

Apple Silicon CPU Optimization Guide: 4.0

2025-06-06

Author
Apple Inc.

Copyright © 2025 Apple Inc. All rights reserved.

This material may contain confidential and proprietary information of Apple Inc., which may include intellectual property of Apple Inc., whether registered or otherwise. Any use, reproduction, disclosure, or distribution of this material is subject to the restrictions imposed as a condition for receiving this document, including but not limited to those restrictions provided in the Apple Developer Program License Agreement, the Apple Developer Agreement, and the Limited License to Apple Silicon CPU Optimization Guide. Use of copyright notice is precautionary and does not imply publication or disclosure. Apple and the Apple logo are trademarks of Apple Inc., registered in the U.S. and other countries.



Contents

Document Release Notes	12
1. Introduction	13
1.1. Chip Series and Family Naming Conventions	13
1.2. Optimization Process	14
1.3. High Impact Recommendations	16
1.4. Branch Terminology	16
1.5. Performance Monitoring Events	18
1.6. Decimal and Binary Data Quantities	18
1.7. Change Log	18
2. ISA Optimization: Overview and Integer Unit	20
2.1. Apple Platform Technologies	20
2.2. Arm AArch64 ISA	21
2.3. Arm Reference Documents	24
2.4. ISA Characteristics	24
2.5. Syntax	26
2.6. Registers	26
2.7. Separate Source and Destination Registers	29
2.8. Addressing Forms, Instruction Immediates, and Operand Shifts	30
2.8.1. Avoid Chains of Pre- and Post-Indexed Operations	32
2.8.2. Constant Generation	34
2.8.3. Long Address Generation	35
2.9. Condition Codes	35
2.10. Conditional Instructions	36
2.11. Fused Op with Add/Accumulate Operations (Integer Unit)	36
2.12. Memory Model, Barriers, and Synchronization	36
2.13. Self-Modifying or JIT-Generated Code Deployment	37
2.14. Data-Independent Timing (FEAT_DIT)	37
2.15. ISA Characterization	38
3. ISA Optimization: Advanced SIMD and FP Unit	39
3.1. Advanced SIMD and FP Data Types	40
3.1.1. Floating Point Data Types	41
3.1.2. Integer Data Types	43
3.2. Arm Advanced SIMD and FP Registers	46
3.3. Overview of Arm Advanced SIMD Mnemonics	47
3.3.1. ASIMD and FP Overall Instruction Data Type: (none) / BF / F / S / U	49
3.3.2. ASIMD Modulo or Saturating Arithmetic: Q	50
3.3.3. ASIMD Truncating or Rounding: R	50
3.3.4. ASIMD Doubling or Halving: D / H	50
3.3.5. ASIMD Polynomial: P	50
3.3.6. FP Negated: N	51
3.3.7. ASIMD Complex Number Arithmetic: C	51
3.3.8. ASIMD Cryptographic: Various	51
3.3.9. ASIMD Load/Store by Element Interleaving and Replicating: 1 / 2 / 3 / 4 / R	51
3.3.10. ASIMD Create Immediate: I	53
3.3.11. ASIMD Comparison Conditionals: EQ / GE / GT / HI / HS / LE / LT	54

3.3.12. FP Special Value Control Suffixes: E / NM / X	54
3.3.13. FP Rounding Mode: A / I / M / N / P / X / Z	55
3.3.14. ASIMD/FP Different Output Type: (none) / F / S / U	55
3.3.15. ASIMD Return High Half: H	55
3.3.16. ASIMD Narrow, Long, Wide: N / L / W	56
3.3.17. ASIMD Lower-Half / Upper-Half: (none) / 1 / 2	57
3.3.18. ASIMD Cross-Lane Paired or Horizontal: P / V	59
3.3.19. ASIMD Bottom or Top Half of Element Pairs: B / T	60
3.4. Compiler Intrinsics for Instructions and Data Types	61
3.4.1. C Vector Types	62
3.4.2. Computation Intrinsics: A General Guide	63
3.4.2.1. Basic Intrinsic Naming	63
3.4.2.2. Intrinsic Functional Notation	65
3.4.3. Computation Intrinsics: Unusual Cases	66
3.4.3.1. Paired Operations	66
3.4.3.2. Comparison Operations	66
3.4.3.3. Multiplication by Element or Scalar Operations	66
3.4.3.4. Shift-by-Immediate Operations	67
3.4.3.5. Type Conversion Operations	67
3.4.4. Vector Manipulation Intrinsics	68
3.4.4.1. Element Insertion and Extraction Operations	69
3.4.4.2. Element Duplication Operations	70
3.4.4.3. Narrowing and Lengthening Element Operations	70
3.4.4.4. Type Casting Operations	71
3.4.4.5. Vector Length Conversion Operations	71
3.4.4.6. Byte Rotations Operations	72
3.4.4.7. Data Transposition Operations	72
3.4.4.8. Table Operations	73
3.4.4.9. Load/Store Operations	73
3.5. Advanced SIMD Coding Recommendations	75
3.5.1. Manage Architectural Register Limits	75
3.5.2. Minimize Moves Between Integer and Vector Registers	76
3.5.3. Horizontal Arithmetic and Test Instructions	77
3.5.4. Fused Op with Add/Accumulate Instructions (Integer and ASIMD and FP Types)	79
3.5.5. Complex Number Instructions	81
3.5.6. Dot Product and Matrix Multiply Instructions	82
3.5.7. Shuffle and Permute Instructions	83
3.5.7.1. Byte Shuffle Example: Transposing Matrices	87
3.5.8. Brain Float Data Type (BFLOAT16) Operations	89
3.5.9. Int8 Matrix Multiplication and Additional Dot Product Operations	90
3.6. Advanced SIMD Coding Examples	90
3.6.1. Matrix Operations	90
3.6.1.1. Matrix-Matrix Multiply	91
3.6.1.2. Matrix-Vector Multiply	92
3.6.1.3. Matrix Operation Improvements	92
3.6.2. Image Filtering	93
3.6.2.1. Window Filtering	94
3.6.2.2. Per-Pixel Filtering	96
3.6.2.3. Per-Pixel Filtering Using DOT Instructions	98

Apple Silicon CPU Optimization Guide: 4.0

4. ISA Optimization: Scalable Matrix Extension (SME)	99
4.1. SSVE/SME Registers	100
4.1.1. SSVE Z Vector Registers (zn)	100
4.1.2. SSVE P and PN Predicate Registers (P(N)n)	102
4.1.3. SME ZA Storage	105
4.1.3.1. ZA Tile Registers (zA<n>.<size>)	105
4.1.3.2. ZA Vector Registers (zA.<type>[row(s)])	107
4.1.4. ZA Storage Addressing Modes	110
4.2. SVE Streaming and ZA Modes	114
4.3. Memory Consistency Model	116
4.4. Predication and Loop Control	116
4.4.1. Predicate and Loop Control Operations	117
4.4.2. Element Activation Control	120
4.4.3. Data Structure Edge Control	121
4.5. SSVE/SME Instruction Fundamentals	125
4.5.1. Register Operand Limitations	125
4.5.2. SSVE Horizontal (Pairwise) Instructions	126
4.5.3. Element Indexing	127
4.6. Overview of Arm SSVE/SME Mnemonics	129
4.6.1. SSVE/SME Narrow, Long, Long Long, Wide: N / L / LL / W	132
4.6.2. SSVE/SME Lower / Upper Half + Bottom / Top: (none) / 1 / 2 / LO / HI / B / T	138
4.6.3. SME Horizontal / Vertical in Vector: H / V (Prefix)	139
4.6.4. SME Horizontal / Vertical in Tile: H / V (Suffix)	140
4.6.5. SSVE Load Replicating / Broadcasting: R / RQ	141
4.6.6. SSVE Comparison Conditionals: EQ / GE / GT / HI / HS / LE / LO / LS / LT / NE / UO	141
4.6.7. SSVE Cross-Lane Paired or Horizontal: P / V	142
4.6.8. SSVE Reversed Inputs: R	143
4.6.9. SSVE/SME Target Specification: A / S / T	143
4.6.10. SSVE Lookup Table Instructions	144
4.6.11. SME ZA Tile and Vector Computation Instructions	146
4.6.11.1. SME Outer Product (MOP) Tile Instructions	149
4.6.11.2. SME Dot Product (DOT) Vector Instructions	151
4.6.11.3. SME Multiply Accumulate (MLA) Vector Instructions	152
4.7. SSVE/SME Coding Examples	155
4.7.1. General Flow of an SSVE/SME Routine	155
4.7.2. SME Matrix Transposition Example	156
4.7.3. Matrix-Matrix Multiply (Same Element Size)	157
4.7.4. Matrix-Matrix Multiply (Double Lengthening Element Size)	159
4.7.5. Matrix-Vector Multiply	163
5. Core Microarchitecture Optimization	166
5.1. Apple Silicon CPU and Chip Overview	166
5.2. Microarchitectural Characteristics	168
5.3. Core Processor Pipeline Fundamentals	169
5.4. Instruction Delivery Optimization	173
5.4.1. L1 Instruction (L1I) TLB	173
5.4.2. L1 Instruction (L1I) Cache	174
5.4.3. Branch Target Alignment	176
5.4.4. Taken Branch Reduction	176

