1 Zvinsert: Vector Extension for Moves between Scalars and Vector Elements

Version *0.94*
Contributors: *Abel Bernabeu, Allen Baum*

The `Zvinsert` optional extension for `Zv` enables data movement between integer scalar registers and arbitrary elements of a vector register.

The register allocator within an optimizing compiler backend must sometimes spill registers into stack memory slots. This event happens often when the register pressure is high. The implementers of this extension provide a convenient alternative to spilling some scalar registers to memory, using vector registers as alternative storage for saving and restoring scalar registers.

1.1 Extension Dependencies

When `XLEN=32` the `Zvinsert` extension depends on `Zvl1024b`. For the case when `XLEN=64` the `Zvinsert` extension depends on `Zvl2048b`. As a general rule the `Zvinsert` extension depends on `Zvl<L>b`, where `L=32*XLEN`.

1.2 Benefit

When running out of integer scalar registers a register allocator can temporarily save a value to another register class for restoring it later. For example on implementations where `XLEN=32`, an integer scalar register like `x1` can be saved to a floating point register like `f7`, and later be restored. Conversely, when running out of floating point registers it is also possible to save and restore the value of a floating point register using an integer scalar register as temporary storage.

This is a valid strategy to save a spill to memory and it has already been implemented by some RISC-V compilers. However the amount of alternative spilling storage the compiler can exploit between the integer scalar and floating point scalar register files is very limited. By contrast the amount of available temporary storage increases notably when data move instructions are available between vector registers and integer scalar registers.

With the present extension we extend the ability of saving and restoring integer scalar registers so that every single bit of the vector register file is usable as temporary storage for alternative spilling. This extension can be seen as a generalization of the `vmv.x.s` and `vmv.s.x` instructions.

At a system level, the combination of this ISA extension and an optimizing compiler exploiting the feature should make the spilling of integer scalar registers to memory a less frequent event.

Note

> As a design principle it is a desirable property of an ISA that every part from every register of a given class A can be moved to and from every register of any other class B, with the width of B matching the width of the parts from A. The graph representing the availability of data move instructions between register classes does not need to be complete, from every

> register class to every other. However the graph needs to be connected so there is a data move path for every source and destination register class, even if the path requires more than one copy step. Abiding by this principle facilitates the implementation of alternative spilling strategies.

1.3 Insertion of Arbitrary Vector Element

The value in a source integer scalar register can be inserted into an arbitrary vector element destination.

```
vinserti.s.x vd, rs2, imm5    # vd[imm5] = x[rs2]
vinsert.s.x  vd, rs2, (rs1)   # vd[rs1] = x[rs2]
```

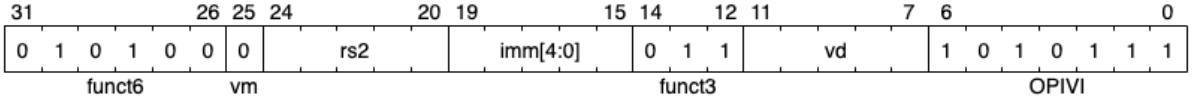
The vinserti.s.x instruction allows the programmer to specify the vector element index as an unsigned 5-bits immediate value. This has the limitation of only accessing up to 32 vector elements.

If the software needs to access elements beyond the 0 to 31 range can instead use `vinsert.s.x`, where the vector element is selected with index from `rs1`.

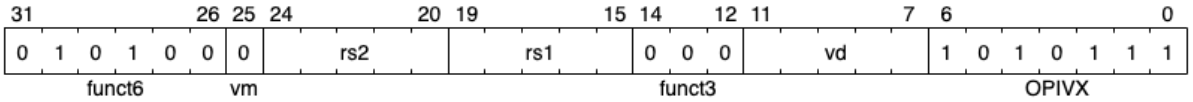
Both instructions use `EEW=XLEN` and `EMUL=1`. In other words, both instructions ignore the `vtype` CSR. These instructions also ignore the vector mask register, which is the reason why on the instruction encodings the vm bit is set to zero, and value one is reserved.

When the software requests to insert into an out of bounds vector element index, the hardware must leave the destination `vd` vector register unmodified.

The instruction encoding for `vinserti.s.x` is:



The instruction encoding for `vinsert.s.x` is:



1.4 Extraction of Arbitrary Vector Element

An arbitrary vector element source can be extracted into a destination integer scalar register.

```
vextracti.x.s rd, vs2, imm5    # x[rd] = vs2[imm5]
vextract.x.s  rd, vs2, (rs1)   # x[rd] = vs2[rs1]
```

The vextracti.x.s instruction allows the programmer to specify the vector element index as an unsigned 5-bits immediate value. This has the limitation of only accessing up to 32 vector elements.

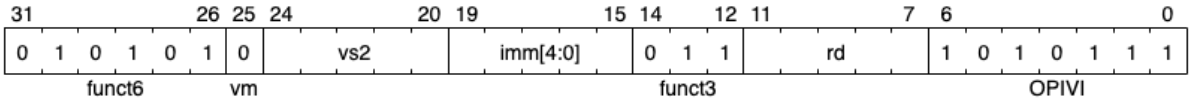
If the software needs to access elements beyond the 0 to 31 range can instead use `vextract.x.s`, where the vector element is selected with the index from `rs1`.

Both instructions use `EEW=XLEN` and `EMUL=1`. In other words, both instructions ignore the `vtype` CSR. These instructions also ignore the vector mask register, which is the reason why on the instruction encodings the vm bit is set to zero, and value one is reserved.

The value written into `rd` is unspecified when accessing out of bounds vector elements.

> Note
>
> When reading out of bounds elements the implementer is advised to store zero on rd. An equally valid handling of the out of bounds condition is to read any element from vs2 like the first or the last.

The instruction encoding for `vextracti.x.s` is:



The instruction encoding for `vextract.x.s` is: