AVISPADO - VPU Interface

SemiDynamics Technology Services SL

Version 1.05

Roger Espasa

Pedro Marcuello

Alberto Moreno

Sebastiano Pomata

Change log

Version	Description of Change
v0.0	Initial proposal
v0.1	Extended proposal
v0.2	Rewritten interface according to discussions and feedback from BSC
v0.3	Fixes after feedback from BSC
v0.4	Added Illegal Instruction section
v1.0	Several fixes and clarifications added from BSC feedback
v1.01	Clarifications in indices sign-extension and immediate values, fixed typos.
v1.02	Clarified data sent in vector store operations when vstart is not zero
v1.03	A kill signal now precludes sending a sync_end signal for a given instruction
V1.04	VLoad retries should now wait until a sync_end has been received
V1.05	Increased instruction ID width to 5 bits

Contents

1. Introduction

The current document presents a detailed description of the interface between the Avispado core implemented by SemiDynamics and the Vector Processing Unit (VPU) implemented by the Barcelona Supercomputing Center. The description contains an exhaustive list of all the signals and buses that connect both systems. This includes all the protocols for data exchange as well as the specific timings.

2. Interface signals and buses

Figure 1 shows a list of all the signals connecting the VPU with Avispado. Some signals are grouped together to indicate that they belong to the same type of transaction. For example, all the signals belonging to the instruction issue are grouped as *ISSUE*.

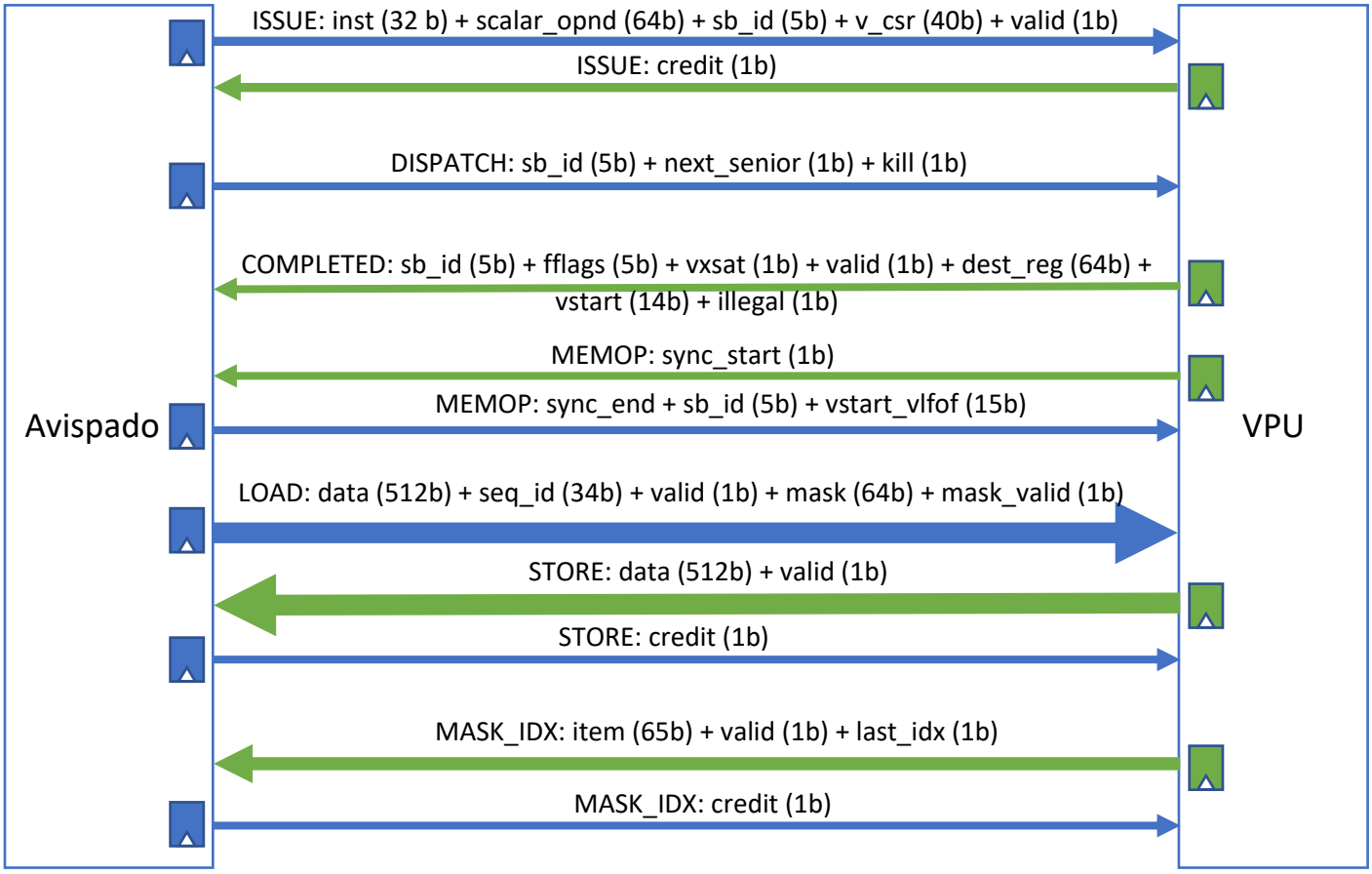

Figure 1. Signals and buses connecting Avispado to the VPU

Besides the signals themselves, flip-flop representations can be seen in both modules. A flip-flop is placed on the output of every signal/bus. This indicates that, on clock arrival, output signals must not traverse any logic in the origin module (i.e. the signal must exit the module directly from a flip-flop).

Table 1 shows a list of signals along with a small description. The direction of the signals is indicated by an arrow (→ from Avispado to the VPU, ← from the VPU to Avispado).

Table 1. Signals and their descriptions

Signal	Size(b)	Group	Direction	Description
inst	32	issue	AV→	Instruction fetched by Avispado
scalar_opnd	64	issue	AV→	Scalar value from Avispado's RF
sb_id	5	issue	AV→	Scoreboard ID used to identify the instruction while on the fly
v_csr	40	issue	AV→	Vector CSR to be used by the VPU
valid	1	issue	AV→	Indicates valid data on *issue* group
credit	1	issue	←VPU	VPU returns a credit to Avispado
sb_id	5	dispatch	AV→	ID of the instruction in reference by the ***dispatch*** group
next_senior	1	dispatch	AV→	An instruction becomes next senior
kill	1	dispatch	AV→	An instruction must be killed
sb_id	5	completed	←VPU	ID of the instruction in reference by the ***completed*** group
fflags	5	completed	←VPU	Floating-point accrued exception flags
vxsat	1	completed	← VPU	Fixed-point accrued saturation flag
valid	1	completed	← VPU	Valid data on ***completed*** group
dest_reg	64	completed	← VPU	Result scalar value
vstart	14	completed	← VPU	vstart value in case of traps or retry
illegal	1	completed	← VPU	Illegal instruction
memop_sync_start	1	memop	← VPU	VPU is ready to execute a memory op
sync_end	1	memop	AV→	All data has been transmitted on current memory operation
sb_id	5	memop	AV→	ID of the instruction in reference by the ***memop*** group
vstart_vlfof	15	memop	AV→	vstart/vl value after memory operation. It can only contain vl on f-o-f loads
data	512	load	AV→	Data fetched from memory
seq_id	34	load	AV→	Sequence ID for a given chunk of data
valid	1	load	AV→	Whether there is valid data on the bus
mask	64	load	AV→	Mask assigned to the load operation
mask_valid	1	load	AV→	Whether the mask is valid
data	512	store	← VPU	Data to store on memory
valid	1	store	← VPU	Whether the store data is valid
credit	1	store	AV→	Avispado returns a store credit
item	65	mask_idx	← VPU	Mask/Index shared bus for indexed memory operations
valid	1	mask_idx	← VPU	Whether ***item*** contains valid data
last_idx	1	mask_idx	← VPU	Whether the mask/index being transmitted is the last one
credit	1	mask_idx	AV→	Avispado returns a mask credit

3. Credit system

In order to prevent potential timing issues, the availability of resources between Avispado and the VPU is communicated by a system of credits. That is, systems that may become busy will assign a specific number of credits. As long as there are credits left, the allocator may assume full availability of a system. In particular, credits are used for:

- **Instruction issue**. The number of credits correspond to the depth of the Instruction Queue of the VPU. Avispado consumes a credit every time it issues an instruction. A credit is recovered for every cycle that the signal ***issue.credit*** is set. This signal is completely decoupled from any other signal (such as the completion of an instruction) and may be set/unset at any moment.
- **Masks**. Avispado has a limited capacity to store masks (i.e. two entries of 64 bits). A credit is consumed for every cycle that the signal ***mask_idx.valid*** is set. Credits are returned by setting the signal ***mask_idx.credit***, which might happen at any moment and completely decoupled of any other signal or transaction.
- **Store**. Avispado can store up to 2 KB of store data, which corresponds to 32 packets of 512b. A store credit is consumed for every cycle that the signal ***store.valid*** is set. The signal ***mask_idx.item store.credit*** is used to return the credits. As in previous cases, this may happen at any moment.

The initial state (i.e. the state after a reset) assumes that resources are totally available, and all credits are implicitly granted.

Note: The signals ***issue.valid***, ***mask_idx.valid*** and ***store.valid*** always consume a credit irrespective of the situation in which they are set (even if those signals are ignored for other purposes). Similarly, signals ***issue.credit***, ***mask_idx.credit*** and ***store.credit*** always return a credit, independently of the context in which they are set.

4. CSRs

The RISC-V spec defines a few CSRs that modify the behavior of vector instruction. These CSR are stored on Avispado, but they are often needed by the VPU, which must keep a copy when necessary. For this reason, the interface provisions for a bus used to transmit these signals between Avispado and the VPU.

On instruction issue, Avispado sets the bus ***issue.v_csr*** which contains, for the issued instruction:

- ***v_csr*** *[13:0]* = ***vstart***
- ***v_csr*** *[28:14]* = ***vl***
- ***v_csr*** *[30:29]* = ***vxrm***
- ***v_csr*** *[33:31]* = ***frm***
- ***v_csr*** *[35:34]* = ***vlmul***
- ***v_csr*** *[38:36]* = ***vsew***
- ***v_csr*** *[39]* = ***vill***

Additionally, instructions executed on the VPU may require updating CSRs. In particular, for instructions that set the ***vxsat*** flag (fixed-point saturation) and the ***fflags*** (floating-point flags), the VPU will send updated copies on instruction completion (see *Finishing a vector instruction*).