5.4.5. Conditional Branch Mispredicts and Conditional Instructions	178
5.4.6. Indirect Branch and Indirect Call Mispredicts	182
5.4.7. Return Address Stack	183
5.4.8. Multi- μ op Instructions	184
5.5. Instruction Processing Map and Execution Optimization	186
5.5.1. Movement of Data from General Purpose Registers to Vector Registers	188
5.5.2. Movement of Data from Vector Registers to General Purpose Registers	189
5.5.3. Preferred Instructions for Common Operations	190
5.5.3.1. Register Data Copy	190
5.5.3.2. Register Zeroing	190
5.5.3.3. Register Constants	191
5.5.3.4. Limited FPCR and FPSR Bandwidth	192
5.5.4. Unroll and Software Pipeline Long Sequential Loop Bodies	192
5.6. Instruction Processing Memory Optimization	192
5.6.1. Address Generation	193
5.6.2. L1 Data (L1D) TLB and Shared L2 TLB	193
5.6.3. L1 Data (L1D) Cache	195
5.6.4. Shared L2 Cache	197
5.6.5. Memory Cache	198
5.6.6. Improving Cache Hierarchy Performance	199
5.6.7. Fast Load-to-Load Latency (Pointer Chasing)	201
5.6.8. Non-Temporal (NT) Accesses	202
5.6.9. Unaligned Accesses	203
5.6.10. Memory Dependence Prediction	204
5.6.10.1. Memory Dependence Prediction Example	210
5.6.11. Store-to-Load Forwarding	211
5.6.12. Prefetching	212
5.6.12.1. Hardware Prefetcher	212
5.6.12.2. Software Prefetch Instructions	213
5.6.12.3. Prefetch Performance Tips	215
6. SME Engine Microarchitecture Optimization	219
6.1. Overall Optimization Guidance	223
6.2. SSVE Vector Execution Unit Optimization	224
6.3. SME Processing Grid Optimization	224
6.4. ZA Extraction Unit Optimization	227
6.5. SME Engine Load/Store Execution Unit Optimization	228
6.5.1. Access Merging	230
6.5.2. Optimizing for Long Memory μ op Latencies	231
6.6. Core Integer Unit Optimization	232
6.7. Core Load/Store Execution Unit Optimization	232
6.7.1. Address Translations and Permission Checks	233
6.7.2. Memory Ordering Between Core and SME Engine	233
6.7.3. SME Engine Instruction Delivery and Packing Fusion	234
6.8. SSVE/SME Instruction Operation Fusion	234
6.9. Memory Access Optimizations	238
6.9.1. Cache Blocking	238
6.9.2. Software Prefetching	239
6.9.3. Non-Temporal Memory Access	239
6.10. Preferred Instructions for Common Operations	239

Apple Silicon CPU Optimization Guide: 4.0

6.10.1. Register Data Copy	239
6.10.2. Register Zeroing	240
6.11. Multithreading and Mode Management	240
7. Asymmetric Multiprocessor (AMP) Optimization	242
7.1. Prioritizing Work	242
7.2. Partitioning Work	242
7.2.1. Avoid Static Partitioning	243
7.2.2. Use Dynamic Partitioning	243
7.3. Avoid Spin-Wait	246
7.4. APIs for Synchronization and Thread Communication	246
7.5. Thread Communication Tradeoffs	247
7.6. Xcode: Sanitizer Tools	249
8. Xcode: Performance Monitoring and Analysis	250
8.1. CPU Counters With Performance Monitoring Events	250
8.1.1. CPU Bottleneck Analysis	251
8.1.1.1. Instruction Delivery Bottleneck Analysis	252
8.1.1.2. Instruction Processing Bottleneck Analysis	253
8.1.2. Performance Monitoring Events	255
8.2. Processor Trace	260
A. Instruction Latency and Bandwidth	262
A.1. Core Integer Execution Units	262
A.2. Core Advanced SIMD and FP Execution Units	266
A.3. Core Load and Store Execution Units	272
A.3.1. Load Latency	272
A.3.2. Store Latency	273
A.3.3. Atomic Instructions	273
A.3.4. Load and Store Execution Unit Bandwidth	275
A.3.5. Memory Hierarchy Access Latencies	276
A.4. SSFP/SSVE Execution Unit	277
A.5. SME ZA Extraction Unit	282
A.6. SME Processing Grid	282
A.7. SME Engine Load and Store Execution Unit Bandwidth	286
B. Dynamic Determination of Chip-Specific Capabilities	289
B.1. ISA Features	290
B.2. Cache and Topology Characteristics	291

List of Figures

- 3.1. Standard Advanced SIMD Addition ADD (vector) 40
- 3.2. ASIMD Integer Processing Example 45
- 3.3. Advanced SIMD Load/Store Element Interleaving and De-Interleaving 52
- 3.4. Advanced SIMD Narrow, Lengthen, and Widen 58
- 3.5. Advanced SIMD Byte Transformations 59
- 3.6. Advanced SIMD Cross Lane Paired and Horizontal Instructions 60
- 3.7. Advanced SIMD Top and Bottom Half of Element Pairs 61
- 3.8. Advanced SIMD Pixel Filter 94
- 4.1. Bit Mask Predicate (Pn) 104
- 4.2. Layout of the ZA storage 107
- 4.3. ZA Tile and Vector Register Mapping for Computation Instructions 110
- 4.4. ZA Tile and Vector Register Mapping for Data Movement Instructions 112
- 4.5. Example of Edge Control Predication 123
- 4.6. ADDP Instruction in Advanced SIMD and SSVE 127
- 4.7. SSVE Vector Indexing 128
- 4.8. SSVE: Narrowing Instructions 133
- 4.9. SME: Narrowing Instructions 134
- 4.10. SSVE: Widening/Lengthening Instructions 135
- 4.11. SME: Widening/Lengthening Instructions 136
- 4.12. SSVE: Lengthening Multiply-Accumulate Instructions 137
- 4.13. SME: Lengthening Multiply-Accumulate Instructions 138
- 4.14. SME Vector: Horizontal and Vertical DOT Product Instructions 140
- 4.15. SSVE: Horizontal (Pairwise) Instructions 143
- 4.16. SSVE Lookup Table Instruction `LUT12` on 4B Elements with Single Vector Destination 144
- 4.17. SSVE Lookup Table Register Usage 145
- 4.18. ZA Tile Element Operation Patterns for 16/32/64-bit Input MOP Instructions 150
- 4.19. ZA Tile Element Operation Patterns for 8-bit Input MOP Instructions 151
- 4.20. ZA Vector Element Patterns for DOT Instructions 152
- 4.21. ZA Vector Element Patterns for MLA Instructions 154
- 4.22. SSVE/SME Data Flow Overview 155
- 4.23. SME Matrix-Matrix Multiply Memory Access Pattern 159
- 4.24. SME Matrix-Matrix Multiply with Double Lengthening Memory Access Pattern 162
- 4.25. SME Matrix-Vector Multiply Memory Access Pattern 164
- 5.1. General Purpose Compute Complex and Related Components 167
- 5.2. Abstract View of the Core Processor Pipeline. 171
- 6.1. Abstract View of the SME Engine and Connectivity 221
- 6.2. Performance and Efficiency SME Engine Processing Element Grids 225
- 6.3. Full-Tile and Single-Vector Group SME Operations in the P SME Engine Processing Grid 226
- 6.4. Access Merging 230
- 7.1. Partitioning Strategies: An Illustrative Example 245
- 8.1. Instruction Processing Bottleneck Categories 255
- 8.2. Example of a Processor Trace Flame Graph 261

