SiFive Dot-Product Extension

Version v0.0.1, 20 September 2023:
# Table of Contents

Colophon .................................................................................................................. 1
Acknowledgments ....................................................................................................... 2
1. Introduction ............................................................................................................ 3
2. Extensions Overview ............................................................................................. 4
   2.1. Xsfvqdotq - Vector quad widening 4D Dot Product ........................................ 4
3. Instructions ........................................................................................................... 6
   3.1. vqdot.[vv,vx] .................................................................................................... 6
   3.2. vqdotu.[vv,vx] ................................................................................................ 8
   3.3. vqdotsu.[vv,vx] .............................................................................................. 10
   3.4. vqdotus.vx ..................................................................................................... 12
4. Bibliography .......................................................................................................... 14
5. Encodings ............................................................................................................. 15
   Appendix A: Vector Dot Product Instructions ...................................................... 15
Colophon

This document describes the Vector Dot Product extensions to the RISC-V Instruction Set Architecture.

This document is Discussion Document. Assume everything can change. This document is not complete yet and was created only for the purpose of conversation outside of the document.

Document Version Information:
Zv-fast-track @ 4ae2021a989ddc90fda47fd4db721c3d850ea322

See https://github.com/sifive/arch-specs/isa/zvdot for more information.
Acknowledgments

Contributors to this specification (in alphabetical order) include:

We are all very grateful to the many other people who have helped to improve this specification through their comments, reviews, feedback and questions.
Chapter 1. Introduction

This document describes the proposed vector dot product extensions for RISC-V.

Convolution operations have seen considerable speedup on other microprocessor architectures with the adoption of specialized dot-product instructions. Such convolutions are used in many AI/ML, image-processing, and DSP algorithms. These algorithms can be found in benchmark suites which are used to measure the relative performance of microprocessor cores with different ISAs against each other. This suite of vector dot product extensions is designed to bring RISC-V vector performance up to or to exceed the level of the competition, depending on the implementation vector-length and number of vector ALUs.

The dot product operation is defined for four INT8 values held in a 32-bit element group with an accumulation back into a single 32-bit value with quad-widening. The dot-product operation takes the 8-bit values at the same index from each source and multiplies them together. The four products are summed together and accumulated into the 32-bit element of the destination register. If the instruction variant has any signed source data then the accumulation is performed into a signed value.

Given that the source data is 8-bits and accumulation is performed into 32-bits, it is presumed that overflow is infrequent and thus no saturation option has been provided.

To save on opcode encoding space the unsigned-signed vector-vector variant is not encoded; it is always possible to swap the order of the input operands to the signed-unsigned vector-vector variant.
Chapter 2. Extensions Overview

The section introduces all of the extensions in the Vector Dot Product Instruction Set Extension Specification.

All the Vector Dot Product Extensions can be built on any embedded (Zve*) or application (“V”) base Vector Extension.

2.1. Xsfvqdotq - Vector quad widening 4D Dot Product

8-bit Integer dot-product instructions performing the dot product between two 4-element vectors of 8-bit integer elements and accumulating it into a 32-bit integer accumulator.

SEW is used to indicate both the size of the accumulator elements and the size of the 4-element byte vectors. These instructions are only defined for SEW=32.

These vector dot product instructions are defined with a fixed SEW value of 32. They work on a packed element group with four 8-bit values packed together in a 32-bit bundle. For each input bundle for the dot product there is a corresponding (same index) SEW-wide element in the accumulator source (and destination). The “q” in the mnemonic indicates that the instruction is quad-widening.

The number of body bundles is determined by vl. The operation can be masked, each mask bit determines whether the corresponding element result is active or not.

Fractional values of LMUL that result in LMUL*VLEN < SEW are not supported and cause an illegal instruction exception.