5. Issuing a vector instruction

This section defines the interaction between Avispado and the VPU on vector instruction issue. Figure 2 shows the basic issue protocol. The row labelled *Pipeline* shows how Avispado issues instructions towards the VPU on the execution stage. If there are credits available (in this case, 4 credits are available, as represented by the row *Credits from VPU*), Avispado sets the following signals:

- ***issue.inst***: The vector instruction, as fetched from the instruction memory.
- ***issue.sb_id***: A scoreboard id that Avispado uses to track an instruction. This value is used between issue and commit in order to identify a specific instruction.
- ***issue.scalar_opnd***: Some instructions require an operand from the scalar register file. In these cases, this bus contains the value from the register file. Note that this field is reserved for registers only. Immediate values can be obtained by decoding the ***issue.inst*** field.
- ***issue.v_csr***: Contains the relevant vector CSRs for the execution of the instruction.
- ***issue.valid***: Only set when the previous signals contain valid values.

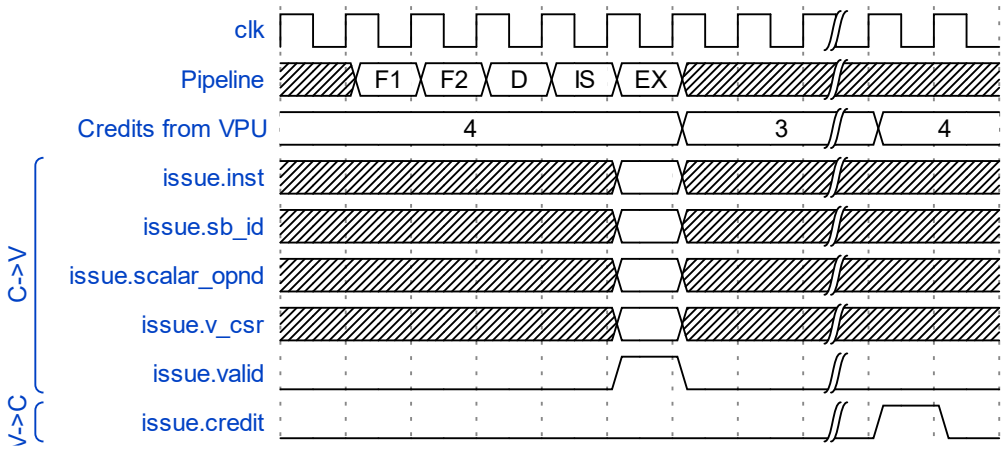

Figure 2. Vector instruction issue

All these signals are set on the same cycle with a duration of one cycle and assume the consumption of one credit. The VPU will return the credit whenever it has freed the allocated resource, at an arbitrary time in the future. This is done by setting signal ***issue.credit*** for one cycle per credit.

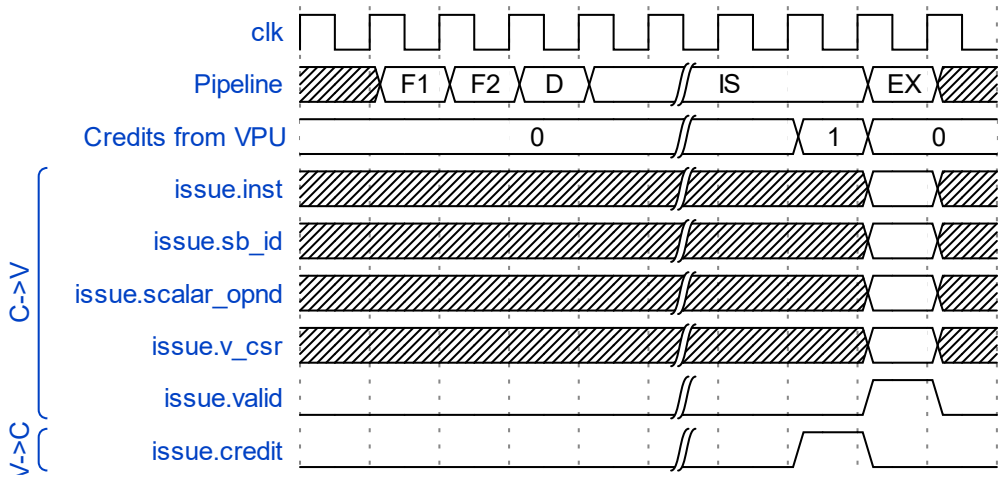

Figure 3. Avispado waits for a credit

Potentially, Avispado might have a vector instruction to issue but no credits are available. In this case, Avispado will stall on the issue stage until the VPU indicates its availability by issuing a credit. Figure 3 shows this scenario. In this case, Avispado stalls until ***issue.credit*** is set, potentially issuing an instruction in the immediate cycle after receiving a credit.

6. Instruction speculation

Avispado core uses speculative execution of instructions in order to increase performance. In some circumstances, Avispado might issue an instruction speculatively towards the VPU. This means that, potentially, an issued instruction might need to be killed, possibly requiring a rollback of the core and VPU states. Note that Avispado executes branches in one cycle. From the VPU perspective, this means that Avispado will never speculate on a branch. It can be assumed that speculation only happens on a trap (e.g. an exception) or, possibly, other rare events that require flushing the pipeline (e.g. some *rcsr* instructions).

The VPU is allowed to decide whether to execute speculatively or not in a per-instruction basis. For this reason, the ***dispatch*** bus has been added to the interface between Avispado and the VPU. This bus consists of three fields:

- ***dispatch.sb_id***: This signal is used to indicate the ID of the specific instruction that the other signals of the bus refer to.
- ***dispatch.next_senior***: If set, the instruction with ID ***dispatch.sb_id*** becomes the *next senior*. This means that any older instruction is now a *senior* and thus cannot generate a trap. Consequently, it is guaranteed that the instruction will not be killed (i.e. the instruction is no longer speculative).
- ***dispatch.kill***: If set, the instruction with ID ***dispatch.sb_id*** was incorrectly issued and must be killed. Any Avispado or VPU state updated by this instruction must be reverted. This signal also implies that any younger instruction will have to be killed too.

It is guaranteed that, for every issued instruction, either *next_senior* or *kill* will eventually be set. Furthermore, they will always arrive in the exact same order that they were issued. In particular, this means that, on instruction kill, Avispado is responsible for sending kill requests for every instruction that is killed as a result of a failed speculation.

Figure 4 shows a speculative instruction. After an arbitrary amount of cycles, the instruction becomes the *next senior*, making it safe to execute non-speculatively. If the VPU decides not to speculate, this signal indicates that the instruction may now be executed non-speculatively.

Figure 5 shows the opposite case. A speculative instruction is found to be incorrectly issued after an arbitrary number of cycles. The signal *kill* is set, and any state updated by this instruction must be reverted. After an instruction is killed, Avispado does not expect any other transaction or signal related to this instruction. Load instructions may still require some additional transactions after a kill signal, as detailed in section *Killing load instructions*.