List of Tables

- 1. Change History 12
- 1.1. Decimal and Binary Data Quantities 18
- 2.1. Supported Arm Base ISA and EL0 Features 21
- 2.2. Load and Store Addressing Modes 31
- 2.3. Condition Codes 35
- 2.4. Common Instruction Mix Metrics 38
- 3.1. FP Types Supported in the Advanced SIMD and FP Unit: By Field 41
- 3.2. FP Types Supported in the Advanced SIMD and FP Unit: By Range 41
- 3.3. FP Rounding, NaN Handling, and Denormal Handling Controls in FPCR 42
- 3.4. Integer Types Supported in the Advanced SIMD and FP Unit 43
- 3.5. Vector Register Layouts 46
- 3.6. Vector Register Interpretation 47
- 3.7. Advanced SIMD Instruction Mnemonic Prefixes 48
- 3.8. Advanced SIMD Base Instructions Mnemonics Grouped By General Function 48
- 3.9. Advanced SIMD Instruction Mnemonic Suffixes 49
- 3.10. Advanced SIMD Move Immediate 53
- 3.11. Advanced SIMD Comparison Conditionals 54
- 3.12. Advanced SIMD Intrinsic Data Types 62
- 3.13. Example Complex Number Operations 82
- 3.14. Advanced SIMD Shuffling Operations 84
- 3.15. Input-Output Patterns for the Advanced SIMD Shuffling Operations: Byte-size Elements 85
- 3.16. Input-Output Patterns for the Advanced SIMD Shuffling Operations: Half-size Elements 85
- 3.17. Input-Output Patterns for the Advanced SIMD Shuffling Operations: Word-size Elements 86
- 3.18. Input-Output Patterns for the Advanced SIMD Shuffling Operations: Double-size Elements 86
- 4.1. Counted Predicate (PNn) 104
- 4.2. Addressing Modes for ZA Storage for Computational Instructions 111
- 4.3. Addressing Modes for ZA Storage for Data Movement Instructions 113
- 4.4. ISA and Register Availability in Non-Streaming Mode and Streaming Mode 115
- 4.5. SSVE/SME Variant Conditional Branch Mnemonics 119
- 4.6. SSVE/SME Instruction Mnemonic Prefixes 130
- 4.7. SSVE/SME Base Instructions Grouped By General Function 130
- 4.8. SSVE/SME Instruction Mnemonic Suffixes 131
- 4.9. SSVE Comparison Conditionals 141
- 4.10. ZA Tile Operation Type Support 147
- 4.11. ZA Vector Operation Type Support 147
- 5.1. Topology Configurations: M-Series Chips 167
- 5.2. Topology Configurations: A-Series Chips 168
- 5.3. L1I 16KiB-Page TLB Organization: M-Series Chips 173
- 5.4. L1I 16KiB-Page TLB Organization: A-Series Chips 173
- 5.5. Common L1 Instruction TLB Metrics 174
- 5.6. L1I Cache Organization: M-Series Chips 175
- 5.7. L1I Cache Organization: A-Series Chips 175
- 5.8. Common L1 Instruction Cache Metrics 175
- 5.9. Common Branch Instruction Mix and Direction Metrics 178
- 5.10. General and Conditional Branch Mispredict Metrics 182
- 5.11. Indirect Branch and Indirect Call Mispredict Metrics 182

Apple Silicon CPU Optimization Guide: 4.0

5.12. Return Mispredict Metrics	184
5.13. μ op Execution Bandwidth: M-Series Chips	186
5.14. μ op Execution Bandwidth: A-Series Chips	187
5.15. Instruction Throughput Metric	188
5.16. Register Data Copy Instructions	190
5.17. Register Data Zeroing Instructions	191
5.18. Register Data Constant Instructions	191
5.19. L1D 16KiB-Page TLB Organization: M-Series Chips	193
5.20. L1D 16KiB-Page TLB Organization: A-Series Chips	193
5.21. Shared L2 16KiB-Page TLB Organization: M-Series Chips	194
5.22. Shared L2 16KiB-Page TLB Organization: A-Series Chips	194
5.23. Common L1D TLB, Shared L2 TLB, and Address Translation Metrics	195
5.24. L1D Cache Organization: M-Series Chips	196
5.25. L1D Cache Organization: A-Series Chips	196
5.26. Common Data Cache Metrics	196
5.27. Shared L2 Cache Organization: M-Series Chips	197
5.28. Shared L2 Cache Organization: A-Series Chips	197
5.29. Memory Cache Organization: M-Series Chips	198
5.30. Memory Cache Organization: A-Series Chips	199
5.31. Common Non-Temporal Access Metrics	203
5.32. Common Access Alignment Metrics	204
5.33. Store-to-Load Dependence Prediction Outcome	205
5.34. Common Memory Dependence Metrics	210
5.35. Recommended Software Prefetch Instructions	213
6.1. SSVE/SME Instruction Operation Fusion Pairings	235
6.2. Register Data Copy Instructions	239
6.3. Register Data Zeroing Instructions	240
7.1. Common Atomic and Exclusive Metrics	247
8.1. Performance Monitoring Events	256
A.1. Integer Execution Classes	263
A.2. Integer Execution Class Availability: M-Series Chips	265
A.3. Integer Execution Class Availability: A-Series Chips	266
A.4. Advanced SIMD and FP Execution Classes	267
A.5. GENERAL Advanced SIMD and FP Class Details	268
A.6. Advanced SIMD and FP Execution Class Availability: M-Series Chips	270
A.7. Advanced SIMD and FP Execution Class Availability: A-Series Chips	271
A.8. Integer Load Latencies	272
A.9. Vector Load Latencies	273
A.10. Integer Store Latencies	273
A.11. Synchronization Instruction Latencies	274
A.12. Memory Execution Unit Bandwidth: P Core	276
A.13. Memory Execution Unit Bandwidth: E Core	276
A.14. Cache and Memory Access Latencies	277
A.15. SSVE Execution Classes	277
A.16. SME ZA Extraction Unit Classes	282
A.17. SME Processing Grid Move From Storage Classes	283
A.18. SME Processing Grid Computation and Move To Storage Classes	284
A.19. SME Engine Load μ op Issue Bandwidth	287
A.20. SME Engine Load Instruction Bandwidth	287

Apple Silicon CPU Optimization Guide: 4.0

A.21. SME Engine Store μ op Issue Bandwidth	287
A.22. SME Engine Store Instruction Bandwidth	288
B.1. Supported Base Instruction Set Parameter	290
B.2. Supported Arm v8/v9 Unofficially Named ELO Feature Parameters	290
B.3. Legacy Feature <code>sysctl</code> Parameter Names	291
B.4. Per Performance Level Sysctl Parameters	291
B.5. Updated Existing Sysctl Parameters	292



Document Release Notes

Table 1. Change History

Edit Date	Owner	Summary
2025-06-06	Apple Inc.	Release for 4.0
2024-03-09	Apple Inc.	Initial Release for 3.0



Chapter 1. Introduction

Apple silicon M-series and recent A-series chip designs include general purpose cores, often referred to as a central processing unit (CPU), along with a graphics processing unit (GPU), a neural engine (ANE), a high-performance fabric, caches, and more. This guide discusses performance optimization opportunities found in the general purpose compute complex, focusing primarily on the general purpose cores, associated vector and matrix acceleration engine (SME engine), and related components. Within a family, such as the M1 family of chips (see [Section 1.1, “Chip Series and Family Naming Conventions”](#)), the general purpose compute core designs are generally the same. However, the core counts, chip topology, bandwidths, and other systems components and characteristics may differ. This guide covers the CPU cores of Apple silicon M-series chips beginning with the M1 family and Apple silicon A-series chips beginning with A14 Bionic.

This guide contains information useful for a wide range of developers. Coverage includes:

- Fundamental CPU and chip characteristics for developers interested in performance and who code primarily in high-level languages, including those developers new to performance optimization
- Detailed instruction descriptions for developers willing to code at the instruction level to leverage the capabilities of the instruction set architecture (ISA)
- Detailed CPU and chip characteristics for developers interested in getting the most from the microarchitecture by coding longer sequences at the instruction level

The discussions in this guide assume that the reader has a basic understanding of instruction level programming, but not of any particular ISA. Likewise, it assumes the reader has knowledge of foundational processor design concepts such as caches, but the guide provides a concise introduction to overall processor microarchitecture.

The guide begins with some background material and then proceeds to discuss the ISA followed by the processor microarchitecture. Within the microarchitecture chapters, the topics are contained to allow the reader to jump to specific sections of interest, but organized to follow the path that instructions flow through the processor.

Note: The optimizations recommended in this guide are intended to be beneficial for all core types in current and future chip families, unless otherwise noted. However, the guide cannot cover all scenarios, and there may be instances where actual performance is different than described. And while future chip families may make different microarchitectural tradeoffs and some performance details may change, the recommendations are anticipated to endure. Any altered recommendations will be clearly noted.

