Unlimited Vector Extension 2.0

February 2024

Contents

Acronyms

CPU Central Processing Unit.

CSR Control Status Register.

EOD End-of-Dimension.

EOS End-of-Stream.

FIFO First-In, First-Out.

HPC High-Performance Computing.

ISA Instruction Set Architecture.

ISS Instruction Set Simulator.

MMU Memory Management Unit.

OoO Out-of-Order.

PC Program Counter.

RAT Register Alias Table.

RTL Register Transfer Level.

RVV RISC-V Vector Extension.

SAT Stream Allocation Table.

SCROB Stream Configuration Reorder Buffer.

SE Streaming Engine.

SIMD Single Instruction, Multiple Data.

SU Streaming Unit.

SVE Scalable Vector Extension.

UVE Unlimited Vector Extension.

Chapter 1

Simulation Infrastructure

Since the Unlimited Vector Extension (UVE) development is in its early stages, the specification is undergoing several improvements and corrections, and a real hardware implementation is not yet available. Therefore, a software simulator is the most adequate tool to continue the development and validation of the extension. In accordance, this chapter starts by presenting the chosen simulator, *Spike*, and the reasons for its selection over the one used in the original work, *gem5*. Then, the developed infrastructure is described, as well as the modifications and additions made to the simulator in order to support existing and new Unlimited Vector Extension (UVE) instructions.

1.1 The RISC-V ISA Simulator: *Spike*

The Instruction Set Simulator (ISS) *Spike* has been chosen as the appropriate tool to validate instructions and overall behaviour of UVE, as well as to continue the development of its specification. *Spike* is the golden reference functional RISC-V Instruction Set Architecture (ISA) simulator and is widely used as the proof-of-concept target for every RISC-V extension [1, 2].

The UVE proof-of-concept was implemented and validated on a different simulator, *gem5*, which is a cycle-accurate simulator whose purpose is to mimic real hardware behaviour. The work by Domingos et al. [3] implements the supporting microarchitecture, namely the Streaming Engine (SE) and the necessary Central Processing Unit (CPU) processing pipeline modifications. Although much of the base work is done and available for further development, the base *gem5* simulator does not provide extensive enough documentation available and its base code is constantly evolving between releases, so the original implementation is now considered deprecated. In addition, some benchmarks used to evaluate the extension return unexpected results, due to the internal workings of the simulator, while others take some time to be executed, which is not ideal for an early development tool. Moreover, a validation that is completely independent of implementation details is necessary (i.e., microarchitecture and pipeline modifications to a specific processor), as it is the only way to ensure that the extension is correctly formalised and that the instructions behave and interact as expected. All of this led to the decision to opt for a simpler and purely functional simulator, thus speeding up the development process and allowing the focus to be solely on the specification, key for future implementations.

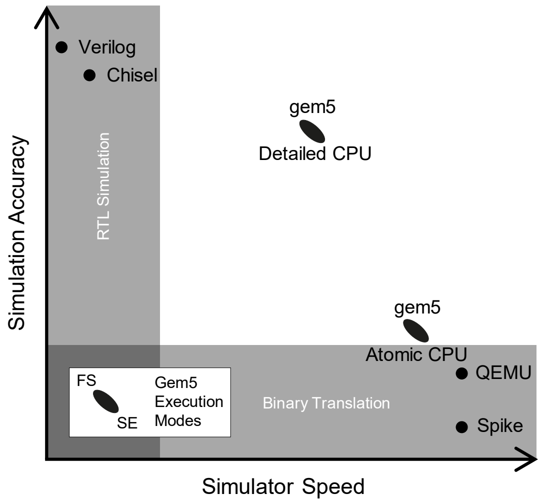


Figure 1.1: Illustration of simulation accuracy vs. speed of multiple simulation platforms [4].

The choice of *Spike* as the base simulation environment for this work resulted from an initial assessment of the advantages and disadvantages between several simulation platforms. As can be seen in Figure 1.1, there is usually a compromise between simulation accuracy and speed when choosing between the available RISC-V simulators available. Whereas Register Transfer Level (RTL) Simulation is the most accurate, binary translation is the fastest. It is clear that *gem5* is the most adequate tool when time performance analysis is pivotal but it is not feasible to create a proper RTL Simulation, as it requires the development and implementation of the entire system. Because the main objective is to perform a functional validation, binary translation is not only enough, as it is preferable. Therefore, although *Spike* does not allow cycle-by-cycle precision, it is suitable for this work. Despite QEMU appearing to be slightly more accurate, it is a much bigger project, as it targets multiple architectures, not only RISC-V, and is thus more difficult to modify, something that is necessary in order to create UVE support.

Spike is currently at Version 1.1.0 and already supports many RISC-V ISA features, including the RISC-V Vector Extension (RVV) which served as a base for some of the developed modules. However, upon analysing the implementation of several extensions on the simulator, it became clear that the UVE implementation structure would be very different. This is mainly due to the way the simulator source code is written, heavily dependent on macros defined in multiple files and with little to no documentation. This resulted in code structured in a very different way than the rest of the simulator and its supported extensions, albeit more comprehensible.

1.2 Simulator Files and Code Structure

The *Spike* simulator is a complex piece of software, with several files and classes, and a large codebase. It emulates a processor through the `processor_t` class, declared and implemented in files `processor.h/cc`. Alterations to this class were minimal, limited to the addition of the Streaming Unit (SU). Another key component which the processor has access to is the Memory Management Unit (MMU), implemented through the `mmu_t` class.

Only a few original files were modified, with the majority of the new code being added in new files. The emulated Streaming Engine (SE) structures were implemented in C++ classes as well, and the *Standard Library* was extensively used. The main classes and their attributes are shown in Figure 1.2, which also indicates in which files each definition and implementation can be found.

The Streaming Unit (SU) and its supporting classes, described in detail in Section 1.3, are defined and implemented in the files `descriptors.h/cc` (dimensions and modifiers) and `streaming_unit.h/cc` (registers and SU). Moreover, files containing the implementation of each instruction were created. These must be inserted in the `riscv/insns` directory and have the same name as the corresponding instruction (e.g., `so.a.add.fp` is implemented in file `so_a_add_fp.h`). The necessary decoding functions are included in the file `decode.h`, described in Section 1.4.1. For the simulator to recognise the new instructions, the file that holds the ISA encoding, `encoding.h`, was updated. To obtain the necessary code, the official RISC-V Opcodes project [5] was used, where the encoding of each instruction was added and UVE's predicate registers and immediate encoding was added to the file `constants.py`.

Furthermore, the new extension was added to file `riscv/riscv.mk.in`, identically to what is done to the native ones, so each new instruction was included in the variable `riscv_insn_ext_uve`. In this file, every new source and header file was also added to variables `riscv_srcs` and `riscv_install_hdrs`, respectively, so that they could be recognised during the compilation of the simulator.

Lastly, the disassembler was extended so that the new registers and instruc-

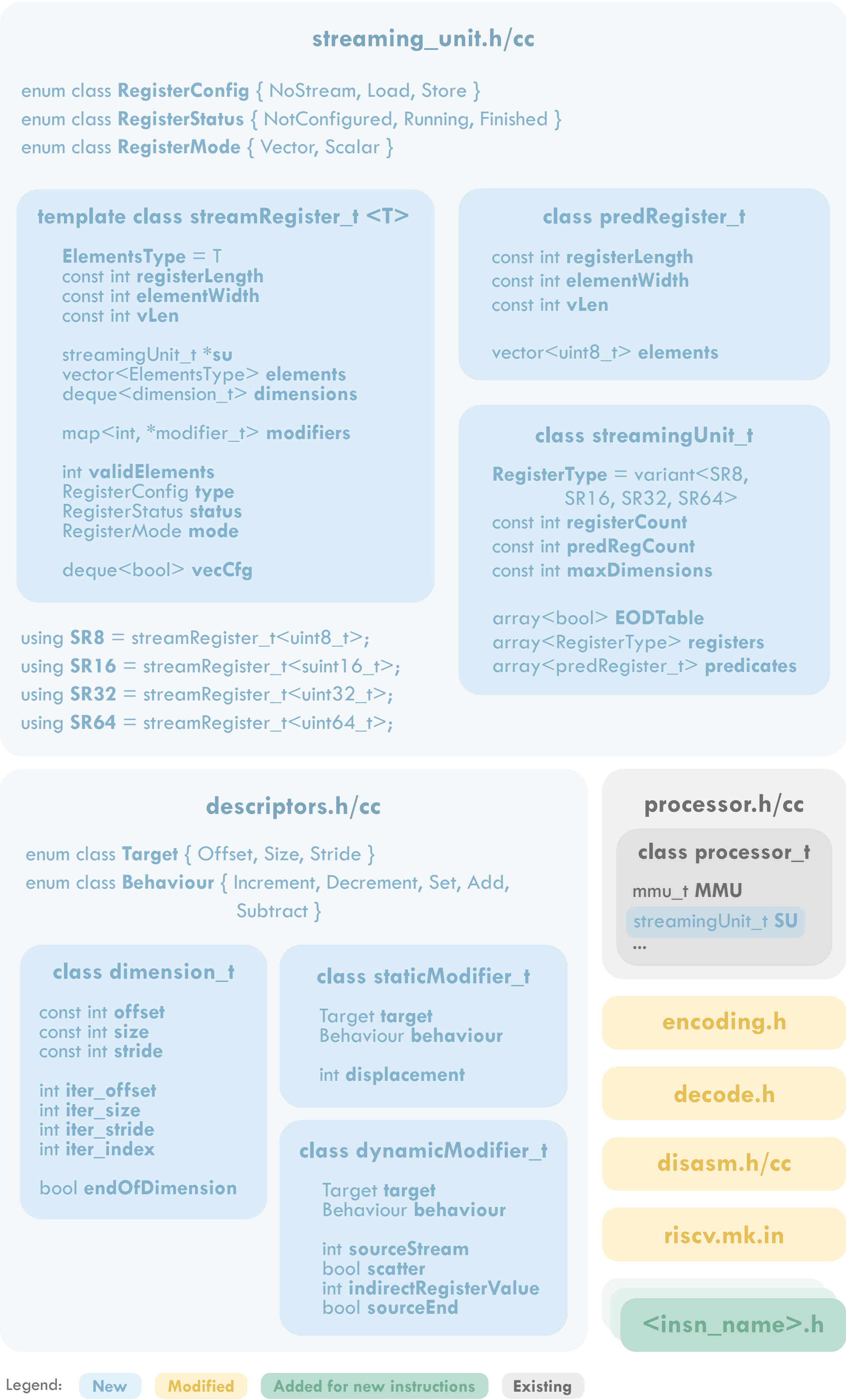


Figure 1.2: Diagram of added and modified structures and files on *Spike*.

tion formats could be recognised by *Spike's* debugger and trace generator. Code related to the disassembler is in files `regnames.cc` and `disasm.cc`, both in the `disasm` directory, and `disasm.h`, which is in the `riscv` folder. These changes are detailed in Section 1.5.

1.3 Streaming Simulation Infrastructure

In order to add UVE to *Spike*, the key necessary addition to the simulator is a set of mechanisms that emulate the SE, responsible for streaming operations. As such, the focal component of the new simulation framework is the Streaming Unit (SU), a new class that has access to the streaming and predicate registers. This unit mimics some parts of the original SE [3], specifically the *Stream Tables* and the *Stream Processing Modules* (see Figure 1.3). Each UVE register may or may not be associated with a stream, and this module is responsible for the implicit loading and storing of data, as well as the iteration of the streams (by the *Address Generator*). For the desired functional evaluation, the *Load/Store FIFOs*, the Stream Configuration Reorder Buffer (SCROB), and the *Stream Scheduler*, represented in gray in Figure 1.3, were not needed, as streams are iterated as they are being consumed, with each computation instruction triggering the iteration of the source streams (implicit loading) and the destination streams (implicit storing). The resulting elements are immediately placed in the associated registers and the *End Of Dimension* flags are updated and saved. The iteration and address generation parts work very similarly to the proposed configuration and are implemented in a different class, *Dimension*, which has access to the *Modifier* class, where static and dynamic modifiers are implemented. Each streaming register, when associated with a stream, is therefore also associated with *n* dimensions and respective modifiers if such is the case.

1.3.1 Stream Iteration and Load/Store Mechanisms

The most important part of the streaming process is the implicit loading/storing of new elements. This was the process that required the most changes to *Spike* to correctly implement and that allowed for the identification of some issues in the old specification. It should be noticed that the current *Spike* implementation works sequentially and not in parallel, which means that these operations are not performed while other instructions are being executed, as would be expected in a real processor. However, that is not necessary for an ISS to perform as desired. With this in mind, a simplistic view of this process is described in the flowchart of Figure 1.4.

As a first approach to the implementation of the stream iteration and element load operations, after the stream configuration was complete, the SU immedi-

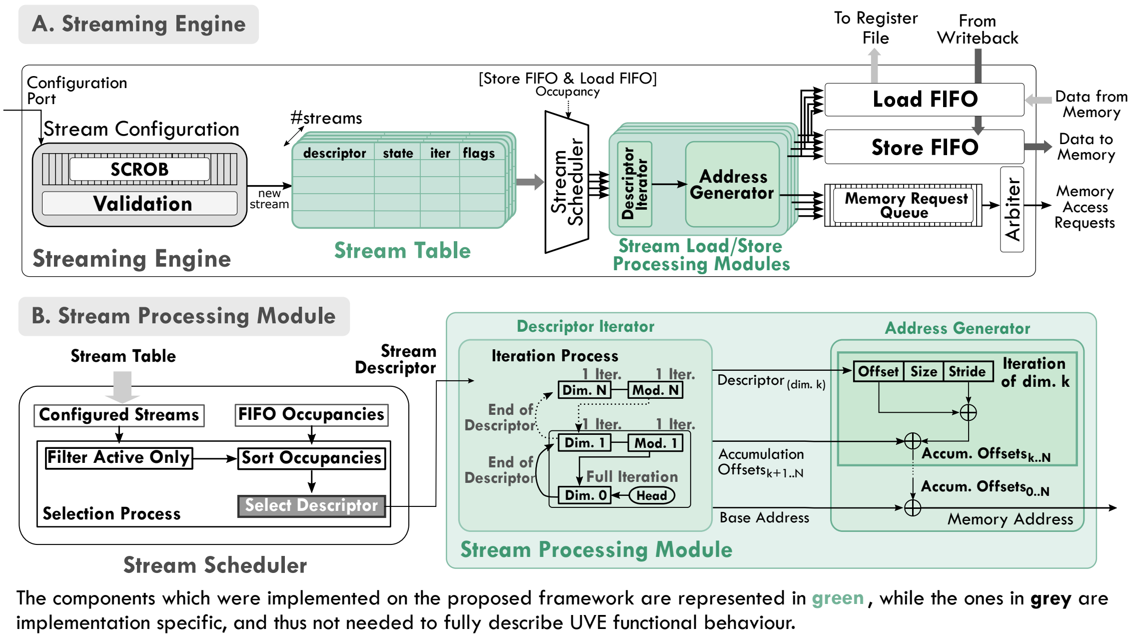


Figure 1.3: (A) Streaming Engine and (B) Stream Processor Module proposed in [3], now emulated on *Spike*.

ately loaded the first elements to the associated register. Then, each computation instruction directly read its operands and, once they had been consumed, the stream was iterated and new values were loaded to the register, a similar process to what was specified and implemented on *gem5*. However, this highlighted a memory coherence problem despite not having *Load/Store First-In, First-Outs (FIFOs)*, because the memory was accessed before a consuming instruction was executed, leaving plenty of time for other instructions to make changes to the elements in memory which were not reflected in the register. Although it is a different issue from the one identified in the originally proposed SE, this was what led to its identification, one of the many instances throughout this work where the implementation of the extension has led to the identification of issues in the specification. While in real hardware a solution for this issue will require more complex changes to the SE, it was noticed that it is necessary to delay the register load operation until the consuming instruction is executed. This is the solution that was implemented and that is currently in use: each instruction that takes a load stream as an operand starts by requesting the SU to load elements from memory to the register.

As previously detailed, the *Stream Processing Modules* are responsible for the iteration and flag setting of each stream. This process was implemented in the SU, at the register level, accessing each dimension of the configured pattern to perform necessary checks and computations. As such, several methods were implemented, namely the ones responsible for offset generation, available in Listing 1.1.

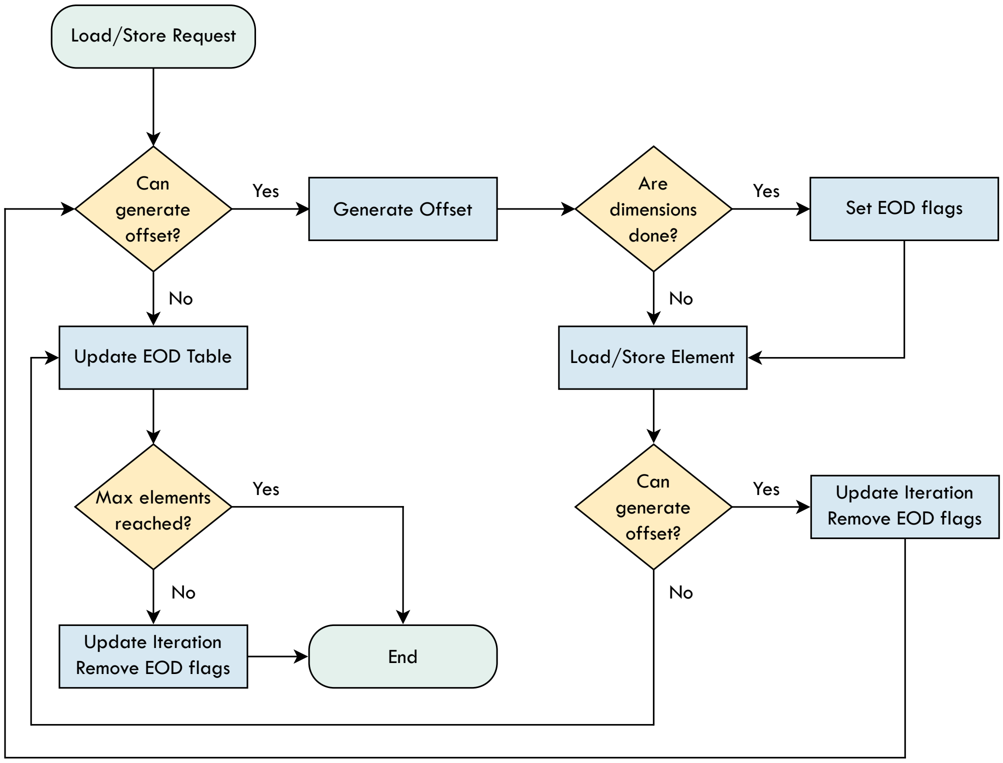


Figure 1.4: Flowchart of a high-level overview of the loading and storing of elements to/from a stream, as implemented on *Spike*.

According to the UVE specification, each register can hold values of four different widths (*byte*, *half-word*, *word* and *double-word*). As such `streamRegister_t` was implemented as a template class, allowing for type flexibility. Because of this, *variants* are often used, namely in the `streamingUnit_t` class, to be able to have an array of registers (emulating a *Register File*) of different and unknown types. Accordingly, to access a register the `std::visit()` *callable* must be used[1]. In contrast, the `predRegister_t` class is a regular one because predicates are always composed of *bytes*.

The `streamRegister_t` class contains a *vector* of `dimension_t` objects, each corresponding to a configured dimension of the memory pattern of the stream, in case the register is configured as `Load` or `Store`. When no stream is associated with a register, it is configured as `NoStream`, and none of these attributes are used, it simply saves values in the `elements` vector. This class also has a structure holding modifiers, which are mapped to the dimension they are associated with. Lastly, `vecCfg` is a mask that indicates which dimensions are vector coupled. This functionality is further explained in Section 2.3.1.

The `streamRegister_t<T>::generateAddress()` method is responsible for

[1]Documentation on these C++ utilities can be found at `https://en.cppreference.com/w/cpp/utility/variant`

the accumulation of all offsets calculated per dimension, as well as the setting of the End-of-Dimension (EOD) flag. Other methods used in this piece of code have very straightforward implementations, returning exactly what is expected from their names. The variables used by `dimension_t::calcAddress()` are attributes of this class and are updated during the stream iteration process (*Update Iteration* in Figure 1.4):

- `iter_index` is incremented by 1 after each iteration, and is reset to 0 when the EOD flag is reset, signaling the start of a new full iteration of the dimension.
- `iter_size` is the number of elements in the dimension and is set during the stream configuration process. It is used as the upper limit for the `iter_index` iterator.
- `iter_size`, `iter_stride`, and `iter_offset` are set during the stream configuration process and are only changed during the dimension iteration if a modifier is applied.

```cpp
size_t dimension_t::calcOffset(size_t width) const {
    return iter_offset + iter_stride * iter_index * width;
}

template <typename T>
size_t streamRegister_t<T>::generateAddress() {
    /* Result will be the final accumulation of all offsets calculated per dimension */
    size_t init = 0;
    int dimN = 0;

    return std::accumulate(dimensions.begin(), dimensions.end(), init, [&](size_t acc, Dimension &dim) {
        if (dim.isLastIteration() && isDimensionFullyDone(dimensions.begin(), dimensions.begin() + dimN)) {
            dim.setEndOfDimension(true);
        }
        ++dimN;
        return acc + dim.calcAddress(elementWidth);
    });
}
```

Listing 1.1: Offset computation C/C++ code.

It is assumed that the *offset* of each dimension is already the value in bytes, as it can be the base address of a descriptor. The remaining values correspond to an element count and must be multiplied by the element width, which is passed as an argument to methods that require it, as dimensions and modifiers have no information about the element width of the stream they are associated with.

Before an *address* is generated, a check is performed, as two situations prevent the SU from generating a new load/store *address*:

- The last iteration of the outermost dimension has been reached, signaling the End-of-Stream (EOS) flag. In this case, the *status* of the register is set to *finished* and its *type* is set to *NoStream*, as the stream has ended and it can be used as a regular vector register again.
- A vector coupled dimension has its EOD flag set, signaling the end of the dimension. In this case, iteration can only resume once the EOD flag is reset, which is done by a new iteration of the stream.

Finally, throughout the iteration of a stream, if present, modifiers are also applied. Dynamic and static modifiers are applied in different moments, as the former must be applied before the start of a dimension, while the latter are applied after a dimension ends. Furthermore, new scatter-gather descriptors were added to the specification and were implemented in the simulator. These behave differently from previously existing modifiers and are therefore also applied in different moments. Their functioning and possible applications are detailed in Section 2.1.3.

1.3.2 Stream Table

In the proposed simulation environment, the *Stream Table* is not a single structure. Instead, the information it keeps is divided into various attributes of the SU and its registers. Besides variables that have information about its state and configuration (*type*, *status*, and *mode*), or if a register is associated with a stream, it has a list of dimensions that build the stream memory access pattern, which each have an EOD flag, `bool endOfDimension` of `dimension_t`, as seen in Figure 1.2. As indicated in the flowchart of Figure 1.4, these flags are set and reset during the iteration process. This means that they can be set and reset several times during the execution of a single instruction, as a dimension may come to an end while there is still space for more elements in the vector register, in which case the iteration process continues and a new dimension is processed, unless the one that ends is configured as vector coupled. However, if a dimension still came to an end, that information cannot be lost, or branch instructions will not be able to capture the EOD signal. As such, the EOD flags are saved in a structure called `EODTable`, which belongs to the SU. This 2D array is responsible for saving all EOD flags for every stream and is also updated during the iteration process. These updates are performed in a way that, in case a given dimension ends, this signal is saved in the table before a new iteration, which inherently resets all EOD flags. This way, they are not lost and can be used by branch instructions, which access `EODTable` instead of the registers.

1.4 Instruction Implementation

Following the standard implementation of instructions on *Spike*, each new instruction was implemented in its separate file. Each instruction has a corresponding *header* file in the `riscv/insns` folder. While compiling the simulator, these files will be used to create copies of the `riscv/insn_template.cc` file for each instruction, responsible for the generation of the various versions of the instruction (e.g., 32/64 bit). The obvious implication is that the developed code for an instruction exists inside an external function, therefore header file inclusion is not allowed and only some variables are accessible, namely the processor, the instruction object, and the Program Counter (PC). It is through the processor that each instruction can access the MMU, as well as the SU and its registers. The instruction that is being executed, an `insn_t` object, has access to the operand decoding functions, and the PC is mainly used in branching instructions.

Furthermore, predication support was developed at the instruction level, which means that the predicate values never reach the SU, for simplicity. A predicate register has a fixed vector size of 64 *bytes*, and a predicate is thus evaluated according to the element width of the instruction's source operands. As a result, in each predicated instruction the predicate register is read for each active lane, and the operation is only performed if it evaluates to 1, as stated by the ISA specification [3].

