3.7 Real stuff: The rest of the ARMv8 arithmetic instructions

> This section has been set as optional by your instructor.

COD Figure 2.40 (The number of instructions of each type…) lists 63 assembly-language instructions for integer multiply, integer divide, and floating-point operations in the full ARMv8 instruction set and an impressive 245 SIMD assembly language instructions.

It also shows 18 floating-point data transfer instructions in machine language but none in assembly language. COD Figure 3.17 (LEGv8 floating-point architecture revealed thus far), however, lists four assembly language instructions for floating-point data transfer. This apparent contradiction occurs because ARMv8 assemblers can use the names of the registers along with the generic data transfer instruction names to generate the proper opcode. For example, the ARMv8 assembler turns three instructions:

```
LDUR S1,   [X23,#100]
LDUR D1,   [X23,#100]
LDUR X1,   [X23,#100]
```

into machine language versions of the following LEGv8 instructions:

```
LDURS S1,   [X23,#100]
LDURD D1,   [X23,#100]
LDUR  X1,   [X23,#100]
```

As mentioned in an earlier elaboration, we use distinct assembly language names for different floating-point machine language instructions in this book in the belief that there will be less confusion about how the hardware works if we keep the relationship between the two levels one-to-one, but the register names alone are sufficient for an assembler to keep things straight.

Full ARMv8 integer and floating-point arithmetic instructions

The figure below shows all 63 assembly-language integer arithmetic and floating-point instructions in the full ARMv8 instruction set, with the 15 ARMv8 arithmetic core instructions highlighted in bold. Pseudoinstructions are italicized.

Figure 3.7.1: Full ARMv8 assembly-language instructions for integer and floating-point arithmetic (COD Figure 3.20).

Instruction in bold font is in the LEGv8 core. Italicized instructions are pseudoinstructions.

Type	Mnemonic	Instruction	Type	Mnemonic	Instruction
Integer Multiply & Divide	***MUL***	Multiply	Integer Mul-Add	MADD	Multiply-add
	SMULH	Signed multiply high		MSUB	Multiply-subtract
	UMULH	Unsigned multiply high		SMADDL	Signed multiply-add long
	SDIV	Signed divide		SMSUBL	Signed multiply-subtract long
	UDIV	Unsigned divide		UMADDL	Unsigned multiply-add long
	SMULL	Signed multiply long		UMSUBL	Unsigned multiply-subtract long
	UMULL	Unsigned multiply long	FP Mul-Add	FMADD	Floating-point fused multiply-add
	MNEG	Multiply-negate		FMSUB	Floating-point fused multiply-subtract
	UMNEGL	Unsigned multiply-negate long		FNMADD	Floating-point negated fused multiply-add
	SMNEGL	Signed multiply-negate long		FNMSUB	Floating-point negated fused multiply-subtract
FP two source operands	**FADDS**	Floating-point add single	FP move	FMOV	Floating-point move to/from integer or FP register
	FSUBS	Floating-point subtract single		FMOVI	Floating-point move immediate
	FMULS	Floating-point multiply single	FP sel	FCSEL	Floating-point conditional select
	FDIVS	Floating-point divide single			
	FADDD	Floating-point add double	FP round	FRINTA	Floating-point round to nearest with ties to odd
	FSUBD	Floating-point subtract double		FRINTI	Floating-point round using current rounding mode
	FNMUL	Floating-point scalar multiply-negate		FRINTM	Floating-point round toward -infinity
	FMULD	Floating-point multiply double		FRINTN	Floating-point round to nearest with ties to even
	FDIVD	Floating-point divide double		FRINTP	Floating-point round toward +infinity
	FCMPS	Floating-point compare single (quiet)		FRINTX	Floating-point exact using current rounding mode
	FCMPD	Floating-point compare double (quiet)		FRINTZ	Floating-point round toward 0
	FCMPE	Floating-point signaling compare	FP convert	FCVTAS	FP convert to signed integer, rounding to nearest odd
	FCCMP	Floating-point conditional quiet compare		FCVTAU	FP convert to unsigned integer, rounding to nearest odd
	FCCMPE	Floating-point conditional signaling compare		FCVTMS	FP convert to signed integer, rounding toward -infinity
FP one operand	FABS	Floating-point scalar absolute value		FCVTMU	FP convert to unsigned integer, rounding toward -infinity
	FNEG	Floating-point scalar negate		FCVTNS	FP convert to signed integer, rounding to nearest even
	FSQRT	Floating-point scalar square root		FCVTNU	FP convert to unsigned integer, rounding to nearest even
FP Min/Max	FMAX	Floating-point scalar maximum		FCVTPS	FP convert to signed integer, rounding toward +infinity
	FMIN	Floating-point scalar minimum		FCVTPU	FP convert to unsigned integer, rounding toward +infinity
	FMAXNM	Floating-point scalar maximum number (NaN = –Inf)		FCVTZS	FP convert to signed integer, rounding toward 0
				FCVTZU	FP convert to unsigned integer, rounding toward 0
	FMINNM	Floating-point scalar minimum number (NaN = +Inf)		SCVTF	Signed integer convert to FP, current rounding mode
				UCVTF	Unsigned integer convert to FP, current rounding mode