NOTE
Future variants could be defined for different SEW values thus:

<table>
<thead>
<tr>
<th>SEW</th>
<th>packed-values</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>8-bit</td>
</tr>
<tr>
<td>64</td>
<td>16-bit</td>
</tr>
<tr>
<td>16</td>
<td>4-bit</td>
</tr>
</tbody>
</table>

All the instructions defined in this extension fall into two schemes: vector-vector or vector-scalar.
### Mnemonic | Instruction
---|---
vqdot.[vv,vx] | Vector 8-bit Signed-Signed Dot Product
vqdotu.[vv,vx] | Vector 8-bit Unsigned Dot Product
vqdotsu.[vv,vx] | Vector 8-bit Signed-Unsigned Dot Product
vqdotus.vx | Vector 8-bit Unsigned-Signed Dot Product
Chapter 3. Instructions

3.1. vqdot.[vv,vx]

Synopsis
Vector 8-bit Signed-Signed Dot Product

Mnemonic
vqdot.vv vd, vs2, vs1, vm
vqdot.vx vd, vs2, rs1, vm

Figure 3. Encoding (Vector-Vector)

Figure 4. Encoding (Vector-Scalar)

Reserved Encodings
- **SEW** is any value other than 32

Arguments

<table>
<thead>
<tr>
<th>Register</th>
<th>Direction</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>vs1/rs1</td>
<td>input</td>
<td>vector of 8-bit signed multipliers</td>
</tr>
<tr>
<td>vs2</td>
<td>input</td>
<td>vector of 8-bit signed multiplicands</td>
</tr>
<tr>
<td>vd</td>
<td>input</td>
<td>vector of 32-bit signed accumulator sources</td>
</tr>
<tr>
<td>vd</td>
<td>output</td>
<td>vector of 32-bit signed accumulated dot products</td>
</tr>
</tbody>
</table>

Description

vqdot performs a partial dot product of two vectors of 8-bit signed integers, accumulating the partial sums into a vector of 32-bit signed integers.

In the vector-vector (.vv) variant, each 32-bit element of vs1 and vs2 is interpreted as a list of four 8-bit signed integer sub-elements, whereas each 32-bit element of vd is interpreted as a single 32-bit signed integer. For each pair of elements taken consecutively from vs1 and vs2, the associated sub-element lists are multiplied componentwise, widening (with sign-extension) to 32 bits. Finally, the four products are accumulated into the associated element of vd, wrapping on signed overflow.

The vector-scalar (.vx) variant reuses a single 32-bit element, read from integer register rs1, instead of a vector (vs1) of such elements.

Operation
function clause execute (VQDOT(vs2, vs1, vd, suffix)) = {
  foreach (i from vstart to vl-1) {
    // TODO sign extension
    let {a0 @ a1 @ a2 @ a3} : bits (4*8) = get_velem(vs2, i)
    let {b0 @ b1 @ b2 @ b3} : bits (4*8) = if suffix =="vv" then get_velem(vs1, i)
      else zext_or_truncate_to_sew(X(rs1));
    let acc : bits (SEW) = get_velem(vd, i);
    let res : bits (SEW) = a0 * b0 + a1 * b1 + a2 * b2 + a3 * b3 + acc;
    set_velem(vd, i, product);
  }
  RETIRE_SUCCESS
}
3.2. vqdotu.[vv,vx]

Synopsis
Vector 8-bit Unsigned Dot Product

Mnemonic
vqdotu.vv vd, vs2, vs1, vm
vqdotu.vx vd, vs2, rs1, vm

Reserved Encodings
• SEW is any value other than 32

Arguments

<table>
<thead>
<tr>
<th>Register</th>
<th>Direction</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>vs1/rs1</td>
<td>input</td>
<td>unsigned multiplier</td>
</tr>
<tr>
<td>vs2</td>
<td>input</td>
<td>unsigned multiplicand</td>
</tr>
<tr>
<td>vd</td>
<td>input</td>
<td>unsigned accumulator source</td>
</tr>
<tr>
<td>vd</td>
<td>output</td>
<td>unsigned accumulated dot product</td>
</tr>
</tbody>
</table>

Description

vqdotu performs a partial dot product of two vectors of 8-bit unsigned integers, accumulating the partial sums into a vector of 32-bit unsigned integers.