1.4.1 Operand Decoding

To execute an instruction, the simulator must first be able to decode its arguments. For this purpose, decoding functions for each operand type were created, according to the ISA encoding. These functions, divided into different types of instructions, followed the same pattern as already existing ones (for other extensions), some even being direct copies so that there is complete flexibility in case the UVE encoding is changed. In case that happens, it is not necessary to alter every instruction if, for example, one of the source registers is differently encoded, and only the decoding function corresponding to its type requires updates. These functions are defined in file `decode.h` and some are shown in Listing 1.2.

The first three functions presented in Listing 1.2 were already implemented on the simulator. The `x()` function takes the first bit to be read and the length of the operand, both defined in the ISA encoding, which can be consulted in Appendix B and is further detailed in Chapter 2. It discards the lower `lo` bits by performing a right shift on `b`, the instruction bits. Then, it applies a mask to the result, which is obtained by subtracting 1 from the result of a left shift of 1 by `len` bits. This way, the function returns the desired operand, which is then used in the instruction's implementation. The `xs()` function is similar, but handles signed values, useful for immediate operands. These functions are used in the decoding of the UVE instructions in a very similar fashion to already existing instructions.

The least straightforward decoding function is the one that handles the immediate operand of the branching instructions. The `uve_branch_imm()` function is used to calculate the offset to be added to the program counter in case the branch is taken. Because the immediate operand is not contiguous in the instruction, the function performs a series of shifts and adds them together to obtain the final value. Because a branch can jump either to a previous or a following instruction, the immediate operand is signed, and the function `xs()` is used to obtain the sign bit in position 28. As a final note, the lower bit of the immediate operand is not used, as it is always 0. This is due to each instruction being 32 bits long (4 bytes), and the PC being incremented by 4 after each instruction is executed. Because RISC-V also has a 16-bit (2 bytes) instruction format, the PC is incremented by 2

```
// Spike defined functions
typedef uint64_t insn_bits_t;
insn_bits_t b; // Current instruction bits
uint64_t x(int lo, int len) { return (b >> lo) & ((insn_bits_t(1) << len) -
↪ 1); }
uint64_t xs(int lo, int len) { return int64_t(b) << (64 - lo - len) >> (64
↪ - len); }

// Registers for arithmetic and logic instructions
uint64_t uve_rd() { return x(7, 5); }
int64_t uve_rs1() { return x(15, 5); }
int64_t uve_rs2() { return x(20, 5); }
int64_t uve_rs3() { return x(27, 5); }
uint64_t uve_pred() { return x(25, 3); }

// Calculate offset for UVE branching instruction
int64_t uve_branch_imm() { return (x(8, 4) << 1) + (x(22, 6) << 5) + (x(7,
↪ 1) << 11) + ( xs(28, 1) << 12); }
```

Listing 1.2: Operand decoding function examples.

in that case. For compatibility, the RISC-V specification states that branch offsets are then always scaled by 2 bytes, even when no 16-bit instructions are used [6]. This means that there is no need to encode the least significant bit in the instruction and that the offset must always suffer a left shift by 1 before being added to the PC, which is performed by the decode instructions on *Spike*.

1.5 Disassembler

The disassembler is a key component of the *Spike* simulator, as it is responsible for translating the binary instructions into human-readable assembly code. This is particularly important important for the desired simulation and validation framework, as they are required for a readable debugger and trace output, which represent important tools for the development and validation of UVE. It should also be mentioned that the *Spike*-generated trace can be used in other different tools to provide other types of simulations, such as [7].

Several functions and macros were added to the source code so that new instructions and operands were correctly recognised and printed in the output trace, as well as during any debugging session. They follow the same pattern as already existing ones, for other extensions.

First, the new register types must be added, as well as the size of the register file. Register names are defined in the `regnames.cc` file, which lists the `u0-31` and `p0-15` registers in two different character arrays, `ur_name` and `pr_name`, respectively. The size of each register file (streaming and predicate) is defined in file `decode.h`.

The operand disassembling functions they take the instruction object and use the decoding functions described in Section 1.4.1 to obtain the index of the register, whose name they get from the previously defined arrays. Immediate operands in UVE are only used for branch instructions and are directly printed in the disassembled instruction, as they are not associated with any register.

To facilitate the disassembly, instructions can be grouped by types with a similar encoding (i.e., operands in the same bit positions). Each type of instruction has a corresponding function where its operands are indicated and macros are used to further simplify the code. UVE instructions are added to the disassembler as illustrated in the example from Listing 1.3, showing the entire process for the `so.a.add` instructions.

```
struct : public arg_t {
  std::string to_string(insn_t insn) const {
    return ur_name[insn.uve_rd()];
  }
} urd;
```

```
struct : public arg_t {
  std::string to_string(insn_t insn) const {
    return ur_name[insn.uve_rs1()];
  }
} urs1;

struct : public arg_t {
  std::string to_string(insn_t insn) const {
    return ur_name[insn.uve_rs2()];
  }
} urs2;

struct : public arg_t {
  std::string to_string(insn_t insn) const {
    return pr_name[insn.uve_pred()];
  }
} upred;

static void NOINLINE add_uve_arith_insn(disassembler_t* d, const char* name, uint32_t match, uint32_t mask) {
  d->add_insn(new disasm_insn_t(name, match, mask, {&urd, &urs1, &urs2, &upred}));
}

void disassembler_t::add_instructions(const isa_parser_t* isa) {
  #define DEFINE_UATYPE(code) add_uve_arith_insn(this, #code, match_##code, mask_##code);
  DEFINE_UATYPE(so_a_add_fp);
  DEFINE_UATYPE(so_a_add_us);
  DEFINE_UATYPE(so_a_add_sg);
}
```

Listing 1.3: Disassembler structures and functions for UVE *add* instructions.

1.6 Summary

In this chapter, the most relevant modifications and additions to the *Spike* simulator were described in detail. The used files and structures were listed and key algorithms that were developed to effectively emulate a Streaming Engine (SE) were described, accompanied by examples in code snippets. Also, instruction implementation and necessary decoding functions were explained. Lastly, changes made to the disassembler were shown, a key component of the framework, as it is required by the debugger and allows trace generation.

Chapter 2

Unlimited Vector Extension Specification Revision

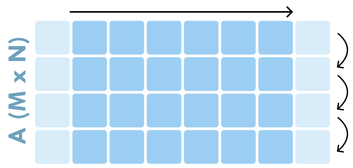
Throughout the simulator development procedure, several limitations were found in the original UVE specification, mainly related to the behaviour of specific instructions, the stream execution model, and the set of benchmark applications. Therefore, the extension was fully revised and several improvements were introduced. Furthermore, some functional aspects that previously were either only envisioned or implicit in instruction definitions, were now formalised. This chapter describes the newly proposed modifications and additions made to the UVE specification, as well as the reasoning behind them.

2.1 Stream Configuration

According to the UVE specification, registers can be associated with streams, which are managed by the Streaming Engine (SE) and are implicitly loaded/stored from/to memory when read or written to. To configure each stream, a dedicated set of instructions is used. As the instruction set was tested, some shortcomings were found in the streaming interface, which led to de modifications proposed in this section.

2.1.1 Base Address and Offset

In its first version, UVE allowed any dimension of a stream pattern to be configured with the base address of the access as its *offset*. Consequently, while generating memory addresses, the SE would add the *offsets* of the dimensions, assuming they all corresponded to a *byte* value, as that is how memory addresses are interpreted. This did not pose a problem until now, as every tested pattern had dimension *offsets* equal to zero, except the one holding the memory base address for the stream data. However, when testing a *convolution* kernel, which involves the padding of the source matrix, the descriptor encoding requires setting *offsets* to non-zero values. A simplified version of an access of this kind is represented in Figure 2.1. It was found that the SE would not correctly compute the memory addresses in this case. Because this value corresponds to an element count, it is an integer and not a *byte* value. As such, to correctly compute the memory address of each element of the stream, the SE would need to multiply it by the element size (in bytes) and add it to the base address. This multiplication was not present in the original specification and would be impossible to perform correctly unless the dimension whose *offset* corresponded to the base address was distinguishable from the others.



(a) Padded pattern.

```
for (i = 0; i < M; ++i)
    for (j = 1; j < N-1; ++j)
        A[i*N+j];
```

(b) C loop.

Dimension 1:	Dimension 2:
Offset = A	Offset = 1
Size = N-2	Size = M
Stride = 1	Stride = N

(c) Stream representation.

Figure 2.1: Padded memory access pattern example, where the *offset* is non-zero.

An initial approach to solving this issue consisted of always defining the base address as the *offset* of the first dimension to be configured. This meant that `stream-start` configuration instructions would receive a memory address that did not require any additional computation. On the other hand, `stream-append` and `stream-end` configuration instructions would always be given *offset* values that required the multiplication by the element size, which was performed immediately. This way, the SE would have correct *offset* values, without additional computation.

Eventually, because stream configuration instructions suffered restructuring, as shown in Section 2.3.1, this idea was maintained and implemented in the new *header* instructions, which replace stream start configuration instructions.

2.1.2 Scalar Streams

Although UVE is a vector extension, it is primarily a streaming extension. With this in mind, there are many complex patterns that, while not vectorisable, can still benefit from data streaming. To support these cases, both the SE and the ISA must handle scalar streams. Although it was possible to implement scalar streams by configuring the stream length to 1, scalar code was not formally supported. Three possible approaches were studied:

- Extending the register bank with new streaming scalar registers, completely separate from the streaming vector registers;
- Adding streaming support to the native RISC-V scalar registers;
- Modifying the streaming vector registers to support scalar elements, with the SE handling the scalar elements as if they were vector elements with a single element;

Of these alternatives, the last one is the simplest and requires little hardware modifications relative to the existing specification, as opposed to the other two. From the SE perspective, everything works the same, as its control is always dependent on the vector length. This puts the burden of handling scalar streams on the ISA, which must guarantee their correct configuration. As such, new information is added to the stream table: a single flag that indicates whether the stream is scalar or vectorial. Because the base ISA is scalar, UVE streaming registers are by default also scalar.

Lastly, the scalar/vector property of a register is transient from source to destination. In accordance, when reduction or scalar instructions are used, the destination vector is always scalar (despite any previous configuration). However, if the destination is a vectorial load stream, an exception must be raised, even though it should not happen in well-structured code. Furthermore, if an arithmetic or logic instruction has at least one scalar source operand, the destination register is also scalar, even if previously configured as vectorial. When the operands are vectorial, the destination register is also configured as vectorial.

2.1.3 Dimensions and Modifiers

Order of dimension configuration

One of the main changes in the new specification is related to the order in which dimensions are appended to a descriptor. To match the order in which they are presented in a typical C/C++ `for` loop, this order has been inverted. This is because the original UVE specification stated that the first dimension is the innermost one (in Listing 2.1 the first dimension of the streams described in Listing 2.2 is the one equivalent to the loop in line 3). Previously, UVE code would present dimensions in the opposite order of what is expected, by appending dimensions from the first to the last one. This often led to confusion and errors, so it was changed. While this minor change does not affect the functionality of the extension, its usability was improved. Furthermore, it led to the elimination of certain instructions, as explained in Section 2.3.1.

```
// data is a N x M matrix; cov is a M x M matrix; double_n is (double)N

for (i = 0; i < M; i++)
    for (j = i; j < M; j++) {
        cov[i * M + j] = 0;
        for (k = 0; k < N; k++)
            cov[i * M + j] += data[k * M + i] * data[k * M + j];
        cov[i * M + j] /= double_n - 1.0;
        cov[j * M + i] = cov[i * M + j];
}
```

Listing 2.1: Snippet of the *covariance* C/C++ kernel code [8].

```
# cov[i * M + j]
ss.sta.st.d         u1, cov, M, M     # D2
ss.app.mod.ofs.inc u1, M, one
ss.app.mod.siz.dec u1, M, one
ss.end              u1, zero, M, one # D1

# cov[j * M + i]
ss.sta.st.d         u2, cov, M, one # D2
ss.app.mod.ofs.inc u2, M, M
ss.app.mod.siz.dec u2, M, one
ss.end              u2, zero, M, M   # D1
```

Listing 2.2: *Covariance* kernel UVE pseudo-assembly store stream configuration of streams with two modifiers per dimension.

Multiple Modifiers per Dimension

The original specification of UVE stated that only one modifier was allowed per dimension of a stream descriptor. This was found to be a limitation when trying to implement the *covariance* kernel (specifically, in the loop presented in Listing 2.1).

For the storing of the cov matrix pattern to be correctly defined, both the *offset* and the *size* of a dimension must be modified simultaneously. This is because, in each outer loop iteration, both the writing matrix index and the number of elements to be stored are changed. This is due to the indexing dependence on the second nested loop: in each outer loop iteration, the lower limit of the iteration is increased, which means that one less iteration is performed each time. The UVE stream configuration of accesses to the cov matrix is represented in Listing 2.2, already with two modifiers appended to the outermost dimension, which will affect the innermost one.

Explicit Target Dimension

When testing a new kernel with three nested loops, shown in Listing 2.3, it was found that some patterns could not be described with the current set of modifiers. This is because a dimension may need to be updated by a modifier that is iterated with a dimension of a higher order and not the dimension directly above it. This limitation can be solved by explicitly indicating the target dimension in the modifier appending instruction. This way, the modifier is still associated with the correct dimension, iterating along with it, but can affect an arbitrary dimension of the pattern. Formally, it also moves the iteration variable updates mirror the variable update order found in the equivalent `for` loop code.

```
1 // C is an N x N matrix; A is an N x M matrix; alpha and beta are scalars
2 for (i = 0; i < N; i++) {
3     for (j = 0; j <= i; j++)
4         C[i*N+j] *= beta;
5     for (k = 0; k < M; k++)
6         for (j = 0; j <= i; j++)
7             C[i*N+j] += alpha * A[i*M+k] * A[j*M+k];
8 }
```

Listing 2.3: *SYRK* (Symetric Rank-K Update) C/C++ computation kernel code [8].

As such, in the example from Listing 2.3, to accurately describe the 3D memory access pattern of `C[i*N+j]`, the *size* of the first dimension (correspondent to the loop in line 7) must be incremented every time the third dimension (correspondent to the loop in line 3) is iterated. This is due to the innermost loop being bounded by `i`, which is incremented two loops above. The result is that while the second dimension is iterated, the size of the first one remains unchanged, as `i` is not updated in the loop on line 6.

Consequently, the `C[i*N+j]` stream access pattern needs a *size* modifier appended to the last dimension, but targeting the first one, as shown in Listing 2.4. For this to be possible, modifier configuration instructions need to be extended to include the target dimension, as presented in Section 2.3.1.

Scatter-gather Dynamic Modifiers

Although the desired behaviour and description of scatter-gather accesses were idealised, the original specification lacked dedicated support for this type of

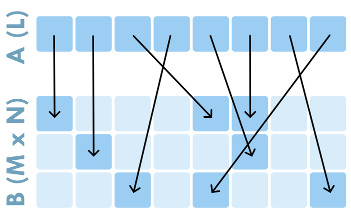
```
// C[i*N]+j
ss.sta.st.d          u3, C, N, N          # D3 (i)
ss.app.mod.siz.inc.1 u3, N, one           # Target: D1
ss.app               u3, zero, M, zero    # D2 (k)
ss.end               u3, zero, one, one   # D1 (j)
ss.cfg.vec           u3                   # vectorial stream
```

Listing 2.4: *SYRK* kernel UVE pseudo-assembly store stream configuration with explicitly defined target modifier.

memory access. Traditional modifiers are applied when the dimension they are associated with is iterated. This means that, in order to describe scatter-gather accesses, the affected dimension must be scalar and the one above it iterates the number of elements to be accessed. This is a poor way of describing accesses that could be vectorial, as it wastes the Single Instruction, Multiple Data (SIMD) capabilities of the UVE extension.

To solve this issue, a new kind of dynamic modifier was introduced. These modifiers are associated directly with the dimension that they affect, meaning that they are applied at each iteration (and each element), allowing registers to be filled with vectors. These modifiers are only defined for *offset* targets, as these are enough to perform scatter-gather accesses. However, other targets may be added in the future, if necessary. An updated example is presented in Figure 2.2. This new descriptor is configured through a new set of scatter-gather modifier instructions.

```
ss.<app/end>.ind.ofs.sg.<behaviour>
```



(a) Scatter-gather pattern

```
for (i = 0; i < L; ++i)
    B[A[i]];
```

(b) C loop

Stream A
Dimension 1:
Offset = A
Size = L
Stride = 1

Stream B (vector)
Dimension 1:
Offset = B
Size = L
Stride = 0
Scatter-gather 1:
Offset, Add, A

(c) Stream representation

Figure 2.2: Scatter-gather memory access representation.

Support for this new descriptor type was added on *Spike* and was used in one of the benchmarks, *SpMV-2*.

2.2 Predication Policies

As mentioned throughout this work, UVE supports lane control through predication. This means that in any SIMD instruction, there is the possibility of controlling which elements are operated on or not. This is achieved through the use of predicate registers, which are used to mask the operation of the instruction. The original specification simply stated that a predicated element would not take part in the computation, and the corresponding lane in the destination vector would be left unchanged. This is known as *merging* predication in Arm Scalable Vector Extension (SVE) [9] and *undisturbed* tail/mask in RISC-V RISC-V Vector Extension (RVV)[10].

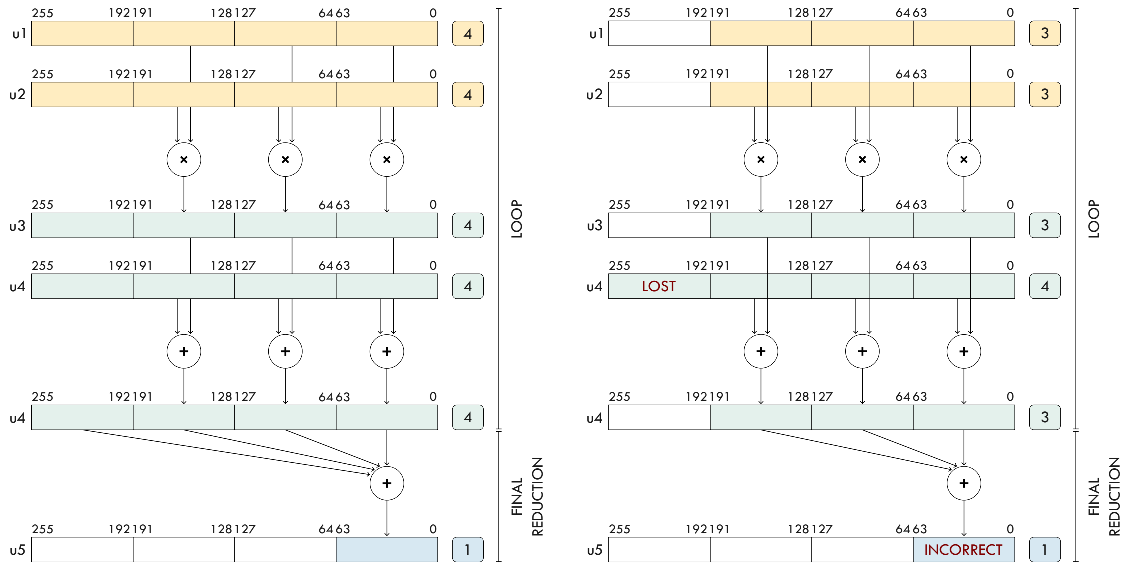
However, a different kind of predication exists, known as *zeroing* predication in SVE, where predicated lanes are set to 0 in the destination. This mode was not present in the original UVE specification, having been left for future work. Furthermore, no implicit vector predication policy was defined or specified, which was revealed to be an issue in most applications.

In particular, although considered in the original specification, an important aspect is related to what happens in the execution of instructions when non-complete vectors are used as operands. Until now, there were no explicit rules on how the number of valid elements of a vector register was managed, so it was entirely possible to have less data in a vector than the maximum allowed (e.g., a vector register configured to handle 8 elements but only 4 are valid). The initial approach was to simply take the number of valid elements of the source registers (i.e., the minimum between the valid elements from the source registers) and only operate on those. Then, this value was copied to the destination register, leaving the remaining elements unchanged. This was defined as *implicit vector predication*.

```
// u1 and u2 are load streams; u5 is a store stream

so.v.dp.d  u4, zero, p0 // fill u4 with 0s

jloop1 :
    so.a.mul.fp u3, u1, u2, p0 // u3 = u1 * u2
    so.a.add.fp u4, u4, u3, p0 // u4 = u4 + u3
    so.b.ndc.1 u1, .jloop1

so.a.adde.fp   u5, u4, p0       // reduce vector to scalar
```

Listing 2.5: Example of a reduction loop in UVE assembly code.

To understand where problems may arise, one can analyse Listing 2.5, where a common reduction loop is presented. In this case, both source operands of the multiplication are streams, which means that in each iteration, new values are implicitly loaded to these registers. However, the number of valid elements in these registers is not known, and although the vector is full in most iterations (Figure 2.3a), edge cases can occur where the vector is not full. This may lead to incorrect results, as partial sums are being stored in u4 at each iteration. The original specification states that the valid elements of the destination are set to the minimum between the valid elements of the sources. As Figure 2.3b illustrates, this results in incorrect values, as the partial sums in invalid lanes of u4 will be lost. On the other hand, if the valid elements of this register remain untouched, the result is correct, as the partial sums are not lost before the horizontal add in line 7.



(a) Computation with full vectors and correct result.

(b) Computation with incomplete vectors, which lead to incorrect result.

Figure 2.3: Reduction loops with flawed original implicit predication policy, assuming a vector length of 256 bits. Load streams are in yellow, auxiliary registers in green, and store streams in blue.

One possible solution to this issue is to redefine the implicit vector predication policy. It was defined that the only way to have invalid elements in a vector register is if is *a)* a scalar register, *b)* a load stream. In every other case, the vector register is always full. In particular, a store stream register may have valid elements with irrelevant data without issues, as the store pattern is assumed to be well-defined and those elements will not be stored in memory by the SE. This begs the question: if a vector register is the destination of an instruction with streams as sources, what results will be stored in lanes where the source register has invalid elements? The simple answer is to fill these lanes with zeroes, i.e., *zeroing* vector predication. With this policy, the reduction loop in Listing 2.5 would always produce correct results, as zero is the neutral element for addition.

This is illustrated in Figure 2.4.

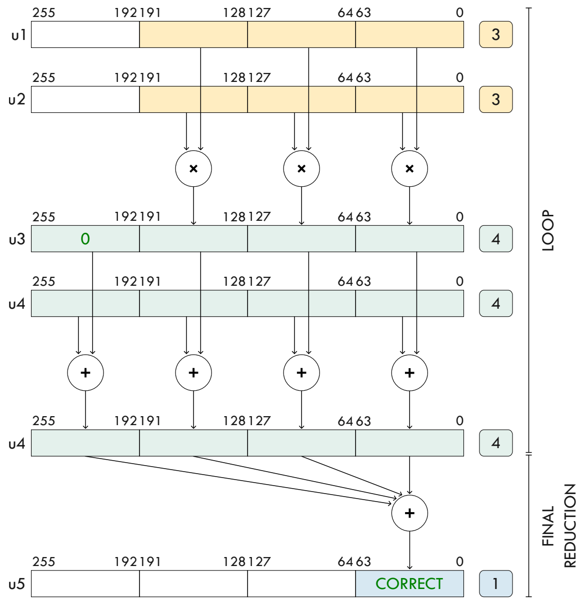


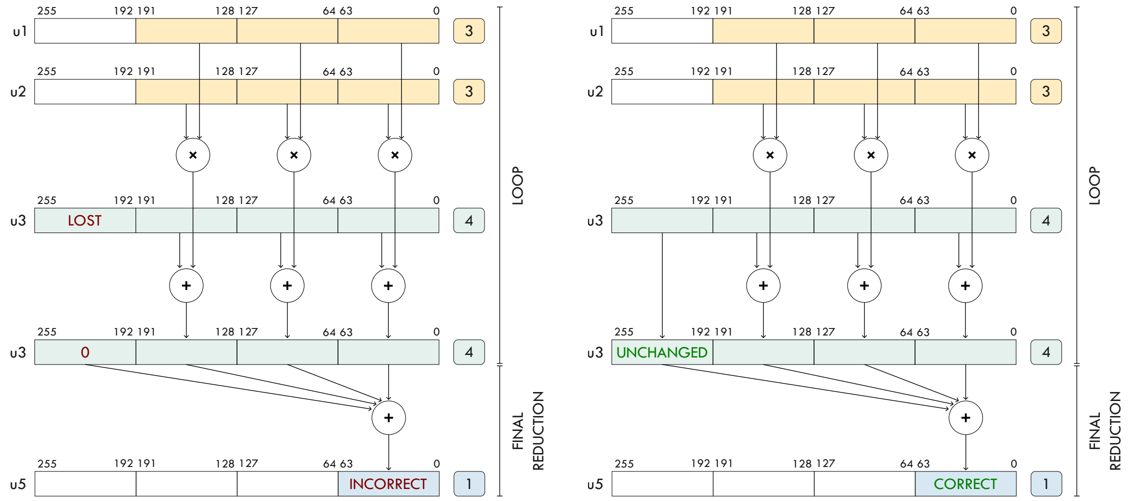
Figure 2.4: Reduction loop with *zeroing* predication.

