FMLSL, FMLSL2 (by element)

Floating-point fused Multiply-Subtract Long from accumulator (by element). This instruction multiplies the negated vector elements in the first source SIMD&FP register by the specified value in the second source SIMD&FP register, and accumulates the product to the corresponding vector element of the destination SIMD&FP register. The instruction does not round the result of the multiply before the accumulation.

A floating-point exception can be generated by this instruction. 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.

In Armv8.2 and Armv8.3, this is an optional instruction. From Armv8.4 it is mandatory for all implementations to support it.

> **Note**
>
> *ID_AA64ISAR0_EL1*.FHM indicates whether this instruction is supported.

It has encodings from 2 classes: FMLSL and FMLSL2

FMLSL
(FEAT_FHM)

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	0	0	1	1	1	1	1	0	L	M	Rm				0	1	0	0	H	0	Rn					Rd				
									sz								S														

FMLSL <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.H[<index>]

```
if !IsFeatureImplemented(FEAT_FHM) then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt('0':Rm);     // Vm can only be in bottom 16 registers.
if sz == '1' then UNDEFINED;
integer index = UInt(H:L:M);

constant integer esize = 32;
constant integer datasize = 64 << UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (S == '1');
integer part = 0;
```

FMLSL2
(FEAT_FHM)

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	1	1	0	L	M	Rm	1	1	0	0	H	0	Rn	Rd
									sz					S						

FMLSL2 <Vd>.<Ta>, <Vn>.<Tb>, <Vm>.H[<index>]

```
if !IsFeatureImplemented(FEAT_FHM) then UNDEFINED;
integer d = UInt(Rd);
integer n = UInt(Rn);
integer m = UInt('0':Rm);     // Vm can only be in bottom 16 registers.
if sz == '1' then UNDEFINED;
integer index = UInt(H:L:M);

constant integer esize = 32;
constant integer datasize = 64 << UInt(Q);
integer elements = datasize DIV esize;

boolean sub_op = (S == '1');
integer part = 1;
```

Assembler Symbols

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

<Ta> Is an arrangement specifier, encoded in “Q”:

Q	<Ta>
0	2S
1	4S

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

<Tb> Is an arrangement specifier, encoded in “Q”:

Q	<Tb>
0	2H
1	4H

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

<index> Is the element index, encoded in the "H:L:M" fields.

Operation

```
CheckFPAdvSIMDEnabled64();
bits(datasize DIV 2) operand1 = Vpart[n, part, datasize DIV 2];
bits(128) operand2 = V[m, 128];
bits(datasize) operand3 = V[d, datasize];
bits(datasize) result;
bits(esize DIV 2) element1;
bits(esize DIV 2) element2 = Elem[operand2, index, esize DIV 2];

for e = 0 to elements-1
    element1 = Elem[operand1, e, esize DIV 2];
    if sub_op then element1 = FPNeg(element1);
    Elem[result, e, esize] = FPMulAddH(Elem[operand3, e, esize], elemen
V[d, datasize] = result;
```