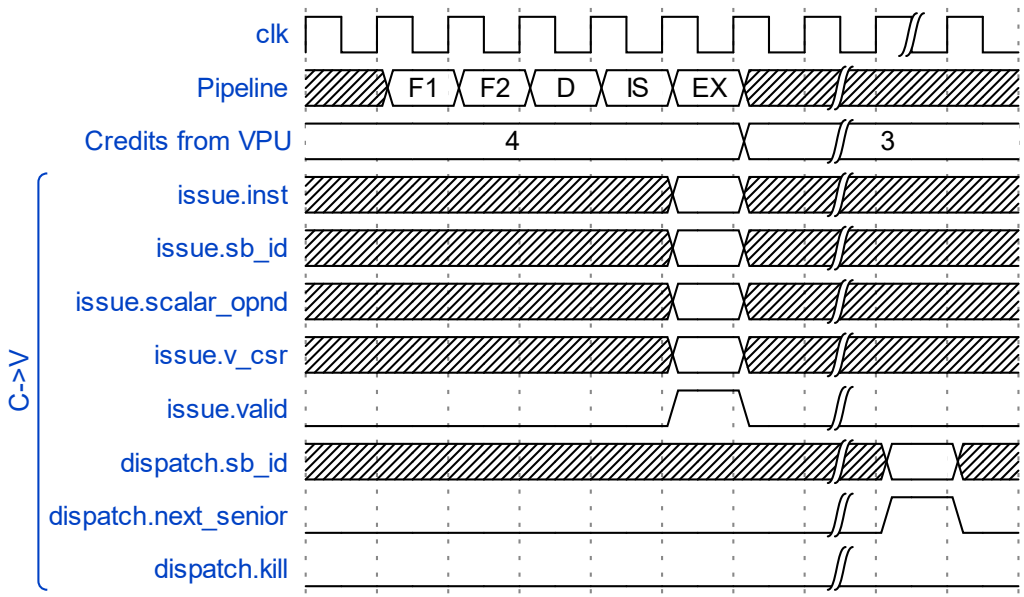

Figure 4. Issued instruction becomes next senior

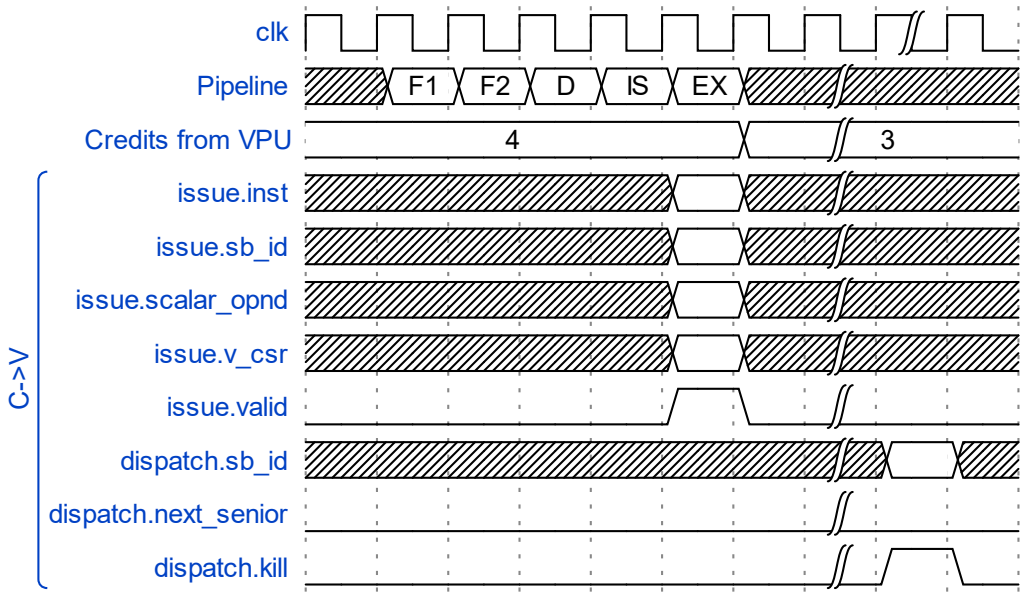

Figure 5. Failed speculation requires killing an issued instruction

It is also possible to issue an instruction in a non-speculative way. This happens when Avispado knows that the instruction will be executed, older instructions cannot cause a trap and this instruction is the next senior. In this case, the signal ***dispatch.next_senior*** may be set on the same cycle that the instruction is issued. This is the case depicted by Figure 6. Note that the analogue case of ***dispatch.kill*** set on the same cycle at which the instruction is issued cannot happen.

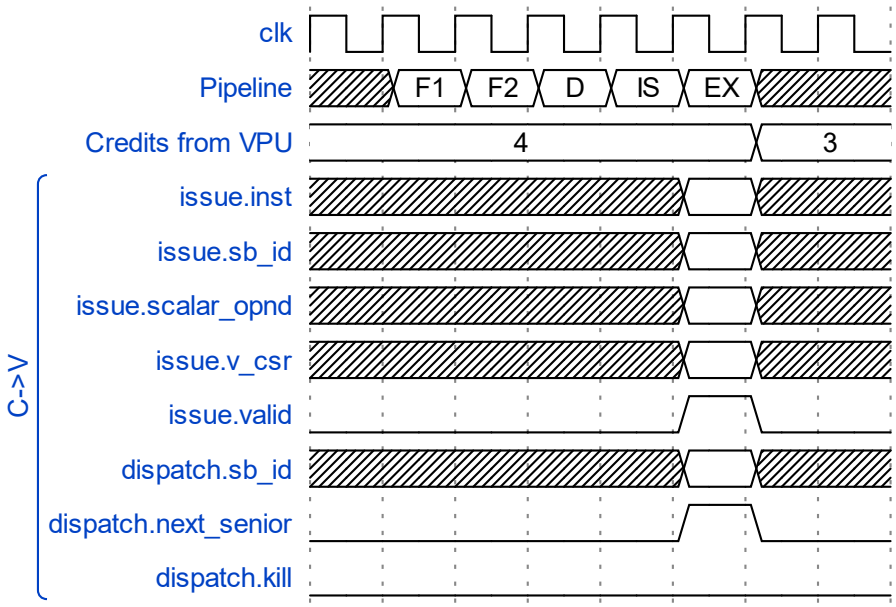

Figure 6. Issuing a non-speculative instruction

7. Finishing a vector instruction

This section describes the protocol to finish an instruction, which must be followed by any vector instruction that is not killed. For killed instructions, see *Instruction speculation*.

Avispado will only commit a vector instruction after the VPU has informed of its completion. This is done by using the ***completed*** bus, which contains the following signals:

- ***completed.valid*****:** Whether the information contained in the *completed* bus is valid.
- ***completed.sb_id***: The scoreboard ID of the completed instruction.
- ***completed.fflags***: Floating-point accrued exception flags, which must be set with the correct value for every vector instruction that might modify them. For instructions that do not modify it, this field must be set to zero.
- ***completed.vxsat***: Fixed-point accrued saturation flag, which must be set with the correct value for every vector instruction that might modify them. For instructions that do not modify it, this field must be set to zero.
- ***completed.vstart***: vstart value after the execution of an instruction. It should always be zero except in case of load retry (see *Retrying load instructions*).
- **completed.dest_reg**: Contains a scalar value in case the instruction writes into the scalar register file.
- **completed.illegal**: Bit value indicating whether the instruction is illegal and thus was not executed.

Note that for any instruction, the ***completed.valid*** signal can only be set after receiving a ***dispatch.next_senior*** with the *sb_id* matching the instruction. Conversely, a ***completed.valid*** signal will not be set for any instruction that receives a ***dispatch.kill***.

Figure 7 shows an example of instruction completion (assuming it was not killed). After an arbitrary amount of time, the VPU completes execution and sets the ***completed*** bus with valid data. Note that the VPU is not required to return a credit on completion and might do so later.

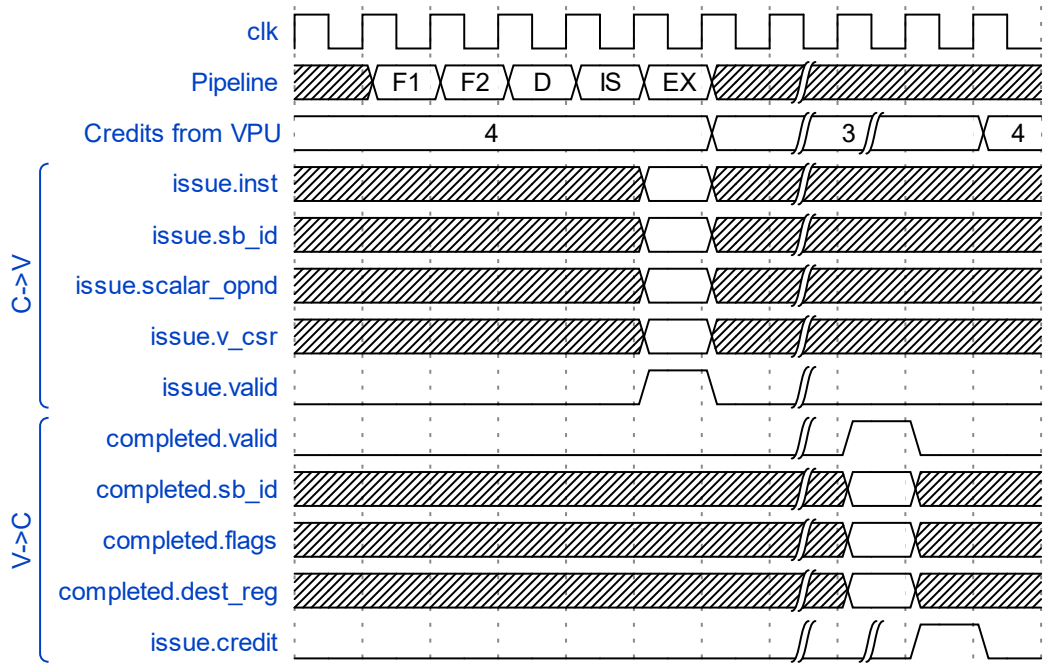

Figure 7. Instruction completion

8. Illegal Instructions

A vector instruction is considered illegal if one of the following happens:

- **Invalid instruction encoding**. This corresponds to an instruction that has a vector opcode, but the encoding violates the specs. For example, by encoding a reserved value in one of the fields (e.g. a LOAD instruction with the *mop* field set to 001, which is reserved, falls into this category).
- **Invalid configuration**. A vector instruction with a valid encoding can still be considered illegal by the spec if it is not supported by the current configuration (i.e. the value in the CSRs). For example, a widening operation with an LMUL of 8.

Note: Some illegal instructions are ambiguous with respect to these categories. In particular, instructions that technically have a valid encoding but require performing an illegal operation regardless of the configuration. This only happens for instructions that require that the register source does not overlap register *v0* when the operation is masked (e.g. widening operations, v*iota, vslideup, vrgather, vrcompress,* etc. instructions). Because detection of these cases requires fine grain decoding, this is considered to fall into the *Invalid configuration* category.

The first kind of illegal instructions can be detected at the decoding stage without performing a thorough decoding of the instruction. This is implemented by Avispado and it is guaranteed that an instruction with an invalid encoding will not be sent to the VPU.

The second type, *Invalid configuration*, can only be detected on the execution stage, as it requires knowledge of the CSRs. Additionally, detection of these issues necessitates a fine grain decoding in order to identify source/destination registers and their relationship. Because the VPU is responsible of complete decoding of vector instructions and has full knowledge of the CSRs at decode time, it is

the natural candidate to raise illegal instruction exceptions when an invalid configuration is detected. Notice that the CSR *vill* must also be considered and an exception should raise when set.

To support this functionality, the ***completed.illegal*** signal contains a bit indicating whether the completed instruction is illegal or not. When this signal is set on finishing an instruction (see *Finishing a vector instruction*), Avispado will raise an exception on the instruction commit. This will trigger a pipeline flush which eventually leads to killing any instruction younger than the one that raised the exception. It is thus guaranteed that any vector instruction that was issued after the illegal instruction will eventually receive a kill signal (see *Instruction speculation*).

Note that the VPU is responsible for detecting invalid configurations even for memory operations. In these cases, the VPU complete the instruction by sending the ***completed*** bus and raising the ***completed.illegal*** signal without setting ***memop.sync_start*** (see below *Load instructions* and *Store instructions*).

9. Types of instructions

This section presents the timings for the interface between Avispado and the VPU concerning all the possible types of vector instructions. Timing diagrams will be shown for every group of instructions, showing standard and corner cases.

9.1. Arithmetic instructions

Arithmetic vector instructions are vector instructions that are executed independently at the VPU. This group of instructions includes any instruction that is not a memory instruction (i.e. load/store) and it is not a configuration instruction (i.e. *vsetvl{i}* and *wcsr*). The role of Avispado on these instructions is exclusively to issue them to the VPU and wait for the completion signal. As such, these are the simplest from the interface perspective.