With this modification, two new scenarios arise: what happens in accumulation instructions, and what happens if zero is not the neutral element for the operation (e.g., multiplication).

The first scenario presents itself in instructions such as `so.a.mac`, which could easily replace the two computation instructions in Listing 2.5, as it performs a multiply-accumulate operation. With *zeroing* predication, the behaviour of a reduction loop is presented in Figure 2.5a. In this case, the register that is subject to predication is itself the accumulator, which means that values must not be lost in any iteration. However, that is exactly what happens with *zeroing* predication in cases where, after full-vector iterations, the source registers have fewer elements. To solve this issue, the previous *merging* policy would be more adequate, while still marking all lanes as valid in the destination. This is illustrated in Figure 2.5b.

To handle the second scenario, two possible paths are possible: either fill invalid lanes with another value (e.g., 1 for multiplication) or reorganise the loop code and perform *merging* predication. The first approach was discarded, as a new predication mode for such specific situations would only convolute the specification. The second approach is ideal and easy to implement, as shown in Listing 2.6. By simply reorganising the multiplication instructions, the values accumulated in `u4` are never lost, and the result is always correct. This is possible whenever operations are commutative, which covers most accumulation cases. In reality, this is the order in which the multiplication would be performed in a scalar code, so it is a natural solution.

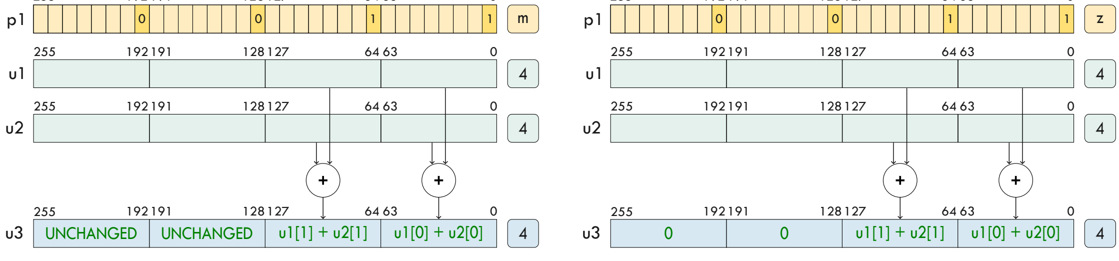
The devised solution aims to solve all the issues presented so far, by allowing



(a) Incorrect result with *zeroing* predication. (b) Correct result with *merging* predication.

Figure 2.5: Reduction loop with *multiply-accumulate* instruction and both possible predication policies.

both *zeroing* and *merging* predication policies to be used. As observed, different situations require different approaches, and a single policy would not be enough to cover all of them. Moreover, a single rule for predication type would not guarantee that the correct policy would be applied every time. In accordance, it was decided that the predication policy would be explicitly encoded in each streaming register. This way, when a register is used as a source operand, the chosen predication policy is used to determine the behaviour of the instruction in the destination. Additionally, both modes are also added to explicit instruction predicates, which means that each predicate register is also configured to use a specific predication policy, as shown in Figure 2.6. This way, the predication policy is always explicit, and thus correctly applied. As a final note, *zeroing* was the policy chosen as the default for all streaming registers, as this is the most common case observed in studied kernels. For predicate registers, the default policy is *merging*, preserving its original behaviour, but allowing for the newly added *zeroing* predication. When executing a predicated instruction with source streams, the



(a) Merging predication. (b) Zeroing predication.

Figure 2.6: Illustration of explicit predication in the `so.a.add` instruction.

predication policy of the stream prevails over the policy of the predicate register passed to the instruction.

```
// u1 and u2 are load streams; u5 is a store stream

so.v.dp.d  u4, zero, p0 // fill u4 with 0s

jloop1 :
    so.a.mul.fp u3, u1, u2, p0 // u3 = u1 * u2 (zeroing)
    so.a.mul.fp u4, u4, u3, p0 // u4 = u4 * u3
    so.b.nc u1, .jloop1

so.v.dp.d  u4, zero, p0 // fill u4 with 0s

jloop2:
    so.a.mul.fp u3, u4, u1, p0 // u3 = u4 * u1 (merging)
    so.a.mul.fp u4, u3, u2, p0 // u4 = u4 * u2 (merging)
    so.b.nc u1, .jloop1
```

Listing 2.6: Example of product accumulation loops in UVE assembly code.

2.3 Instruction Set Overview

Having laid out all the modifications to the UVE specification, this section shows how they are reflected in the encoding of the instructions. Because a thorough definition of the encoding was not provided in the original specification, this section will also serve as a reference for the encoding of instructions that did not suffer any alterations.

The encoding of these instructions is divided into several fields that are used to specify the operation to be performed and its operands. To simplify the representation of the encoding, some fields are presented with a specific notation. Additionally, the full encoding and bit fields are presented in Appendix B. This section closely follows the notation and conventions used in the official RISC-V specification.

2.3.1 Stream Configuration

Instructions responsible for the configuration of a stream, including its dimensions and modifiers, are at the core of data streaming from the programmer perspective and constitute the preamble of any streaming computational kernel.

These instructions are found in the *custom-2* opcode region of RISC-V, called *StreamSet* in the context of UVE, with the mnemonic SS. To differentiate the different types of stream configuration instructions, the *tc* field is used. The *tc* field is a 2-bit field that encodes the type of stream configuration instruction, detailed in Table 2.1.

Table 2.1: Original *tc* field encoding.

***tc* field**	**Prefix**	**Meaning**
00	APP	Append dimension/modifier to configuration
01	END	Append dimension/modifier and end configuration
10	STA	Start configuration and append dimension
11	LD/ST	1D Load/Store configuration

Table 2.2: *width* field encoding.

***width* field**	**Suffix (WTH)**	**Meaning**
00	B	Byte (8 bits)
01	H	Half-word (16 bits)
10	W	Word (32 bits)
11	D	Double-word (64 bits)

Original Specification

In the original UVE, instructions that solely configure dimensions needed four operands: the destination register, the *size* and *stride* of the dimension, and the base address or *offset*. The `ss.ld`, `ss.st`, `ss.sta.ld`, and `ss.sta.st` instructions were used to start a stream associated with the given destination register. Since they were the first (or only) instructions of the stream definition, they also required the element width to be defined in the encoding, through a suffix in the instruction name (e.g., `ss.ld.w` or `ss.sta.st.d`). This information occupied the two least-significant bits of *funct3* and is here represented by the mnemonic WTH, detailed in Table 2.2.

The `ss.app` and `ss.end` instructions appended a dimension or modifier (distinguished in the *funct3* field) to the stream configuration associated with the given destination register.

31 27	26 25	24 20	19 15	14 12	11 7	6 0
rs3	tc	rs2	rs1	funct3	vd	opcode
5	2	5	5	3	5	7
stride	11	size	base addr	LD.[WTH]	dest	SS
stride	11	size	base addr	ST.[WTH]	dest	SS
stride	STA	size	base addr	LD.[WTH]	dest	SS
stride	STA	size	base addr	ST.[WTH]	dest	SS
stride	APP	size	offset	000	dest	SS
stride	END	size	offset	000	dest	SS

Each static modifier configuration instruction took three operands: the destination register, the *size*, and the *displacement* value, whereas a dynamic modifier only required two: the destination and the source registers. The *funct3* and *l* fields encoded the type of modifier. The latter was a single bit field present in *dynamic modifier* instructions encoding, indicating whether the modifier was linked to the coupled dimension or if it required a *dimension hop (dhop)* which indicated to which dimension the modifier was linked. In the case of static modifiers, this information was encoded in the *funct3* field as well, and the only difference from typical modifiers was that the *size* operand was omitted and instead inferred from the dimension it was appended to. Furthermore, there were several different instructions for each type of modifier, with different *behaviour* and *target* values. These distinctions were made in the homonymous fields detailed in Table 2.3[1] and Table 2.4.

Table 2.3: *behaviour* field encoding.

***b* field**	**Suffix (B)**	**Meaning**
000	INC	Increment
001	DEC	Decrement
010	ADD	Add to base value
011	SUB	Subtract from base value
100	SET	Set to value

Table 2.4: *target* field encoding.

***ta* field**	**Suffix (T)**	**Meaning**
00	SIZ	Size
01	STR	Stride
10	OFS	Offset

31 27	26 25	24 22	21 20	19 15	14 12	11 7	6 0
rs3	tc	b	ta	rs1	funct3	vd	opcode
5	2	3	2	5	3	5	7
displacement	APP	B	T	size	MOD	dest	SS
displacement	END	B	T	size	MOD	dest	SS
displacement	APP	B	T	0	MODL	dest	SS
displacement	END	B	T	0	MODL	dest	SS

31	30 28	27	26 25	24 22	21 20	19 15	14 12	11 7	6 0
-	dh	l	tc	b	ta	rs1	funct3	vd	opcode
1	3	1	2	3	2	5	3	5	7
0	dimh	0	APP	B	T	src	IND	dest	SS
0	dimh	0	END	B	T	src	IND	dest	SS
0	000	1	APP	B	T	src	IND	dest	SS
0	000	1	END	B	T	src	IND	dest	SS

[1] The presented encoding of this field is updated relative to the original, which had two different encodings for INC and DEC in each type of modifier, despite having the same behaviour.

Finally, three additional configuration instructions existed, which either targeted a whole stream (`ss.cfg.ind` and `ss.cfg.mem`) or a single stream dimension (`ss.cfg.vec`). The first one was used to indicate if a stream was to be used as the source of a dynamic modifier of another stream, which in a real implementation would mean that data does not need to be moved to the CPU, never leaving the Streaming Engine (SE). The second one configured the cache-level access for that stream. The last one was used to indicate that a certain stream dimension was vector-coupled. A vector-coupled dimension stops the stream iteration once it reaches EOD, resuming once another request is made to the SE. This differs from the default behavior, which fills the register with values from the next dimension iteration. This instruction allowed for dimensions to be consumed individually, without mixing elements from other iterations to fill the vector. It was also first proposed that this instruction indicated that a stream is vectorial and not scalar, as discussed in Section 2.1.

31 27	26 25	24 18	17 12	11 7	6 0
-	tc	-	funct6	vd	opcode
5	2	7	6	5	7
0	00	0	CFG.IND	dest	SS
0	00	0	CFG.MEM[l]	dest	SS
0	00	0	CFG.VEC	dest	SS

Updates to the Specification

To accommodate for the newly proposed changes and additions, such as scatter-gather modifiers and the inversion of the dimension/modifier appending order, as well as to reduce the number of required configuration instructions, the encoding of the stream configuration instructions was modified.

Firstly, several configuration instructions were collapsed in one single *header* instruction, which is used to start a stream configuration, similarly to the `ss.sta` instructions, whose name was kept, but without a dimension or modifier. Instead, the encoding space was used to define what previously required the `ss.cfg` instructions, as well as new information about the predication mode (see Section 2.2). This was done by adding several new fields:

- ***pm* field**: a 1-bit field that indicates the predication mode of the stream. If set, the stream suffers from merging predication, otherwise, it suffers from zeroing predication, the default. If set, this field translates into the optional M suffix in the instruction name (i.e., `ss.sta.m`).
- ***vec* field**: a 1-bit field that indicates if the stream is vectorial. If set, the stream is vectorial, otherwise, it is scalar. If set, this field translates into the optional V suffix in the instruction name (i.e., `ss.sta.v`).

- ***vdim* field**: a 3-bit field that indicates the vector-coupled dimension of the stream. If no dimension is vector-coupled, this field is set to 111, as the default behaviour of the outermost dimension is similar to that of a vector-coupled dimension. A number is added to the instruction name to indicate the vector-coupled dimension (i.e., `ss.sta.v.1`), unless the field is set to 8, in which case the number is omitted.

- ***inds* field**: a 1-bit field that indicates if the stream is to be used as the source of a dynamic modifier of another stream. If set, this field translates into the optional INDS suffix in the instruction name (i.e., `ss.sta.inds`). For now, a stream can only be the source of a dynamic modifier if it is scalar, as only one value is used to modify the target per iteration, so an entire vector cannot be loaded. Improvements to this behaviour are left to future work, as the SE architecture complexity should be taken into account.

- ***mem* field**: a 2-bit field that indicates the cache-level access for that stream, from 0 (default) to 3. This field translates into the optional MEM[l] suffix in the instruction name (i.e., `ss.sta.mem2`), absent when it is 0.

While these instructions do not configure any dimension or modifier, they still configure the base address of the stream, consistent with the behaviour proposed in Section 2.1.1. Because of the removal of `ss.<ld/st>` instructions, the *tc* field encoding was updated and is summarised in Table 2.5.

Table 2.5: Updated *tc* field encoding.

***tc* field**	**Prefix**	**Meaning**
00	STA	Stream header instructions
01	APP	Append dimension/modifier
10	END	Append dimension/modifier and end configuration
11	-	Reserved

31	30	29 27	26 25	24	23 22	21 20	19 15	14 12	11 7	6 0
pm	vec	vdim	tc	inds	mem	-	rs1	funct3	vd	opcode
1	1	3	2	1	2	2	5	3	5	7
PM	V	vdim	00	INDS	MEM[l]	00	base addr	LOAD.[WTH]	dest	SS
PM	V	vdim	00	0	MEM[l]	00	base addr	STORE.[WTH]	dest	SS

While instructions that append dimensions did not suffer any alterations, modifier instructions were updated to include the *tdim* field, replacing the operand that previously indicated the *size* of the modifier. This is a 3-bit field that encodes that *target* dimension of a modifier (i.e., the dimension it modifies). This takes advantage of the fact that the *size* of the modifier is in most cases the same as the *size* of the dimension it is linked to to solve the target dimension issue highlighted in Section 2.1.3. As such, previously existing `ss.app.modl` and `ss.app.indl` were removed, since every modifier instruction now has an explicitly defined target dimension and implicit *size*. This decision was made after observing that no studied pattern required a modifier to be applied to only a few iterations of a dimension. This behaviour is directly related to the induction variable dependencies in the loops that modifiers represent. In fact, the modifier *size* field was never used on *Spike* and has since been removed. Furthermore, due to the inversion of the order in which dimensions appear, `ss.end.modl` and `ss.end.indl` instructions were removed, as they are no longer necessary.

31 27	26 25	24 22	21 20	19 18	17 15	14 12	11 7	6 0
rs3	tc	b	ta	-	tdim	funct3	vd	opcode
5	2	3	2	2	3	3	5	7
displacement	APP	B	T	0	tdim	MOD	dest	SS

Lastly, with the encoding space freed by the removal of the aforementioned instructions, it was possible to add the new scatter-gather instructions. These instructions have the same encoding as ordinary dynamic modifiers but with the *sg* field set to 1. In this case, the suffix SGI is added to the instruction name (e.g., `ss.app.sgi`). These instructions do not need a *tdim* field, as they are linked to the target dimension, as described in Section 2.1.

31	30 28	27	26 25	24 22	21 20	19 15	14 12	11 7	6 0
-	tdim	sg	tc	b	ta	vs1	funct3	vd	opcode
1	3	1	2	3	2	5	3	5	7
0	tdim	0	APP	B	T	origin	IND	dest	SS
0	0	SG	APP	B	10	origin	IND	dest	SS

2.3.2 Loop Control – Branching

These instructions control the flow of every UVE computation kernel, using EOD and EOS flags raised by the SE to branch accordingly, allowing to loop over data streams. These instructions belong to the *StreamOps* opcode region, with the mnemonic SO, and take two operands: the destination register and the offset to the target instruction, an immediate value scattered across the instruction encoding, similar to the original RISC-V *B-type* instructions [6]. The *n* field is a single bit that differentiates between dimension *complete* and *not complete* instructions (i.e., `so.b.dc.3` and `so.b.ndc.3`). Any pattern dimension can be used in the branch comparison, and its index is encoded in the 3-bit *d* field. In the case where this field is 111, the EOS flag is used to determine the branch outcome, and both the D prefix and the dimension index are omitted from the instruction name (i.e., `so.b.c` and `so.b.nc`).

31 29	28	27 22	21	20	19 15	14 12	11 8	7	6 0
funct3	imm[12]	imm[10:5]	n	d[2]	vs1	d[1:0] -	imm[4:1]	imm[11]	opcode
3	1	6	1	1	5	3	4	1	7
111	offset[12 | 10:5]		N	d[2]	src1	d[1:0]	offset[11 | 4:1]		SO

While revising the specification, it became clear that the *d* field could be encoded contiguously, as there was an unused bit at the right of *d[1:0]*. As such, the instruction was updated and *n* was also moved to the right, leading to a clearer encoding.

<table>
<tr><td>31 29</td><td>28</td><td>27 22</td><td>21</td><td>20</td><td>19 15</td><td>14 12</td><td>11 8</td><td>7</td><td>6 0</td></tr>
<tr><td>funct3</td><td>imm[12]</td><td>imm[10:5]</td><td>-</td><td>n</td><td>vs1</td><td>d</td><td>imm[4:1]</td><td>imm[11]</td><td>opcode</td></tr>
<tr><td>3</td><td>1</td><td>6</td><td>1</td><td>1</td><td>5</td><td>3</td><td>4</td><td>1</td><td>7</td></tr>
<tr><td>111</td><td colspan="2">offset[12 | 10:5]</td><td>0</td><td>N</td><td>src1</td><td>d</td><td colspan="2">offset[11 | 4:1]</td><td>SO</td></tr>
</table>

As a final note, some initially proposed predicate-based branch instructions were removed, as they were deemed unnecessary and redundant, given the new predication policies and valid element settings on vector registers.

2.3.3 Lane Control – Predication

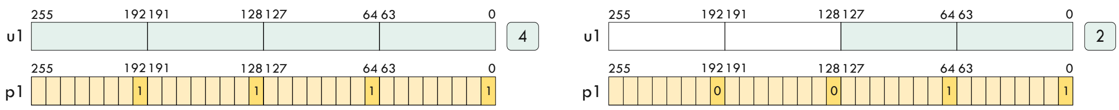
One key aspect of UVE is the ability to predicate instructions, which is done through the use of predicate registers that need to be configured before use. Belonging to the *StreamOps* opcode region, these instructions are responsible for the population of predicates.

Original Specification

Most instructions from the original specification retain their functionality in the new specification. The most simple predicate instructions, `so.p.zero` and `so.p.one`, simply take the destination register and set it to all zeroes or ones, respectively. They can also be predicated themselves, which means that a *predicate* is an operand of these instructions. The `so.p.not` instruction takes two operands: the destination register and the source register, and sets the former to the bitwise negation of the latter. The `so.p.mv` and `so.p.mvt` instructions take the same operands and move the source predicate into the destination, either directly or reversed, respectively.

There is also another instruction that takes an additional argument, a source vector register, and creates a mask from all its valid elements, which is then stored in the destination predicate register. This instruction is called `so.p.vr` and its behaviour is illustrated in Figure 2.7. This instruction can also be predicated, in which case the destination predicate register is always set to zero in lanes where the predicate mask is null.

```
so.p.vr p1, u1, p0 #dest, src1, pred
```



(a) Full source vector.

(b) Incomplete source vector.

Figure 2.7: Illustration of `so.p.vr` instruction with different source vectors, assuming true instruction predicate (`p0`).

31 28	27 25	24 20	19 15	14 12	11 7	6 0
funct4	ps3	-	vs1	funct4	pd	opcode
4	3	5	5	4	4	7
1000	pred	0	0	ZERO	dest	SO
1000	pred	0	0	ONE	dest	SO
1000	pred	0	src1	VR	dest	SO

31 28	27 25	24 19	18 15	14 12	11 7	6 0
funct4	ps3	-	ps1	funct4	pd	opcode
4	3	6	4	4	4	7
1000	pred	0	src1	NOT	dest	SO
1000	pred	0	src1	MV	dest	SO
1000	pred	0	src1	MVT	dest	SO

Originally, predicates could also be generated from vector-vector and vector-scalar comparisons, which took an extra operand for the second source register. Additionally, because arithmetic operations were involved, the type of computation was encoded in the *funct3* field, according to Table 2.6. It should be noted that two versions of each instruction were available, one that took two vector registers and performed an element-wise comparison, and one that took a scalar register whose value was compared to each value in the *src1* vector register.

Table 2.6: *fps* field encoding.

***fps* field**	**Suffix (FPS)**	**Meaning**
00	US	Unsigned integer operation
01	FP	Floating-point operation
10	SG	Signed integer operation
11	-	Reserved

31 28	27 25	24 20	19 15	14 12	11 7	6 0
funct4	ps3	vs2	vs1	funct4	pd	opcode
4	3	5	5	4	4	7
1000	pred	src2	src1	EGT.[FPS]	dest	SO
1001	pred	src2	src1	EQ.[FPS]	dest	SO
1001	pred	src2	src1	LT.[FPS]	dest	SO

31 28	27 25	24 20	19 15	14 12	11 7	6 0
funct4	ps3	rs2	vs1	funct4	pd	opcode
4	3	5	5	4	4	7
1000	pred	src2	src1	EGTS.[FPS]	dest	SO
1001	pred	src2	src1	EQS.[FPS]	dest	SO
1001	pred	src2	src1	LTS.[FPS]	dest	SO

Lastly, because predicates may result from arithmetic operations on vectors with an arbitrary element width, despite not having a defined data type themselves, conversion instructions are necessary to adapt a previously defined predicate to be applied to other vector operands of a different data type. This conversion was done through the use of the `so.p.cv` instruction, which took the source predicate and the destination register as operands. It also encoded the element width to which the source predicate was to be converted in the *width* field, according to Table 2.2, which was traduced into a suffix in the instruction name (e.g., `so.p.cv.b`).

31 28	27 25	24 22	21 20	19	18 15	14 11	10 7	6 0
funct4	ps3	-	width	-	ps1	funct4	pd	opcode
4	3	3	2	1	4	4	4	7
1000	0	0	WTH	0	src1	CV	dest	SO

Updates to the Specification

Starting with the conversion instructions, it was noticed during the development of this work that these instructions lacked information about the data type used to configure the source register. While this is not a problem in vector conversion instructions (see Section 2.3.4), predicate registers do not contain any information about the data type, contrary to UVE vector registers. To perform a conversion, the data width of the source predicate must be known, as it determines how the source predicate is interpreted and converted. As such, the instruction now features two width fields, *dw* and *sw*, for the destination and source predicates, respectively. The behaviour of these instructions is exemplified in Figure 2.8.

`so.p.cv.d.w p2, p1` `so.p.cv.w.d p2, p1`

31 28	27 25	24	23 22	21 20	19	18 15	14 11	10 7	6 0
funct4	-	pm	dw	sw	-	ps1	funct4	pd	opcode
4	3	1	2	2	1	4	4	4	7
1000	0	PM	WTH	WTH	0	src1	CV	dest	SO

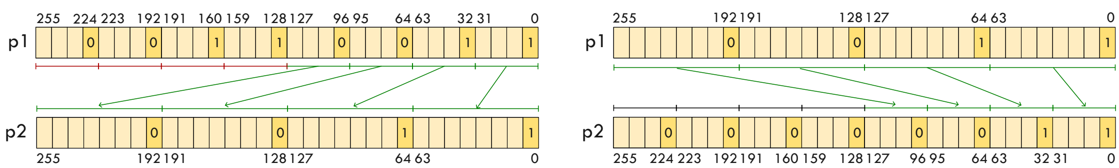
Additionally, a change that is present in every predicate instruction is the inclusion of the *pm* field, which indicates the predication mode of the stream, similar to the stream configuration instructions in Section 2.3.1, and described in Section 2.2. In this case, because merging is the default behaviour, if the bit is set then zeroing is chosen and a suffix is added to the instruction name (i.e., `so.p.zero.z`).

31 28	27 25	24	23 20	19 15	14 12	11 7	6 0
funct4	ps3	pm	-	vs1	funct4	pd	opcode
4	3	1	4	5	4	4	7
1000	pred	PM	0	0	ZERO	dest	SO
1000	pred	PM	0	0	ONE	dest	SO
1000	pred	PM	0	src1	VR	dest	SO

31 28	27 25	24	23 19	18 15	14 12	11 7	6 0
funct4	ps3	pm	-	ps1	funct4	pd	opcode
4	3	1	5	4	4	4	7
1000	pred	PM	0	src1	NOT	dest	SO
1000	pred	PM	0	src1	MV	dest	SO
1000	pred	PM	0	src1	MVT	dest	SO

The comparison instructions were also revised, as there was no available encoding space for the required *pm* field. Upon analysis of the instruction set, scalar comparisons were deemed dispensable, as they can be replaced with an `so.v.mvsv` instruction followed by a regular vector predicate comparison. As such, the scalar comparison instructions were removed, freeing up a bit 11, previously belonging to *funct4* field. The naming of the *greater or equal* comparison instruction was also changed to more closely resemble typical ISA naming conventions.