1.1 Chip Series and Family Naming Conventions

To simplify references to multiple chips, the following naming conventions are used in this guide:

- **Mx family of chips:** All Mx, Mx Pro, Mx Max, and Mx Ultra chips, such as the M1, M1 Pro, M1 Max, and M1 Ultra chips

- Note that for any particular family, only a subset of the chip classes may be available
- **M-series chips:** All Mx families of chips, starting with the M1 family
- **Ax family of chips:** All Ax, Ax Bionic, and Ax Pro chips, such as the A18 and A18 Pro chips
 - Note that for any particular family, only a subset of the chip classes may be available. When only one chip is available for a particular family, the specific name is generally used over the family name.
- **A-series chips:** All Ax families of chips, starting with the A14 family
- **Apple silicon:** M-series and A-series chips

1.2 Optimization Process

Prior to pursuing optimization, set performance goals and measurement strategies. Performance goals may be absolute, such as " n frames per second", or relative, such as " n % faster than previous generation with new features added". Goals may be related to throughput, such as " n transactions per second", or to latency, such as "screen updated in n milliseconds". Goals may include managing outlier behavior, such as "no more than n samples dropped per second". Once goals are determined, implement one or more effective methods to measure and track performance against the goal. Instrument applications to self-measure and report performance. Use `mach_absolute_time()` library function to measure time within applications.

To achieve high performance, consider two interrelated facets:

- **Instruction set architecture (ISA):** The ISA contains the commands that software uses to accomplish tasks along with the environment in which those commands execute. It includes instruction definitions, register sets, virtual memory organization, memory model, and more. The ISA is commonly referred to as the hardware-software interface. Broadly, choose instructions and instruction sequences to most efficiently represent the desired functionality.
- **Microarchitecture:** The microarchitecture is the underlying hardware mechanism that implements the ISA. It fetches the instructions from memory and executes the operations, updating values in registers and memory. In order to achieve a high performance implementation of the ISA, the microarchitecture executes many instructions at the same time, even out of order with the specified program order, but importantly preserving the appearance of single-stepped instruction execution. Broadly, choose instruction patterns and use memory access patterns that optimally use available microarchitectural resources, such as cache capacity, instruction type bandwidth, and instruction latencies.

Although distinct, the two facets are inexorably linked. For instance, the microarchitecture may implement certain instructions or instruction sequences more efficiently or in a more performant manner than others.

When considering options for performance improvement, generally pursue opportunities in this order:

1. **Apply compiler optimization:** Enable general compiler optimizations that optimize the code for the platform. "Debug" configurations typically map to lower levels of optimization, which results in less efficient code. Use "Release" configurations

broadly to enable more optimizations. Further, enable the compiler to use specific optimizations, transformations, or advanced instructions on select portions of the code that are performance critical. Many of these are tuned for both the ISA and the microarchitecture. For example, apply `-O3` or `-Ofast` to the files that contain the performance-critical functions. (See [note](#) on the implications of `-ffast-math`). These aggressive optimizations may increase code size (and thus cache footprint and cache misses), which can lead to slower performance for infrequently executed code. Consider using [Profile Guided Optimization \(PGO\)](#) and [Link Time Optimization \(LTO\)](#).

2. **Leverage frameworks and libraries:** Use platform-tuned frameworks and libraries for common tasks. For example, use Accelerate's ZLIB, BLAS, and vDSP frameworks, and Grand Central Dispatch for task management. (See [Section 2.1, "Apple Platform Technologies"](#), [Section 7.4, "APIs for Synchronization and Thread Communication"](#), and [Section 2.1, "Apple Platform Technologies"](#)) And, for example, use the platform's `memcpy()` routine for copying blocks of memory.
3. **Profile code using Instruments to identify the time consuming routines and algorithms:** If optimizing existing code, and before investing in further optimization steps, profile the code using Instruments. Focus efforts on the routines and data structures where the processor is spending the most time. It is common to find unexpected "hot" routines. (See [Chapter 8, Xcode: Performance Monitoring and Analysis](#))
4. **Design data structures and algorithms for performance, and decorate data:** Explore options for efficient storage and location of information. For example, use a tree structure instead of a linked list where appropriate. Also, decorate data to provide hints to the compiler. For example, use `const`, `restrict`, `pragma unroll`, and `pragma vectorize`. Details can be found in the [Clang Language Extension](#) guide.
5. **Develop custom code sequences using intrinsics or assembly instructions, leveraging the full ISA, especially if the compiler does not vectorize the sequence:** Insert sequences of custom code using intrinsics that map directly to processor instructions. Use predefined data types via the `<simd.h>` or `<arm_neon.h>` header files that map directly to processor register types. (See [Section 2.1, "Apple Platform Technologies"](#) and [Section 3.4, "Compiler Intrinsics for Instructions and Data Types"](#)) Use assembly instructions when necessary, and leverage the CPU's vector and matrix acceleration capabilities (SME engine) when beneficial.
6. **Adjust algorithms for better caching:** Consider data working set sizes and use data in patterns that leverage caches for fast data access. Ensure prefetchable patterns and selectively include software prefetches. (See [Section 5.6.3, "L1 Data \(L1D\) Cache"](#), [Shared L2 Cache](#), and [Section 5.6.12, "Prefetching"](#))
7. **Optimized for the microarchitecture, tuning for maximum throughput:** Use Xcode's performance monitoring and analysis tools in [Instruments](#) to identify inefficiencies. Carefully craft code sequences to achieve maximum execution throughput based on microarchitectural characteristics. (See topics across [Chapter 5, Core Microarchitecture Optimization](#))

This guide covers aspects of each of the optimizations steps, with a focus on the processors themselves and on the more detailed optimization opportunities. It is structured according to the two facets, ISA then Microarchitecture. In cases where the topics are best covered by other documentation, such as higher level software constructs

or tool references, links are provided. Before delving into the more detailed opportunities, first consider whether better algorithms, automation, and tuned libraries can help achieve performance goals.

Recommendation: Set performance goals to focus optimization effort.

[Magnitude: High | Applicability: High] Develop performance goals and methods for evaluating performance against those goals. Objectives may include throughput, latency, and outlier behavior targets for whole applications or portions of applications. Goals focus optimization efforts on the opportunities that matter.

1.3 High Impact Recommendations

Although there are many potential optimization opportunities, consider these high magnitude or high applicability items:

- Set performance goals to focus optimization effort: [Section 1.2, “Optimization Process”](#)
- Explore platform technologies and libraries for optimized common functions and algorithms: [Section 2.1, “Apple Platform Technologies”](#)
- High IPC does not necessarily mean high performance. Improve the fundamental algorithm or implementation of the algorithm: [Section 5.5, “Instruction Processing Map and Execution Optimization”](#)
- Use barrier instructions to enforce memory ordering due weakly ordered memory model: [Section 2.12, “Memory Model, Barriers, and Synchronization”](#)
- Use intrinsic functions to manually vectorize the core algorithm in high-level languages: [Section 3.4, “Compiler Intrinsics for Instructions and Data Types”](#)
- Use op-add instructions (e.g., multiply-accumulate) to increase code density and reduce latency: [Section 2.11, “Fused Op with Add/Accumulate Operations \(Integer Unit\)”](#)
- Reduce taken branch density to improve instruction fetch bandwidth: [Section 5.4.4, “Taken Branch Reduction”](#)
- Use conditional instructions to reduce branch mispredictions: [Section 5.4.5, “Conditional Branch Mispredicts and Conditional Instructions”](#)
- Avoid false sharing by allocating independent shared variables to different 128-byte cache lines: [Section 5.6.6, “Improving Cache Hierarchy Performance”](#)
- Pack hot variables into the smallest set of cache lines for improved cache hierarchy performance: [Section 5.6.6, “Improving Cache Hierarchy Performance”](#)
- Query `sysctl` parameters for determining ISA feature support and microarchitectural characteristics: [Appendix B, *Dynamic Determination of Chip-Specific Capabilities*](#)

1.4 Branch Terminology

Many analyses and recommendations involve instructions that alter the control flow of the application. Those discussions rely on these definitions which reflect the characteristics of the Arm v8/v9 AArch64 ISA (see [Section 2.2, “Arm AArch64 ISA”](#)):