In fact, all the necessary interaction for this type of instructions has already been described. As with every vector instruction, the interaction between Avispado and the VPU consists of the timings described at sections *Issuing a vector instruction*, *Instruction speculation* and *Finishing a vector instruction*. Figure 8 shows a complete example for an arithmetic instruction.

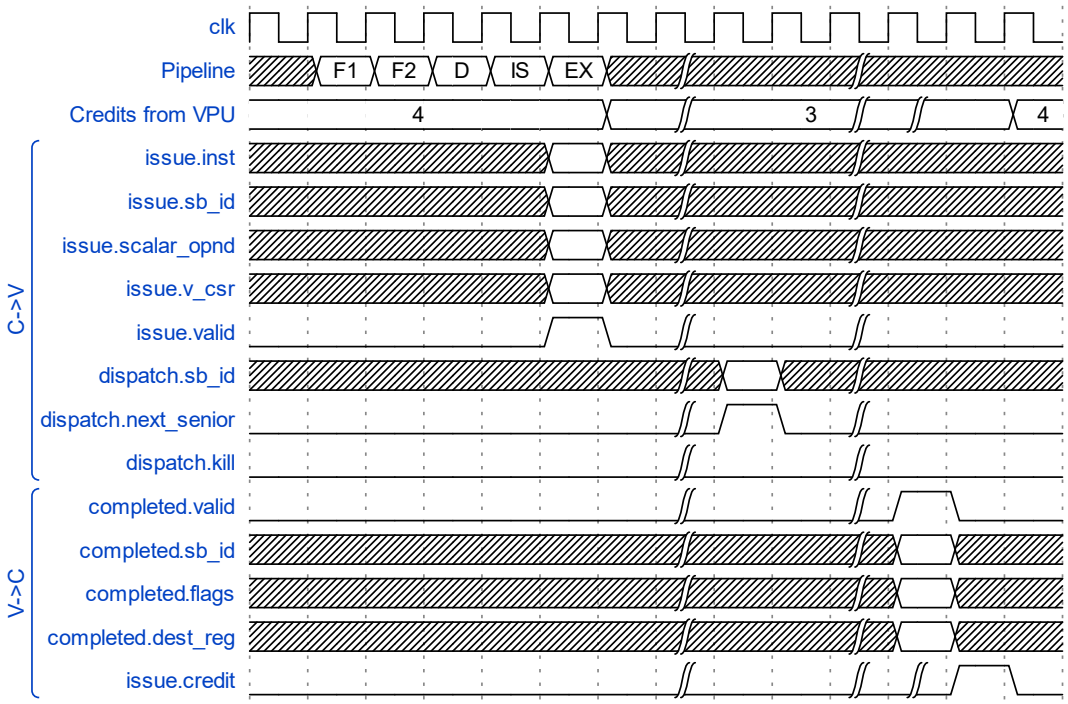

Figure 8. Arithmetic instruction

9.2. Load instructions

Vector load instructions are those that require fetching data from memory. These instructions must be executed in parallel by both Avispado and the VPU.

From the perspective of the interface between Avispado and the VPU, there are two general types of load instructions:

- Strided loads are instructions that access a memory element given by an address, and subsequent elements. The stride defines the offset that separates the subsequent elements. Supported values for strided loads are 1, 2, 4, -1, -2 and -4 times the Standard Element Width (SEW). Different values for the stride are possible but will be transmitted in a per-element basis, not taking advantage of the stride.
- Indexed loads are instructions that access individual elements defined by a group of addresses. Data from indexed loads are always served in a per-element basis.

Additionally, any load operation might be masked and unmasked.

Any vector memory operation (load or store) whose elements are not naturally aligned to memory in their Standard Element Width will eventually cause an UNALIGNED exception.

9.2.1. Sequence ID

Before showing specific timings for a load operation, it is important to first define the *sequence ID*, which is used to identify the specific memory elements being transmitted to the VPU. The need for an identifier appears due to the memory system not guaranteeing the in-order arrival of elements.

Thus, for every chunk of data being transmitted between Avispado and the VPU, a sequence ID is transmitted by the ***load.seq_id*** bus containing the following information:

- Vector register (***v_reg***): Identifies the logical vector register that the data should be written to. In case of LMUL > 1, this field will take into account that a single instruction will have multiple logical registers.
- Element number (***el_id***): A sequential identifier for the lowest (i.e. smallest id) valid element contained in the chunk of data being transmitted. The identifier is relative to the vector register ***v_reg***.
- Offset element (***el_off***): An offset in the chunk of data being transmitted identifying the first valid element. The offset is calculated according to the current Standard Element Width (SEW).
- Number of elements (***el_count***): The number of valid elements being transmitted. Note that masked elements (i.e. elements in which a mask is 0) are also valid elements.
- Scoreboard ID (***sb_id***): The scoreboard ID of the load instruction that requested the data.

The sequence ID is then a 34-bit element composed of the following sub-elements:

- ***seq_id****[4:0]* = ***v_reg***
- ***seq_id****[15:5]* = ***el_id***
- ***seq_id****[21:16]* = ***el_off***
- ***seq_id****[28:22]* = ***el_count***
- ***seq_id****[33:29]* = ***sb_id***

Examples of sequence ID for SEW=32b, LMUL=1:

The following examples assume a SEW of 32b, which implies, for a 512b cache line, up to 16 elements. Additionally, an LMUL=1 means that a load fetches up to 512 elements. Negative strides are included along with positive strides to show how it affects the interpretation of the sequence ID. As a general rule, a negative stride "mirrors" the sequence ID with respect to the line. In particular, element ID still refers to the element with the smallest ID contained in the line, and element count to the total number of elements. But the offset of an element is specified "from left to right" instead of from "right to left" (assuming "right" refers to the less significant bit in the line):

Element offset with positive stride:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Element offset with negative stride:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

The examples show a chunk of 512b of data transmitted between Avispado and the VPU, split by elements of 32b. Cells in blue indicate valid elements, along with a value representing the element number. White cells represent invalid data that must be discarded. Two rows are specified, row "ID", which refers to the element ID, and row "@", which refers to the offset (in bytes) with respect

to the base address specified by the vector instruction. Additionally, a possible base address that would generate such a sequence ID is included.

Unit stride, v_reg = A, el_id= 32, el_off = 0, el_count=16, sb_id = B

(e.g.: Base@=10240)

ID	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
@	188	184	180	176	172	168	164	160	156	152	148	144	140	136	132	128

-1SEW-Stride (stride -4), v_reg = A, el_id= 32, el_off = 0, el_count=16, sb_id = B

(e.g.: Base@=10236)

ID	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
@	-128	-132	-136	-140	-144	-148	-152	-156	-160	-164	-168	-172	-176	-180	-184	-188

Unit stride, v_reg = A, el_id= 0, el_off = 11, el_count=5, sb_id = B

(e.g.: Base@=10284)

ID	4	3	2	1	0											
@	16	12	8	4	0											

-1SEW-Stride (stride -4), v_reg = A, el_id= 0, el_off = 11, el_count=5, sb_id = B

(e.g.: Base@=10256)

ID												0	1	2	3	4
@												0	-4	-8	-12	-16

Unit stride, v_reg = A, el_id= 508, el_off = 0, el_count=4, sb_id = B

(e.g.: Base@=10256)

ID													511	510	509	508
@													2044	2040	2036	2032

-1SEW-Stride (stride -4), v_reg = A, el_id= 508, el_off = 0, el_count=4, sb_id = B

(e.g.: Base@=10284)

ID	508	509	510	511												
@	-2032	-2036	-2040	-2044												

2SEW-Stride (stride 8), v_reg = A, el_id= 16, el_off = 0, el_count=8, sb_id = B

(e.g.: Base@=10240)

ID		23		22		21		20		19		18		17		16
@		184		176		168		160		152		144		136		128

-2SEW-Stride (stride -8), v_reg = A, el_id= 16, el_off = 0, el_count=8, sb_id = B

(e.g.: Base@=10236)

ID	16		17		18		19		20		21		22		23	
@	-128		-136		-144		-152		-160		-168		-176		-184	

2SEW-Stride (stride 8), v_reg = A, el_id= 0, el_off = 10, el_count=3, sb_id = B

(e.g.: Base@=10280)

ID		2		1		0										
@		16		8		0										

-2SEW-Stride (stride -8), v_reg = A, el_id= 0, el_off = 10, el_count=3, sb_id = B

(e.g.: Base@=10260)

ID											0		1		2	
@											0		-8		-16	

2SEW-Stride (stride 8), v_reg = A, el_id= 507, el_off = 0, el_count=5, sb_id = B

(e.g.: Base@=10280)

ID								511		510		509		508		507
@								4088		4080		4072		4064		4056

-2SEW-Stride (stride -8), v_reg = A, el_id= 507, el_off = 0, el_count=5, sb_id = B

(e.g.: Base@=10260)

ID	507		508		509		510		511							
@	-4056		-4064		-4072		-4080		-4088							

3SEW-Stride (stride 12), v_reg = A, el_id= 123, el_off = 13, el_count=1, sb_id = B

(e.g.: Base@=10288)

ID			123													
@			1476													

-3SEW-Stride (stride -12), v_reg = A, el_id= 123, el_off = 13, el_count=1, sb_id = B

(e.g.: Base@=10252)

ID														123		
@														-1476		

9.2.2. Strided load instructions

Strided load operations are the simplest kind of instruction from the perspective of the interface. This is because the addresses for a strided load are stored on Avispado and do not need to be transmitted from the VPU. As such, the additional signals required to complete these instructions are limited to synchronization signals and the data from memory (along with the sequence id).

For simplicity, the following figures hide most of the signals related to issue/finish that have been already defined. Additionally, it is assumed that there is no speculation unless otherwise stated.

Figure 9 shows an example of a load instruction without mask. After the instruction has been issued, Avispado waits for a pulse of ***memop.sync_start*** before executing the instruction (i.e. before generating addresses). With that signal, the VPU acknowledges that it is ready to start receiving data from the load until the completion of the instruction. Avispado starts sending data an undetermined number of cycles afterwards by driving signals ***load.valid***, ***load.data*** and ***load.seq_id***.

Note: In case of illegal instruction is detected by the VPU, the signal ***completed.illegal*** (along with the rest of the information of the ***completed*** bus) must be sent instead of raising ***memop.sync_start***.

After the transfer is completed, Avispado sets signal ***memop.sync_end*** to indicate that no more data is being transmitted. This signal is coupled with ***memop.sb_id*** in order to allow the VPU to identify which load operation is finishing. This enables completing different vector loads in disorder. Finally, the signal ***memop.vstart_vlfof*** is set with the correct vstart value. In general, vstart is 0 on an exception-free execution, but it might have a different value if an exception occurred.

As in the general case for vector instructions, the VPU must acknowledge the end of the instruction by correctly setting the ***completed*** bus any number of cycles after the reception of ***memop.sync_end*** (see *Finishing a vector instruction*).

Figure 10 shows a similar example, but in this case the instruction produces an exception in address 4. Most of the execution is performed in the same way, with the only difference being the number of elements transmitted and a vstart value corresponding to the address that caused the exception.

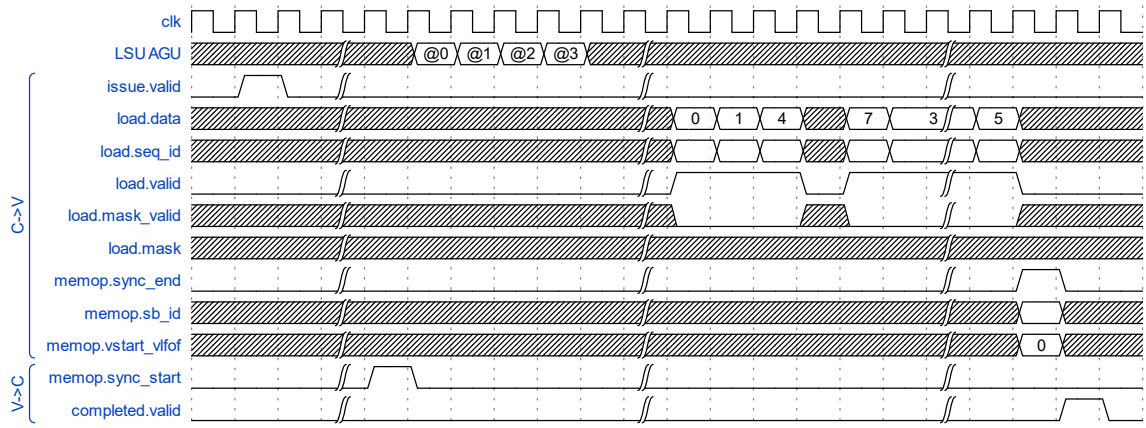

Figure 9. Strided load instruction

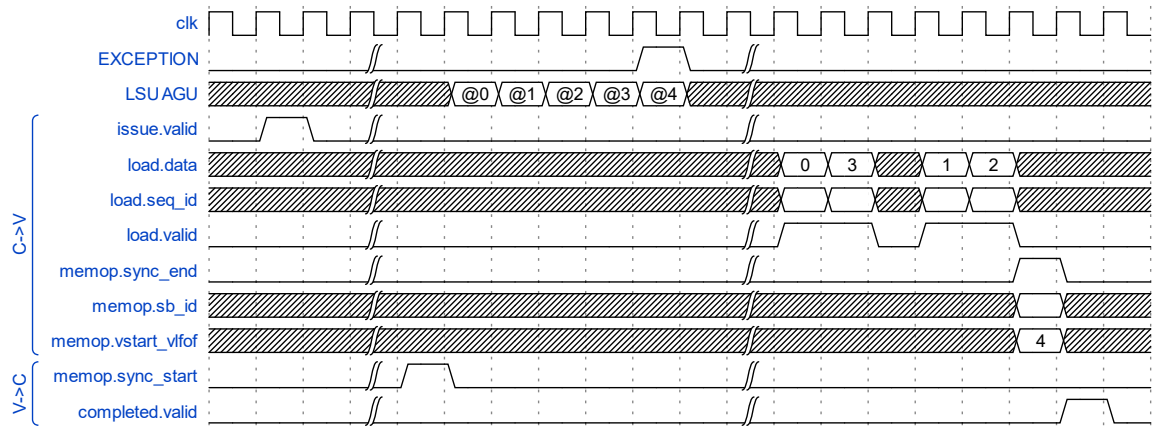

Figure 10. Strided load with exception

Strided load operations also allow the use of masks. Masks are transmitted from the VPU to Avispado, which has a limited capacity for buffering. Because of that, another system of credits is required to indicate the availability of resources to store them. In particular, Avispado has a capacity of 2 64-bit masks, which is the amount of credits granted to the VPU. If there are available credits, a mask is transmitted through the least significant 64 bits of the bus ***mask_idx.item*** (the bit 65 is only used for indexed memory access and is ignored otherwise) when signal ***mask_idx.valid*** is set. This implicitly consumes one credit, which is later returned by Avispado by setting the signal ***mask_idx.credit*** one cycle per returned credit.

For the case of *vstart* different than 0, masks for elements smaller than *vstart* are not sent, i.e. the least significant bits of ***mask_idx.item*** correspond to elements above or equal to *vstart*.

An additional bus, ***load.mask*** is used for masked load instructions. This bus is used by Avispado to transmit a mask back to the VPU. This mask belongs to the specific chunk of data it is associated with but represents exclusively valid elements (see Sequence ID). In case the mask is smaller than the 64 bits, the valid bits are contained in the least significant bits.

Note: The signal ***load.valid*** is used to indicate that there is a package transmitted on the ***load*** bus. A value of zero indicates that there is no information being transmitted on the bus, including no mask. The signal ***load.mask_valid*** is zero when the bus ***load.mask*** should be ignored and one when it contains a real mask. If ***load.valid*** is zero, the value of ***load.mask_value*** is considered *don't care*.

Example:

Assume a 2-strided masked load. The first mask sent by the VPU contains:

mask_idx.item[4:0] = 5'b01101

After a few cycles, the chunk of data returned by Avispado through the ***load.data*** bus is of the form:

2SEW-Stride (stride 8), SEW=32b, el_id= 0, el_off = 6, el_count=5

Then the mask returned along with this chunk of data will be:

load.mask[63:0] = {59'b0 ,5'b01101}

This mask should be interpreted as the bit *n* of ***load.mask*** corresponds to the valid element *n* of the chunk of data.

Figure 11 shows a complete example for a strided load. In this case, signal ***memop.sync_start*** indicates that the VPU is ready to receive data. From this moment, the ***mask_idx*** bus can be used to transmit the mask towards Avispado. In the example, this happens on the same cycle that ***memop.sync_start*** is set, but this may happen any number of cycles afterward. The VPU may send masks as long as it has credits available but must stall otherwise. Avispado grants credits as resources become available, but it may start returning data before the whole mask is transmitted. Transmission of data is identical to the non-masked case, with the addition of the retransmission of the mask through the ***load.mask*** bus.

The transmission of data ends like in the non-masked load and must be similarly acknowledged by the VPU through the ***completed*** bus. The procedure in case of exception is identical to the previous case (i.e. setting vstart to the element that caused the exception).

Note: The VPU must transmit the mask for every element even if the transmitted mask if composed exclusively of zeroes. Similarly, Avispado will always transmit data even when masks are composed of zeroes. In this case, the data transmitted by Avispado may contain arbitrary values.

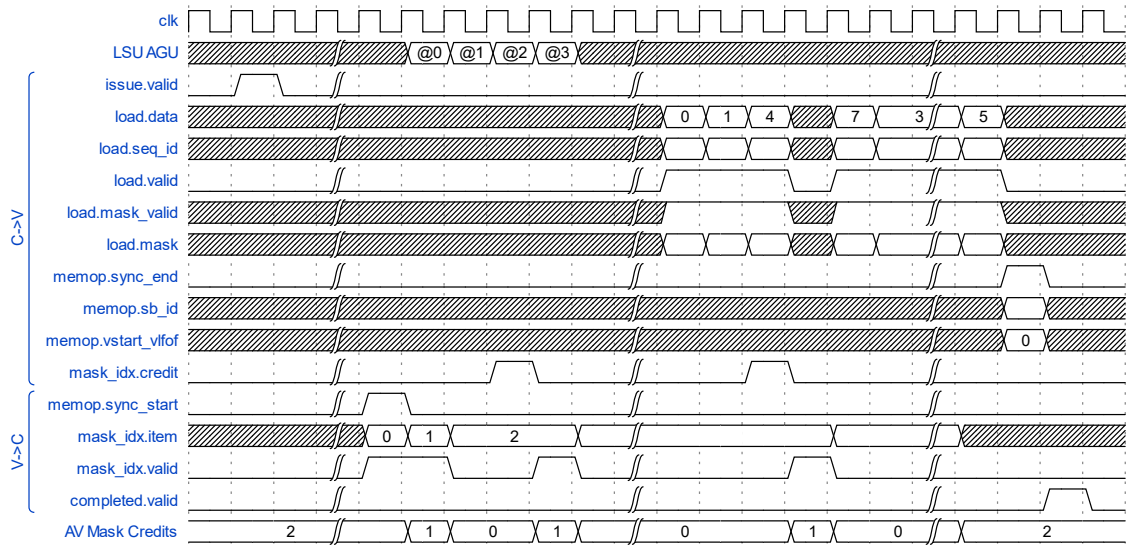

Figure 11. Strided load with mask

It is important to note that load data transmitted from Avispado to the VPU usually contains more than one valid element as long as the stride belongs to the set {1, 2, 4, -1, -2, -4} times the SEW. In case any other stride is used, regardless of SEW, only one valid element per chunk of data is transmitted (see Sequence ID).

9.2.3. Indexed load instructions (gather)

Indexed loads instructions differ from strided ones in that every element has a unique index, which is stored on the VPU. This requires the VPU to send indices towards Avispado by using the 64 least significant bits of the bus ***mask_idx.item***. The most significant bit of ***mask_idx.item*** is used to encode the mask of the element:

- ***mask_idx.item***[63:0] = Index
- ***mask_idx.item***[64] = Mask

When SEW < 64b, indices are sign-extended to 64b by the VPU. For every index sent, signal ***mask_idx.valid*** must be set. As is the case for strided loads, setting this signal for one cycle consumes one mask credit, which must be returned by Avispado by setting signal ***mask_idx.credit***. The VPU is required to send indices even when masked to zero. This is necessary so that Avispado can track the element number and be able to generate a valid sequence ID. In case of *vstart* different than 0, indices of elements with ID smaller than *vstart* are not sent. Nonetheless, when all the remaining elements are masked to zero, the VPU does not need to send them. Because of that optimization, Avispado cannot know the number of elements that it will receive. This is solved by the VPU setting signal ***mask_idx.last_idx*** along with the last element sent. Note that ***mask_idx.last_idx*** must be zero otherwise.