31 28	27 25	24 20	19 15	14 12	11	10 5	6 0
funct4	ps3	vs2	vs1	funct3	pm	pd	opcode
4	3	5	5	3	1	4	7
1000	pred	src2	src1	GE.[FPS]	PM	dest	SO
1001	pred	src2	src1	EQ.[FPS]	PM	dest	SO
1001	pred	src2	src1	LT.[FPS]	PM	dest	SO



(a) Element widening (word to double-word). (b) Element narrowing (double-word to word).

Figure 2.8: Illustration of `so.p.cv` instruction with different source and destination widths.

2.3.4 Vector Manipulation

These instructions allow for the transferring of data between vector registers, as well as the conversion between vectors of different data types. They belong to the *StreamOps* opcode region, with the mnemonic SO, and remain unchanged from the original specification, apart from the removal of some instructions. Vector load/store instructions were deemed unnecessary, as they were simple non-streaming vector memory access instructions, which can be easily replicated with linear streams. Moreover, move instructions had *"no stream"* variants, which did not trigger the iteration of source/destination streams, as UVE instructions do by default when reading or writing from/to a register associated with a stream. Because data streaming is the main focus of UVE, these instructions were deemed deprecated for now, simplifying the instruction set.

The remaining instructions take up to three operands: the destination, source, and predicate registers. Two simple *move* instructions are available, `so.v.mv`, which moves the source vector into the destination, and `so.v.mvt` which first reverses the vector. To perform vector-to-scalar and scalar-to-vector moves, `so.v.mvvs` and `so.v.mvsv` are available, respectively. It should be noted that in these cases, the destination register remains configured as scalar. Another instruction is available to create a vector from a scalar, `so.v.dp`, which broadcasts the scalar value to the destination vector register, which is configured as vectorial.

31 27	26 23	22 20	19 15	14 12	11 7	6 0
funct5	funct4	ps2	vs1	funct3	vd	opcode
5	4	3	5	3	5	7
10101	MV	pred	src1	0	dest	SO
10101	MVT	pred	src1	0	dest	SO

31 27	26 23	22 20	19 15	14 12	11 7	6 0
funct5	funct4	ps2	vs1	funct3	rd	opcode
5	4	3	5	3	5	7
10101	DP	pred	src1	WTH	dest	SO

31 27	26 23	22 20	19 15	14 12	11 7	6 0
funct5	funct4	ps2	vs1	funct3	rd	opcode
5	4	3	5	3	5	7
10101	MVVS	0	src1	0	dest	SO

31 27	26 23	22 20	19 15	14 12	11 7	6 0
funct5	funct4	ps2	rs1	funct3	vd	opcode
5	4	3	5	3	5	7
10101	MVSV	0	src1	WTH	dest	SO

To be able to convert the elements of a vector to different data types, the `so.v.cv` instruction is available, which takes the source vector and the destination register as operands. This instruction performs the necessary narrowing or widening of the source vector to fit the destination element width, which is encoded in the *width* field, according to Table 2.2. These operations have a similar behaviour as predicate conversion ones, illustrated in Figure 2.8. However, the behaviour of the block of data that would be lost is not clearly defined yet. It was initially proposed that the elements in the source register were implicitly right-shifted after the conversion and, if associated with a stream, new values were loaded to the now empty lanes. However, this behaviour has not been validated and further testing is necessary, which is left for future work.

31 — 27	26 — 23	22 — 20	19 — 15	14 — 12	11 — 7	6 — 0
funct5	funct4	ps2	rs1	funct3	vd	opcode
5	4	3	5	3	5	7
10110	CV.[FPS]	0	src1	WTH	dest	SO

2.3.5 Vector Control

In the *StreamOps* encoding space, two instructions to configure the vector length are available, both to set and get the value from the `VLEN` Control Status Register (CSR), `so.c.setvl` and `so.c.getvl`, respectively. The `so.c.setvl` instruction takes the source register as an operand, which contains the new value for `VLEN`, in bytes, and the destination register, which is used to store the new value of `VLEN`. This value is the minimum between the requested value and the maximum allowed, `VLMAX`, similar to the equivalent RVV instructions. After the execution of the instruction, the `VLEN` CSR is updated with the new value, and every vector register is configured to the new length.

31 — 27	26 — 20	19 — 15	14 — 12	11 — 7	6 — 0
funct5	-	rs1	funct3	rd	opcode
5	7	5	3	5	7
10110	0	src1	SETVL	dest	SO
10110	0	src1	GETVL	dest	SO

The remaining instructions allow for explicit control over the streams, such as suspension and resuming, as well as definite breaking of the stream: `so.v.suspd`, `so.v.resum`, and `so.v.break`, respectively. Additionally, the `so.v.vload` and `so.v.vstor` instructions are available to load and store data from and to suspended streams.

31 — 27	26 — 15	14 — 12	11 — 7	6 — 0
funct5	-	funct3	vd	opcode
5	12	3	5	7
10110	0	SUSPD	dest	SO
10110	0	RESUM	dest	SO
10110	0	BREAK	dest	SO
10110	0	VLOAD	dest	SO
10110	0	VSTOR	dest	SO

2.3.6 Arithmetic and Logic Instructions

The instructions that perform SIMD arithmetic and logic operations on UVE vector registers (part of the *StreamOps* encoding space) are the most common in computation loops. Although the element width is encoded in each register, the data type as seen in Table 2.6 is not, so it must be present in each arithmetic instruction. Simple instructions take two source operands and a destination, as well as a predicate register, and perform an element-wise operation. Particularly, `so.a.mac` is a multiply-accumulate instruction, which multiplies the source operands and adds the result to the value already in the destination register.

There is also an instruction to obtain the absolute value of every vector element, which takes only one source operand, `so.a.abs`, which does not have an *unsigned* version, as it can only operate on *signed* types. The `so.a.inc` and `so.a.dec` instructions increment or decrement its only source operand, respectively, storing the result in the destination register.

31–28	27–25	24–20	19–15	14–12	11–7	6–0
funct4	ps3	vs2	vs1	funct3	vd	opcode
4	3	5	5	3	5	7
0000	pred	src2	src1	ADD.[FPS]	dest	SO
0000	pred	src2	src1	SUB.[FPS]	dest	SO
0001	pred	src2	src1	MUL.[FPS]	dest	SO
0001	pred	src2	src1	DIV.[FPS]	dest	SO
0011	pred	0	src1	ABS.[FP]	dest	SO
0011	pred	src2	src1	MAC.[FPS]	dest	SO
0100	pred	src2	src1	MIN.[FPS]	dest	SO
0100	pred	src2	src1	MAX.[FPS]	dest	SO
0110	pred	0	src1	INC.[FP]	dest	SO
0110	pred	0	src1	DEC.[FPS]	dest	SO

Some reduction instructions are also present in this extension, which instead of taking two source operands and performing an element-wise operation, take only one source operand and a destination, and perform a reduction operation on the vector elements, storing the scalar result in the destination stream register, which is therefore configured as scalar.

Addition is a special case in this set of instructions, as it is the only reduction operation that can accumulate the result with the destination register (`so.a.adde.acc`), which can also be a regular RISC-V scalar register (`so.a.adds.acc` and `so.a.adds.acc`). In particular, in the floating-point variants of these instructions (`so.a.adds.fp` and `so.a.adds.acc.fp`), a RISC-V floating-point register is required as the destination. This versatility is useful for accumulating the result of a reduction operation in a loop, as it can be done in a single instruction, something very common in computation kernels (e.g., *SGD, GEMVER, covariance*).

31–28	27–25	24–20	19–15	14–12	11–7	6–0
funct4	ps3	acc	vs1	funct3	vd	opcode
4	3	5	5	3	5	7
0010	pred	ACC	src1	ADDE.[FPS]	dest	SO
0101	pred	0	src1	MINE.[FPS]	dest	SO
0101	pred	0	src1	MAXE.[FPS]	dest	SO

31–28	27–25	24–20	19–15	14–12	11–7	6–0
funct4	ps3	acc	vs1	funct3	fd	opcode
4	3	5	5	3	5	7
0010	pred	ACC	src1	ADDS.[FPS]	dest	SO

Lastly, several SIMD logic instructions are available, which perform bit-wise operations on the source operands, storing the result in the destination register. In detail, both logical (zero-extending) and arithmetic (sign-extending) shift right instructions are available, as seen in RVV, (`so.a.srl`) and `so.a.sra`, respectively. The shift-amount value is encoded in the second source operand, which can be a vector or regular RISC-V register.

31–28	27–25	24–20	19–15	14–12	11–7	6–0
funct4	ps3	vs2	vs1	funct3	vd	opcode
4	3	5	5	3	5	7
1100	pred	src2	src1	NAND	dest	SO
1100	pred	src2	src1	AND	dest	SO
1100	pred	src2	src1	OR	dest	SO
1100	pred	src2	src1	NOR	dest	SO
1100	pred	src2	src1	XOR	dest	SO
1100	pred	0	src1	NOT	dest	SO
1101	pred	src2	src1	SLL	dest	SO
1101	pred	src2	src1	SRL	dest	SO
1101	pred	src2	src1	SRA	dest	SO

31–28	27–25	24–20	19–15	14–12	11–7	6–0
funct4	ps3	rs2	vs1	funct3	vd	opcode
4	3	5	5	3	5	7
1101	pred	src2	src1	SLLS	dest	SO
1101	pred	src2	src1	SRLS	dest	SO
1101	pred	src2	src1	SRAS	dest	SO

2.4 Summary

This chapter detailed how some issues were found, through tests performed on the simulator presented in Chapter 1, which revealed caveats in the original specification. Furthermore, while attempting to describe more complex patterns in new computation kernels, several features were found to be missing. These include:

- The need for multiple modifiers per dimension;
- The necessity of explicitly indicating the target dimension of a modifier;
- Scatter-gather dynamic modifiers;
- The need for scalar streams;
- New predication policies.

Then, the new ISA encoding was presented, with all the necessary changes to support these features. Most of these features were also implemented on the simulator, which was used to test the new specification.

It is important to note that the new specification is not yet final, and as more complex patterns are tested, new features may be added. However, the current specification is already a significant improvement over the original one, and it represents the base for the first stable release of the UVE extension. Moreover, support for some of the new features, such as the new scatter-gather dynamic modifiers, has yet to be added to the microarchitecture of UVE and tested in hardware. While this is outside the scope of this work, it is expected that the new features will be implemented in the future, and hardware constraints may lead to a new revision of the extension.

References

[1] RISC-V. *RISC-V ISA Simulator*. C/C++. Dec. 2021. URL: `https://github.com/riscv-software-src/riscv-isa-sim`.

[2] Benjamin W. Mezger, Douglas A. Santos, Luigi Dilillo, Cesar A. Zeferino and Douglas R. Melo. 'A Survey of the RISC-V Architecture Software Support'. In: *IEEE Access* 10 (2022), pp. 51394–51411. DOI: `10.1109/ACCESS.2022.3174125`.

[3] Joao Mário Domingos, Nuno Neves, Nuno Roma and Pedro Tomás. 'Unlimited Vector Extension with Data Streaming Support'. In: *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. Valencia, Spain: IEEE, June 2021, pp. 209–222. ISBN: 978-1-66543-333-4. DOI: `10.1109/ISCA52012.2021.00025`. URL: `https://ieeexplore.ieee.org/document/9499750/`.

[4] Alec Roelke and Mircea R Stan. *RISC5: Implementing the RISC-V ISA in gem5*. 2017.

[5] RISC-V. *RISC-V Opcodes*. URL: `https://github.com/riscv/riscv-opcodes`.

[6] Andrew Waterman and Krste Asanovic. *The RISC-V Instruction Set Manual - Volume I: Unprivileged ISA*. 2019. URL: `https://github.com/riscv/riscv-isa-manual/`.

[7] *Olympia*. C++. Feb. 2024. URL: `https://github.com/riscv-software-src/riscv-perf-model`.

[8] Louis-Noël Pouchet. *PolyBench/C*. 2012. URL: `https://web.cs.ucla.edu/~pouchet/software/polybench/`.

[9] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanael Premillieu, Alastair Reid, Alejandro Rico and Paul Walker. 'The ARM Scalable Vector Extension'. In: *IEEE Micro* 37.2 (2017), pp. 26–39. DOI: `10.1109/MM.2017.35`.

[10] RISC-V. *Working draft of the proposed RISC-V V vector extension*. 2023. URL: `https://github.com/riscv/riscv-v-spec`.

[11] Dezsö Sima. 'The Design Space of Register Renaming Techniques'. In: *IEEE Micro* 20.5 (Sept. 2000), pp. 70–83. ISSN: 0272-1732. DOI: 10.1109/40.877952. URL: https://doi.org/10.1109/40.877952.

Appendices

Appendix A

Unlimited Vector Extension Supporting Microarchitecture

The Streaming Engine (SE) is responsible for most streaming operations and is the main hardware component of the proposed extension. Aside from that, some modifications were made to the Central Processing Unit (CPU) processing pipeline, in order to support streaming. An Out-of-Order (OoO) processing pipeline was chosen as the target of the proof-of-concept implementation of Unlimited Vector Extension (UVE), as it is more common in High-Performance Computing (HPC), thus providing a more valid assessment of the extension in its main target applications, albeit more complex to implement. The proposed microarchitecture is illustrated in Figure A.1, where the modifications listed below are also highlighted.

- **Decoders, register file and execution units:** Support for the decoding of added instructions, vector registers, as well as necessary logic, arithmetic, and branch functional units (similar to RISC-V Vector Extension (RVV) and Scalable Vector Extension (SVE)).
- **Rename stage:** Support for vector register and stream renaming, allowing for speculative configuration of streams. Register renaming is an important mechanism that is already present in most processors. It allows for the elimination of certain data dependencies, by separating architectural and physical registers [11]. This principle is applied to stream configurations, thus making it possible to speculatively configure new streams while others that share the same logical name are still executing.
- **Commit stage:** Support for the commit and squash of streams, through the signaling of all misspeculation and commit events related to the processing of streams to the SE. As a result of an eventual misspeculation, stream configurations or iterations may be incorrectly performed. In the first case, all

the structures involved are released and a new configuration may be accepted by the SE. However, two actions are necessary in the second case. On the one hand, the pipeline is responsible for reverting the physical register to the previously committed value. On the other hand, the SE must be notified of the squash so that it can revert the speculated pointers on the load/store circular buffers to the current commit point. This means that buffered data is never impacted by misspeculations on loads, as well as generated addresses on stores. As streaming data patterns are deterministic, the fact that they are consumed in the wrong order does not change data validity inside said buffers, and it can be re-used with no need for new loads.

In order to fully understand the role of the SE, it is important to first understand how a stream is supposed to behave from the moment it is configured until it is terminated. Furthermore, its implementation on an OoO core is not straightforward. The most important aspects of stream operation in the proposed model are hereby detailed.

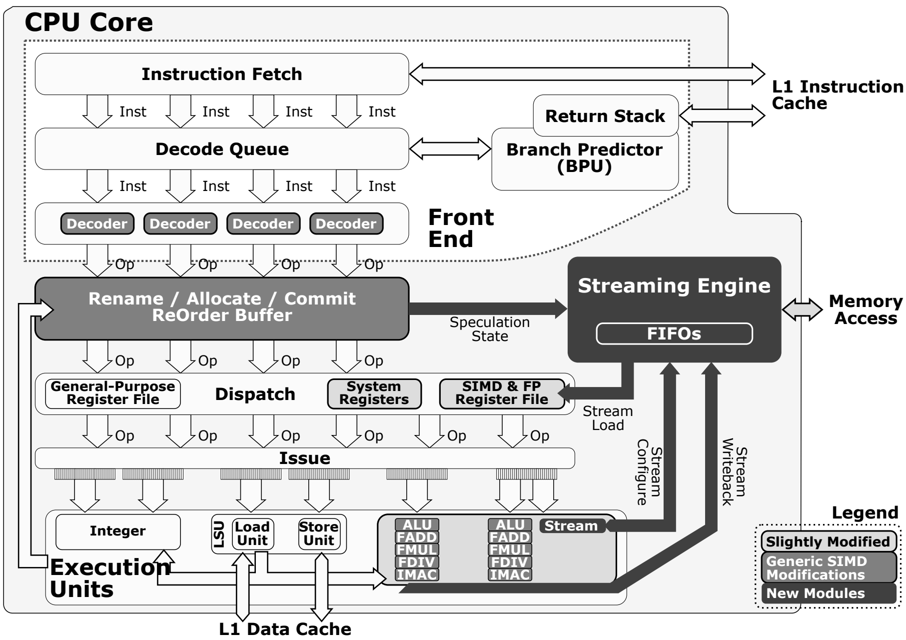


Figure A.1: UVE supporting microarchitecture overview, highlighting modifications introduced in a traditional OoO processing pipeline [3].

Stream Configuration

Multiple instructions are needed to configure any pattern that is not trivial, and these instructions must be executed in order so that the descriptors are correctly chained. In an OoO architecture, this means that some mechanism must be put in place to ensure in-order execution. However, speculative configuration of streams is desirable as it improves performance. As previously mentioned, stream renaming was implemented to support this. At this stage, each stream configuration instruction is inserted in the *Stream Configuration Reorder Buffer (SCROB)*, a new structure embedded in the SE. It processes each configuration instruction in order as soon as the corresponding operands are available, similarly to a re-order buffer. Then, after its configuration, the stream is processed by the SE. This encompasses the pre-loading of data in the case of a load stream, or the computation of store addresses for store streams, that will then await for the committing of store data.

Lastly, each stream configuration also results in two stream state iterators: *speculative* or *commit*. These are dynamically iterated once a stream manipulation instruction reaches the rename and commit stages, respectively, to allow for speculative execution.

Stream Renaming

When a certain stream is beginning the respective configuration, its corresponding identification register may still be occupied by another running stream, due to misspeculation or even pipeline latency. To mitigate possible pipeline blockings, a Stream Allocation Table (SAT) is included. This structure, very similar to a Register Alias Table (RAT), is responsible for the mapping between each physical and logical stream identification register. Moreover, the SAT is also designated for keeping information about which registers are currently associated with active streams, which is necessary for the distinction between stream operations (involving reads/writes from a stream) and regular register operations, which are not handled by the SE.

Stream Iteration

Iterating a stream is the process of reading from input streams (read streams) and writing to output streams (store streams). This occurs during the rename stage, where the *speculative* iterator is incremented. In the case of a load stream, when an instruction that consumes values from a stream enters the rename stage, they are immediately read from a register that holds the pre-loaded data, all while new data is already being pre-loaded to a different physical register. This is thanks to vector register renaming, which was extended beyond the standard of only performing renaming for destination registers to support the renaming of source registers as well. The newly renamed physical register is passed to the load queue of the SE, which then loads the next values from memory.

Stream Termination

The end of a stream is reached at the commit stage, either by an explicit termination instruction or because an instruction with an End-of-Stream (EOS) signal was committed, due to the reaching of the end of the streaming pattern. When this happens, every structure associated with the stream is released.

Memory hierarchy

UVE allows for the configuration of data loading from any cache level. Besides, to simplify the implementation, as well as minimising the impact on caches, the proposed model joins stream requests and typical memory loads/stores before the L1 cache is accessed. This is possible because, most of the time, conventional memory accesses are mutually exclusive from stream accesses, as streaming loops generally do not require scalar memory operations. The memory hierarchy is illustrated in Figure A.2.

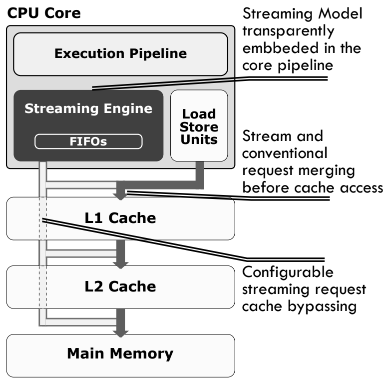


Figure A.2: System overview, featuring the SE embedded in an OoO core and respective connections to the memory hierarchy [3].

Memory coherence

Eventual stream load/store dependencies are handled through typical mechanisms found in modern architectures: request delays, replays, and squashes. This ensures that data resulting from conventional operations can be read by input streams straight away. Likewise, data written by store streams is readily available for conventional load instructions to use. Cache coherence is guaranteed by a MOESI protocol. Coherence mechanisms at the level of the stream First-In, First-Out (FIFO) buffers are not defined, under the assumption that data that has been preloaded has already been consumed by the core, as it would with register pre-fetching or loop unrolling. However, this raises the issue of how load and store streams from the same memory access will behave.

Appendix B

UVE Instruction Listing

31 28	27 25	24 20	19 15	14 12	11 7	6 0	
funct4	ps3	vs2	vs1	funct3	vd/rd	opcode	UA-type

Arithmetic and Logic Instructions

0000	ps3	vs2	vs1	000	vd	0101011	SO.A.ADD.US
0000	ps3	vs2	vs1	001	vd	0101011	SO.A.ADD.FP
0000	ps3	vs2	vs1	010	vd	0101011	SO.A.ADD.SG
0000	ps3	vs2	vs1	100	vd	0101011	SO.A.SUB.US
0000	ps3	vs2	vs1	101	vd	0101011	SO.A.SUB.FP
0000	ps3	vs2	vs1	110	vd	0101011	SO.A.SUB.SG
0001	ps3	vs2	vs1	000	vd	0101011	SO.A.MUL.US
0001	ps3	vs2	vs1	001	vd	0101011	SO.A.MUL.FP
0001	ps3	vs2	vs1	010	vd	0101011	SO.A.MUL.SG
0001	ps3	vs2	vs1	100	vd	0101011	SO.A.DIV.US
0001	ps3	vs2	vs1	101	vd	0101011	SO.A.DIV.FP
0001	ps3	vs2	vs1	110	vd	0101011	SO.A.DIV.SG
0010	ps3	00000	vs1	000	vd	0101011	SO.A.ADDE.US
0010	ps3	00000	vs1	001	vd	0101011	SO.A.ADDE.FP
0010	ps3	00000	vs1	010	vd	0101011	SO.A.ADDE.SG
0010	ps3	00001	vs1	000	vd	0101011	SO.A.ADDE.ACC.US
0010	ps3	00001	vs1	001	vd	0101011	SO.A.ADDE.ACC.FP
0010	ps3	00001	vs1	010	vd	0101011	SO.A.ADDE.ACC.SG
0010	ps3	00000	vs1	100	rd	0101011	SO.A.ADDS.US
0010	ps3	00000	vs1	101	rd	0101011	SO.A.ADDS.FP
0010	ps3	00000	vs1	110	rd	0101011	SO.A.ADDS.SG
0010	ps3	00001	vs1	100	rd	0101011	SO.A.ADDS.ACC.US
0010	ps3	00001	vs1	101	rd	0101011	SO.A.ADDS.ACC.FP
0010	ps3	00001	vs1	110	rd	0101011	SO.A.ADDS.ACC.SG
0011	ps3	00000	vs1	001	vd	0101011	SO.A.ABS.FP
0011	ps3	00000	vs1	000	vd	0101011	SO.A.ABS.SG
0011	ps3	vs2	vs1	100	vd	0101011	SO.A.MAC.US
0011	ps3	vs2	vs1	101	vd	0101011	SO.A.MAC.FP
0011	ps3	vs2	vs1	110	vd	0101011	SO.A.MAC.SG

31 29	28 27 25	24 22 21 20	19 15	14 12	11 7	6 0	
funct4	ps3	vs2	vs1	funct3	vd/rd	opcode	UA-type

funct3	imm[12|10:5]	-	n	vs1	funct3	imm[4:1|11]	opcode	UB-type

Arithmetic and Logic Instructions (Continuation)

1100	ps3	vs2	vs1	000	vd	0101011	SO.A.NAND
1100	ps3	vs2	vs1	001	vd	0101011	SO.A.AND
1100	ps3	vs2	vs1	010	vd	0101011	SO.A.NOR
1100	ps3	vs2	vs1	011	vd	0101011	SO.A.OR
1100	ps3	00000	vs1	100	vd	0101011	SO.A.NOT
1100	ps3	vs2	vs1	101	vd	0101011	SO.A.XOR
1101	ps3	vs2	vs1	000	vd	0101011	SO.A.SLL
1101	ps3	rs2	vs1	001	vd	0101011	SO.A.SLLS
1101	ps3	vs2	vs1	010	vd	0101011	SO.A.SRL
1101	ps3	rs2	vs1	011	vd	0101011	SO.A.SRLS
1101	ps3	vs2	vs1	100	vd	0101011	SO.A.SRA
1101	ps3	rs2	vs1	101	vd	0101011	SO.A.SRAS
0100	ps3	vs2	vs1	000	vd	0101011	SO.A.MIN.US
0100	ps3	vs2	vs1	001	vd	0101011	SO.A.MIN.FP
0100	ps3	vs2	vs1	010	vd	0101011	SO.A.MIN.SG
0100	ps3	vs2	vs1	100	vd	0101011	SO.A.MAX.US
0100	ps3	vs2	vs1	101	vd	0101011	SO.A.MAX.FP
0100	ps3	vs2	vs1	110	vd	0101011	SO.A.MAX.SG
0101	ps3	00000	vs1	000	vd	0101011	SO.A.MINE.US
0101	ps3	00000	vs1	001	vd	0101011	SO.A.MINE.FP
0101	ps3	00000	vs1	010	vd	0101011	SO.A.MINE.SG
0101	ps3	00000	vs1	100	vd	0101011	SO.A.MAXE.US
0101	ps3	00000	vs1	101	vd	0101011	SO.A.MAXE.FP
0101	ps3	00000	vs1	110	vd	0101011	SO.A.MAXE.SG
0110	ps3	00000	vs1	000	vd	0101011	SO.A.INC.US
0110	ps3	00000	vs1	001	vd	0101011	SO.A.INC.FP
0110	ps3	00000	vs1	010	vd	0101011	SO.A.INC.SG
0110	ps3	00000	vs1	100	vd	0101011	SO.A.DEC.US
0110	ps3	00000	vs1	101	vd	0101011	SO.A.DEC.FP
0110	ps3	00000	vs1	110	vd	0101011	SO.A.DEC.SG