- **Target Address:** The next Program Counter (PC) address specified by a control flow altering instruction.
- **Direct Branch:** A control flow altering instruction where the target address is specified in the instruction itself as an offset from the current PC.
- **Indirect Branch:** A control flow altering instruction where the target address is read from a register.
- **Call Branch (Call Instruction, Branch with Link Instruction):** A control flow altering instruction used for entering a subroutine. The PC+4 address is written to a specific register, x30. Some call branches are indirect where the target address is read from a register.
- **Return Branch (Return Instruction):** A special case of an indirect branch used for returning from a subroutine. In the Arm ISA, the target address is obtained from a specific register, x30.
- **Unconditional Branch:** A control flow altering instruction where the next PC is always the Target Address. Note that in the Arm v8/v9 ISA, all indirect branches are unconditional.
- **Conditional Branch:** A control flow altering instruction where the next PC is the Target Address or the current PC+4, depending on evaluation of a condition.
- **Fall Through Branch / Skip Branch:** A conditional branch with a `FALSE` condition. The PC is not updated to the Target Address but rather continues sequentially to PC+4.
- **Taken Branch:** An unconditional branch or a conditional branch with a `TRUE` condition. The PC is updated to the Target Address.
- **Predicted Branch:** A branch for which the processor makes an "educated guess" as to the condition or indirect target address of the branch. The processor begins fetching instructions after fetching a branch according to the prediction. The processor checks the prediction against the actual outcome when the branch executes. Generally, all branches are predicted with the exception of direct unconditional branches.
- **Correctly Predicted Branch:** A predicted branch for which the actual outcome matches that of the prediction.
- **Mispredicted (Incorrectly Predicted) Branch:** A predicted branch for which the actual outcome does not match the prediction. Instructions fetched after the branch must be flushed from the machine.
- **Correct Path:** The stream of instructions that follow a predicted branch according to the correct outcome of the branch. For a conditional branch, the stream may start on either the Target Address or PC+4 depending on the actual outcome. For an indirect branch, the stream starts on the correct Target Address.
- **Wrong (Incorrect) Path:** The stream of instructions that follow a predicted branch according to an incorrect outcome of the branch. For a conditional branch, the stream may start on either the Target Address or PC+4 depending on the actual outcome. For an indirect branch, the stream may start on an incorrect Target Address.
- **Retired Branch:** A branch that executes on the correct path and is committed to program state. Such a branch may be correctly or incorrectly predicted, that is, the next instruction in the predicted stream may be on the correct or wrong path.

- ## 1.5 Performance Monitoring Events

1.6 Decimal and Binary Data Quantities

Table 1.1. Decimal and Binary Data Quantities

Factor	Name	Symbol	Definition
10^3	kilo Byte	KB	$10^3 = 1,000$ Bytes
2^{10}	kibi Byte	KiB	$2^{10} = 1,024$ Bytes
10^6	mega Byte	MB	$10^6 = 1,000,000$ Bytes
2^{20}	mebi Byte	MiB	$2^{20} = 1,048,576$ Bytes
10^9	giga Byte	GB	$10^9 = 1,000,000,000$ Bytes
2^{30}	gibi Byte	GiB	$2^{30} = 1,073,741,824$ Bytes
10^{12}	tera Byte	TB	$10^{12} = 1,000,000,000,000$ Bytes
2^{40}	tebi Byte	TiB	$2^{40} = 1,099,511,627,776$ Bytes

1.7 Change Log

- Added a reference for using the Data-Independent Timing (FEAT_DIT) feature
- Added coverage of A17 Pro, the M4 family of chips, and the A18 family of chips
- Added a description of the Scalable Matrix Extension (SME) ISA and microarchitecture
- Added more details on both floating point and integer element data types

- Corrected:
 - The size of the L1D TLB in the P core of the M3 family of chips: [Table 5.19](#)
 - The μ op execution bandwidth capabilities for several entries in summary tables: [Table 5.13](#) and [Table 5.14](#)
 - The number of Advanced SIMD and FP MUL-class and MOVE2GPR-class execution units in the M3 Max E core: [Table A.6](#)
 - Some performance monitoring event definitions in [Section 8.1.2, “Performance Monitoring Events”](#)



Chapter 2. ISA Optimization: Overview and Integer Unit

The following chapter describes aspects of the processor's instruction set architecture (ISA). However, before coding and tuning custom software, explore the optimized universally installed frameworks and libraries provided for common data types, functions, and algorithms.

2.1 Apple Platform Technologies

Xcode is Apple's integrated development environment (IDE). It offers developers access to a large number of optimized universally installed platform [technologies](#) that include data types, library classes and functions, and services for many common tasks. Prior to writing custom code, explore the provided tuned [technologies](#) to speed development.

- **Accelerate:** Of particular interest, the [Accelerate](#) framework consists of routines for neural networks (BNNS), image and video processing (vImage), digital signal processing (vDSP), transcendentals (vForce), and linear algebra (Sparse Solvers, BLAS, and LAPACK). Outside of Accelerate, Xcode offers multithreaded compression functions (Apple Archive), compression algorithms for LZFS, LZ4, LZMA, and ZLIB (Compression), and small vector and matrix operations (simd).
- **simd:** The [simd](#) module provides basic data types and simple functions related to small vectors and matrices.
- **Grand Central Dispatch and Background Task:** The [Grand Central Dispatch](#) framework and [Background Task](#) framework provide support for scheduling tasks. Also see [Chapter 7, Asymmetric Multiprocessor \(AMP\) Optimization](#)
- **Many Other Technologies:** For example: Core Image, Core Media, Create ML, Image I/O, ML Compute. Use the Metal framework for rendering advanced 3D graphics and performing data-parallel computations using graphics processors.
- **memcpy ():** Use highly tuned `memcpy ( )` for copying blocks of data.

Recommendation: Explore platform technologies and libraries for optimized common functions and algorithms.

[Magnitude: High | Applicability: High] Xcode provides developers access to tuned libraries of mathematical data types, functions, and algorithms. These algorithms are optimized for the available hardware and offer the most portable solution between Apple products and families. Before developing custom solutions, explore the Accelerate and other platform [technologies](#). They may speed both development time as well as application performance without the need for extensive custom code.

2.2 Arm AArch64 ISA

Apple silicon CPUs support the Arm A-Profile 64b (AArch64) ISA. The cores do not support AARCH32 nor ISA prior to Arm v8.0. When used, the term "Arm ISA" refers to Arm AArch64 v8 and follow-on Arm AArch64 v9.

All Apple silicon CPUs documented in this guide (see [Section 1.1, "Chip Series and Family Naming Conventions"](#)) support floating point, floating point half-precision converts, Advanced SIMD, and the CRC32 instruction.

Table 2.1: "Supported Arm Base ISA and EL0 Features" lists the base Arm ISA version and supported EL0 (unprivileged, colloquially *user code*) features for each CPU. Note that the base Arm ISA version is for reference only, and should not be used as the compiler target. Instead, the Clang compiler can automatically enable the appropriate set of ISA features for a given chip by using the `-mcpu` option, such as `-mcpu=apple-m1`. To see the list of chips available in an installed version of the compiler, use `clang --print-supported-cpus`. The ISA feature set enabled by a specified chip can be modified with `"<Clang ISA Extension Name>"`, such as `-mcpu=apple-m1+bf16`. Use chip `apple-latest` only in rare circumstances. It maps to a particular CPU that may change from one compiler generation to the next, to one that may be newer than your CPU, and contain features that are not yet fully stable. Further, when changing compilers, it could lead to inconsistency in ISA version between individual files.

Note that specifying a newer chip for a whole application may prevent it from running on an older chip. Instead, provide multiple code paths for small sections of performance-critical code. Software can dynamically check for the existence of features via the `sysctl` interface (see [Appendix B, Dynamic Determination of Chip-Specific Capabilities](#) for details) and then select between different versions of the code. There are a range of common strategies for creating the various versions from replication and customization of key code snippets within a function to replication, custom naming, and targeted compilation of whole files. To enable select ISA features in specific functions, use:

```
__attribute__((target("OPTION")))
```

`OPTION` is the compiler-accepted name for the particular ISA feature. This strategy requires additional code to test for the availability of the ISA feature combined with calls to the proper version. For an example of this strategy, see [Enable DIT for constant time cryptographic operations](#). And, see Clang documentation for [targeting specific CPUs and ISA features](#).

Table 2.1. Supported Arm Base ISA and EL0 Features

Feature	Description [Clang ISA Extension Name]	CPU Support			
		M1 Family A14 Bionic	M2 Family A15 Bionic	M3 Family A16 Bionic A17 Pro	M4 Family A18 Family
	Base ARM ISA (for reference only)	v8.5 (excluding FEAT_BTI)	v8.6	v8.6	v9.2
FEAT_AES	Advanced SIMD advanced encryption standard (AES) instructions	Yes	Yes	Yes	Yes

Table 2.1. Supported Arm Base ISA and EL0 Features (cont.)