As with the case of strided load, Avispado sets data on the ***load*** bus along with the mask, in the case of masked gather. Since only one element is transmitted, only the least significant bit of the bus ***load.mask*** is used in this case.

Figure 12 depicts the waveform of an indexed load. Like in the masked strided load, the VPU sets the signal ***memop.sync_start*** along with the first index, which prompts Avispado to generate addresses and request data. When this data comes back from memory, Avispado drives the ***load*** bus. After the last element is sent back to the VPU, Avispado sets the ***memop*** bus with the scoreboard ID and vstart (0 if no exception occurred). This must be acknowledged by the VPU via the ***completed*** bus (see *Finishing a vector instruction*).

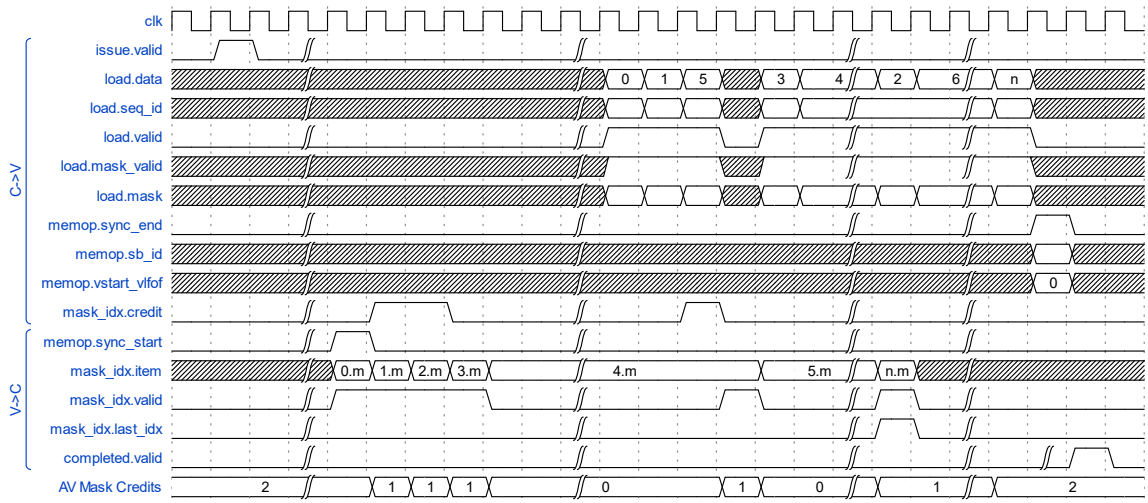

Figure 12. Indexed load

Note: Exceptions for indexed loads are handled in the same way as the strided counterparts. If an exception is detected, the ***memop*** bus is appropriately set with a vstart value indicating the element that had an exception.

For the indexed loads, it is possible that an exception is detected and ***memop*** is set before the VPU has finished sending indices. In this case, the VPU may continue sending indices (which will be ignored by Avispado) until it is ready to acknowledge the completion of the instruction by setting the ***completed*** bus. Additionally, the VPU must guarantee that it will not set ***memop.sync_start*** for a different memory instruction until it has stopped sending indices.

9.2.4. Fault-only-first load instructions

Fault-only-first load instructions are a specific type of load that can only trap in element 0. If an element greater than 0 causes an exception, the CSR *vl* is updated to the previous element and no trap is taken. This type of instruction requires that Avispado informs the VPU of a new *vl* value. In order to do this, the bus ***memop.vstart_vlfof*** is used. Typically, this bus contains *vstart* at the end of an execution, so it is important to specify exactly when it may contain a *vl* value instead:

- If an instruction is not a fault-only-first load it contains vstart
- If an instruction is a fault-only first load and ***memop.vstart_vlfof*** drives a zero, it contains vstart. In this case it might mean that the instruction finished correctly or that the first element raised an exception.
- If an instruction is a fault-only-first load and ***memop.vstart_vlfof*** has a value different than zero, then it contains a *vl* value.

Figure 13 shows an example of a fault-only-first load raising an exception and setting a *vl* value. From the interface point of view, this instruction is almost identical to a strided load with exception. The only difference for this case is that the ***memop.vstart_vlfof*** bus points to the last element that was processed, rather than the element that caused an exception.

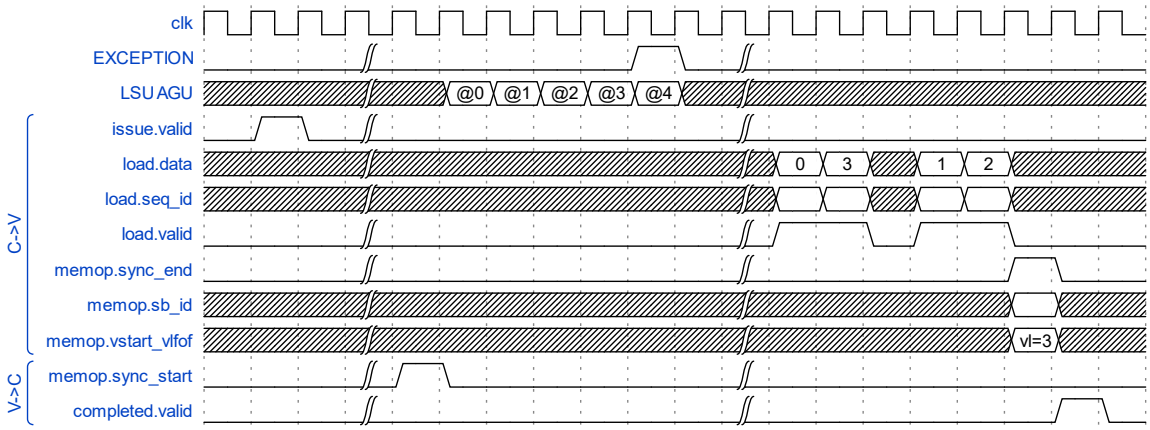

Figure 13. Fault-only-first load with exception

9.2.5. Retrying load instructions

In some cases, the VPU might require that Avispado retries a load from a given point. For this, the VPU may set the ***completed*** bus with a vstart indicating the first value that has to be fetched again. When retrying, the VPU must guarantee the following:

- The intention to retry is communicated after receiving the signal ***memop.sync_end***.
- It must guarantee forward progress between retries. Two retries for the same instruction are possible, but two retries for the same element of the same instruction are not allowed. Additionally, the VPU is not allowed under any circumstance to retry the first element (as this violates forward progress).

When the VPU retries a load, Avispado will set the vstart value to the one provided by the VPU (regardless of possible exceptions) and restart the instruction from the fetch stage. This requires flushing the pipeline, re-fetch the offending instruction and issuing it again to the VPU.

Figure 14 shows an example of a load with retry. The last row of the waveform with name *Retry Needed* symbolizes the VPU becoming aware of the necessity of a retry of element 5. The VPU waits until the signal ***memop.sync_end*** is received setting the bus ***complete***, including ***completed.vstart*** to 5. Note that the vstart value set by Avispado in the case of a retry should be ignored.

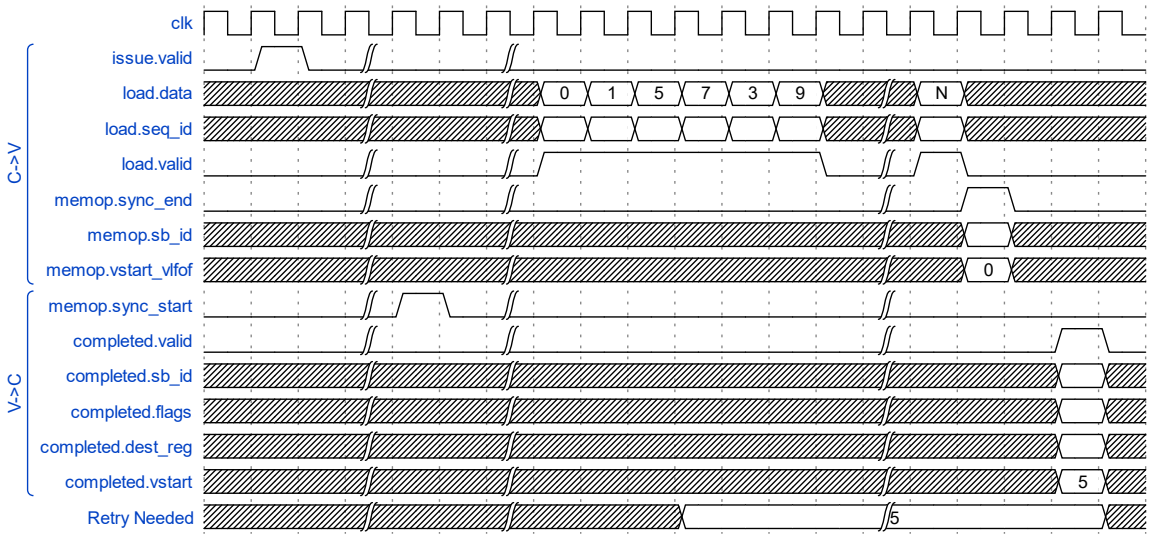

Figure 14. Load with retry

It is possible to receive an exception in the same instruction in which a retry is requested, as shown in Figure 15. In the example, the address for element 5 generates an exception and Avispado sets ***memop.sync_end***. A few cycles later, the VPU sets ***completed.valid*** with a vstart of 4, which indicates a retry for that element. Avispado will then set vstart to 4, as requested by the VPU.