Like many other ARMv8 instruction categories, there is a version of the integer multiply instruction that supplies the negative of the result (`MNEG`). There are also "long" versions of the four multiply instructions, where the operands are 32 bits (`long`) instead of 64 bits (`long long`). ARMv8 also has six instructions that do both an integer multiply *and* an add or subtract for either three long (32-bit) or three long long (64-bit) operands. In fact, six of the integer multiply instructions are just pseudoinstructions of the multiply-add instructions with one of the three operands being the zero register (`XZR`). These 11 new ARMv8 integer instructions join the five multiply and divide instructions in the ARMv8 arithmetic core account for 16 instructions.

Like integer multiply, there is a version of floating-point multiply that produces a negative product (`FNMUL`). Like integer conditional compare instruction (`CCMP`), there is a conditional version of compare that (confusingly) does a comparison only if the initial condition is true (`FCCMP`). To allow programmers to check to see if an operand is a *Not-A-Number* (NaN), there are two versions of the comparison instructions: one that doesn't cause an exception whenever one of the operands is the value NaN (quiet compare) as well as one that does (signaling compare) in case the programmer wants an exception whenever a NaN is found. Again like integer multiply, there are four floating-point instructions that do multiply followed by an add or a subtract. ARMv8 has three more instructions for floating-point operations with a single operand: absolute value, negate, and square root. These 11 instructions join the 10 other existing instructions from the ARMv8 arithmetic core to bring our running total to 37.

Minimum and maximum floating-point operations are again a bit more complicated due to NaNs. There are two instructions that trap if an operand is a NaN and two that treat NaN as an extreme number: minus infinity for maximum or plus infinity for minimum. Not only does ARMv8 have one floating-point move instruction that can copy a value between floating-point registers or between integer registers and floating-point registers, it has one that can load a floating-point constant into a register. And once again like integer conditional select (`CSEL`), there is a floating-point version (`FCSEL`). These seven instructions get us to 44 arithmetic assembly-language instructions.

The final two categories are for rounding floating-point numbers and converting between integers and floating-point numbers. To round according to the many modes of IEEE 754, ARMv8 has seven instructions. To cover all combinations of conversions of signed and unsigned integers for the different rounding modes requires 12 more instructions. These last two categories bring us to 63 arithmetic assembly language instructions, which matches COD Figure 2.40 (The number of instructions of each type…).

PARTICIPATION ACTIVITY 3.7.1: ARMv8 assembly-language arithmetic instructions.

FCCMP FNMUL 63 15

Number of assembly-language integer arithmetic and floating-point instructions in ARMv8

Floating-point multiply instruction that produces a negative product

Number of ARMv8 arithmetic core instructions

Conditional compare that only compares if the initial condition is true

Reset

Full ARMv8 SIMD instructions

The figure below shows all 245 assembly-language SIMD instructions in the full ARMv8 instruction set. To fit these 245 instructions into a single table, we use regular expressions to represent several instructions within a single table entry. The figure caption reviews the three regular expression operators we use.

Figure 3.7.2: Full ARMv8 assembly language for SIMD instructions, which uses the term vector to distinguish them from single operands (scalar) (COD Figure 3.21).

To fit all 245 assembly language instructions into this small space, we use regular expressions to show the valid combinations. Question mark means 0 or 1 copies of the letter before it, so `F?ADD` represents the two instructions `ADD` and `FADD`. Square brackets mean there is a version for each letter in the brackets, so `[US]QADD` represents the two instructions `UQADD` and `SQADD`. Finally, curly brackets and a vertical line to separate the options show how to form multiple letter versions of the instructions. For example, `REV{16|32|64}` stands for the three instructions `REV16, REV32,` and `REV64`.