Loop Control Branching Instructions

111	imm[12|10:5]	0	1	vs1	000	imm[4:1|11]	0101011	SO.B.NDC.1
111	imm[12|10:5]	0	1	vs1	001	imm[4:1|11]	0101011	SO.B.NDC.2
111	imm[12|10:5]	0	1	vs1	010	imm[4:1|11]	0101011	SO.B.NDC.3
111	imm[12|10:5]	0	1	vs1	011	imm[4:1|11]	0101011	SO.B.NDC.4
111	imm[12|10:5]	0	1	vs1	100	imm[4:1|11]	0101011	SO.B.NDC.5
111	imm[12|10:5]	0	1	vs1	101	imm[4:1|11]	0101011	SO.B.NDC.6
111	imm[12|10:5]	0	1	vs1	110	imm[4:1|11]	0101011	SO.B.NDC.7
111	imm[12|10:5]	0	0	vs1	000	imm[4:1|11]	0101011	SO.B.DC.1
111	imm[12|10:5]	0	0	vs1	001	imm[4:1|11]	0101011	SO.B.DC.2
111	imm[12|10:5]	0	0	vs1	010	imm[4:1|11]	0101011	SO.B.DC.3
111	imm[12|10:5]	0	0	vs1	011	imm[4:1|11]	0101011	SO.B.DC.4
111	imm[12|10:5]	0	0	vs1	100	imm[4:1|11]	0101011	SO.B.DC.5
111	imm[12|10:5]	0	0	vs1	101	imm[4:1|11]	0101011	SO.B.DC.6
111	imm[12|10:5]	0	0	vs1	110	imm[4:1|11]	0101011	SO.B.DC.7
111	imm[12|10:5]	0	1	vs1	111	imm[4:1|11]	0101011	SO.B.NC
111	imm[12|10:5]	0	0	vs1	111	imm[4:1|11]	0101011	SO.B.C

31	28	27	25	24	23 22	21 20	19	18 15	14 12	11	10 7	6 0	
funct4		ps3		z	dw	sw	-	ps1	funct4		pd	opcode	UP1-type
funct4		ps3		z	-		vs1		funct4		pd	opcode	UP2-type
funct4		ps3		vs2			vs1		funct3	z	pd	opcode	UP3-type

Lane Control Predication Instructions

1000		ps3		0	000000000				0000		pd	0101011	SO.P.ZERO
1000		ps3		1	000000000				0000		pd	0101011	SO.P.ZERO.Z
1000		ps3		0	000000000				0001		pd	0101011	SO.P.ONE
1000		ps3		1	000000000				0001		pd	0101011	SO.P.ONE.Z
1000		ps3		0	0000		vs1		0010		pd	0101011	SO.P.VR
1000		ps3		1	0000		vs1		0010		pd	0101011	SO.P.VR.Z
1000		ps3		0	00000			ps1	0011		pd	0101011	SO.P.NOT
1000		ps3		1	00000			ps1	0011		pd	0101011	SO.P.NOT.Z
1000		ps3		0	00000			ps1	0100		pd	0101011	SO.P.MV
1000		ps3		1	00000			ps1	0100		pd	0101011	SO.P.MV.Z
1000		ps3		0	00000			ps1	0101		pd	0101011	SO.P.MVT
1000		ps3		1	00000			ps1	0101		pd	0101011	SO.P.MVT.Z
1	000	0	00	001		00	0	ps1	011	0	pd	0101011	SO.P.CV.B.H
1	000	0	00	101		00	0	ps1	011	0	pd	0101011	SO.P.CV.B.H.Z
1	000	0	00	010		00	0	ps1	011	0	pd	0101011	SO.P.CV.B.W
1	000	0	00	110		00	0	ps1	011	0	pd	0101011	SO.P.CV.B.W.Z
1	000	0	00	011		00	0	ps1	011	0	pd	0101011	SO.P.CV.B.D
1	000	0	00	111		00	0	ps1	011	0	pd	0101011	SO.P.CV.B.D.Z
1	000	0	00	000		01	0	ps1	011	0	pd	0101011	SO.P.CV.H.B
1	000	0	00	100		01	0	ps1	011	0	pd	0101011	SO.P.CV.H.B.Z
1	000	0	00	010		01	0	ps1	011	0	pd	0101011	SO.P.CV.H.W
1	000	0	00	110		01	0	ps1	011	0	pd	0101011	SO.P.CV.H.W.Z
1	000	0	00	011		01	0	ps1	011	0	pd	0101011	SO.P.CV.H.D
1	000	0	00	111		01	0	ps1	011	0	pd	0101011	SO.P.CV.H.D.Z
1	000	0	00	000		10	0	ps1	011	0	pd	0101011	SO.P.CV.W.B
1	000	0	00	100		10	0	ps1	011	0	pd	0101011	SO.P.CV.W.B.Z
1	000	0	00	001		10	0	ps1	011	0	pd	0101011	SO.P.CV.W.H
1	000	0	00	101		10	0	ps1	011	0	pd	0101011	SO.P.CV.W.H.Z
1	000	0	00	011		10	0	ps1	011	0	pd	0101011	SO.P.CV.W.D
1	000	0	00	111		10	0	ps1	011	0	pd	0101011	SO.P.CV.W.D.Z
1	000	0	00	000		11	0	ps1	011	0	pd	0101011	SO.P.CV.D.B
1	000	0	00	100		11	0	ps1	011	0	pd	0101011	SO.P.CV.D.B.Z
1	000	0	00	001		11	0	ps1	011	0	pd	0101011	SO.P.CV.D.H
1	000	0	00	101		11	0	ps1	011	0	pd	0101011	SO.P.CV.D.H.Z
1	000	0	00	010		11	0	ps1	011	0	pd	0101011	SO.P.CV.D.W
1	000	0	00	110		11	0	ps1	011	0	pd	0101011	SO.P.CV.D.W.Z
1000		ps3		vs2			vs1		100	0	pd	0101011	SO.P.GE.US
1000		ps3		vs2			vs1		100	1	pd	0101011	SO.P.GE.US.Z
1000		ps3		vs2			vs1		101	0	pd	0101011	SO.P.GE.FP
1000		ps3		vs2			vs1		101	1	pd	0101011	SO.P.GE.FP.Z
1000		ps3		vs2			vs1		110	0	pd	0101011	SO.P.GE.SG
1000		ps3		vs2			vs1		110	1	pd	0101011	SO.P.GE.SG.Z
1001		ps3		vs2			vs1		000	0	pd	0101011	SO.P.EQ.US
1001		ps3		vs2			vs1		000	1	pd	0101011	SO.P.EQ.US.Z
1001		ps3		vs2			vs1		001	0	pd	0101011	SO.P.EQ.FP
1001		ps3		vs2			vs1		001	1	pd	0101011	SO.P.EQ.FP.Z
1001		ps3		vs2			vs1		010	0	pd	0101011	SO.P.EQ.SG
1001		ps3		vs2			vs1		010	1	pd	0101011	SO.P.EQ.SG.Z
1001		ps3		vs2			vs1		100	0	pd	0101011	SO.P.LT.US
1001		ps3		vs2			vs1		100	1	pd	0101011	SO.P.LT.US.Z
1001		ps3		vs2			vs1		101	0	pd	0101011	SO.P.LT.FP
1001		ps3		vs2			vs1		101	1	pd	0101011	SO.P.LT.FP.Z
1001		ps3		vs2			vs1		110	0	pd	0101011	SO.P.LT.SG
1001		ps3		vs2			vs1		110	1	pd	0101011	SO.P.LT.SG.Z

Appendix B

31	30	29 27	26 25	24	23 22	21 20	19 15	14 12	11 7	6 0	
funct5			-				rs1	funct3	rd	opcode	UC-type
funct5			funct4		ps2		vs1/rs1	funct3	vd/rd	opcode	UV-type
m	v	vdim	funct2	inds	mem	-	rs1	funct3	vd	opcode	USTA-type

Vector Control Instructions

10110	0000000	rs1	000	rd	0101011	SO.C.SETVL
10110	000000000000		111	rd	0101011	SO.C.GETVL
10110	000000000000		001	vd	0101011	SO.C.SUSPD
10110	000000000000		010	vd	0101011	SO.C.RESUM
10110	000000000000		011	vd	0101011	SO.C.BREAK
10110	000000000000		100	vd	0101011	SO.C.VLOAD
10110	000000000000		101	vd	0101011	SO.C.VSTOR

Vector Manipulation Instructions

10101	1000	ps2	rs1	000	vd	0101011	SO.V.DP.B
10101	1000	ps2	rs1	001	vd	0101011	SO.V.DP.H
10101	1000	ps2	rs1	010	vd	0101011	SO.V.DP.W
10101	1000	ps2	rs1	011	vd	0101011	SO.V.DP.D
10101	0000	ps2	vs1	000	vd	0101011	SO.V.MV
10101	0001	ps2	vs1	000	vd	0101011	SO.V.MVT
10101	0010	000	vs1	000	rd	0101011	SO.V.MVVS
10101	0011	000	rs1	000	vd	0101011	SO.V.MVSV.B
10101	0011	000	rs1	001	vd	0101011	SO.V.MVSV.H
10101	0011	000	rs1	010	vd	0101011	SO.V.MVSV.W
10101	0011	000	rs1	011	vd	0101011	SO.V.MVSV.D
10101	0100	000	vs1	000	vd	0101011	SO.V.CV.US.B
10101	0100	000	vs1	001	vd	0101011	SO.V.CV.US.H
10101	0100	000	vs1	010	vd	0101011	SO.V.CV.US.W
10101	0100	000	vs1	011	vd	0101011	SO.V.CV.US.D
10101	0101	000	vs1	000	vd	0101011	SO.V.CV.FP.B
10101	0101	000	vs1	001	vd	0101011	SO.V.CV.FP.H
10101	0101	000	vs1	010	vd	0101011	SO.V.CV.FP.W
10101	0101	000	vs1	011	vd	0101011	SO.V.CV.FP.D
10101	0110	000	vs1	000	vd	0101011	SO.V.CV.SG.B
10101	0110	000	vs1	001	vd	0101011	SO.V.CV.SG.H
10101	0110	000	vs1	010	vd	0101011	SO.V.CV.SG.W
10101	0110	000	vs1	011	vd	0101011	SO.V.CV.SG.D

Stream Configuration Instructions

0	0	000	00	0	00	00	rs1	100	vd	0001011	SS.STA.LD.B
0	0	000	00	0	01	00	rs1	100	vd	0001011	SS.STA.LD.B.MEM1
0	0	000	00	0	10	00	rs1	100	vd	0001011	SS.STA.LD.B.MEM2
0	0	000	00	0	11	00	rs1	100	vd	0001011	SS.STA.LD.B.MEM3
0	0	000	00	1	00	00	rs1	100	vd	0001011	SS.STA.LD.B.INDS
0	0	000	00	1	01	00	rs1	100	vd	0001011	SS.STA.LD.B.INDS.MEM1
0	0	000	00	1	10	00	rs1	100	vd	0001011	SS.STA.LD.B.INDS.MEM2
0	0	000	00	1	11	00	rs1	100	vd	0001011	SS.STA.LD.B.INDS.MEM3
1	0	000	00	0	00	00	rs1	100	vd	0001011	SS.STA.LD.B.M
1	0	000	00	0	01	00	rs1	100	vd	0001011	SS.STA.LD.B.M.MEM1
1	0	000	00	0	10	00	rs1	100	vd	0001011	SS.STA.LD.B.M.MEM2
1	0	000	00	0	11	00	rs1	100	vd	0001011	SS.STA.LD.B.M.MEM3
1	0	000	00	1	00	00	rs1	100	vd	0001011	SS.STA.LD.B.M.INDS
1	0	000	00	1	01	00	rs1	100	vd	0001011	SS.STA.LD.B.M.INDS.MEM1
1	0	000	00	1	10	00	rs1	100	vd	0001011	SS.STA.LD.B.M.INDS.MEM2
1	0	000	00	1	11	00	rs1	100	vd	0001011	SS.STA.LD.B.M.INDS.MEM3
0	0	000	00	0	00	00	rs1	101	vd	0001011	SS.STA.LD.H
0	0	000	00	0	01	00	rs1	101	vd	0001011	SS.STA.LD.H.MEM1
0	0	000	00	0	10	00	rs1	101	vd	0001011	SS.STA.LD.H.MEM2
0	0	000	00	0	11	00	rs1	101	vd	0001011	SS.STA.LD.H.MEM3
0	0	000	00	1	00	00	rs1	101	vd	0001011	SS.STA.LD.H.INDS
0	0	000	00	1	01	00	rs1	101	vd	0001011	SS.STA.LD.H.INDS.MEM1
0	0	000	00	1	10	00	rs1	101	vd	0001011	SS.STA.LD.H.INDS.MEM2
0	0	000	00	1	11	00	rs1	101	vd	0001011	SS.STA.LD.H.INDS.MEM3
1	0	000	00	0	00	00	rs1	101	vd	0001011	SS.STA.LD.H.M
1	0	000	00	0	01	00	rs1	101	vd	0001011	SS.STA.LD.H.M.MEM1
1	0	000	00	0	10	00	rs1	101	vd	0001011	SS.STA.LD.H.M.MEM2
1	0	000	00	0	11	00	rs1	101	vd	0001011	SS.STA.LD.H.M.MEM3
1	0	000	00	1	00	00	rs1	101	vd	0001011	SS.STA.LD.H.M.INDS
1	0	000	00	1	01	00	rs1	101	vd	0001011	SS.STA.LD.H.M.INDS.MEM1
1	0	000	00	1	10	00	rs1	101	vd	0001011	SS.STA.LD.H.M.INDS.MEM2
1	0	000	00	1	11	00	rs1	101	vd	0001011	SS.STA.LD.H.M.INDS.MEM3
0	0	000	00	0	00	00	rs1	110	vd	0001011	SS.STA.LD.W

31	30	29 27	26 25	24	23 22	21 20	19 15	14 12	11 7	6 0	
m	v	vdim	funct2	inds	mem	-	rs1	funct3	vd	opcode	USTA-type

Stream Configuration Instructions (Continuation)

0	0	000	00	0	01	00	rs1	110	vd	0001011	SS.STA.LD.W.MEM1
0	0	000	00	0	10	00	rs1	110	vd	0001011	SS.STA.LD.W.MEM2
0	0	000	00	0	11	00	rs1	110	vd	0001011	SS.STA.LD.W.MEM3
0	0	000	00	1	00	00	rs1	110	vd	0001011	SS.STA.LD.W.INDS
0	0	000	00	1	01	00	rs1	110	vd	0001011	SS.STA.LD.W.INDS.MEM1
0	0	000	00	1	10	00	rs1	110	vd	0001011	SS.STA.LD.W.INDS.MEM2
0	0	000	00	1	11	00	rs1	110	vd	0001011	SS.STA.LD.W.INDS.MEM3
1	0	000	00	0	00	00	rs1	110	vd	0001011	SS.STA.LD.W.M
1	0	000	00	0	01	00	rs1	110	vd	0001011	SS.STA.LD.W.M.MEM1
1	0	000	00	0	10	00	rs1	110	vd	0001011	SS.STA.LD.W.M.MEM2
1	0	000	00	0	11	00	rs1	110	vd	0001011	SS.STA.LD.W.M.MEM3
1	0	000	00	1	00	00	rs1	110	vd	0001011	SS.STA.LD.W.M.INDS
1	0	000	00	1	01	00	rs1	110	vd	0001011	SS.STA.LD.W.M.INDS.MEM1
1	0	000	00	1	10	00	rs1	110	vd	0001011	SS.STA.LD.W.M.INDS.MEM2
1	0	000	00	1	11	00	rs1	110	vd	0001011	SS.STA.LD.W.M.INDS.MEM3
0	0	000	00	0	00	00	rs1	111	vd	0001011	SS.STA.LD.D
0	0	000	00	0	01	00	rs1	111	vd	0001011	SS.STA.LD.D.MEM1
0	0	000	00	0	10	00	rs1	111	vd	0001011	SS.STA.LD.D.MEM2
0	0	000	00	0	11	00	rs1	111	vd	0001011	SS.STA.LD.D.MEM3
0	0	000	00	1	00	00	rs1	111	vd	0001011	SS.STA.LD.D.INDS
0	0	000	00	1	01	00	rs1	111	vd	0001011	SS.STA.LD.D.INDS.MEM1
0	0	000	00	1	10	00	rs1	111	vd	0001011	SS.STA.LD.D.INDS.MEM2
0	0	000	00	1	11	00	rs1	111	vd	0001011	SS.STA.LD.D.INDS.MEM3
1	0	000	00	0	00	00	rs1	111	vd	0001011	SS.STA.LD.D.M
1	0	000	00	0	01	00	rs1	111	vd	0001011	SS.STA.LD.D.M.MEM1
1	0	000	00	0	10	00	rs1	111	vd	0001011	SS.STA.LD.D.M.MEM2
1	0	000	00	0	11	00	rs1	111	vd	0001011	SS.STA.LD.D.M.MEM3
1	0	000	00	1	00	00	rs1	111	vd	0001011	SS.STA.LD.D.M.INDS
1	0	000	00	1	01	00	rs1	111	vd	0001011	SS.STA.LD.D.M.INDS.MEM1
1	0	000	00	1	10	00	rs1	111	vd	0001011	SS.STA.LD.D.M.INDS.MEM2
1	0	000	00	1	11	00	rs1	111	vd	0001011	SS.STA.LD.D.M.INDS.MEM3
0	1	000	00	0	00	00	rs1	100	vd	0001011	SS.STA.LD.B.V.1
0	1	000	00	0	01	00	rs1	100	vd	0001011	SS.STA.LD.B.V.1.MEM1
0	1	000	00	0	10	00	rs1	100	vd	0001011	SS.STA.LD.B.V.1.MEM2
0	1	000	00	0	11	00	rs1	100	vd	0001011	SS.STA.LD.B.V.1.MEM3
1	1	000	00	0	00	00	rs1	100	vd	0001011	SS.STA.LD.B.V.1.M
1	1	000	00	0	01	00	rs1	100	vd	0001011	SS.STA.LD.B.V.1.M.MEM1
1	1	000	00	0	10	00	rs1	100	vd	0001011	SS.STA.LD.B.V.1.M.MEM2
1	1	000	00	0	11	00	rs1	100	vd	0001011	SS.STA.LD.B.V.1.M.MEM3
0	1	000	00	0	00	00	rs1	101	vd	0001011	SS.STA.LD.H.V.1
0	1	000	00	0	01	00	rs1	101	vd	0001011	SS.STA.LD.H.V.1.MEM1
0	1	000	00	0	10	00	rs1	101	vd	0001011	SS.STA.LD.H.V.1.MEM2
0	1	000	00	0	11	00	rs1	101	vd	0001011	SS.STA.LD.H.V.1.MEM3
1	1	000	00	0	00	00	rs1	101	vd	0001011	SS.STA.LD.H.V.1.M
1	1	000	00	0	01	00	rs1	101	vd	0001011	SS.STA.LD.H.V.1.M.MEM1
1	1	000	00	0	10	00	rs1	101	vd	0001011	SS.STA.LD.H.V.1.M.MEM2
1	1	000	00	0	11	00	rs1	101	vd	0001011	SS.STA.LD.H.V.1.M.MEM3
0	1	000	00	0	00	00	rs1	110	vd	0001011	SS.STA.LD.W.V.1
0	1	000	00	0	01	00	rs1	110	vd	0001011	SS.STA.LD.W.V.1.MEM1
0	1	000	00	0	10	00	rs1	110	vd	0001011	SS.STA.LD.W.V.1.MEM2
0	1	000	00	0	11	00	rs1	110	vd	0001011	SS.STA.LD.W.V.1.MEM3
1	1	000	00	0	00	00	rs1	110	vd	0001011	SS.STA.LD.W.V.1.M
1	1	000	00	0	01	00	rs1	110	vd	0001011	SS.STA.LD.W.V.1.M.MEM1
1	1	000	00	0	10	00	rs1	110	vd	0001011	SS.STA.LD.W.V.1.M.MEM2
1	1	000	00	0	11	00	rs1	110	vd	0001011	SS.STA.LD.W.V.1.M.MEM3
0	1	000	00	0	00	00	rs1	111	vd	0001011	SS.STA.LD.D.V.1
0	1	000	00	0	01	00	rs1	111	vd	0001011	SS.STA.LD.D.V.1.MEM1
0	1	000	00	0	10	00	rs1	111	vd	0001011	SS.STA.LD.D.V.1.MEM2
0	1	000	00	0	11	00	rs1	111	vd	0001011	SS.STA.LD.D.V.1.MEM3
1	1	000	00	0	00	00	rs1	111	vd	0001011	SS.STA.LD.D.V.1.M
1	1	000	00	0	01	00	rs1	111	vd	0001011	SS.STA.LD.D.V.1.M.MEM1
1	1	000	00	0	10	00	rs1	111	vd	0001011	SS.STA.LD.D.V.1.M.MEM2
1	1	000	00	0	11	00	rs1	111	vd	0001011	SS.STA.LD.D.V.1.M.MEM3