9.2.6. Killing load instructions

Avispado supports speculative execution of a load instruction. The VPU may decide, like in any other instruction, to wait for the *next senior* signal in order to execute it non-speculatively. In case of speculative execution of a load, the kill signal may arrive at any moment before ***dispatch.next_senior*** is sent from Avispado. This includes killing a load after all the data has been sent from Avispado. It is important to note that if a ***dispatch.kill*** is sent, the VPU must not send any ***memop.sync_start*** after two cycles from the kill signal. The VPU may still send other data (indices, masks) for several cycles. In this case, the next ***memop.sync_start*** will indicate when data is relevant again. Also note that in the case of kill signal sent before a ***memop.sync_end***, Avispado will not send ***memop.sync_end***.

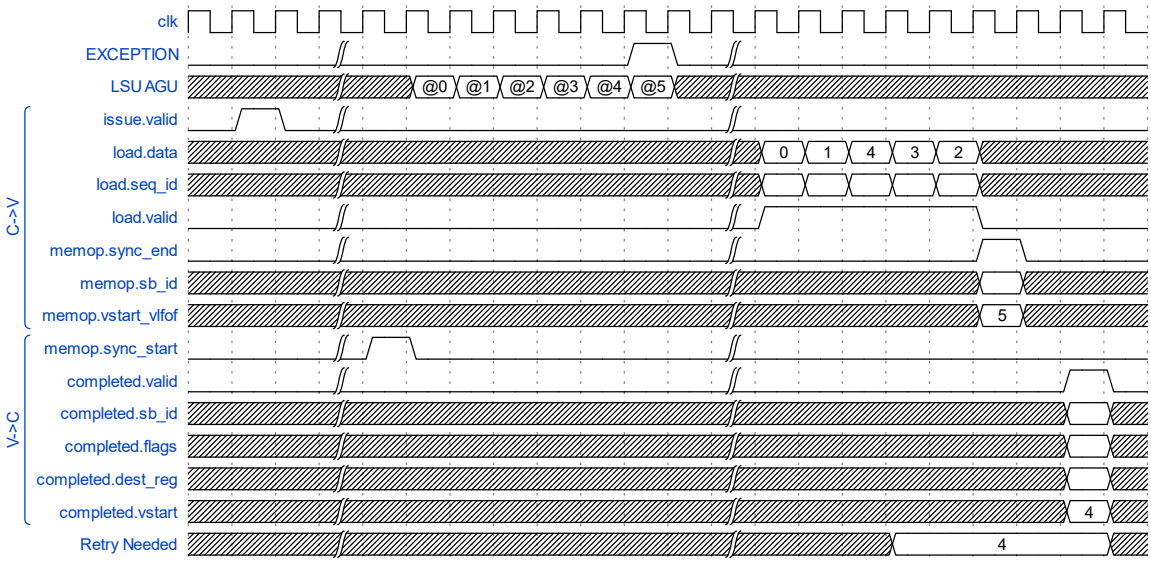

Figure 15. Load with retry and exception

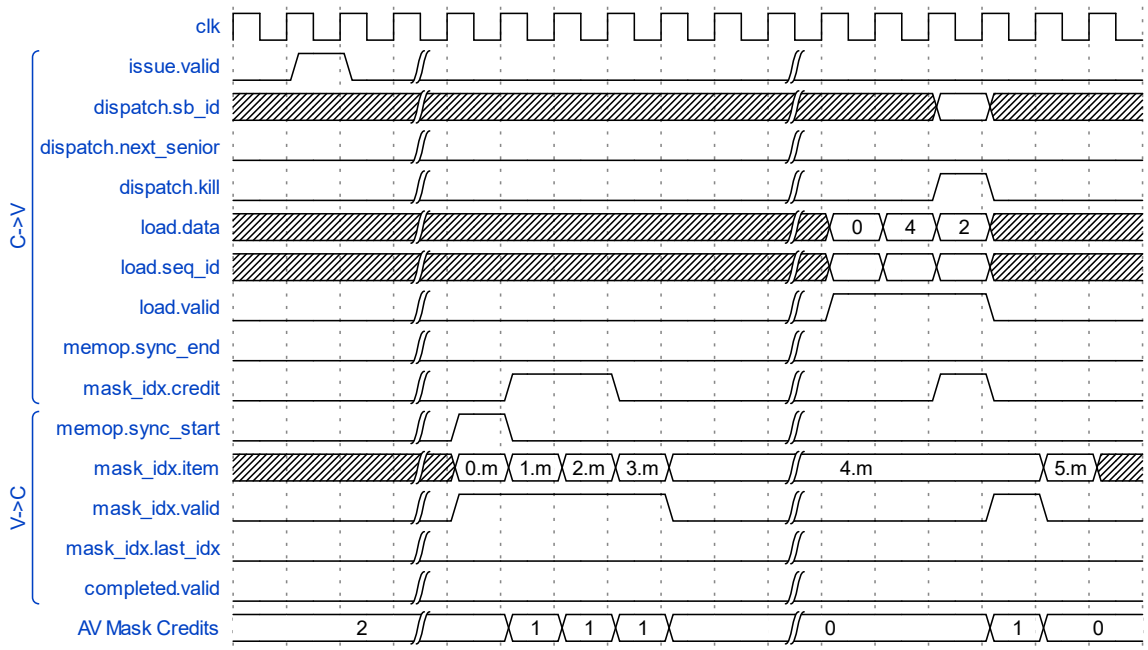

Figure 16. Indexed load killed

Figure 16 shows an example of an indexed load that is killed. After receiving the kill signal, the VPU might still send data for several cycles. The credit system works as usual: a pulse of signal ***mask_idx.valid*** consumes a credit that must be returned by signal ***mask_idx.credit***.

It is possible to receive a kill signal after a retry has been requested by the VPU. In this case, the kill signal prevails, and it is the responsibility of the VPU to restore its state.

9.3. Store instructions

Store operations are those that require writing data to memory. From the perspective of the interface, the difference between store and load operations lay in the direction that data flows (i.e.

from VPU to Avispado). Thus, the type of stores can also be divided between strided and indexed (scatter) memory accesses. Unlike in the case of loads, data is always transmitted in packed chunks and no sequence id is involved.

Because of the limited buffering resources of Avispado, an additional system of credits is used for store operations. Avispado can buffer up to 2 KB of data to be committed to memory. In total, 32 credits (32x512b=16,384b) are granted for store buffering.

Data is be transmitted from the VPU by using the bus ***store.data*** and setting ***store.valid*** as long as there are available credits. Avispado will return credits when resources are freed by setting ***store.credit*** signal for once cycle per credit.

9.3.1. Strided store instructions

The initialization and completion of strided store instructions is similar to that of load instructions. The VPU sets ***memop.sync_start*** when it's ready to send data. At that same cycle, data is loaded into ***store.data*** bus and setting ***store.valid***. Data may be sent in a continuous burst as long as there are credits available, which are returned by Avispado by setting signal ***store.credit***. If LMUL is 1, all data contained in the vector register is always sent regardless of the *vstart* value. In case of LMUL greater than 1 and *vstart* different than 0, vector registers inside the register group that do not contain any element that needs to be stored (i.e. all its element IDs fall below *vstart*) are not transmitted. Avispado sets signal ***memop.sync_start*** along with ***memop.sb_id*** and ***memop.vstart*** (to zero, if no exception occurred) whenever it finishes the execution. This must be acknowledged by the VPU by setting the ***completed*** bus in the same way as any other instruction (see *Finishing a vector instruction*). Figure 17 shows an example of strided store instruction.

Note: As is the case for Load instructions, detection of illegal instructions by the VPU is signaled by ***completed.illegal***, which must be set instead of raising ***memop.sync_start***.

Note: Avispado is not guaranteed to return all the credits before setting signal ***memop.sync_end***.

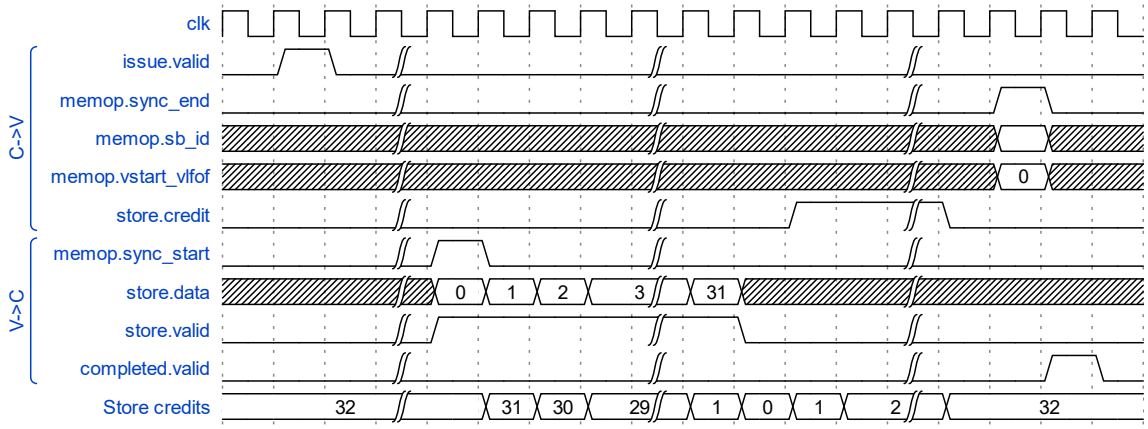

Figure 17. Store strided

A masked store operation is depicted in Figure 18. Note that the only difference is the inclusion of the bus ***mask_idx.item*** to transmit the mask, along with the signals ***mask_idx.valid*** and

mask_idx.credit. The procedure to transmit masks behaves in the same way as the case for strided masked loads.

Note: As in the case of strided masked loads, the mask for every element (excluding elements under vstart) must be transmitted even if it only contains zeroes. The amount of data sent is not affected by masks.