Type	Description	Name	Type	Description	Name
Add / Subtract	Vector add	F?ADD	Saturating Arithmetic	Integer saturating vector add	[US]QADD
	Integer vector add returning high, narrow	ADDHN2?		Integer saturating vector subtract	[US]QSUB
	Integer vector add long	[US]ADDL2?		Signed integer saturating vector accumulate of unsigned value	SUQADD
	Integer vector add wide	[US]ADDW2?		Unsigned integer saturating vector accumulate of unsigned value	USQADD
	Vector add pair	FADDP		Signed integer saturating vector absolute	SQABS
	Integer vector add long pair	[US]ADDLP		Signed integer saturating vector doubling multiply-add long	SQDMLAL2?
	Integer vector add and accumulate long pair	[US]ADALP		Signed integer saturating vector doubling multiply-subtract long	SQDMLSL2?
	Vector subtract	F?SUB		Signed integer saturating vector doubling multiply high half	SQDMULH
	Integer vector subtract returning high, narrow	SUBHN2?		Signed integer saturating vector doubling multiply long	SQDMULL2?
	Integer vector subtract long	[US]SUBL2?		Integer saturating vector narrow	[US]QXTN2?
	Integer vector subtract wide	[US]SUBW2?		Signed integer saturating vector and unsigned narrow	SQXTUN2?
	Vector negate	F?NEG		Signed integer saturating vector negate	SQNEG
Multiply / Divide / Square Root	Vector multiply	[FP]MUL		Signed integer vector saturating rounding doubling multiply high half	SQRDMULH
	FP vector multiply extended (0xINF→2)	FMULX	Multiply-Add	Vector chained multiply-add	MLA
	Vector multiply long	[USP]MULL2?		Vector fused multiply-add	FMLA
	Vector FP divide	FDIV		Integer vector multiply-add long	[US]MLAL2?
	FP vector square root	FSQRT		Vector chained multiply-subtract	MLS
	FP reciprocal square root	FRSQRTS		Vector fused multiply-subtract	FMLS
	Vector reciprocal square root estimate	[UF]RSQRTE		Integer vector multiply-subtract long	[US]MLSL2?
	Vector reciprocal estimate	[UF]RECPE	Reduction	Integer sum elements in vector	ADDV
	FP vector reciprocal step	FRECPS		Integer sum elements in vector long	[US]ADDLV
	FP vector reciprocal exponent	FRECPX		Maximum element in vector	[USF]MAXV
Compare	FP vector absolute compare greater than or equal	FACGE		FP maxNum element in vector	FMAXNMV
	FP vector absolute compare greater than	FACGT		Minimum element in vector	[USF]FMINV
	FP vector absolute compare less than or equal	FACLE		FP minNum element in vector	FMINNMV
	FP vector absolute compare less than	FACLT	Saturating Shifts	Integer saturating vector rounding shift left	[US]QRSHL
	Vector compare equal	F?CMEQ		Integer saturating vector shift right rounded narrow	[US]QRSHRN
	Vector compare greater than or equal	{CMHS|F?CMGE}		Signed integer saturating vector shift right rounded unsigned narrow	SQRSHRUN
	Vector compare greater than	{CMHI|F?CMGT}		Integer saturating vector shift left	[US]QSHL
	Vector compare less than or equal	{CMLS|F?CMLE}		Signed integer saturating vector shift left unsigned	SQSHLU
	Vector compare less than	{CMLO|F?CMLT}		Integer saturating vector shift right narrow	[US]QSHRN
	Vector test bits	CMTST		Signed integer saturating vector shift right unsigned narrow	SQSHRUN
Logical	Bitwise vector AND	AND	Min / Max	Vector maximum	[USF]MAX
	Bitwise vector bit clear	BIC		FP vector maxNum	FMAXNM
	Bitwise vector OR	ORR		Vector minimum	[USF]MIN
	Bitwise vector OR NOT	ORN		FP vector minNum	FMINNM
	Bitwise vector exclusive OR	EOR		Vector max pair	[USF]MAXP
	Bitwise vector NOT	MVN		FP vector maxNum pair	FMAXNMP
Shifts	Integer vector shift left	SHL		Vector min pair	[USF]MINP
	Integer vector shift left long	[US]SHLL		FP vector minNum pair	FMINNMP
	Integer vector shift right	[US]SHR	Convert	Vector FP convert to {signed | unsigned} integer (round to {0|x})	FCVT[Zx][SU]
	Integer vector shift right narrow	SHRN2?		Integer vector convert to FP	[US]CVTF
	Integer vector shift left and insert	SLI		Vector convert FP precision	FCVT[NL]
	Integer vector shift right and accumulate	[US]SRA		Vector convert double to single-precision (rounding to odd)	FCVTXN
	Integer vector shift right and insert	SRI	Round	Vector FP round to integral FP value (towards x)	FRINTx
	Integer rounding vector shift left	[US]RSHL		Integer rounding vector shift right and accumulate	[US]RSRA
	Integer rounding vector shift right	[US]RSHR		Integer rounding vector subtract returning high, narrow	RSUBHN2
	Integer rounding vector shift right narrow	RSHRN2?		Integer rounding vector halving add	[US]RHADD
Absolute value / difference	Integer vector absolute difference and accumulate	[US]ABA		Integer vector rounding add returning high, narrow	RADDHN
	Integer vector absolute difference and accumulate long	[US]ABAL2?	Insert / Extract / Reverse / Duplicate	Bitwise vector select	BSL
	Vector absolute difference	[USF]ABD		Bitwise vector extract	EXT
	Integer vector absolute difference long	[USF]ABDL2?		Bitwise vector insert if (true | false)	BI[TF]
	Vector absolute value	F?ABS		Vector reverse bits in bytes	RBIT
Data Transfer	Vector load (pair | register)	LD[PR]		Vector reverse elements	REV{16|32|64}
	Vector store (pair | register)	ST[PR]		Duplicate single vector element to all elements	DUP
	Vector move	[USF]MOV		Insert single element in another element	INS
	Vector structure/element load	LD[1234]		Vector element transpose	TRN{12}
	Vector replicated element load	LD[1234]R		Vector element zip	ZIP2?
	Vector structure store	ST[1234]		Vector element unzip	UZP{12}
Count Bits	Integer vector count leading {sign | zero} bits	CL{SZ}	Vector Length	Integer vector lengthen	[US]XTL2?
	Vector count non-zero bits	CNT		Integer vector narrow	XTN
Table Lookup	Vector table lookup	TBL		Integer vector halving add	[US]HADD
	Vector table extension	TBX		Integer vector halving subtract	[US]HSUB