31	30	29 27	26 25	24	23 22	21 20	19 15	14 12	11 7	6 0	
m	v	vdim	funct2	inds	mem	-	rs1	funct3	vd	opcode	USTA-type
Stream Configuration Instructions (Continuation)											
0	1	001	00	0	00	00	rs1	100	vd	0001011	SS.STA.LD.B.V.2
0	1	001	00	0	01	00	rs1	100	vd	0001011	SS.STA.LD.B.V.2.MEM1
0	1	001	00	0	10	00	rs1	100	vd	0001011	SS.STA.LD.B.V.2.MEM2
0	1	001	00	0	11	00	rs1	100	vd	0001011	SS.STA.LD.B.V.2.MEM3
1	1	001	00	0	00	00	rs1	100	vd	0001011	SS.STA.LD.B.V.2.M
1	1	001	00	0	01	00	rs1	100	vd	0001011	SS.STA.LD.B.V.2.M.MEM1
1	1	001	00	0	10	00	rs1	100	vd	0001011	SS.STA.LD.B.V.2.M.MEM2
1	1	001	00	0	11	00	rs1	100	vd	0001011	SS.STA.LD.B.V.2.M.MEM3
0	1	001	00	0	00	00	rs1	101	vd	0001011	SS.STA.LD.H.V.2
0	1	001	00	0	01	00	rs1	101	vd	0001011	SS.STA.LD.H.V.2.MEM1
0	1	001	00	0	10	00	rs1	101	vd	0001011	SS.STA.LD.H.V.2.MEM2
0	1	001	00	0	11	00	rs1	101	vd	0001011	SS.STA.LD.H.V.2.MEM3
1	1	001	00	0	00	00	rs1	101	vd	0001011	SS.STA.LD.H.V.2.M
1	1	001	00	0	01	00	rs1	101	vd	0001011	SS.STA.LD.H.V.2.M.MEM1
1	1	001	00	0	10	00	rs1	101	vd	0001011	SS.STA.LD.H.V.2.M.MEM2
1	1	001	00	0	11	00	rs1	101	vd	0001011	SS.STA.LD.H.V.2.M.MEM3
0	1	001	00	0	00	00	rs1	110	vd	0001011	SS.STA.LD.W.V.2
0	1	001	00	0	01	00	rs1	110	vd	0001011	SS.STA.LD.W.V.2.MEM1
0	1	001	00	0	10	00	rs1	110	vd	0001011	SS.STA.LD.W.V.2.MEM2
0	1	001	00	0	11	00	rs1	110	vd	0001011	SS.STA.LD.W.V.2.MEM3
1	1	001	00	0	00	00	rs1	110	vd	0001011	SS.STA.LD.W.V.2.M
1	1	001	00	0	01	00	rs1	110	vd	0001011	SS.STA.LD.W.V.2.M.MEM1
1	1	001	00	0	10	00	rs1	110	vd	0001011	SS.STA.LD.W.V.2.M.MEM2
1	1	001	00	0	11	00	rs1	110	vd	0001011	SS.STA.LD.W.V.2.M.MEM3
0	1	001	00	0	00	00	rs1	111	vd	0001011	SS.STA.LD.D.V.2
0	1	001	00	0	01	00	rs1	111	vd	0001011	SS.STA.LD.D.V.2.MEM1
0	1	001	00	0	10	00	rs1	111	vd	0001011	SS.STA.LD.D.V.2.MEM2
0	1	001	00	0	11	00	rs1	111	vd	0001011	SS.STA.LD.D.V.2.MEM3
1	1	001	00	0	00	00	rs1	111	vd	0001011	SS.STA.LD.D.V.2.M
1	1	001	00	0	01	00	rs1	111	vd	0001011	SS.STA.LD.D.V.2.M.MEM1
1	1	001	00	0	10	00	rs1	111	vd	0001011	SS.STA.LD.D.V.2.M.MEM2
1	1	001	00	0	11	00	rs1	111	vd	0001011	SS.STA.LD.D.V.2.M.MEM3
0	1	010	00	0	00	00	rs1	100	vd	0001011	SS.STA.LD.B.V.3
0	1	010	00	0	01	00	rs1	100	vd	0001011	SS.STA.LD.B.V.3.MEM1
0	1	010	00	0	10	00	rs1	100	vd	0001011	SS.STA.LD.B.V.3.MEM2
0	1	010	00	0	11	00	rs1	100	vd	0001011	SS.STA.LD.B.V.3.MEM3
1	1	010	00	0	00	00	rs1	100	vd	0001011	SS.STA.LD.B.V.3.M
1	1	010	00	0	01	00	rs1	100	vd	0001011	SS.STA.LD.B.V.3.M.MEM1
1	1	010	00	0	10	00	rs1	100	vd	0001011	SS.STA.LD.B.V.3.M.MEM2
1	1	010	00	0	11	00	rs1	100	vd	0001011	SS.STA.LD.B.V.3.M.MEM3
0	1	010	00	0	00	00	rs1	101	vd	0001011	SS.STA.LD.H.V.3
0	1	010	00	0	01	00	rs1	101	vd	0001011	SS.STA.LD.H.V.3.MEM1
0	1	010	00	0	10	00	rs1	101	vd	0001011	SS.STA.LD.H.V.3.MEM2
0	1	010	00	0	11	00	rs1	101	vd	0001011	SS.STA.LD.H.V.3.MEM3
1	1	010	00	0	00	00	rs1	101	vd	0001011	SS.STA.LD.H.V.3.M
1	1	010	00	0	01	00	rs1	101	vd	0001011	SS.STA.LD.H.V.3.M.MEM1
1	1	010	00	0	10	00	rs1	101	vd	0001011	SS.STA.LD.H.V.3.M.MEM2
1	1	010	00	0	11	00	rs1	101	vd	0001011	SS.STA.LD.H.V.3.M.MEM3
0	1	010	00	0	00	00	rs1	110	vd	0001011	SS.STA.LD.W.V.3
0	1	010	00	0	01	00	rs1	110	vd	0001011	SS.STA.LD.W.V.3.MEM1
0	1	010	00	0	10	00	rs1	110	vd	0001011	SS.STA.LD.W.V.3.MEM2
0	1	010	00	0	11	00	rs1	110	vd	0001011	SS.STA.LD.W.V.3.MEM3
1	1	010	00	0	00	00	rs1	110	vd	0001011	SS.STA.LD.W.V.3.M
1	1	010	00	0	01	00	rs1	110	vd	0001011	SS.STA.LD.W.V.3.M.MEM1
1	1	010	00	0	10	00	rs1	110	vd	0001011	SS.STA.LD.W.V.3.M.MEM2
1	1	010	00	0	11	00	rs1	110	vd	0001011	SS.STA.LD.W.V.3.M.MEM3
0	1	010	00	0	00	00	rs1	111	vd	0001011	SS.STA.LD.D.V.3
0	1	010	00	0	01	00	rs1	111	vd	0001011	SS.STA.LD.D.V.3.MEM1
0	1	010	00	0	10	00	rs1	111	vd	0001011	SS.STA.LD.D.V.3.MEM2
0	1	010	00	0	11	00	rs1	111	vd	0001011	SS.STA.LD.D.V.3.MEM3
1	1	010	00	0	00	00	rs1	111	vd	0001011	SS.STA.LD.D.V.3.M
1	1	010	00	0	01	00	rs1	111	vd	0001011	SS.STA.LD.D.V.3.M.MEM1
1	1	010	00	0	10	00	rs1	111	vd	0001011	SS.STA.LD.D.V.3.M.MEM2
1	1	010	00	0	11	00	rs1	111	vd	0001011	SS.STA.LD.D.V.3.M.MEM3

31	30	29 27	26 25	24	23 22	21 20	19 15	14 12	11 7	6 0	
m	v	vdim	funct2	inds	mem	-	rs1	funct3	vd	opcode	USTA-type
					Stream Configuration Instructions (Continuation)						
0	1	011	00	0	00	00	rs1	100	vd	0001011	SS.STA.LD.B.V.4
0	1	011	00	0	01	00	rs1	100	vd	0001011	SS.STA.LD.B.V.4.MEM1
0	1	011	00	0	10	00	rs1	100	vd	0001011	SS.STA.LD.B.V.4.MEM2
0	1	011	00	0	11	00	rs1	100	vd	0001011	SS.STA.LD.B.V.4.MEM3
1	1	011	00	0	00	00	rs1	100	vd	0001011	SS.STA.LD.B.V.4.M
1	1	011	00	0	01	00	rs1	100	vd	0001011	SS.STA.LD.B.V.4.M.MEM1
1	1	011	00	0	10	00	rs1	100	vd	0001011	SS.STA.LD.B.V.4.M.MEM2
1	1	011	00	0	11	00	rs1	100	vd	0001011	SS.STA.LD.B.V.4.M.MEM3
0	1	011	00	0	00	00	rs1	101	vd	0001011	SS.STA.LD.H.V.4
0	1	011	00	0	01	00	rs1	101	vd	0001011	SS.STA.LD.H.V.4.MEM1
0	1	011	00	0	10	00	rs1	101	vd	0001011	SS.STA.LD.H.V.4.MEM2
0	1	011	00	0	11	00	rs1	101	vd	0001011	SS.STA.LD.H.V.4.MEM3
1	1	011	00	0	00	00	rs1	101	vd	0001011	SS.STA.LD.H.V.4.M
1	1	011	00	0	01	00	rs1	101	vd	0001011	SS.STA.LD.H.V.4.M.MEM1
1	1	011	00	0	10	00	rs1	101	vd	0001011	SS.STA.LD.H.V.4.M.MEM2
1	1	011	00	0	11	00	rs1	101	vd	0001011	SS.STA.LD.H.V.4.M.MEM3
0	1	011	00	0	00	00	rs1	110	vd	0001011	SS.STA.LD.W.V.4
0	1	011	00	0	01	00	rs1	110	vd	0001011	SS.STA.LD.W.V.4.MEM1
0	1	011	00	0	10	00	rs1	110	vd	0001011	SS.STA.LD.W.V.4.MEM2
0	1	011	00	0	11	00	rs1	110	vd	0001011	SS.STA.LD.W.V.4.MEM3
1	1	011	00	0	00	00	rs1	110	vd	0001011	SS.STA.LD.W.V.4.M
1	1	011	00	0	01	00	rs1	110	vd	0001011	SS.STA.LD.W.V.4.M.MEM1
1	1	011	00	0	10	00	rs1	110	vd	0001011	SS.STA.LD.W.V.4.M.MEM2
1	1	011	00	0	11	00	rs1	110	vd	0001011	SS.STA.LD.W.V.4.M.MEM3
0	1	011	00	0	00	00	rs1	111	vd	0001011	SS.STA.LD.D.V.4
0	1	011	00	0	01	00	rs1	111	vd	0001011	SS.STA.LD.D.V.4.MEM1
0	1	011	00	0	10	00	rs1	111	vd	0001011	SS.STA.LD.D.V.4.MEM2
0	1	011	00	0	11	00	rs1	111	vd	0001011	SS.STA.LD.D.V.4.MEM3
1	1	011	00	0	00	00	rs1	111	vd	0001011	SS.STA.LD.D.V.4.M
1	1	011	00	0	01	00	rs1	111	vd	0001011	SS.STA.LD.D.V.4.M.MEM1
1	1	011	00	0	10	00	rs1	111	vd	0001011	SS.STA.LD.D.V.4.M.MEM2
1	1	011	00	0	11	00	rs1	111	vd	0001011	SS.STA.LD.D.V.4.M.MEM3
0	1	100	00	0	00	00	rs1	100	vd	0001011	SS.STA.LD.B.V.5
0	1	100	00	0	01	00	rs1	100	vd	0001011	SS.STA.LD.B.V.5.MEM1
0	1	100	00	0	10	00	rs1	100	vd	0001011	SS.STA.LD.B.V.5.MEM2
0	1	100	00	0	11	00	rs1	100	vd	0001011	SS.STA.LD.B.V.5.MEM3
1	1	100	00	0	00	00	rs1	100	vd	0001011	SS.STA.LD.B.V.5.M
1	1	100	00	0	01	00	rs1	100	vd	0001011	SS.STA.LD.B.V.5.M.MEM1
1	1	100	00	0	10	00	rs1	100	vd	0001011	SS.STA.LD.B.V.5.M.MEM2
1	1	100	00	0	11	00	rs1	100	vd	0001011	SS.STA.LD.B.V.5.M.MEM3
0	1	100	00	0	00	00	rs1	101	vd	0001011	SS.STA.LD.H.V.5
0	1	100	00	0	01	00	rs1	101	vd	0001011	SS.STA.LD.H.V.5.MEM1
0	1	100	00	0	10	00	rs1	101	vd	0001011	SS.STA.LD.H.V.5.MEM2
0	1	100	00	0	11	00	rs1	101	vd	0001011	SS.STA.LD.H.V.5.MEM3
1	1	100	00	0	00	00	rs1	101	vd	0001011	SS.STA.LD.H.V.5.M
1	1	100	00	0	01	00	rs1	101	vd	0001011	SS.STA.LD.H.V.5.M.MEM1
1	1	100	00	0	10	00	rs1	101	vd	0001011	SS.STA.LD.H.V.5.M.MEM2
1	1	100	00	0	11	00	rs1	101	vd	0001011	SS.STA.LD.H.V.5.M.MEM3
0	1	100	00	0	00	00	rs1	110	vd	0001011	SS.STA.LD.W.V.5
0	1	100	00	0	01	00	rs1	110	vd	0001011	SS.STA.LD.W.V.5.MEM1
0	1	100	00	0	10	00	rs1	110	vd	0001011	SS.STA.LD.W.V.5.MEM2
0	1	100	00	0	11	00	rs1	110	vd	0001011	SS.STA.LD.W.V.5.MEM3
1	1	100	00	0	00	00	rs1	110	vd	0001011	SS.STA.LD.W.V.5.M
1	1	100	00	0	01	00	rs1	110	vd	0001011	SS.STA.LD.W.V.5.M.MEM1
1	1	100	00	0	10	00	rs1	110	vd	0001011	SS.STA.LD.W.V.5.M.MEM2
1	1	100	00	0	11	00	rs1	110	vd	0001011	SS.STA.LD.W.V.5.M.MEM3
0	1	100	00	0	00	00	rs1	111	vd	0001011	SS.STA.LD.D.V.5
0	1	100	00	0	01	00	rs1	111	vd	0001011	SS.STA.LD.D.V.5.MEM1
0	1	100	00	0	10	00	rs1	111	vd	0001011	SS.STA.LD.D.V.5.MEM2
0	1	100	00	0	11	00	rs1	111	vd	0001011	SS.STA.LD.D.V.5.MEM3
1	1	100	00	0	00	00	rs1	111	vd	0001011	SS.STA.LD.D.V.5.M
1	1	100	00	0	01	00	rs1	111	vd	0001011	SS.STA.LD.D.V.5.M.MEM1
1	1	100	00	0	10	00	rs1	111	vd	0001011	SS.STA.LD.D.V.5.M.MEM2
1	1	100	00	0	11	00	rs1	111	vd	0001011	SS.STA.LD.D.V.5.M.MEM3
0	1	101	00	0	00	00	rs1	100	vd	0001011	SS.STA.LD.B.V.6
0	1	101	00	0	01	00	rs1	100	vd	0001011	SS.STA.LD.B.V.6.MEM1
0	1	101	00	0	10	00	rs1	100	vd	0001011	SS.STA.LD.B.V.6.MEM2
0	1	101	00	0	11	00	rs1	100	vd	0001011	SS.STA.LD.B.V.6.MEM3

31	30	29 27	26 25	24	23 22	21 20	19 15	14 12	11 7	6 0	
m	v	vdim	funct2	inds	mem	-	rs1	funct3	vd	opcode	USTA-type

Stream Configuration Instructions (Continuation)

1	1	101	00	0	00	00	rs1	100	vd	0001011	SS.STA.LD.B.V.6.M
1	1	101	00	0	01	00	rs1	100	vd	0001011	SS.STA.LD.B.V.6.M.MEM1
1	1	101	00	0	10	00	rs1	100	vd	0001011	SS.STA.LD.B.V.6.M.MEM2
1	1	101	00	0	11	00	rs1	100	vd	0001011	SS.STA.LD.B.V.6.M.MEM3
0	1	101	00	0	00	00	rs1	101	vd	0001011	SS.STA.LD.H.V.6
0	1	101	00	0	01	00	rs1	101	vd	0001011	SS.STA.LD.H.V.6.MEM1
0	1	101	00	0	10	00	rs1	101	vd	0001011	SS.STA.LD.H.V.6.MEM2
0	1	101	00	0	11	00	rs1	101	vd	0001011	SS.STA.LD.H.V.6.MEM3
1	1	101	00	0	00	00	rs1	101	vd	0001011	SS.STA.LD.H.V.6.M
1	1	101	00	0	01	00	rs1	101	vd	0001011	SS.STA.LD.H.V.6.M.MEM1
1	1	101	00	0	10	00	rs1	101	vd	0001011	SS.STA.LD.H.V.6.M.MEM2
1	1	101	00	0	11	00	rs1	101	vd	0001011	SS.STA.LD.H.V.6.M.MEM3
0	1	101	00	0	00	00	rs1	110	vd	0001011	SS.STA.LD.W.V.6
0	1	101	00	0	01	00	rs1	110	vd	0001011	SS.STA.LD.W.V.6.MEM1
0	1	101	00	0	10	00	rs1	110	vd	0001011	SS.STA.LD.W.V.6.MEM2
0	1	101	00	0	11	00	rs1	110	vd	0001011	SS.STA.LD.W.V.6.MEM3
1	1	101	00	0	00	00	rs1	110	vd	0001011	SS.STA.LD.W.V.6.M
1	1	101	00	0	01	00	rs1	110	vd	0001011	SS.STA.LD.W.V.6.M.MEM1
1	1	101	00	0	10	00	rs1	110	vd	0001011	SS.STA.LD.W.V.6.M.MEM2
1	1	101	00	0	11	00	rs1	110	vd	0001011	SS.STA.LD.W.V.6.M.MEM3
0	1	101	00	0	00	00	rs1	111	vd	0001011	SS.STA.LD.D.V.6
0	1	101	00	0	01	00	rs1	111	vd	0001011	SS.STA.LD.D.V.6.MEM1
0	1	101	00	0	10	00	rs1	111	vd	0001011	SS.STA.LD.D.V.6.MEM2
0	1	101	00	0	11	00	rs1	111	vd	0001011	SS.STA.LD.D.V.6.MEM3
1	1	101	00	0	00	00	rs1	111	vd	0001011	SS.STA.LD.D.V.6.M
1	1	101	00	0	01	00	rs1	111	vd	0001011	SS.STA.LD.D.V.6.M.MEM1
1	1	101	00	0	10	00	rs1	111	vd	0001011	SS.STA.LD.D.V.6.M.MEM2
1	1	101	00	0	11	00	rs1	111	vd	0001011	SS.STA.LD.D.V.6.M.MEM3
0	1	110	00	0	00	00	rs1	100	vd	0001011	SS.STA.LD.B.V.7
0	1	110	00	0	01	00	rs1	100	vd	0001011	SS.STA.LD.B.V.7.MEM1
0	1	110	00	0	10	00	rs1	100	vd	0001011	SS.STA.LD.B.V.7.MEM2
0	1	110	00	0	11	00	rs1	100	vd	0001011	SS.STA.LD.B.V.7.MEM3
1	1	110	00	0	00	00	rs1	100	vd	0001011	SS.STA.LD.B.V.7.M
1	1	110	00	0	01	00	rs1	100	vd	0001011	SS.STA.LD.B.V.7.M.MEM1
1	1	110	00	0	10	00	rs1	100	vd	0001011	SS.STA.LD.B.V.7.M.MEM2
1	1	110	00	0	11	00	rs1	100	vd	0001011	SS.STA.LD.B.V.7.M.MEM3
0	1	110	00	0	00	00	rs1	101	vd	0001011	SS.STA.LD.H.V.7
0	1	110	00	0	01	00	rs1	101	vd	0001011	SS.STA.LD.H.V.7.MEM1
0	1	110	00	0	10	00	rs1	101	vd	0001011	SS.STA.LD.H.V.7.MEM2
0	1	110	00	0	11	00	rs1	101	vd	0001011	SS.STA.LD.H.V.7.MEM3
1	1	110	00	0	00	00	rs1	101	vd	0001011	SS.STA.LD.H.V.7.M
1	1	110	00	0	01	00	rs1	101	vd	0001011	SS.STA.LD.H.V.7.M.MEM1
1	1	110	00	0	10	00	rs1	101	vd	0001011	SS.STA.LD.H.V.7.M.MEM2
1	1	110	00	0	11	00	rs1	101	vd	0001011	SS.STA.LD.H.V.7.M.MEM3
0	1	110	00	0	00	00	rs1	110	vd	0001011	SS.STA.LD.W.V.7
0	1	110	00	0	01	00	rs1	110	vd	0001011	SS.STA.LD.W.V.7.MEM1
0	1	110	00	0	10	00	rs1	110	vd	0001011	SS.STA.LD.W.V.7.MEM2
0	1	110	00	0	11	00	rs1	110	vd	0001011	SS.STA.LD.W.V.7.MEM3
1	1	110	00	0	00	00	rs1	110	vd	0001011	SS.STA.LD.W.V.7.M
1	1	110	00	0	01	00	rs1	110	vd	0001011	SS.STA.LD.W.V.7.M.MEM1
1	1	110	00	0	10	00	rs1	110	vd	0001011	SS.STA.LD.W.V.7.M.MEM2
1	1	110	00	0	11	00	rs1	110	vd	0001011	SS.STA.LD.W.V.7.M.MEM3
0	1	110	00	0	00	00	rs1	111	vd	0001011	SS.STA.LD.D.V.7
0	1	110	00	0	01	00	rs1	111	vd	0001011	SS.STA.LD.D.V.7.MEM1
0	1	110	00	0	10	00	rs1	111	vd	0001011	SS.STA.LD.D.V.7.MEM2
0	1	110	00	0	11	00	rs1	111	vd	0001011	SS.STA.LD.D.V.7.MEM3
1	1	110	00	0	00	00	rs1	111	vd	0001011	SS.STA.LD.D.V.7.M
1	1	110	00	0	01	00	rs1	111	vd	0001011	SS.STA.LD.D.V.7.M.MEM1
1	1	110	00	0	10	00	rs1	111	vd	0001011	SS.STA.LD.D.V.7.M.MEM2
1	1	110	00	0	11	00	rs1	111	vd	0001011	SS.STA.LD.D.V.7.M.MEM3
0	1	111	00	0	00	00	rs1	100	vd	0001011	SS.STA.LD.B.V
0	1	111	00	0	01	00	rs1	100	vd	0001011	SS.STA.LD.B.V.MEM1
0	1	111	00	0	10	00	rs1	100	vd	0001011	SS.STA.LD.B.V.MEM2
0	1	111	00	0	11	00	rs1	100	vd	0001011	SS.STA.LD.B.V.MEM3

31	30	29 27	26 25	24	23 22	21 20	19 15	14 12	11 7	6 0	
m	v	vdim	funct2	inds	mem	-	rs1	funct3	vd	opcode	USTA-type
					Stream Configuration Instructions (Continuation)						
1	1	111	00	0	00	00	rs1	100	vd	0001011	SS.STA.LD.B.V.M
1	1	111	00	0	01	00	rs1	100	vd	0001011	SS.STA.LD.B.V.M.MEM1
1	1	111	00	0	10	00	rs1	100	vd	0001011	SS.STA.LD.B.V.M.MEM2
1	1	111	00	0	11	00	rs1	100	vd	0001011	SS.STA.LD.B.V.M.MEM3
0	1	111	00	0	00	00	rs1	101	vd	0001011	SS.STA.LD.H.V
0	1	111	00	0	01	00	rs1	101	vd	0001011	SS.STA.LD.H.V.MEM1
0	1	111	00	0	10	00	rs1	101	vd	0001011	SS.STA.LD.H.V.MEM2
0	1	111	00	0	11	00	rs1	101	vd	0001011	SS.STA.LD.H.V.MEM3
1	1	111	00	0	00	00	rs1	101	vd	0001011	SS.STA.LD.H.V.M
1	1	111	00	0	01	00	rs1	101	vd	0001011	SS.STA.LD.H.V.M.MEM1
1	1	111	00	0	10	00	rs1	101	vd	0001011	SS.STA.LD.H.V.M.MEM2
1	1	111	00	0	11	00	rs1	101	vd	0001011	SS.STA.LD.H.V.M.MEM3
0	1	111	00	0	00	00	rs1	110	vd	0001011	SS.STA.LD.W.V
0	1	111	00	0	01	00	rs1	110	vd	0001011	SS.STA.LD.W.V.MEM1
0	1	111	00	0	10	00	rs1	110	vd	0001011	SS.STA.LD.W.V.MEM2
0	1	111	00	0	11	00	rs1	110	vd	0001011	SS.STA.LD.W.V.MEM3
1	1	111	00	0	00	00	rs1	110	vd	0001011	SS.STA.LD.W.V.M
1	1	111	00	0	01	00	rs1	110	vd	0001011	SS.STA.LD.W.V.M.MEM1
1	1	111	00	0	10	00	rs1	110	vd	0001011	SS.STA.LD.W.V.M.MEM2
1	1	111	00	0	11	00	rs1	110	vd	0001011	SS.STA.LD.W.V.M.MEM3
0	1	111	00	0	00	00	rs1	111	vd	0001011	SS.STA.LD.D.V
0	1	111	00	0	01	00	rs1	111	vd	0001011	SS.STA.LD.D.V.MEM1
0	1	111	00	0	10	00	rs1	111	vd	0001011	SS.STA.LD.D.V.MEM2
0	1	111	00	0	11	00	rs1	111	vd	0001011	SS.STA.LD.D.V.MEM3
1	1	111	00	0	00	00	rs1	111	vd	0001011	SS.STA.LD.D.V.M
1	1	111	00	0	01	00	rs1	111	vd	0001011	SS.STA.LD.D.V.M.MEM1
1	1	111	00	0	10	00	rs1	111	vd	0001011	SS.STA.LD.D.V.M.MEM2
1	1	111	00	0	11	00	rs1	111	vd	0001011	SS.STA.LD.D.V.M.MEM3
0	0	000	00	0	00	00	rs1	000	vd	0001011	SS.STA.ST.B
0	0	000	00	0	01	00	rs1	000	vd	0001011	SS.STA.ST.B.MEM1
0	0	000	00	0	10	00	rs1	000	vd	0001011	SS.STA.ST.B.MEM2
0	0	000	00	0	11	00	rs1	000	vd	0001011	SS.STA.ST.B.MEM3
1	0	000	00	0	00	00	rs1	000	vd	0001011	SS.STA.ST.B.M
1	0	000	00	0	01	00	rs1	000	vd	0001011	SS.STA.ST.B.M.MEM1
1	0	000	00	0	10	00	rs1	000	vd	0001011	SS.STA.ST.B.M.MEM2
1	0	000	00	0	11	00	rs1	000	vd	0001011	SS.STA.ST.B.M.MEM3
0	0	000	00	0	00	00	rs1	001	vd	0001011	SS.STA.ST.H
0	0	000	00	0	01	00	rs1	001	vd	0001011	SS.STA.ST.H.MEM1
0	0	000	00	0	10	00	rs1	001	vd	0001011	SS.STA.ST.H.MEM2
0	0	000	00	0	11	00	rs1	001	vd	0001011	SS.STA.ST.H.MEM3
1	0	000	00	0	00	00	rs1	001	vd	0001011	SS.STA.ST.H.M
1	0	000	00	0	01	00	rs1	001	vd	0001011	SS.STA.ST.H.M.MEM1
1	0	000	00	0	10	00	rs1	001	vd	0001011	SS.STA.ST.H.M.MEM2
1	0	000	00	0	11	00	rs1	001	vd	0001011	SS.STA.ST.H.M.MEM3
0	0	000	00	0	00	00	rs1	010	vd	0001011	SS.STA.ST.W
0	0	000	00	0	01	00	rs1	010	vd	0001011	SS.STA.ST.W.MEM1
0	0	000	00	0	10	00	rs1	010	vd	0001011	SS.STA.ST.W.MEM2
0	0	000	00	0	11	00	rs1	010	vd	0001011	SS.STA.ST.W.MEM3
1	0	000	00	0	00	00	rs1	010	vd	0001011	SS.STA.ST.W.M
1	0	000	00	0	01	00	rs1	010	vd	0001011	SS.STA.ST.W.M.MEM1
1	0	000	00	0	10	00	rs1	010	vd	0001011	SS.STA.ST.W.M.MEM2
1	0	000	00	0	11	00	rs1	010	vd	0001011	SS.STA.ST.W.M.MEM3
0	0	000	00	0	00	00	rs1	011	vd	0001011	SS.STA.ST.D
0	0	000	00	0	01	00	rs1	011	vd	0001011	SS.STA.ST.D.MEM1
0	0	000	00	0	10	00	rs1	011	vd	0001011	SS.STA.ST.D.MEM2
0	0	000	00	0	11	00	rs1	011	vd	0001011	SS.STA.ST.D.MEM3
1	0	000	00	0	00	00	rs1	011	vd	0001011	SS.STA.ST.D.M
1	0	000	00	0	01	00	rs1	011	vd	0001011	SS.STA.ST.D.M.MEM1
1	0	000	00	0	10	00	rs1	011	vd	0001011	SS.STA.ST.D.M.MEM2
1	0	000	00	0	11	00	rs1	011	vd	0001011	SS.STA.ST.D.M.MEM3
0	1	000	00	0	00	00	rs1	000	vd	0001011	SS.STA.ST.B.V.1
0	1	000	00	0	01	00	rs1	000	vd	0001011	SS.STA.ST.B.V.1.MEM1
0	1	000	00	0	10	00	rs1	000	vd	0001011	SS.STA.ST.B.V.1.MEM2
0	1	000	00	0	11	00	rs1	000	vd	0001011	SS.STA.ST.B.V.1.MEM3