Feature	Description [Clang ISA Extension Name]	CPU Support			
		M1 Family A14 Bionic	M2 Family A15 Bionic	M3 Family A16 Bionic A17 Pro	M4 Family A18 Family
FEAT_AFP	Alternate floating-point behavior	No	No	Yes	Yes
FEAT_BF16	Storage and arithmetic instructions of the brain floating point (BFfloat16) data type [bf16]	No	Yes	Yes	Yes
FEAT_BTI	Instructions to guard against the execution of instructions that aren't the intended target of a branch	No	Yes	Yes	Yes
FEAT_DIT	Data-independent timing instructions	Yes	Yes	Yes	Yes
FEAT_DPB	Clean data cache by address to point of persistence (DC CVAP) instruction	Yes	Yes	Yes	Yes
FEAT_DPB2	Clean data cache by address to point of deep persistence (DC CVADP) instruction	Yes	Yes	Yes	Yes
FEAT_DotProd	Advanced SIMD Int8 dot product instructions	Yes	Yes	Yes	Yes
FEAT_ECV	Support for enhanced counter virtualization, which enhances the Generic Timer architecture	No	Yes	Yes	Yes
FEAT_FCMA	Floating-point complex number instructions	Yes	Yes	Yes	Yes
FEAT_FHM	Floating-point half-precision multiplication instructions	Yes	Yes	Yes	Yes
FEAT_FlagM	Condition flag manipulation instructions	Yes	Yes	Yes	Yes
FEAT_FlagM2	Enhancements to condition flag manipulation instructions	Yes	Yes	Yes	Yes
FEAT_FP16	General half-precision floating-point data processing instructions	Yes	Yes	Yes	Yes
FEAT_FRINTTS	Floating-point to integral valued floating-point number rounding instructions	Yes	Yes	Yes	Yes
FEAT_I8MM	Advanced SIMD Int8 matrix multiplication instructions [i8mm]	No	Yes	Yes	Yes
FEAT_JSCVT	JavaScript conversion instruction	Yes	Yes	Yes	Yes
FEAT_LRCPC	Load-acquire release consistency processor consistent (RCpc) instructions	Yes	Yes	Yes	Yes
FEAT_LRCPC2	Load-acquire release consistency processor consistent (RCpc) instructions version 2	Yes	Yes	Yes	Yes
FEAT_LSE	Atomic instructions to support large systems	Yes	Yes	Yes	Yes

Table 2.1. Supported Arm Base ISA and EL0 Features (cont.)

Feature	Description [Clang ISA Extension Name]	CPU Support			
		M1 Family A14 Bionic	M2 Family A15 Bionic	M3 Family A16 Bionic A17 Pro	M4 Family A18 Family
FEAT_LSE2	Changes to single-copy atomicity and alignment requirements for loads and stores for large systems	Yes	Yes	Yes	Yes
FEAT_PMULL	Advanced SIMD polynomial multiply long (PMULL) instructions	Yes	Yes	Yes	Yes
FEAT_RDM	Advanced SIMD rounding double multiply accumulate instructions	Yes	Yes	Yes	Yes
FEAT_RPRES	Increased precision of reciprocal estimate and reciprocal square root Estimate	No	No	Yes	Yes
FEAT_SB	Barrier instruction to control speculation	Yes	Yes	Yes	Yes
FEAT_SHA1	Advanced SIMD SHA1 instructions	Yes	Yes	Yes	Yes
FEAT_SHA256	Advanced SIMD SHA256 instructions	Yes	Yes	Yes	Yes
FEAT_SHA3	Advanced SIMD SHA-3 instructions	Yes	Yes	Yes	Yes
FEAT_SHA512	Advanced SIMD SHA512 instructions	Yes	Yes	Yes	Yes
FEAT_SME	Scalable Matrix Extension	No	No	No	Yes
FEAT_SME2	Scalable Matrix Extension version 2	No	No	No	Yes
SME_B16F32 ¹	SME accumulate BFloat16 outer products into FP32 single-precision floating-point tiles. Included with FEAT_SME.	No	No	No	Yes
SME_BI32I32 ¹	SME2 accumulate thirty-two 1-bit binary outer products into 32-bit integer tiles. Included with FEAT_SME2.	No	No	No	Yes
SME_F16F32 ¹	SME accumulate FP16 half-precision floating-point outer products into FP32 single-precision floating-point tiles. Included with FEAT_SME.	No	No	No	Yes
SME_F32F32 ¹	SME accumulate FP32 single-precision floating-point outer products into single-precision floating-point tiles. Included with FEAT_SME.	No	No	No	Yes
FEAT_SME_F64F64	SME double-precision floating-point outer product instructions	No	No	No	Yes
SME_I8I32 ¹	SME accumulate 8-bit integer outer products into 32-bit integer tiles. Included with FEAT_SME.	No	No	No	Yes
SME_I16I32 ¹	SME2 accumulating 16-bit outer products into 32-bit integer tiles. Included with FEAT_SME2.	No	No	No	Yes

Table 2.1. Supported Arm Base ISA and EL0 Features (cont.)

Feature	Description [Clang ISA Extension Name]	CPU Support			
		M1 Family A14 Bionic	M2 Family A15 Bionic	M3 Family A16 Bionic A17 Pro	M4 Family A18 Family
FEAT_SME_I16I64	SME 16-bit to 64-bit integer widening outer product instructions	No	No	No	Yes
FEAT_SSBS	Instructions to control speculation of loads and stores	Yes	Yes	Yes	No
FEAT_WFXT	"Wait for" WFE and WFI instructions with timeout	No	No	No	Yes

¹References a specific named indicator bit in an Arm ISA Feature Register that is not referred to individually as "FEAT_" in Arm's ISA documentation. Added for compatibility purposes and is available in sysctl.

2.3 Arm Reference Documents

Arm provides numerous reference and technical documents about the ISA, including these relevant topics:

- [A-Profile ISA](#)
- [Architecture Reference Manual](#)
- [Advanced SIMD and FP ISA Overview \(also referred to as NEON\)](#)
- [Advanced SIMD and FP ISA High-Level Language Intrinsics](#)
- [Scalable Matrix Extension Programmer's Guide](#)
- [Scalable Matrix Extension Multi-Part Overview](#)
- [Software Application Binary Interface \(ABI\)](#), specifically "Procedure Call Standard for the ARM 64-bit Architecture" and Advanced SIMD portions of "Vector Function Application Binary Interface Specification for AArch64"
- [Memory Model](#)
- [Virtual Memory Management](#)

2.4 ISA Characteristics

The foundations of the Arm ISA are similar to other ISAs. Commonalities include:

- **Rich instruction set:** The Arm ISA features an extensive instruction library of both compiler targetable and special purpose instructions. Although the details may be subtly different, many operations found in other architectures are available with the greatest differences being found in the Single Instruction Multiple Data (SIMD) (which includes integer and floating point data types) and floating point (FP) scalar instruction set.
- **Flexible memory address computation:** The Arm ISA offers memory address computation with up to two registers, an immediate value, and optional scaling.
- **Subset support for general purpose registers:** General purpose registers are accessible as full 64-bit registers or 32-bit subsets.

- **SIMD unit with both integer and floating point types, including subset scalar options:** Vector registers are accessible as single element scalars in floating point types as well as both integer and floating point packed Single Instruction Multiple Data (SIMD) full registers. The Arm ISA supports 16b half, 32b single, and 64b double-precision floating point types. Arm floating point technology is fully IEEE-754 compliant.
- **Status flags register:** Process State (PSTATE) contains the ALU status flags.
- **Little-endian data format:** Although the Arm ISA offers some flexibility, Apple implementations natively support little endian instructions and data.
- **Non-temporal memory access instructions:** Software can specify which data items are likely to be used only once and therefore do not need to be cached.
- **Software prefetch:** The Arm ISA offers the ability to specify data prefetches directly in the software.

However there are characteristics to be aware of:

- **RISC load-store architecture:** Aligned with Reduced Instruction Set Computing, the Arm instruction set generally requires explicit load and store instructions to move data in and out of registers. There are some exceptions, however, such as Compare and Swap atomic instructions (ex., CAS) and the Load-/Store-and-{operation} atomic instructions (ex., LDSMIN/STSMIN).
- **32b fixed length instructions:** Arm instructions are always 32b in length and are always 4B aligned. Fixed instruction lengths make the processor simpler and more efficient. However, most of those 32b are used for encoding the operation and operand specifiers, leaving limited remaining bits for immediate values.
 - **Limited immediate bit range for offsets and constants:** Sequences of instructions may be required to build constants or offsets from immediate fragments. Alternatively, constants or offsets can be loaded from memory using PC-relative addressing, which allows larger immediate offsets. ([Section 2.8, "Addressing Forms, Instruction Immediates, and Operand Shifts"](#))
- **128b Advanced SIMD and FP unit vector registers :** Vector registers are 128b in length. These registers support scalar and packed 64b, 32b, and 16b floating point values and packed 64b, 32b, 16b, and 8b integer values. The Arm ISA does not support 80b extended floating point precision.
- **Separate source and destination registers (i.e., nondestructive operations):** Arm instructions typically specify a destination register separate from the source register(s). In other ISAs, one of the source registers is often also the destination register. As a side benefit of separate destination registers, fewer move operations are typically required to preserve registers values before using them as source/destination. ([Section 2.7, "Separate Source and Destination Registers "](#))
- **Register-based subroutine return addresses:** Arm subroutine calls write the return address into a special general purpose "link" register, and subroutine returns read the return address from the same register. The Arm approach offers more flexibility than approaches that directly use the memory stack, especially for leaf functions where writing to the stack may be unnecessary. However, additional instructions may be required to move the return value to the memory-based stack when needed.