In the vector-vector (.vv) variant, each 32-bit element of vs1 and vs2 is interpreted as a list of four 8-bit unsigned integer sub-elements, whereas each 32-bit element of vd is interpreted as a single 32-bit unsigned integer. For each pair of elements taken consecutively from vs1 and vs2, the associated sub-element lists are multiplied componentwise, widening (with zero-extension) to 32 bits. Finally, the four products are accumulated into the associated element of vd, wrapping on unsigned overflow.

The vector-scalar (.vx) variant reuses a single 32-bit element, read from integer register rs1, instead of a vector (vs1) of such elements.

Operation

```plaintext
function clause execute (VQDOTU(vs2, vs1, vd, suffix)) = {
    foreach (i from vstart to vl-1) {
```
let \{a_0 @ a_1 @ a_2 @ a_3\} : \text{bits (4*8)} = \text{get\_velem(vs2, i)};
let \{b_0 @ b_1 @ b_2 @ b_3\} : \text{bits (4*8)} = \text{if suffix }=\text{"vv" then get\_velem(vs1, i) else zext\_or\_truncate\_to\_sew(X(rs1))};
let \text{acc : bits (SEW) = get\_velem(vd, i)};
let \text{res : bits (SEW) = a_0 \times b_0 + a_1 \times b_1 + a_2 \times b_2 + a_3 \times b_3 + acc};
\text{set\_velem(vd, i, product);}
\}
\text{RETIRE\_SUCCESS}
\}

\text{Included in}
\hspace{1cm}\text{Xsfvqdotq}
3.3. \texttt{vqdotsu.[vv,vx]}

Synopsis
Vector 8-bit Signed-Unsigned Dot Product

Mnemonic
\begin{verbatim}
vqdotsu.vv vd, vs2, vs1, vm
vqdotsu.vx vd, vs2, rs1, vm
\end{verbatim}

Reserved Encodings
- \texttt{SEW} is any value other than 32

Arguments

<table>
<thead>
<tr>
<th>Register</th>
<th>Direction</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>vs1/rs1</td>
<td>input</td>
<td>vector of 8-bit unsigned multipliers</td>
</tr>
<tr>
<td>vs2</td>
<td>input</td>
<td>vector of 8-bit signed multiplicands</td>
</tr>
<tr>
<td>vd</td>
<td>input</td>
<td>vector of 32-bit signed accumulator sources</td>
</tr>
<tr>
<td>vd</td>
<td>output</td>
<td>vector of 32-bit signed accumulated dot products</td>
</tr>
</tbody>
</table>

Description
\texttt{vqdotsu} performs a partial dot product of a vector of 8-bit signed integers by a vector of 8-bit unsigned integers, accumulating the partial sums into a vector of 32-bit signed integers.

In the vector-vector (\texttt{.vv}) variant, each 32-bit element of \texttt{vs1} (respectively \texttt{vs2}) is interpreted as a list of four 8-bit unsigned (respectively signed) integer sub-elements, whereas each 32-bit element of \texttt{vd} is interpreted as a single 32-bit signed integer. For each pair of elements taken consecutively from \texttt{vs1} and \texttt{vs2}, the associated sub-element lists are multiplied componentwise, widening (with sign-extension) to 32 bits. Finally, the four products are accumulated into the associated element of \texttt{vd}, wrapping on signed overflow.

The vector-scalar (\texttt{.vx}) variant reuses a single 32-bit element, read from integer register \texttt{rs1}, instead of a vector (\texttt{vs1}) of such elements.