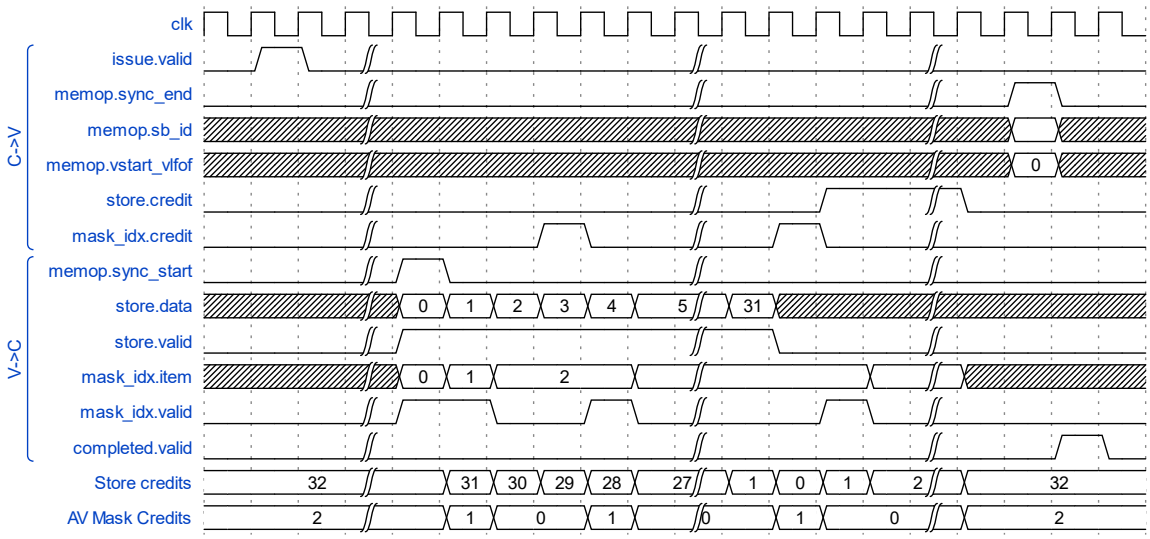

Figure 18. Store strided with mask

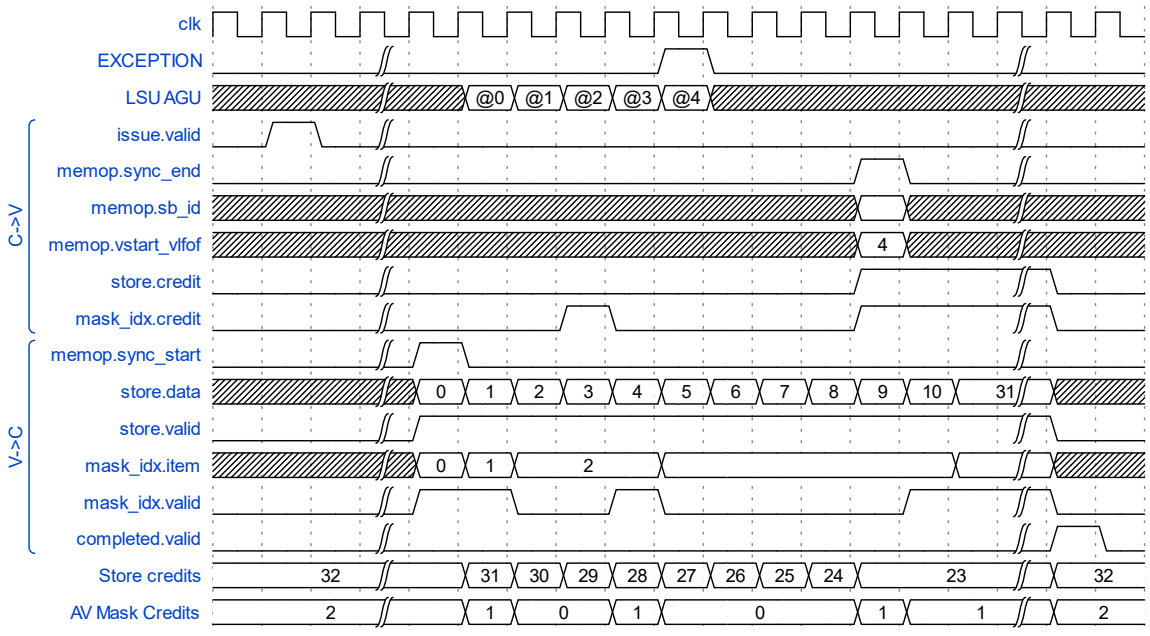

Figure 19. Store strided with exception

It is also possible for exceptions to occur on a store. These are dealt with in the same way as in the load scenario, by setting ***memop.vstart*** to the element that raised the exception. Figure 19 shows this scenario for a strided store. Note that Avispado may set ***memop.sync_end*** before the VPU has finished transmitting data or mask indices. In this case, the VPU may continue sending indices or

data, which will be ignored by Avispado. As with every instruction, the bus ***completed*** is used to inform Avispado that the instruction is done and may be committed.

9.3.2. Indexed store instructions (scatter)

Like in the indexed load instructions, scatter instructions require the transmission of indices and masks through the ***mask_idx.item*** bus (64 least significant bits for indices, most significant bit for mask). As before, scatters use the mask credit system to indicate availability of resources. As is the case for gathers, the last index sent must come along with a setting of signal ***mask_idx.last_idx***. For reference Figure 20 shows the waveform of a scatter.

Note: Data is sent following the same rules than normal strided stores (see 9.3.1).

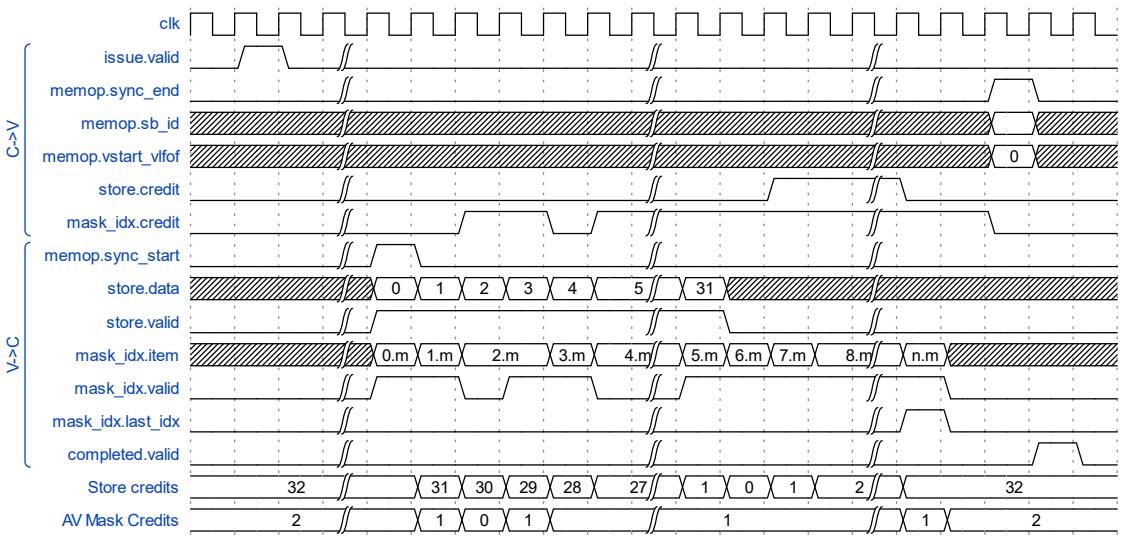

Figure 20. Indexed store (scatter)

9.4. Configuration-setting instructions (vsetvl{i} and *wcsr*)

The RISC-V vector spec defines one configuration-setting instruction that comes in two versions: with scalar register (*vsetvl*) and with an immediate value (*vsetvli*). Additionally, wcsr instructions may configure the execution of some instructions by writing into CSR registers. All these instructions are executed exclusively on the Avispado side. The effects of these instructions are propagated to the VPU on instruction issue (see *Issuing a vector instruction*).

10. License

Solderpad Hardware License v2.1

This license operates as a wraparound license to the Apache License Version 2.0 (the "Apache License") and incorporates the terms and conditions of the Apache License (which can be found here: http://apache.org/licenses/LICENSE-2.0), with the following additions and modifications. It must be read in conjunction with the Apache License. Section 1 below modifies definitions and terminology in the Apache License and Section 2 below replaces Section 2 of the Apache License. The Appendix replaces the Appendix in the Apache License. You may, at your option, choose to treat any Work released under this license as released under the Apache License (thus ignoring all sections written below entirely).

1. Terminology in the Apache License is supplemented or modified as follows:

"Authorship": any reference to "authorship" shall be taken to read "authorship or design".

"Copyright owner": any reference to "copyright owner" shall be taken to read "Rights owner".

"Copyright statement": the reference to "copyright statement" shall be taken to read "copyright or other statement pertaining to Rights".

The following new definition shall be added to the Definitions section of the Apache License:

"Rights" means copyright and any similar right including design right (whether registered or unregistered), rights in semiconductor topographies (mask works) and database rights (but excluding Patents and Trademarks).

The following definitions shall replace the corresponding definitions in the Apache License:

"License" shall mean this Solderpad Hardware License version 2.1, being the terms and conditions for use, manufacture, instantiation, adaptation, reproduction, and distribution as defined by Sections 1 through 9 of this document.

"Licensor" shall mean the owner of the Rights or entity authorized by the owner of the Rights that is granting the License.

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship or design. For the purposes of this License, Derivative Works shall not include works that remain reversibly separable from, or merely link (or bind by name) or physically connect to or interoperate with the Work and Derivative Works thereof.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form or the application of a Source form to physical material, including but not

limited to compiled object code, generated documentation, the instantiation of a hardware design or physical object or material and conversions to other media types, including intermediate forms such as bytecodes, FPGA bitstreams, moulds, artwork and semiconductor topographies (mask works).

"Source" form shall mean the preferred form for making modifications, including but not limited to source code, net lists, board layouts, CAD files, documentation source, and configuration files.

"Work" shall mean the work of authorship or design, whether in Source or Object form, made available under the License, as indicated by a notice relating to Rights that is included in or attached to the work (an example is provided in the Appendix below).

2. Grant of License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable license under the Rights to reproduce, prepare Derivative Works of, make, adapt, repair, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form and do anything in relation to the Work as if the Rights did not exist.

APPENDIX

Copyright [2020] [SemiDynamics]

SPDX-License-Identifier: Apache-2.0 WITH SHL-2.1

Licensed under the Solderpad Hardware License v 2.1 (the "License"); you may not use this file except in compliance with the License, or, at your option, the Apache License version 2.0. You may obtain a copy of the License at

https://solderpad.org/licenses/SHL-2.1/

Unless required by applicable law or agreed to in writing, any work distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.