FCMLA

Floating-point Complex Multiply Accumulate.

This instruction operates on complex numbers that are represented in SIMD&FP registers as pairs of elements, with the more significant element holding the imaginary part of the number and the less significant element holding the real part of the number. Each element holds a floating-point value. It performs the following computation on the corresponding complex number element pairs from the two source registers and the destination register:

- Considering the complex number from the second source register on an Argand diagram, the number is rotated counterclockwise by 0, 90, 180, or 270 degrees.
- The two elements of the transformed complex number are multiplied by:
  - The real element of the complex number from the first source register, if the transformation was a rotation by 0 or 180 degrees.
  - The imaginary element of the complex number from the first source register, if the transformation was a rotation by 90 or 270 degrees.
- The complex number resulting from that multiplication is added to the complex number from the destination register.

The multiplication and addition operations are performed as a fused multiply-add, without any intermediate rounding.

This instruction can generate a floating-point exception. Depending on the settings in *FPCR*, the exception results in either a flag being set in *FPSR* or a synchronous exception being generated. For more information, see *Floating-point exception traps*.

Depending on the settings in the *CPACR_EL1*, *CPTR_EL2*, and *CPTR_EL3* registers, and the current Security state and Exception level, an attempt to execute the instruction might be trapped.

Vector
(FEAT_FCMA)

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	Q	1	0	1	1	1	0	size		0			Rm			1	1	0	rot		1			Rn					Rd		

FCMLA <Vd>.<T>, <Vn>.<T>, <Vm>.<T>, #<rotate>

```
if !IsFeatureImplemented(FEAT_FCMA) then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt(Rm);
if size == '00' then UNDEFINED;
if Q == '0' && size == '11' then UNDEFINED;
```

```
    constant integer esize = 8 << UInt(size);
    if !IsFeatureImplemented(FEAT_FP16) && esize == 16 then UNDEFINED;
    constant integer datasize = 64 << UInt(Q);
    integer elements = datasize DIV esize;
```

Assembler Symbols

<Vd>   Is the name of the SIMD&FP destination register, encoded in the "Rd" field.

<T>   Is an arrangement specifier, encoded in "size:Q":

size	Q	<T>
00	x	RESERVED
01	0	4H
01	1	8H
10	0	2S
10	1	4S
11	0	RESERVED
11	1	2D

<Vn>   Is the name of the first SIMD&FP source register, encoded in the "Rn" field.

<Vm>   Is the name of the second SIMD&FP source register, encoded in the "Rm" field.

<rotate>   Is the rotation, encoded in "rot":

rot	<rotate>
00	0
01	90
10	180
11	270

Operation

```
    CheckFPAdvSIMDEnabled64();
    bits(datasize) operand1 = V[n, datasize];
    bits(datasize) operand2 = V[m, datasize];
    bits(datasize) operand3 = V[d, datasize];
    bits(datasize) result;
    bits(esize) element1;
    bits(esize) element2;
    bits(esize) element3;
    bits(esize) element4;
    FPCRType fpcr = FPCR[];

    for e = 0 to (elements DIV 2)-1
        case rot of
            when '00'
                element1 = Elem[operand2, e*2, esize];
                element2 = Elem[operand1, e*2, esize];
```

```
                element3 = Elem[operand2, e*2+1, esize];
                element4 = Elem[operand1, e*2, esize];
            when '01'
                element1 = FPNeg(Elem[operand2, e*2+1, esize]);
                element2 = Elem[operand1, e*2+1, esize];
                element3 = Elem[operand2, e*2, esize];
                element4 = Elem[operand1, e*2+1, esize];
            when '10'
                element1 = FPNeg(Elem[operand2, e*2, esize]);
                element2 = Elem[operand1, e*2, esize];
                element3 = FPNeg(Elem[operand2, e*2+1, esize]);
                element4 = Elem[operand1, e*2, esize];
            when '11'
                element1 = Elem[operand2, e*2+1, esize];
                element2 = Elem[operand1, e*2+1, esize];
                element3 = FPNeg(Elem[operand2, e*2, esize]);
                element4 = Elem[operand1, e*2+1, esize];

        Elem[result, e*2, esize] = FPMulAdd(Elem[operand3, e*2, esize], ele
        Elem[result, e*2+1, esize] = FPMulAdd(Elem[operand3, e*2+1, esize],

    V[d, datasize] = result;
```