Operation

\begin{verbatim}
function clause execute (VQDOTSU(vs2, vs1, vd, suffix)) = {
    foreach (i from vstart to vl-1) {
        // TODO sign extension
        let {a0 @ a1 @ a2 @ a3} : bits (4*8) = get_velem(vs2, i)
    }
}
\end{verbatim}
let {b0 @ b1 @ b2 @ b3} : bits (4*8) = if suffix == "vv" then get_velem(vs1, i) else zext_or_truncate_to_sew(X(rs1));
let acc : bits (SEW) = get_velem(vd, i);
let res : bits (SEW) = a0 * b0 + a1 * b1 + a2 * b2 + a3 * b3 + acc;
set_velem(vd, i, product);
}'

Included in

Xsfvqdotq
3.4. vqdotus.vx

Synopsis
Vector 8-bit Unsigned-Signed Dot Product

Mnemonic
vqdotus.vx vd, vs2, rs1, vm

Reserved Encodings
- SEW is any value other than 32

Arguments

<table>
<thead>
<tr>
<th>Register</th>
<th>Direction</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>rs1</td>
<td>input</td>
<td>vector of 8-bit signed multipliers</td>
</tr>
<tr>
<td>vs2</td>
<td>input</td>
<td>vector of 8-bit unsigned multiplicands</td>
</tr>
<tr>
<td>vd</td>
<td>input</td>
<td>vector of 32-bit signed accumulator sources</td>
</tr>
<tr>
<td>vd</td>
<td>output</td>
<td>vector of 32-bit signed accumulated dot products</td>
</tr>
</tbody>
</table>

The vector-vector variant of vqdotus is not specified because it can be obtained by swapping the vs1 and vs2 operand of the vqdotsu.vv instruction.

Description
vqdotus performs a partial dot product of a vector of 8-bit unsigned integers by a vector of 8-bit signed integers, accumulating the partial sums into a vector of 32-bit signed integers.

Each 32-bit element of vs2 is interpreted as a list of four 8-bit unsigned integer sub-elements; the 32-bit element read from general purpose register rs1 is interpreted as a list of four 8-bit signed integer sub-elements; whereas each 32-bit element of vd is interpreted as a single 32-bit signed integer. For each element taken consecutively from vs2, the associated sub-element list is multiplied componentwise by the element list from rs1, widening (with sign-extension) to 32 bits. Finally, the four products are accumulated into the associated element of vd, wrapping on signed overflow.

Operation

```c
function clause execute (VQDOTUS(vs2, vs1, vd)) = {
  foreach (i from vstart to vl-1) {
    // TODO sign extension
    let {a0 @ a1 @ a2 @ a3} : bits (4*8) = get_velem(vs2, i)
    let {b0 @ b1 @ b2 @ b3} : bits (4*8) = zext_or_truncate_to_sew(X(rs1));
    let acc : bits (SEW) = get_velem(vd, i);
    let res : bits (SEW) = a0 * b0 + a1 * b1 + a2 * b2 + a3 * b3 + acc;
  }
}
```
set_velem(vd, i, product);
}
RETIRE_SUCCESS
}
Chapter 4. Bibliography
Chapter 5. Encodings

Appendix A: Vector Dot Product Instructions

OP-V (0x57) Vector dot product instruction

<table>
<thead>
<tr>
<th>Integer</th>
<th>Integer</th>
<th>FP</th>
</tr>
</thead>
<tbody>
<tr>
<td>funct3</td>
<td>funct3</td>
<td>funct3</td>
</tr>
<tr>
<td>OPIVV</td>
<td>V</td>
<td>OPMVV</td>
</tr>
<tr>
<td>OPIVX</td>
<td>X</td>
<td>OPMVX</td>
</tr>
<tr>
<td>OPIVI</td>
<td>I</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>funct6</th>
<th>funct6</th>
<th>funct6</th>
</tr>
</thead>
<tbody>
<tr>
<td>10100</td>
<td>V</td>
<td>X</td>
</tr>
<tr>
<td>10101</td>
<td>V</td>
<td>X</td>
</tr>
<tr>
<td>10110</td>
<td>V</td>
<td>X</td>
</tr>
<tr>
<td>10111</td>
<td>V</td>
<td>X</td>
</tr>
</tbody>
</table>