64	1
96	2
128	3

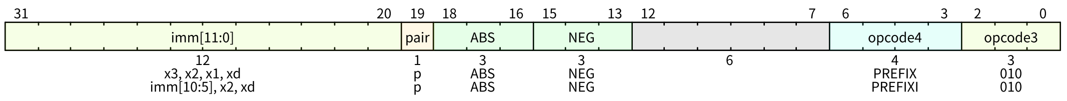
The `cachehint` field of the instruction serves as a hardware cache policy guide but may not take effect.



Custom Compute Prefix Instructions

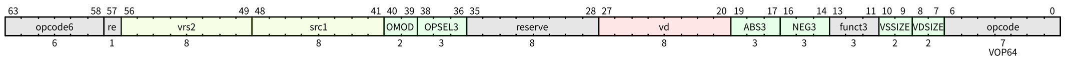
The `PREFIX` and `PREFIXI` instructions are used to extend the register encoding and immediate value encoding of the subsequent instruction and modify the result of the next instruction. In the `PREFIX` instruction, the 12-bit immediate value is split into four 3-bit segments, which are concatenated with the high bits of the `rs3/vs3`, `rs2/vs2`, `rs1/vs1`, and `rd/vd` encodings in the next instruction. The `PREFIXI` instruction targets instructions that use a 5-bit immediate value, with the prefix containing the high 6 bits of the immediate value and the high bits of the `rs2/vs2` and `rd/vd` encodings. The 11-bit immediate value is obtained by directly concatenating the high bits of the immediate value in the prefix instruction with the 5-bit immediate value.

Additionally, when no extension is used, the `vs3` in vector floating-point operations is the `[11:7]` segment of `vd`. However, during vector extension, the high 3 bits of `vs3` and `vd` are stored separately.



Field	Meaning
`pair`	Whether the next instruction uses a 64-bit data pair from register concatenation: 0 - No, 1 - Yes
`ABS`	Whether to take the absolute value of the input data for the next instruction: Bit 0 for `src1`, Bit 1 for `src2`, Bit 2 for `src3`: 0 - No, 1 - Yes
`NEG`	Whether to negate the input data for the next instruction: Bit 0 for `src1`, Bit 1 for `src2`, Bit 2 for `src3`: 0 - No, 1 - Yes

Custom 64-bit Compute Instructions



The 64-bit compute instruction supports integer and floating-point addition, subtraction, multiplication, max, and min operations. It is mainly used to extend computations for data types such as `fp16` and `int8`, and supports operand negation, absolute value, rounding, and scaling through modifiers. The fields are described as follows:

- `NEG`: 3 bits wide, indicating whether to negate the three `src` operands (currently, regular compute instructions only have two operands, with one bit reserved for future extension).
- `ABS`: 3 bits wide.
- `OPSEL`: The `OPSEL` field is 3 bits wide, corresponding to the selection of `vs2`, `src1`, and `vd`. When set to 0, the lower 16 bits of each element are valid; when set to 1, the higher 16 bits are valid. This field is meaningless when the data width is 32 bits or larger.

The `funct3` and `opcode6` fields align with the vector definition, representing the data type (integer or floating-point) and the operation function (addition, subtraction, multiplication, max, min), respectively. The instruction currently reserves 9 bits, which can be used to extend three-operand compute instructions.

RV64I

The following table summarizes the new instructions and semantics added in RV64I compared to RV32I:

Instruction	Semantics
`SLLW`	The source and destination operands are 64-bit data formed by concatenating adjacent register pairs, with even alignment.
`SRLW`	The source and destination operands are 64-bit data formed by concatenating adjacent register pairs, with even alignment.
`SRAIW`	The source and destination operands are 64-bit data formed by concatenating adjacent register pairs, with even alignment.
`SRAW`	The source and destination operands are 64-bit data formed by concatenating adjacent register pairs, with even alignment.
`ADDW`	The source and destination operands are 64-bit data formed by concatenating adjacent register pairs, with even alignment.
`ADDIW`	The source and destination operands are 64-bit data formed by concatenating adjacent register pairs, with even alignment.
`SUBW`	The source and destination operands are 64-bit data formed by concatenating adjacent register pairs, with even alignment.
`LD`	The source operand is 64-bit data formed by concatenating adjacent register pairs with even alignment. It fetches 32-bit data from the 64-bit address space and stores it in the destination register.
`SD`	The address operand `rs1` is 64-bit data formed by concatenating adjacent register pairs with even alignment. It stores 32-bit data from `rs2` into the 64-bit address space.

RV64A

The following table summarizes the new instructions and semantics added in RV64A compared to RV32A:

Instruction Set	Semantics
`RV64A`	The address operand `rs1` is 64-bit data formed by concatenating adjacent register pairs with even alignment. `rs2` and `rd` are 32-bit data.

RVV Widening

The following table summarizes the widening instructions in RVV and their alignment with the RVV definition:

Instruction Scope	Semantics	Aligns with RVV Definition?
`vwop.vv`	Source operands are 32-bit vector elements, and the destination operand is a 64-bit vector element: 64 = 32 op 32.	Yes
`vwop.vx`	Source operands are a 32-bit vector element and a 32-bit scalar element, and the destination operand is a 64-bit vector element: 64 = 32 op 32.	Yes
`vwop.wv`	Source operands are 64-bit vector elements, and the destination operand is a 64-bit vector element: 64 = 64 op 64.	No
`vwop.wx`	Source operands are a 64-bit vector element and a 32-bit scalar element, and the destination operand is a 64-bit vector element: 64 = 64 op 64.	No