- **Weakly ordered memory model:** The Arm ISA employs a weakly ordered memory model where software executes specific instructions to preserve ordering only when it matters, generally improving memory performance over stricter ordering such as Total Store Ordering. ([Section 2.12, “Memory Model, Barriers, and Synchronization”](#))
- **Synchronization required for self-modifying code:** Cores may not immediately observe modifications to instruction memory. To implement self-modifying code (SMC) or just-in-time (JIT) compiled code, software must execute specific synchronization instructions. Although somewhat more complicated for software, it leads to more efficient processor implementations since the processor does not have to constantly monitor for updated instruction bytes. ([Section 2.13, “Self-Modifying or JIT-Generated Code Deployment”](#))
- **Dedicated long vector and two-dimensional matrix acceleration ISA extension:** Some chips feature a separate set of registers (vectors and a 2D array) along with new instructions for accelerating vector and matrix operations (SSVE/SME). Use of these features requires a mode switch that disables 128b Advanced SIMD and FP but offers higher bandwidth.

2.5 Syntax

Arm instructions follow a common syntax. For a complete description, see the Arm Architecture Reference Manual Chapter C1.2. Some common elements include:

- **{ }:** Any item enclosed by curly brackets is optional.
- **[]:** Any items enclosed by square brackets constitute a list of alternative characters. In some case the square brackets are part of the syntax itself, such as addressing modes or vector elements.
- **a|b:** Alternative words are separated by a vertical bar and can be surrounded by parentheses to delimit them. For example, $U(ADD|SUB)W$ represents $UADDW$ or $USUBW$. $(S|U)$ commonly indicates signed or unsigned (zero) extension. $(W|H|B|X)$ commonly indicates extension base size.
- **#:** Optional character to introduce a constant immediate operand.
- **imm *n*:** An immediate value.
- **(LSL|LSR|ASR):** A operand shift specification. See [Section 2.8, “Addressing Forms, Instruction immediates, and Operand Shifts”](#).
- ***XT*:** A operand extension specification. See [Section 2.8, “Addressing Forms, Instruction immediates, and Operand Shifts”](#).

In some examples, *register sizes are denoted on the instruction mnemonic* instead of repeated on each operand. See [Section 2.6, “Registers”](#).

In some examples, *intrinsic functions are used in high-level languages* instead of instruction mnemonics. See [Section 3.4.2, “Computation Intrinsics: A General Guide”](#)

2.6 Registers

The Arm ISA uses registers to hold values for processing, and most instructions require explicit source and destination registers. In some cases, specific register sizes or vector

element organizations are required. For a complete description, see the Arm Architecture Reference Manual Chapter C1.2. For information on parameter passing conventions, see [Writing Arm64 Core for Apple Platforms](#).

Relevant registers, sizes, and organizations include:

Program Counter (PC): Register that holds the virtual address of the current instruction. Software cannot write directly to the PC. It can only be updated on a branch, exception entry, or exception return.

General Purpose Registers (Rn): The 31 general purpose registers (0-30) used for general purpose computation. The general purpose registers and related instructions are often simply referred to as "integer" registers and instructions. In a general-purpose register field, the value 31 represents either the current stack pointer or the zero register, depending on the instruction and the operand position. Note that R30 serves as the link register for procedure calls.

Size qualified names for the general purpose registers are:

- Wm: The 32b subset of Rm
- Xn: The full 64b version of Rn
- WZR: The 32b zero register
- XZR: The 64b zero register
- WSP: The 32b stack pointer (unused on Apple silicon)
- XSP: The 64b stack pointer

When the data size is 32 bits, the lower 32 bits of the register are used. The upper 32 bits are ignored on a read and cleared to 0 on a write.

The ARM standard allows certain decisions to be made by the platform designers. Apple platforms adhere to the following choices:

- The platform reserves register x18. Do not use this register.
- The frame pointer register (x29) must always address a valid frame record. Some functions — such as leaf functions or tail calls — may opt not to create an entry in this list. As a result, stack traces are always meaningful, even without debug information.

Advanced SIMD and FP Registers (vn): The 32 Advanced SIMD and floating point registers (0-31) used for scalar floating point as well as integer and floating point SIMD computation. The Advanced SIMD and FP registers and related instructions are often simply referred to as "vector" registers and instructions, hence the "V" notation. In the context of this document, the term vector does not imply only SIMD operations. It also includes scalar operations that use the same registers and datapath. Compiler intrinsic data types for these registers are described in [Section 3.4, "Compiler Intrinsics for Instructions and Data Types"](#).

Scalar names for Advanced SIMD and FP registers are:

- Bn: 8b scalar subset name of vn
- Hn: 16b scalar subset name of vn
- Sn: 32b scalar subset name of vn

- `Dn`: 64b scalar subset name of `vn`
- `Qn`: 128b scalar name of `vn`

SIMD and floating point instructions that operate on scalar data only access the lower bits of a SIMD and floating point register. The unused high bits are ignored on a read and cleared to 0 on a write.

Operations that operate on vector registers often require an organization, that is, an element size multiplied by the number of elements, such as:

- `vn.8B`: 8b x 8 lanes vector organization of `vn`

For a full description of the vector registers and organizational options, see [Section 3.1, "Advanced SIMD and FP Data Types"](#).

The element size (in bits) multiplied by the number of elements (lanes) must equal either 64 or 128. When 64b, the upper 64 bits of the register are ignored on a read and cleared to zero on a write.

A few load, store, and table instructions operate on *multi-vectors*, that is, a group of two or four sequential vector registers. Although sequential, the list may "wrap around" from the upper end of the register file to the beginning, such as:

- `LD4R { V31.S, V0.S, V1.S, V2.S }, [X0]`

For many instructions, the source and destination vector registers share the same (element size, lanes) specification. Apple assemblers and disassemblers use a simplified coding scheme where the registers are named generically, such as `vn`, and the (element, lanes) specification is attached to the instruction mnemonic instead. For example, the following are equivalent:

```
ADD      V2.4S, V3.4S, V4.4S
ADD.4S   V2, V3, V4
```

Note that this simplified scheme of attaching the vector register organization to mnemonic instead of each operand is not part of the Arm notation, but some examples use it to improve readability.

In cases of mismatched (element size, lanes) specifications between sources and destinations, Apple assemblers apply the specification attached to the instruction mnemonic of the destination.

Process State (`PSTATE`): The register that holds the contents of the arithmetic flags: Negative Condition (`N`), Zero Condition (`Z`), Carry Condition (`C`) and Overflow Condition (`V`). Note though that the Carry Condition (`C`) in the Arm ISA has opposite semantics during subtraction compared with other ISAs. `PSTATE` also includes two SSVE/SME-related fields: Streaming SVE mode (`SM`) and tile/array enabled mode (`ZA`).

SSVE/SME Registers: The SSVE 64B/512b vector registers (`z0–z31`), 64b predicate registers (`p0–p15`), and SME two-dimensional registers (`zA`, 4KiB, number of registers depends on the data type) used for accelerating certain mathematical algorithms. The 512b register `zT0` supports the lookup table feature.

Details on these registers and associated instructions can be found in [Chapter 4, ISA Optimization: Scalable Matrix Extension \(SME\)](#).

2.7 Separate Source and Destination Registers

Arm instructions typically specify a destination register separate from the source register(s). Other architectures have a combined source/destination register. This is often referred to as "destructive", as the source value is "destroyed" when the instruction writes it's result.

In the Arm ISA, source registers are generally not automatically overwritten unless the destination register explicitly matches the source register. As a side benefit, few move operations are required to preserve registers values before using them as source/destination.