31	30	29 27	26 25	24	23 22	21 20	19 15	14 12	11 7	6 0	
m	v	vdim	funct2	inds	mem	-	rs1	funct3	vd	opcode	USTA-type

Stream Configuration Instructions (Continuation)

1	1	000	00	0	00	00	rs1	000	vd	0001011	SS.STA.ST.B.V.1.M
1	1	000	00	0	01	00	rs1	000	vd	0001011	SS.STA.ST.B.V.1.M.MEM1
1	1	000	00	0	10	00	rs1	000	vd	0001011	SS.STA.ST.B.V.1.M.MEM2
1	1	000	00	0	11	00	rs1	000	vd	0001011	SS.STA.ST.B.V.1.M.MEM3
0	1	000	00	0	00	00	rs1	001	vd	0001011	SS.STA.ST.H.V.1
0	1	000	00	0	01	00	rs1	001	vd	0001011	SS.STA.ST.H.V.1.MEM1
0	1	000	00	0	10	00	rs1	001	vd	0001011	SS.STA.ST.H.V.1.MEM2
0	1	000	00	0	11	00	rs1	001	vd	0001011	SS.STA.ST.H.V.1.MEM3
1	1	000	00	0	00	00	rs1	001	vd	0001011	SS.STA.ST.H.V.1.M
1	1	000	00	0	01	00	rs1	001	vd	0001011	SS.STA.ST.H.V.1.M.MEM1
1	1	000	00	0	10	00	rs1	001	vd	0001011	SS.STA.ST.H.V.1.M.MEM2
1	1	000	00	0	11	00	rs1	001	vd	0001011	SS.STA.ST.H.V.1.M.MEM3
0	1	000	00	0	00	00	rs1	010	vd	0001011	SS.STA.ST.W.V.1
0	1	000	00	0	01	00	rs1	010	vd	0001011	SS.STA.ST.W.V.1.MEM1
0	1	000	00	0	10	00	rs1	010	vd	0001011	SS.STA.ST.W.V.1.MEM2
0	1	000	00	0	11	00	rs1	010	vd	0001011	SS.STA.ST.W.V.1.MEM3
1	1	000	00	0	00	00	rs1	010	vd	0001011	SS.STA.ST.W.V.1.M
1	1	000	00	0	01	00	rs1	010	vd	0001011	SS.STA.ST.W.V.1.M.MEM1
1	1	000	00	0	10	00	rs1	010	vd	0001011	SS.STA.ST.W.V.1.M.MEM2
1	1	000	00	0	11	00	rs1	010	vd	0001011	SS.STA.ST.W.V.1.M.MEM3
0	1	000	00	0	00	00	rs1	011	vd	0001011	SS.STA.ST.D.V.1
0	1	000	00	0	01	00	rs1	011	vd	0001011	SS.STA.ST.D.V.1.MEM1
0	1	000	00	0	10	00	rs1	011	vd	0001011	SS.STA.ST.D.V.1.MEM2
0	1	000	00	0	11	00	rs1	011	vd	0001011	SS.STA.ST.D.V.1.MEM3
1	1	000	00	0	00	00	rs1	011	vd	0001011	SS.STA.ST.D.V.1.M
1	1	000	00	0	01	00	rs1	011	vd	0001011	SS.STA.ST.D.V.1.M.MEM1
1	1	000	00	0	10	00	rs1	011	vd	0001011	SS.STA.ST.D.V.1.M.MEM2
1	1	000	00	0	11	00	rs1	011	vd	0001011	SS.STA.ST.D.V.1.M.MEM3
0	1	001	00	0	00	00	rs1	000	vd	0001011	SS.STA.ST.B.V.2
0	1	001	00	0	01	00	rs1	000	vd	0001011	SS.STA.ST.B.V.2.MEM1
0	1	001	00	0	10	00	rs1	000	vd	0001011	SS.STA.ST.B.V.2.MEM2
0	1	001	00	0	11	00	rs1	000	vd	0001011	SS.STA.ST.B.V.2.MEM3
1	1	001	00	0	00	00	rs1	000	vd	0001011	SS.STA.ST.B.V.2.M
1	1	001	00	0	01	00	rs1	000	vd	0001011	SS.STA.ST.B.V.2.M.MEM1
1	1	001	00	0	10	00	rs1	000	vd	0001011	SS.STA.ST.B.V.2.M.MEM2
1	1	001	00	0	11	00	rs1	000	vd	0001011	SS.STA.ST.B.V.2.M.MEM3
0	1	001	00	0	00	00	rs1	001	vd	0001011	SS.STA.ST.H.V.2
0	1	001	00	0	01	00	rs1	001	vd	0001011	SS.STA.ST.H.V.2.MEM1
0	1	001	00	0	10	00	rs1	001	vd	0001011	SS.STA.ST.H.V.2.MEM2
0	1	001	00	0	11	00	rs1	001	vd	0001011	SS.STA.ST.H.V.2.MEM3
1	1	001	00	0	00	00	rs1	001	vd	0001011	SS.STA.ST.H.V.2.M
1	1	001	00	0	01	00	rs1	001	vd	0001011	SS.STA.ST.H.V.2.M.MEM1
1	1	001	00	0	10	00	rs1	001	vd	0001011	SS.STA.ST.H.V.2.M.MEM2
1	1	001	00	0	11	00	rs1	001	vd	0001011	SS.STA.ST.H.V.2.M.MEM3
0	1	001	00	0	00	00	rs1	010	vd	0001011	SS.STA.ST.W.V.2
0	1	001	00	0	01	00	rs1	010	vd	0001011	SS.STA.ST.W.V.2.MEM1
0	1	001	00	0	10	00	rs1	010	vd	0001011	SS.STA.ST.W.V.2.MEM2
0	1	001	00	0	11	00	rs1	010	vd	0001011	SS.STA.ST.W.V.2.MEM3
1	1	001	00	0	00	00	rs1	010	vd	0001011	SS.STA.ST.W.V.2.M
1	1	001	00	0	01	00	rs1	010	vd	0001011	SS.STA.ST.W.V.2.M.MEM1
1	1	001	00	0	10	00	rs1	010	vd	0001011	SS.STA.ST.W.V.2.M.MEM2
1	1	001	00	0	11	00	rs1	010	vd	0001011	SS.STA.ST.W.V.2.M.MEM3
0	1	001	00	0	00	00	rs1	011	vd	0001011	SS.STA.ST.D.V.2
0	1	001	00	0	01	00	rs1	011	vd	0001011	SS.STA.ST.D.V.2.MEM1
0	1	001	00	0	10	00	rs1	011	vd	0001011	SS.STA.ST.D.V.2.MEM2
0	1	001	00	0	11	00	rs1	011	vd	0001011	SS.STA.ST.D.V.2.MEM3
1	1	001	00	0	00	00	rs1	011	vd	0001011	SS.STA.ST.D.V.2.M
1	1	001	00	0	01	00	rs1	011	vd	0001011	SS.STA.ST.D.V.2.M.MEM1
1	1	001	00	0	10	00	rs1	011	vd	0001011	SS.STA.ST.D.V.2.M.MEM2
1	1	001	00	0	11	00	rs1	011	vd	0001011	SS.STA.ST.D.V.2.M.MEM3
0	1	010	00	0	00	00	rs1	000	vd	0001011	SS.STA.ST.B.V.3
0	1	010	00	0	01	00	rs1	000	vd	0001011	SS.STA.ST.B.V.3.MEM1
0	1	010	00	0	10	00	rs1	000	vd	0001011	SS.STA.ST.B.V.3.MEM2
0	1	010	00	0	11	00	rs1	000	vd	0001011	SS.STA.ST.B.V.3.MEM3
1	1	010	00	0	00	00	rs1	000	vd	0001011	SS.STA.ST.B.V.3.M
1	1	010	00	0	01	00	rs1	000	vd	0001011	SS.STA.ST.B.V.3.M.MEM1
1	1	010	00	0	10	00	rs1	000	vd	0001011	SS.STA.ST.B.V.3.M.MEM2
1	1	010	00	0	11	00	rs1	000	vd	0001011	SS.STA.ST.B.V.3.M.MEM3

31	30	29 27	26 25	24	23 22	21 20	19 15	14 12	11 7	6 0	
m	v	vdim	funct2	inds	mem	-	rs1	funct3	vd	opcode	USTA-type

Stream Configuration Instructions (Continuation)

0	1	010	00	0	00	00	rs1	001	vd	0001011	SS.STA.ST.H.V.3
0	1	010	00	0	01	00	rs1	001	vd	0001011	SS.STA.ST.H.V.3.MEM1
0	1	010	00	0	10	00	rs1	001	vd	0001011	SS.STA.ST.H.V.3.MEM2
0	1	010	00	0	11	00	rs1	001	vd	0001011	SS.STA.ST.H.V.3.MEM3
1	1	010	00	0	00	00	rs1	001	vd	0001011	SS.STA.ST.H.V.3.M
1	1	010	00	0	01	00	rs1	001	vd	0001011	SS.STA.ST.H.V.3.M.MEM1
1	1	010	00	0	10	00	rs1	001	vd	0001011	SS.STA.ST.H.V.3.M.MEM2
1	1	010	00	0	11	00	rs1	001	vd	0001011	SS.STA.ST.H.V.3.M.MEM3
0	1	010	00	0	00	00	rs1	010	vd	0001011	SS.STA.ST.W.V.3
0	1	010	00	0	01	00	rs1	010	vd	0001011	SS.STA.ST.W.V.3.MEM1
0	1	010	00	0	10	00	rs1	010	vd	0001011	SS.STA.ST.W.V.3.MEM2
0	1	010	00	0	11	00	rs1	010	vd	0001011	SS.STA.ST.W.V.3.MEM3
1	1	010	00	0	00	00	rs1	010	vd	0001011	SS.STA.ST.W.V.3.M
1	1	010	00	0	01	00	rs1	010	vd	0001011	SS.STA.ST.W.V.3.M.MEM1
1	1	010	00	0	10	00	rs1	010	vd	0001011	SS.STA.ST.W.V.3.M.MEM2
1	1	010	00	0	11	00	rs1	010	vd	0001011	SS.STA.ST.W.V.3.M.MEM3
0	1	010	00	0	00	00	rs1	011	vd	0001011	SS.STA.ST.D.V.3
0	1	010	00	0	01	00	rs1	011	vd	0001011	SS.STA.ST.D.V.3.MEM1
0	1	010	00	0	10	00	rs1	011	vd	0001011	SS.STA.ST.D.V.3.MEM2
0	1	010	00	0	11	00	rs1	011	vd	0001011	SS.STA.ST.D.V.3.MEM3
1	1	010	00	0	00	00	rs1	011	vd	0001011	SS.STA.ST.D.V.3.M
1	1	010	00	0	01	00	rs1	011	vd	0001011	SS.STA.ST.D.V.3.M.MEM1
1	1	010	00	0	10	00	rs1	011	vd	0001011	SS.STA.ST.D.V.3.M.MEM2
1	1	010	00	0	11	00	rs1	011	vd	0001011	SS.STA.ST.D.V.3.M.MEM3
0	1	011	00	0	00	00	rs1	000	vd	0001011	SS.STA.ST.B.V.4
0	1	011	00	0	01	00	rs1	000	vd	0001011	SS.STA.ST.B.V.4.MEM1
0	1	011	00	0	10	00	rs1	000	vd	0001011	SS.STA.ST.B.V.4.MEM2
0	1	011	00	0	11	00	rs1	000	vd	0001011	SS.STA.ST.B.V.4.MEM3
1	1	011	00	0	00	00	rs1	000	vd	0001011	SS.STA.ST.B.V.4.M
1	1	011	00	0	01	00	rs1	000	vd	0001011	SS.STA.ST.B.V.4.M.MEM1
1	1	011	00	0	10	00	rs1	000	vd	0001011	SS.STA.ST.B.V.4.M.MEM2
1	1	011	00	0	11	00	rs1	000	vd	0001011	SS.STA.ST.B.V.4.M.MEM3
0	1	011	00	0	00	00	rs1	001	vd	0001011	SS.STA.ST.H.V.4
0	1	011	00	0	01	00	rs1	001	vd	0001011	SS.STA.ST.H.V.4.MEM1
0	1	011	00	0	10	00	rs1	001	vd	0001011	SS.STA.ST.H.V.4.MEM2
0	1	011	00	0	11	00	rs1	001	vd	0001011	SS.STA.ST.H.V.4.MEM3
1	1	011	00	0	00	00	rs1	001	vd	0001011	SS.STA.ST.H.V.4.M
1	1	011	00	0	01	00	rs1	001	vd	0001011	SS.STA.ST.H.V.4.M.MEM1
1	1	011	00	0	10	00	rs1	001	vd	0001011	SS.STA.ST.H.V.4.M.MEM2
1	1	011	00	0	11	00	rs1	001	vd	0001011	SS.STA.ST.H.V.4.M.MEM3
0	1	011	00	0	00	00	rs1	010	vd	0001011	SS.STA.ST.W.V.4
0	1	011	00	0	01	00	rs1	010	vd	0001011	SS.STA.ST.W.V.4.MEM1
0	1	011	00	0	10	00	rs1	010	vd	0001011	SS.STA.ST.W.V.4.MEM2
0	1	011	00	0	11	00	rs1	010	vd	0001011	SS.STA.ST.W.V.4.MEM3
1	1	011	00	0	00	00	rs1	010	vd	0001011	SS.STA.ST.W.V.4.M
1	1	011	00	0	01	00	rs1	010	vd	0001011	SS.STA.ST.W.V.4.M.MEM1
1	1	011	00	0	10	00	rs1	010	vd	0001011	SS.STA.ST.W.V.4.M.MEM2
1	1	011	00	0	11	00	rs1	010	vd	0001011	SS.STA.ST.W.V.4.M.MEM3
0	1	011	00	0	00	00	rs1	011	vd	0001011	SS.STA.ST.D.V.4
0	1	011	00	0	01	00	rs1	011	vd	0001011	SS.STA.ST.D.V.4.MEM1
0	1	011	00	0	10	00	rs1	011	vd	0001011	SS.STA.ST.D.V.4.MEM2
0	1	011	00	0	11	00	rs1	011	vd	0001011	SS.STA.ST.D.V.4.MEM3
1	1	011	00	0	00	00	rs1	011	vd	0001011	SS.STA.ST.D.V.4.M
1	1	011	00	0	01	00	rs1	011	vd	0001011	SS.STA.ST.D.V.4.M.MEM1
1	1	011	00	0	10	00	rs1	011	vd	0001011	SS.STA.ST.D.V.4.M.MEM2
1	1	011	00	0	11	00	rs1	011	vd	0001011	SS.STA.ST.D.V.4.M.MEM3
0	1	100	00	0	00	00	rs1	000	vd	0001011	SS.STA.ST.B.V.5
0	1	100	00	0	01	00	rs1	000	vd	0001011	SS.STA.ST.B.V.5.MEM1
0	1	100	00	0	10	00	rs1	000	vd	0001011	SS.STA.ST.B.V.5.MEM2
0	1	100	00	0	11	00	rs1	000	vd	0001011	SS.STA.ST.B.V.5.MEM3
1	1	100	00	0	00	00	rs1	000	vd	0001011	SS.STA.ST.B.V.5.M
1	1	100	00	0	01	00	rs1	000	vd	0001011	SS.STA.ST.B.V.5.M.MEM1
1	1	100	00	0	10	00	rs1	000	vd	0001011	SS.STA.ST.B.V.5.M.MEM2
1	1	100	00	0	11	00	rs1	000	vd	0001011	SS.STA.ST.B.V.5.M.MEM3
0	1	100	00	0	00	00	rs1	001	vd	0001011	SS.STA.ST.H.V.5
0	1	100	00	0	01	00	rs1	001	vd	0001011	SS.STA.ST.H.V.5.MEM1
0	1	100	00	0	10	00	rs1	001	vd	0001011	SS.STA.ST.H.V.5.MEM2
0	1	100	00	0	11	00	rs1	001	vd	0001011	SS.STA.ST.H.V.5.MEM3

31	30	29 27	26 25	24	23 22	21 20	19 15	14 12	11 7	6 0	
m	v	vdim	funct2	inds	mem	-	rs1	funct3	vd	opcode	USTA-type

Stream Configuration Instructions (Continuation)

1	1	100	00	0	00	00	rs1	001	vd	0001011	SS.STA.ST.H.V.5.M
1	1	100	00	0	01	00	rs1	001	vd	0001011	SS.STA.ST.H.V.5.M.MEM1
1	1	100	00	0	10	00	rs1	001	vd	0001011	SS.STA.ST.H.V.5.M.MEM2
1	1	100	00	0	11	00	rs1	001	vd	0001011	SS.STA.ST.H.V.5.M.MEM3
0	1	100	00	0	00	00	rs1	010	vd	0001011	SS.STA.ST.W.V.5
0	1	100	00	0	01	00	rs1	010	vd	0001011	SS.STA.ST.W.V.5.MEM1
0	1	100	00	0	10	00	rs1	010	vd	0001011	SS.STA.ST.W.V.5.MEM2
0	1	100	00	0	11	00	rs1	010	vd	0001011	SS.STA.ST.W.V.5.MEM3
1	1	100	00	0	00	00	rs1	010	vd	0001011	SS.STA.ST.W.V.5.M
1	1	100	00	0	01	00	rs1	010	vd	0001011	SS.STA.ST.W.V.5.M.MEM1
1	1	100	00	0	10	00	rs1	010	vd	0001011	SS.STA.ST.W.V.5.M.MEM2
1	1	100	00	0	11	00	rs1	010	vd	0001011	SS.STA.ST.W.V.5.M.MEM3
0	1	100	00	0	00	00	rs1	011	vd	0001011	SS.STA.ST.D.V.5
0	1	100	00	0	01	00	rs1	011	vd	0001011	SS.STA.ST.D.V.5.MEM1
0	1	100	00	0	10	00	rs1	011	vd	0001011	SS.STA.ST.D.V.5.MEM2
0	1	100	00	0	11	00	rs1	011	vd	0001011	SS.STA.ST.D.V.5.MEM3
1	1	100	00	0	00	00	rs1	011	vd	0001011	SS.STA.ST.D.V.5.M
1	1	100	00	0	01	00	rs1	011	vd	0001011	SS.STA.ST.D.V.5.M.MEM1
1	1	100	00	0	10	00	rs1	011	vd	0001011	SS.STA.ST.D.V.5.M.MEM2
1	1	100	00	0	11	00	rs1	011	vd	0001011	SS.STA.ST.D.V.5.M.MEM3
0	1	101	00	0	00	00	rs1	000	vd	0001011	SS.STA.ST.B.V.6
0	1	101	00	0	01	00	rs1	000	vd	0001011	SS.STA.ST.B.V.6.MEM1
0	1	101	00	0	10	00	rs1	000	vd	0001011	SS.STA.ST.B.V.6.MEM2
0	1	101	00	0	11	00	rs1	000	vd	0001011	SS.STA.ST.B.V.6.MEM3
1	1	101	00	0	00	00	rs1	000	vd	0001011	SS.STA.ST.B.V.6.M
1	1	101	00	0	01	00	rs1	000	vd	0001011	SS.STA.ST.B.V.6.M.MEM1
1	1	101	00	0	10	00	rs1	000	vd	0001011	SS.STA.ST.B.V.6.M.MEM2
1	1	101	00	0	11	00	rs1	000	vd	0001011	SS.STA.ST.B.V.6.M.MEM3
0	1	101	00	0	00	00	rs1	001	vd	0001011	SS.STA.ST.H.V.6
0	1	101	00	0	01	00	rs1	001	vd	0001011	SS.STA.ST.H.V.6.MEM1
0	1	101	00	0	10	00	rs1	001	vd	0001011	SS.STA.ST.H.V.6.MEM2
0	1	101	00	0	11	00	rs1	001	vd	0001011	SS.STA.ST.H.V.6.MEM3
1	1	101	00	0	00	00	rs1	001	vd	0001011	SS.STA.ST.H.V.6.M
1	1	101	00	0	01	00	rs1	001	vd	0001011	SS.STA.ST.H.V.6.M.MEM1
1	1	101	00	0	10	00	rs1	001	vd	0001011	SS.STA.ST.H.V.6.M.MEM2
1	1	101	00	0	11	00	rs1	001	vd	0001011	SS.STA.ST.H.V.6.M.MEM3
0	1	101	00	0	00	00	rs1	010	vd	0001011	SS.STA.ST.W.V.6
0	1	101	00	0	01	00	rs1	010	vd	0001011	SS.STA.ST.W.V.6.MEM1
0	1	101	00	0	10	00	rs1	010	vd	0001011	SS.STA.ST.W.V.6.MEM2
0	1	101	00	0	11	00	rs1	010	vd	0001011	SS.STA.ST.W.V.6.MEM3
1	1	101	00	0	00	00	rs1	010	vd	0001011	SS.STA.ST.W.V.6.M
1	1	101	00	0	01	00	rs1	010	vd	0001011	SS.STA.ST.W.V.6.M.MEM1
1	1	101	00	0	10	00	rs1	010	vd	0001011	SS.STA.ST.W.V.6.M.MEM2
1	1	101	00	0	11	00	rs1	010	vd	0001011	SS.STA.ST.W.V.6.M.MEM3
0	1	101	00	0	00	00	rs1	011	vd	0001011	SS.STA.ST.D.V.6
0	1	101	00	0	01	00	rs1	011	vd	0001011	SS.STA.ST.D.V.6.MEM1
0	1	101	00	0	10	00	rs1	011	vd	0001011	SS.STA.ST.D.V.6.MEM2
0	1	101	00	0	11	00	rs1	011	vd	0001011	SS.STA.ST.D.V.6.MEM3
1	1	101	00	0	00	00	rs1	011	vd	0001011	SS.STA.ST.D.V.6.M
1	1	101	00	0	01	00	rs1	011	vd	0001011	SS.STA.ST.D.V.6.M.MEM1
1	1	101	00	0	10	00	rs1	011	vd	0001011	SS.STA.ST.D.V.6.M.MEM2
1	1	101	00	0	11	00	rs1	011	vd	0001011	SS.STA.ST.D.V.6.M.MEM3
0	1	110	00	0	00	00	rs1	000	vd	0001011	SS.STA.ST.B.V.7
0	1	110	00	0	01	00	rs1	000	vd	0001011	SS.STA.ST.B.V.7.MEM1
0	1	110	00	0	10	00	rs1	000	vd	0001011	SS.STA.ST.B.V.7.MEM2
0	1	110	00	0	11	00	rs1	000	vd	0001011	SS.STA.ST.B.V.7.MEM3
1	1	110	00	0	00	00	rs1	000	vd	0001011	SS.STA.ST.B.V.7.M
1	1	110	00	0	01	00	rs1	000	vd	0001011	SS.STA.ST.B.V.7.M.MEM1
1	1	110	00	0	10	00	rs1	000	vd	0001011	SS.STA.ST.B.V.7.M.MEM2
1	1	110	00	0	11	00	rs1	000	vd	0001011	SS.STA.ST.B.V.7.M.MEM3
0	1	110	00	0	00	00	rs1	001	vd	0001011	SS.STA.ST.H.V.7
0	1	110	00	0	01	00	rs1	001	vd	0001011	SS.STA.ST.H.V.7.MEM1
0	1	110	00	0	10	00	rs1	001	vd	0001011	SS.STA.ST.H.V.7.MEM2
0	1	110	00	0	11	00	rs1	001	vd	0001011	SS.STA.ST.H.V.7.MEM3
1	1	110	00	0	00	00	rs1	001	vd	0001011	SS.STA.ST.H.V.7.M
1	1	110	00	0	01	00	rs1	001	vd	0001011	SS.STA.ST.H.V.7.M.MEM1
1	1	110	00	0	10	00	rs1	001	vd	0001011	SS.STA.ST.H.V.7.M.MEM2
1	1	110	00	0	11	00	rs1	001	vd	0001011	SS.STA.ST.H.V.7.M.MEM3