Many SIMD instructions offer three versions:

- *Wide* means the width of the elements in the destination register and the first source registers is twice the width of the elements in the *second* source register.
- *Long* means the width of the elements in the destination register is twice the width of the elements in *all* source registers.
- *Narrow* means the width of the elements in the destination register is *half* the width of the elements in all source registers.

These three options are naturally enough indicated by the three suffixes W, L, and N. Like the non-SIMD instructions, *pair* means do the operation to a pair of SIMD registers. These instructions use the suffix P.

When dealing with narrow operations, the instruction could operate on either the lower half or the upper half of the 128-bit SIMD register containing the narrower elements. The default is use the lower half, with the suffix 2 indicating the operation is working on the upper half of the SIMD register.

The *prefixes* U, S, and F refer to the data types of unsigned integers, signed integers, and floating-point. When the elements are just plain bits, there is no prefix. There are also three instructions that use the type polynomial, and they use P as the prefix.

Most SIMD categories are self-explanatory, so we'll explain just the ones that may not be obvious. Instead of setting condition codes, SIMD compare vector sets the destination vector element to all 1s if the condition holds, or to 0 if not. While ARMv8 has only half of the comparisons (HS, GE, HI, GT), the programmer gets the others (LS, LE, LO, LT) by reversing the operands and using the complementary comparison. That is, A < B is the same as B ≥ A. Reductions do the operations across the elements *within a single* SIMD register rather than *between* elements of *different* SIMD registers as is the case for the rest of the SIMD instructions. As the category name suggests, these instructions perform the classic reduction operations of sum, minimum, and maximum. Finally, the table lookup instructions use one to four SIMD registers to act as a table. The elements of the other source register hold the indexes of the table and then the results of the parallel table lookups are stored in the elements of the destination register.

With the information above describing the terms wide, long, narrow, pair, and so on, the descriptions of the instructions within the categories are usually understandable. Here are a few that might not be:

- The elaboration in COD Section 3.5 (Floating point) explains how fused multiply-add only does one rounding instead of two as you might expect from doing two operations in one instruction. As is often the case, rather than choosing between them, ARMv8 offers both *fused* multiply-adds (one rounding) and *chained* multiply-adds (two roundings).
- Integer vector shift left and insert (`SLI`) provides a way to combine bits from two vectors.
- Integer vector shift right and accumulate instructions (`SSRA, USRA`) are useful when intermediate calculations are made at a higher precision before the result is added to a lower precision accumulator.
- Vector structure/element load instructions (`LD1, LD2, LD3, LD4`) loads structures of one, two, three, or four elements into SIMD registers.

PARTICIPATION ACTIVITY 3.7.2: ARMv8 assembly-language SIMD instructions.

1) ARMv8 has ____ assembly-language SIMD instructions.
   - ○ 63
   - ○ 245

2) Which SIMD instruction version means the width of the elements in the destination register is twice the width of the elements in all source registers?
   - ○ Wide
   - ○ Long

3) Each SIMD version is denoted by a ____.
   - ○ prefix
   - ○ suffix

4) The prefixes U, S, and F refer to ____.
   - ○ data types
   - ○ operand sizes

Provide feedback on this section