## SQRDCMLAH (indexed)

Saturating rounding doubling complex integer multiply-add high with rotate (indexed)

Multiply without saturation the duplicated real components for rotations 0 and 180, or imaginary components for rotations 90 and 270, of the integral numbers in each 128-bit segment of the first source vector by the specified complex number in the corresponding the second source vector segment rotated by 0, 90, 180 or 270 degrees in the direction from the positive real axis towards the positive imaginary axis, when considered in polar representation.
Then double and add the products to the corresponding components of the complex numbers in the addend vector. Destructively place the most significant rounded half of the results in the corresponding elements of the addend vector. Each result element is saturated to the N-bit element's signed integer range $-2^{(N-1)}$ to $(2^{(N-1)})-1$. This instruction is unpredicated.
These transformations permit the creation of a variety of multiply-add and multiply-subtract operations on complex numbers by combining two of these instructions with the same vector operands but with rotations that are 90 degrees apart.
Each complex number is represented in a vector register as an even/odd pair of elements with the real part in the even-numbered element and the imaginary part in the odd-numbered element.
It has encodings from 2 classes: 16-bit and 32-bit

### 16-bit

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | i2 | | Zm | | | 0 | 1 | 1 | 1 | rot | | Zn | | | | | Zda | | | | |
| | | | | | | | | size<1> | size<0> | | | | | | | | | | | | | | | | | | | | | | |

```
SQRDCMLAH <Zda>.H, <Zn>.H, <Zm>.H[<imm>], <const>
```

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 16;
integer index = UInt(i2);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel_a = UInt(rot<0>);
integer sel_b = UInt(NOT(rot<0>));
boolean sub_r = (rot<0> != rot<1>);
boolean sub_i = (rot<1> == '1');
```

### 32-bit

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | i1 | Zm | | | | 0 | 1 | 1 | 1 | rot | | Zn | | | | | Zda | | | | |
| | | | | | | | | size<1> | size<0> | | | | | | | | | | | | | | | | | | | | | | |

**SQRDCMLAH <Zda>.S, <Zn>.S, <Zm>.S[<imm>], <const>**

```
if !HaveSVE2() && !HaveSME() then UNDEFINED;
constant integer esize = 32;
integer index = UInt(i1);
integer n = UInt(Zn);
integer m = UInt(Zm);
integer da = UInt(Zda);
integer sel_a = UInt(rot<0>);
integer sel_b = UInt(NOT(rot<0>));
boolean sub_r = (rot<0> != rot<1>);
boolean sub_i = (rot<1> == '1');
```

### Assembler Symbols

| | |
|---|---|
| <Zda> | Is the name of the third source and destination scalable vector register, encoded in the "Zda" field. |
| <Zn> | Is the name of the first source scalable vector register, encoded in the "Zn" field. |
| <Zm> | For the 16-bit variant: is the name of the second source scalable vector register Z0-Z7, encoded in the "Zm" field.<br>For the 32-bit variant: is the name of the second source scalable vector register Z0-Z15, encoded in the "Zm" field. |
| <imm> | For the 16-bit variant: is the element index, in the range 0 to 3, encoded in the "i2" field.<br>For the 32-bit variant: is the element index, in the range 0 to 1, encoded in the "i1" field. |
| <const> | Is the const specifier, encoded in “rot”: |

| rot | <const> |
|---|---|
| 00 | #0 |
| 01 | #90 |
| 10 | #180 |
| 11 | #270 |

### Operation

```
CheckSVEEnabled();
constant integer VL = CurrentVL;
constant integer PL = VL DIV 8;
constant integer pairs = VL DIV (2 * esize);
constant integer pairspersegment = 128 DIV (2 * esize);
bits(VL) operand1 = Z[n, VL];
bits(VL) operand2 = Z[m, VL];
bits(VL) operand3 = Z[da, VL];
bits(VL) result;

integer res_r, res_i;
```

```
for p = 0 to pairs-1
    integer segmentbase = p - (p MOD pairspersegment);
    integer s = segmentbase + index;
    integer elt1_a = SInt(Elem[operand1, 2 * p + sel_a, esize]);
    integer elt2_a = SInt(Elem[operand2, 2 * s + sel_a, esize]);
    integer elt2_b = SInt(Elem[operand2, 2 * s + sel_b, esize]);
    bits(esize) elt3_r = Elem[operand3, 2 * p + 0, esize];
    bits(esize) elt3_i = Elem[operand3, 2 * p + 1, esize];
    integer product_r =  elt1_a * elt2_a;
    integer product_i =  elt1_a * elt2_b;
    if sub_r then
        res_r = (SInt(elt3_r) << esize) - 2 * product_r;
    else
        res_r = (SInt(elt3_r) << esize) + 2 * product_r;
    res_r = (res_r + (1 << (esize-1))) >> esize;
    Elem[result, 2 * p + 0, esize] = SignedSat(res_r, esize);
    if sub_i then
        res_i = (SInt(elt3_i) << esize) - 2 * product_i;
    else
        res_i = (SInt(elt3_i) << esize) + 2 * product_i;
    res_i = (res_i + (1 << (esize-1))) >> esize;
    Elem[result, 2 * p + 1, esize] = SignedSat(res_i, esize);

Z[da, VL] = result;
```

### Operational information

This instruction might be immediately preceded in program order by a MOVPRFX instruction. The MOVPRFX instruction must conform to all of the following requirements, otherwise the behavior of the MOVPRFX and this instruction is unpredictable:

- The MOVPRFX instruction must be unpredicated.
- The MOVPRFX instruction must specify the same destination register as this instruction.
- The destination register must not refer to architectural register state referenced by any other source operand register of this instruction.

---