31	30	29 27	26 25	24	23 22	21 20	19 18	17 15	14 12	11 7	6 0	
m	v	vdim	funct2	inds	mem	-	rs1		funct3	vd	opcode	USTA-type
rs3			funct2	b		t	-	tdim	funct3	vd	opcode	UMOD-type
rs3			funct2	rs2			rs1		funct3	vd	opcode	UAE-type

Stream Configuration Instructions (Continuation)

0	1	110	00	0	00	00	rs1		010	vd	0001011	SS.STA.ST.W.V.7
0	1	110	00	0	01	00	rs1		010	vd	0001011	SS.STA.ST.W.V.7.MEM1
0	1	110	00	0	10	00	rs1		010	vd	0001011	SS.STA.ST.W.V.7.MEM2
0	1	110	00	0	11	00	rs1		010	vd	0001011	SS.STA.ST.W.V.7.MEM3
1	1	110	00	0	00	00	rs1		010	vd	0001011	SS.STA.ST.W.V.7.M
1	1	110	00	0	01	00	rs1		010	vd	0001011	SS.STA.ST.W.V.7.M.MEM1
1	1	110	00	0	10	00	rs1		010	vd	0001011	SS.STA.ST.W.V.7.M.MEM2
1	1	110	00	0	11	00	rs1		010	vd	0001011	SS.STA.ST.W.V.7.M.MEM3
0	1	110	00	0	00	00	rs1		011	vd	0001011	SS.STA.ST.D.V.7
0	1	110	00	0	01	00	rs1		011	vd	0001011	SS.STA.ST.D.V.7.MEM1
0	1	110	00	0	10	00	rs1		011	vd	0001011	SS.STA.ST.D.V.7.MEM2
0	1	110	00	0	11	00	rs1		011	vd	0001011	SS.STA.ST.D.V.7.MEM3
1	1	110	00	0	00	00	rs1		011	vd	0001011	SS.STA.ST.D.V.7.M
1	1	110	00	0	01	00	rs1		011	vd	0001011	SS.STA.ST.D.V.7.M.MEM1
1	1	110	00	0	10	00	rs1		011	vd	0001011	SS.STA.ST.D.V.7.M.MEM2
1	1	110	00	0	11	00	rs1		011	vd	0001011	SS.STA.ST.D.V.7.M.MEM3
0	1	111	00	0	00	00	rs1		000	vd	0001011	SS.STA.ST.B.V
0	1	111	00	0	01	00	rs1		000	vd	0001011	SS.STA.ST.B.V.MEM1
0	1	111	00	0	10	00	rs1		000	vd	0001011	SS.STA.ST.B.V.MEM2
0	1	111	00	0	11	00	rs1		000	vd	0001011	SS.STA.ST.B.V.MEM3
1	1	111	00	0	00	00	rs1		000	vd	0001011	SS.STA.ST.B.V.M
1	1	111	00	0	01	00	rs1		000	vd	0001011	SS.STA.ST.B.V.M.MEM1
1	1	111	00	0	10	00	rs1		000	vd	0001011	SS.STA.ST.B.V.M.MEM2
1	1	111	00	0	11	00	rs1		000	vd	0001011	SS.STA.ST.B.V.M.MEM3
0	1	111	00	0	00	00	rs1		001	vd	0001011	SS.STA.ST.H.V
0	1	111	00	0	01	00	rs1		001	vd	0001011	SS.STA.ST.H.V.MEM1
0	1	111	00	0	10	00	rs1		001	vd	0001011	SS.STA.ST.H.V.MEM2
0	1	111	00	0	11	00	rs1		001	vd	0001011	SS.STA.ST.H.V.MEM3
1	1	111	00	0	00	00	rs1		001	vd	0001011	SS.STA.ST.H.V.M
1	1	111	00	0	01	00	rs1		001	vd	0001011	SS.STA.ST.H.V.M.MEM1
1	1	111	00	0	10	00	rs1		001	vd	0001011	SS.STA.ST.H.V.M.MEM2
1	1	111	00	0	11	00	rs1		001	vd	0001011	SS.STA.ST.H.V.M.MEM3
0	1	111	00	0	00	00	rs1		010	vd	0001011	SS.STA.ST.W.V
0	1	111	00	0	01	00	rs1		010	vd	0001011	SS.STA.ST.W.V.MEM1
0	1	111	00	0	10	00	rs1		010	vd	0001011	SS.STA.ST.W.V.MEM2
0	1	111	00	0	11	00	rs1		010	vd	0001011	SS.STA.ST.W.V.MEM3
1	1	111	00	0	00	00	rs1		010	vd	0001011	SS.STA.ST.W.V.M
1	1	111	00	0	01	00	rs1		010	vd	0001011	SS.STA.ST.W.V.M.MEM1
1	1	111	00	0	10	00	rs1		010	vd	0001011	SS.STA.ST.W.V.M.MEM2
1	1	111	00	0	11	00	rs1		010	vd	0001011	SS.STA.ST.W.V.M.MEM3
0	1	111	00	0	00	00	rs1		011	vd	0001011	SS.STA.ST.D.V
0	1	111	00	0	01	00	rs1		011	vd	0001011	SS.STA.ST.D.V.MEM1
0	1	111	00	0	10	00	rs1		011	vd	0001011	SS.STA.ST.D.V.MEM2
0	1	111	00	0	11	00	rs1		011	vd	0001011	SS.STA.ST.D.V.MEM3
1	1	111	00	0	00	00	rs1		011	vd	0001011	SS.STA.ST.D.V.M
1	1	111	00	0	01	00	rs1		011	vd	0001011	SS.STA.ST.D.V.M.MEM1
1	1	111	00	0	10	00	rs1		011	vd	0001011	SS.STA.ST.D.V.M.MEM2
1	1	111	00	0	11	00	rs1		011	vd	0001011	SS.STA.ST.D.V.M.MEM3
rs3			01	rs2			rs1		000	vd	0001011	SS.APP
rs3			01	000		00	00	000	100	vd	0001011	SS.APP.MOD.SIZ.INC.1
rs3			01	001		00	00	000	100	vd	0001011	SS.APP.MOD.SIZ.DEC.1
rs3			01	000		01	00	000	100	vd	0001011	SS.APP.MOD.STR.INC.1
rs3			01	001		01	00	000	100	vd	0001011	SS.APP.MOD.STR.DEC.1
rs3			01	000		10	00	000	100	vd	0001011	SS.APP.MOD.OFS.INC.1
rs3			01	001		10	00	000	100	vd	0001011	SS.APP.MOD.OFS.DEC.1
rs3			01	000		00	00	001	100	vd	0001011	SS.APP.MOD.SIZ.INC.2
rs3			01	001		00	00	001	100	vd	0001011	SS.APP.MOD.SIZ.DEC.2
rs3			01	000		01	00	001	100	vd	0001011	SS.APP.MOD.STR.INC.2
rs3			01	001		01	00	001	100	vd	0001011	SS.APP.MOD.STR.DEC.2
rs3			01	000		10	00	001	100	vd	0001011	SS.APP.MOD.OFS.INC.2
rs3			01	001		10	00	001	100	vd	0001011	SS.APP.MOD.OFS.DEC.2
rs3			01	000		00	00	010	100	vd	0001011	SS.APP.MOD.SIZ.INC.3
rs3			01	001		00	00	010	100	vd	0001011	SS.APP.MOD.SIZ.DEC.3
rs3			01	000		01	00	010	100	vd	0001011	SS.APP.MOD.STR.INC.3
rs3			01	001		01	00	010	100	vd	0001011	SS.APP.MOD.STR.DEC.3
rs3			01	000		10	00	010	100	vd	0001011	SS.APP.MOD.OFS.INC.3

31		27	26 25	24 22	21 20	19 18	17 15	14 12	11 7	6 0	
rs3			funct2	b	t	-	tdim	funct3	vd	opcode	UMOD-type
-	tdim	sg	funct2	b	t	vs1		funct3	vd	opcode	UIND-type

Stream Configuration Instructions (Continuation)

rs3			01	001	10	00	010	100	vd	0001011	SS.APP.MOD.OFS.DEC.3
rs3			01	000	00	00	011	100	vd	0001011	SS.APP.MOD.SIZ.INC.4
rs3			01	001	00	00	011	100	vd	0001011	SS.APP.MOD.SIZ.DEC.4
rs3			01	000	01	00	011	100	vd	0001011	SS.APP.MOD.STR.INC.4
rs3			01	001	01	00	011	100	vd	0001011	SS.APP.MOD.STR.DEC.4
rs3			01	000	10	00	011	100	vd	0001011	SS.APP.MOD.OFS.INC.4
rs3			01	001	10	00	011	100	vd	0001011	SS.APP.MOD.OFS.DEC.4
rs3			01	000	00	00	100	100	vd	0001011	SS.APP.MOD.SIZ.INC.5
rs3			01	001	00	00	100	100	vd	0001011	SS.APP.MOD.SIZ.DEC.5
rs3			01	000	01	00	100	100	vd	0001011	SS.APP.MOD.STR.INC.5
rs3			01	001	01	00	100	100	vd	0001011	SS.APP.MOD.STR.DEC.5
rs3			01	000	10	00	100	100	vd	0001011	SS.APP.MOD.OFS.INC.5
rs3			01	001	10	00	100	100	vd	0001011	SS.APP.MOD.OFS.DEC.5
rs3			01	000	00	00	101	100	vd	0001011	SS.APP.MOD.SIZ.INC.6
rs3			01	001	00	00	101	100	vd	0001011	SS.APP.MOD.SIZ.DEC.6
rs3			01	000	01	00	101	100	vd	0001011	SS.APP.MOD.STR.INC.6
rs3			01	001	01	00	101	100	vd	0001011	SS.APP.MOD.STR.DEC.6
rs3			01	000	10	00	101	100	vd	0001011	SS.APP.MOD.OFS.INC.6
rs3			01	001	10	00	101	100	vd	0001011	SS.APP.MOD.OFS.DEC.6
rs3			01	000	00	00	110	100	vd	0001011	SS.APP.MOD.SIZ.INC.7
rs3			01	001	00	00	110	100	vd	0001011	SS.APP.MOD.SIZ.DEC.7
rs3			01	000	01	00	110	100	vd	0001011	SS.APP.MOD.STR.INC.7
rs3			01	001	01	00	110	100	vd	0001011	SS.APP.MOD.STR.DEC.7
rs3			01	000	10	00	110	100	vd	0001011	SS.APP.MOD.OFS.INC.7
rs3			01	001	10	00	110	100	vd	0001011	SS.APP.MOD.OFS.DEC.7
rs3			01	000	00	00	111	100	vd	0001011	SS.APP.MOD.SIZ.INC.L
rs3			01	001	00	00	111	100	vd	0001011	SS.APP.MOD.SIZ.DEC.L
rs3			01	000	01	00	111	100	vd	0001011	SS.APP.MOD.STR.INC.L
rs3			01	001	01	00	111	100	vd	0001011	SS.APP.MOD.STR.DEC.L
rs3			01	000	10	00	111	100	vd	0001011	SS.APP.MOD.OFS.INC.L
rs3			01	001	10	00	111	100	vd	0001011	SS.APP.MOD.OFS.DEC.L
0	000	0	01	000	00	vs1		110	vd	0001011	SS.APP.IND.SIZ.INC.1
0	000	0	01	001	00	vs1		110	vd	0001011	SS.APP.IND.SIZ.DEC.1
0	000	0	01	010	00	vs1		110	vd	0001011	SS.APP.IND.SIZ.ADD.1
0	000	0	01	011	00	vs1		110	vd	0001011	SS.APP.IND.SIZ.SUB.1
0	000	0	01	100	00	vs1		110	vd	0001011	SS.APP.IND.SIZ.SET.1
0	000	0	01	000	01	vs1		110	vd	0001011	SS.APP.IND.STR.INC.1
0	000	0	01	001	01	vs1		110	vd	0001011	SS.APP.IND.STR.DEC.1
0	000	0	01	010	01	vs1		110	vd	0001011	SS.APP.IND.STR.ADD.1
0	000	0	01	011	01	vs1		110	vd	0001011	SS.APP.IND.STR.SUB.1
0	000	0	01	100	01	vs1		110	vd	0001011	SS.APP.IND.STR.SET.1
0	000	0	01	000	10	vs1		110	vd	0001011	SS.APP.IND.OFS.INC.1
0	000	0	01	001	10	vs1		110	vd	0001011	SS.APP.IND.OFS.DEC.1
0	000	0	01	010	10	vs1		110	vd	0001011	SS.APP.IND.OFS.ADD.1
0	000	0	01	011	10	vs1		110	vd	0001011	SS.APP.IND.OFS.SUB.1
0	000	0	01	100	10	vs1		110	vd	0001011	SS.APP.IND.OFS.SET.1
0	001	0	01	000	00	vs1		110	vd	0001011	SS.APP.IND.SIZ.INC.2
0	001	0	01	001	00	vs1		110	vd	0001011	SS.APP.IND.SIZ.DEC.2
0	001	0	01	010	00	vs1		110	vd	0001011	SS.APP.IND.SIZ.ADD.2
0	001	0	01	011	00	vs1		110	vd	0001011	SS.APP.IND.SIZ.SUB.2
0	001	0	01	100	00	vs1		110	vd	0001011	SS.APP.IND.SIZ.SET.2
0	001	0	01	000	01	vs1		110	vd	0001011	SS.APP.IND.STR.INC.2
0	001	0	01	001	01	vs1		110	vd	0001011	SS.APP.IND.STR.DEC.2
0	001	0	01	010	01	vs1		110	vd	0001011	SS.APP.IND.STR.ADD.2
0	001	0	01	011	01	vs1		110	vd	0001011	SS.APP.IND.STR.SUB.2
0	001	0	01	100	01	vs1		110	vd	0001011	SS.APP.IND.STR.SET.2
0	001	0	01	000	10	vs1		110	vd	0001011	SS.APP.IND.OFS.INC.2
0	001	0	01	001	10	vs1		110	vd	0001011	SS.APP.IND.OFS.DEC.2
0	001	0	01	010	10	vs1		110	vd	0001011	SS.APP.IND.OFS.ADD.2
0	001	0	01	011	10	vs1		110	vd	0001011	SS.APP.IND.OFS.SUB.2
0	001	0	01	100	10	vs1		110	vd	0001011	SS.APP.IND.OFS.SET.2

31	30 28	27	26 25	24 22	21 20	19 15	14 12	11 7	6 0	
-	tdim	sg	funct2	b	t	vs1	funct3	vd	opcode	UIND-type

Stream Configuration Instructions (Continuation)

0	010	0	01	000	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.INC.3
0	010	0	01	001	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.DEC.3
0	010	0	01	010	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.ADD.3
0	010	0	01	011	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.SUB.3
0	010	0	01	100	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.SET.3
0	010	0	01	000	01	vs1	110	vd	0001011	SS.APP.IND.STR.INC.3
0	010	0	01	001	01	vs1	110	vd	0001011	SS.APP.IND.STR.DEC.3
0	010	0	01	010	01	vs1	110	vd	0001011	SS.APP.IND.STR.ADD.3
0	010	0	01	011	01	vs1	110	vd	0001011	SS.APP.IND.STR.SUB.3
0	010	0	01	100	01	vs1	110	vd	0001011	SS.APP.IND.STR.SET.3
0	010	0	01	000	10	vs1	110	vd	0001011	SS.APP.IND.OFS.INC.3
0	010	0	01	001	10	vs1	110	vd	0001011	SS.APP.IND.OFS.DEC.3
0	010	0	01	010	10	vs1	110	vd	0001011	SS.APP.IND.OFS.ADD.3
0	010	0	01	011	10	vs1	110	vd	0001011	SS.APP.IND.OFS.SUB.3
0	010	0	01	100	10	vs1	110	vd	0001011	SS.APP.IND.OFS.SET.3
0	011	0	01	000	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.INC.4
0	011	0	01	001	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.DEC.4
0	011	0	01	010	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.ADD.4
0	011	0	01	011	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.SUB.4
0	011	0	01	100	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.SET.4
0	011	0	01	000	01	vs1	110	vd	0001011	SS.APP.IND.STR.INC.4
0	011	0	01	001	01	vs1	110	vd	0001011	SS.APP.IND.STR.DEC.4
0	011	0	01	010	01	vs1	110	vd	0001011	SS.APP.IND.STR.ADD.4
0	011	0	01	011	01	vs1	110	vd	0001011	SS.APP.IND.STR.SUB.4
0	011	0	01	100	01	vs1	110	vd	0001011	SS.APP.IND.STR.SET.4
0	011	0	01	000	10	vs1	110	vd	0001011	SS.APP.IND.OFS.INC.4
0	011	0	01	001	10	vs1	110	vd	0001011	SS.APP.IND.OFS.DEC.4
0	011	0	01	010	10	vs1	110	vd	0001011	SS.APP.IND.OFS.ADD.4
0	011	0	01	011	10	vs1	110	vd	0001011	SS.APP.IND.OFS.SUB.4
0	011	0	01	100	10	vs1	110	vd	0001011	SS.APP.IND.OFS.SET.4
0	100	0	01	000	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.INC.5
0	100	0	01	001	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.DEC.5
0	100	0	01	010	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.ADD.5
0	100	0	01	011	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.SUB.5
0	100	0	01	100	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.SET.5
0	100	0	01	000	01	vs1	110	vd	0001011	SS.APP.IND.STR.INC.5
0	100	0	01	001	01	vs1	110	vd	0001011	SS.APP.IND.STR.DEC.5
0	100	0	01	010	01	vs1	110	vd	0001011	SS.APP.IND.STR.ADD.5
0	100	0	01	011	01	vs1	110	vd	0001011	SS.APP.IND.STR.SUB.5
0	100	0	01	100	01	vs1	110	vd	0001011	SS.APP.IND.STR.SET.5
0	100	0	01	000	10	vs1	110	vd	0001011	SS.APP.IND.OFS.INC.5
0	100	0	01	001	10	vs1	110	vd	0001011	SS.APP.IND.OFS.DEC.5
0	100	0	01	010	10	vs1	110	vd	0001011	SS.APP.IND.OFS.ADD.5
0	100	0	01	011	10	vs1	110	vd	0001011	SS.APP.IND.OFS.SUB.5
0	100	0	01	100	10	vs1	110	vd	0001011	SS.APP.IND.OFS.SET.5
0	101	0	01	000	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.INC.6
0	101	0	01	001	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.DEC.6
0	101	0	01	010	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.ADD.6
0	101	0	01	011	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.SUB.6
0	101	0	01	100	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.SET.6
0	101	0	01	000	01	vs1	110	vd	0001011	SS.APP.IND.STR.INC.6
0	101	0	01	001	01	vs1	110	vd	0001011	SS.APP.IND.STR.DEC.6
0	101	0	01	010	01	vs1	110	vd	0001011	SS.APP.IND.STR.ADD.6
0	101	0	01	011	01	vs1	110	vd	0001011	SS.APP.IND.STR.SUB.6
0	101	0	01	100	01	vs1	110	vd	0001011	SS.APP.IND.STR.SET.6
0	101	0	01	000	10	vs1	110	vd	0001011	SS.APP.IND.OFS.INC.6
0	101	0	01	001	10	vs1	110	vd	0001011	SS.APP.IND.OFS.DEC.6
0	101	0	01	010	10	vs1	110	vd	0001011	SS.APP.IND.OFS.ADD.6
0	101	0	01	011	10	vs1	110	vd	0001011	SS.APP.IND.OFS.SUB.6
0	101	0	01	100	10	vs1	110	vd	0001011	SS.APP.IND.OFS.SET.6

31	30 28	27	26 25	24 22	21 20	19 15	14 12	11 7	6 0	
-	tdim	sg	funct2	b	t	vs1	funct3	vd	opcode	UIND-type
rs3			funct2	rs2		rs1	funct3	vd	opcode	UAE-type
				Stream Configuration Instructions (Continuation)						
0	110	0	01	000	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.INC.7
0	110	0	01	001	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.DEC.7
0	110	0	01	010	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.ADD.7
0	110	0	01	011	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.SUB.7
0	110	0	01	100	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.SET.7
0	110	0	01	000	01	vs1	110	vd	0001011	SS.APP.IND.STR.INC.7
0	110	0	01	001	01	vs1	110	vd	0001011	SS.APP.IND.STR.DEC.7
0	110	0	01	010	01	vs1	110	vd	0001011	SS.APP.IND.STR.ADD.7
0	110	0	01	011	01	vs1	110	vd	0001011	SS.APP.IND.STR.SUB.7
0	110	0	01	100	01	vs1	110	vd	0001011	SS.APP.IND.STR.SET.7
0	110	0	01	000	10	vs1	110	vd	0001011	SS.APP.IND.OFS.INC.7
0	110	0	01	001	10	vs1	110	vd	0001011	SS.APP.IND.OFS.DEC.7
0	110	0	01	010	10	vs1	110	vd	0001011	SS.APP.IND.OFS.ADD.7
0	110	0	01	011	10	vs1	110	vd	0001011	SS.APP.IND.OFS.SUB.7
0	110	0	01	100	10	vs1	110	vd	0001011	SS.APP.IND.OFS.SET.7
0	111	0	01	000	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.INC.L
0	111	0	01	001	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.DEC.L
0	111	0	01	010	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.ADD.L
0	111	0	01	011	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.SUB.L
0	111	0	01	100	00	vs1	110	vd	0001011	SS.APP.IND.SIZ.SET.L
0	111	0	01	000	01	vs1	110	vd	0001011	SS.APP.IND.STR.INC.L
0	111	0	01	001	01	vs1	110	vd	0001011	SS.APP.IND.STR.DEC.L
0	111	0	01	010	01	vs1	110	vd	0001011	SS.APP.IND.STR.ADD.L
0	111	0	01	011	01	vs1	110	vd	0001011	SS.APP.IND.STR.SUB.L
0	111	0	01	100	01	vs1	110	vd	0001011	SS.APP.IND.STR.SET.L
0	111	0	01	000	10	vs1	110	vd	0001011	SS.APP.IND.OFS.INC.L
0	111	0	01	001	10	vs1	110	vd	0001011	SS.APP.IND.OFS.DEC.L
0	111	0	01	010	10	vs1	110	vd	0001011	SS.APP.IND.OFS.ADD.L
0	111	0	01	011	10	vs1	110	vd	0001011	SS.APP.IND.OFS.SUB.L
0	111	0	01	100	10	vs1	110	vd	0001011	SS.APP.IND.OFS.SET.L
0	000	1	01	000	10	vs1	110	vd	0001011	SS.APP.SGI.OFS.INC
0	000	1	01	001	10	vs1	110	vd	0001011	SS.APP.SGI.OFS.DEC
0	000	1	01	010	10	vs1	110	vd	0001011	SS.APP.SGI.OFS.ADD
0	000	1	01	011	10	vs1	110	vd	0001011	SS.APP.SGI.OFS.SUB
0	000	1	01	100	10	vs1	110	vd	0001011	SS.APP.SGI.OFS.SET
0	000	1	10	000	10	vs1	110	vd	0001011	SS.END.SGI.OFS.INC
0	000	1	10	001	10	vs1	110	vd	0001011	SS.END.SGI.OFS.DEC
0	000	1	10	010	10	vs1	110	vd	0001011	SS.END.SGI.OFS.ADD
0	000	1	10	011	10	vs1	110	vd	0001011	SS.END.SGI.OFS.SUB
0	000	1	10	100	10	vs1	110	vd	0001011	SS.END.SGI.OFS.SET
rs3			10	rs2		rs1	000	vd	0001011	SS.END

Table B.1: Unlimited Vector Extension (UVE) instruction listing for RISC-V