For example, a basic "Add (register)" takes a destination register `xd` as well as two source registers `xn` and `xm`:

```
ADD Xd, Xn, Xm    // ([Xd] = [Xn] + [Xm])
```

Some Advanced SIMD and FP operations even take 3 source registers along with a destination register, such as "floating point fused Multiply-Add to accumulator":

```
FMADD Dd, Dn, Dm, Da    // ([Dd] = [Da] + ([Dn] * [Dm]))
```

Note that some operations do exist in the ISA where a source/destination register is specified, such as "Signed Absolute difference and Accumulate" that accumulate into a register:

```
SABA Vd.T, Vn.T, Vm.T    // ([Vd] = [Vd] + ABS([Vn] - [Vm]))
                        // where T is the element organization
```

The Arm ISA generally denotes these source/destination registers just as destination registers `<*d>` but the values are also read as sources according to the instruction pseudocode.

Note that destructive destinations are more common in SSVE/SME instructions and are not covered here.

Non-SSVE/SME instructions with a destination register that also serves as a source register include:

- **Bitfield Operations:**
 - BFM, BIT, BIF, BSL, SLI, SRI
- **Add/Subtract/Shift/Multiply with Add/Accumulates:** (see [Section 3.5.4, "Fused Op with Add/Accumulate Instructions \(Integer and ASIMD and FP Types\)"](#))
 - SABA, SABAL, SADALP, SRSRA, SSRA
 - UABA, UABAL, UADALP, URSRA, USRA
 - FMLA, FMLS
 - FMLAL(2), FMLSL(2)
 - BFMLALB, BFMLALT

- MLA, MLS
- SMLAL(2), SMLSL(2), SQDMLAL(2), SQDMLSL(2)
- UMLAL(2), UMLSL(2)
- SQRDMLAH, SQRDMLSH
- **Dot Products:**
 - UDOT, SDOT, USDOT, SUDOT, BFDOT
- **Matrix Multiplies:**
 - SMMLA, UMMLA, USMMLA, BFMMLA
- **Partial Immediate Moves:**
 - BIC (vector immediate), MOVK, ORR (vector immediate)
- **Vector Table Lookup and Inserts:** (see [Section 3.5.7, “Shuffle and Permute Instructions”](#))
 - TBX
- **Vector Inserts:**
 - INS
- **Second Half of Narrowing Operations:** (see [Section 3.3.16, “ASIMD Narrow, Long, Wide: N / L / W”](#))
 - FCVTN2, FCVTXN2, BFCVTN2
 - ADDHN2, RADDHN2, RSHRN2, RSUBHN2, SHRN2, SQRSHRN2, SQRSHRUN2, SQSHRN2, SQSHRUN2, SUBHN2, UQRSHRN2, UQSHRN2
 - SQXTN2, SQXTUN2, UQXTN2, XTN2

2.8 Addressing Forms, Instruction immediates, and Operand Shifts

For memory addresses, the Arm ISA allows a 64-bit index register to be added to a 64-bit base register, with optional scaling of the index by the access size. Additionally it allows for sign-extension or zero-extension of a 32-bit value within an index register, followed by optional scaling. It also supports PC-relative addressing. For a complete description, see the [Arm Architecture Reference Manual Chapter C1.3](#).

Leveraging these addressing modes improves code density by incorporating offsets and scale factors directly into the memory instructions, instead of requiring a chain of multiple instructions. However, some combinations result in additional microoperations (μops) to update pointers, and in one case, an additional μop to calculate the address. See [Section 5.4.8, “Multi-μop Instructions”](#) and [Section 5.6.1, “Address Generation”](#) for more details.

The addressing modes have different ranges from the base register (or PC, in the case of PC-relative addressing). Some offsets are signed and thus the range is centered on the base address, whereas others are unsigned and the range extends only to higher addresses from the base register. Some offsets are specified in number of bytes and

others are in number of elements. When elements, the encoded immediate is scaled by the transfer size, but the assembly still reads bytes. The modes with their ranges and example instructions are:

Table 2.2. Load and Store Addressing Modes

Addressing Mode / Form	Immediate	Example
Base Only Address = base [base]	None.	Exclusive, atomic, acquire, release LDSET X1, X2, [X0]
Base Plus Offset (immediate) Address = base + (imm offset) [base{, #imm}]	12b offset, zero extended, implied scale by transfer size (0 to 4095 <i>elements</i>)	Single register operations LDR X1, [X0, #8]
	9b offset, sign extended (-256 to 255 <i>bytes</i>)	Single register unscaled operations LDUR X1, [X0, #-6]
	7b offset, sign extended, implied scale by transfer size (-64 to 63 <i>elements</i>)	Paired register operations LDP X1, X2, [X0, #-8]
Base Plus Offset (register) Address = base + extended and scaled reg offset [base, {W X}{,<xt> {amt}}]	No offset. UXTW LSL SXTW SXTX extend, optional scale by transfer size specified as amount #2 #3 [†]	Single register operations LDR X2, [X0, W1, UXTW #3]
Pre-indexed (immediate) Address = base + imm offset Base updated with address [base{, #imm}]!	9b offset, sign extended (-256 to 255 <i>bytes</i>)	Single register operations LDR X1, [X0, #8]!
	7b offset, sign extended, implied scale by transfer size (-64 to 63 <i>elements</i>).	Paired register operations LDP X1, X2, [X0, #16]!
Post-indexed (immediate) Address = base Base updated with base + imm offset [base], #imm	9b offset, sign extended (-256 to 255 <i>bytes</i>)	Single register operations LDR X1, [X0], #8
	7b offset, sign extended, implied scale by transfer size (-64 to 63 <i>elements</i>).	Paired register operations LDP X1, X2, [X0], #16
	Offset must equal structure size	ASIMD and FP operations only LD1 {Q1.16B,Q2.16B}, [X0], #32
Post-indexed (register) Address = base Base updated with base + reg offset [base], X	None	ASIMD and FP operations only LD1 {Q2.16B,Q3.16B}, [X0], X1
Literal (PC-relative) Address = label label (converted to #imm)	19b offset, sign extended word (4B) (-1MiB to 1MiB-1 <i>bytes</i>)	Single register operations LDR <literal>

[†]**Extended register offsets:** the memory address equals the sum of the base register with the extended and scaled offset register. In most cases, several different assembly annotations are allowed for the same operation. The extension is performed before the shift.

33

the space last after the last load (or along with the last load). See [Writing Arm64 Code for Apple Platforms](#) for a discussion of the Arm ABI.

Recommendation: Use a single pre- or post-indexed pointer adjustment operation per sequence of stores or loads.

[Magnitude: Medium | Applicability: Medium] Avoid sequences of pre- or post-indexed stores or loads because the chain of address updates creates unnecessary register dependencies. For sequential memory locations, access data through base plus positive immediate offsets before adjusting the pointer at the end of the sequence. For stack operations, use either a pre-index store operation or an `ADD` or `SUB` at the beginning of a push sequence to allocate needed stack space. Use either a post-index load operation or an `ADD` or `SUB` at the end of a pop sequence to deallocated unneeded stack space.

2.8.2 Constant Generation

A common practice with the Arm ISA is to store constants within binaries using small data blocks placed into the code pages where space allows – for example, in between functions. The values can be loaded into registers using nearby PC-relative loads. This can be a good strategy because it minimizes code size: a single instruction can load a full-size constant value.

However, short sequences of move, logic, and shift instructions may be advantageous:

- 16-bit MOV instructions:** Many constants are small and can easily be encoded into the 16b immediate fields available in the "wide immediate" `MOVZ`, `MOVN`, and `MOVK` instructions. Of note, the `MOVK` instruction accepts a 16b immediate, optionally shifts the immediate 16, 32, or 48 bits to the left, and merges it with the existing value in the destination register. To create a 64b constant `0x0123456789ABCDEF` in `X1`:

```
MOVZ X1, #0123 LSL 48
MOVK X1, #4567 LSL 32
MOVK X1, #89AB LSL 16
MOVK X1, #CDEF
```

In this worst case of needing 4 moves, the sequence executes with the same latency as the alternative load operation; see [Section A.3.1, "Load Latency"](#). In most cases, fewer than 4 instructions are needed and the throughput and latency are better than what one can achieve with loads.

- Better cache performance:** Constant values obtained through loads must be read from the data cache, even though they are within the instruction code space. This results in some redundancy between the caches as portions of the code space must be loaded into the data cache. More critically, the "islands" of data within the instruction code space are not automatically prefetched by the instruction-side prefetcher. And, they are hard for the data-side prefetchers to prefetch effectively because they are scattered randomly throughout the code space. This typically results in at least one or more data cache misses for each new island of immediate data that is accessed. In contrast, the few extra move operations are naturally fetched as part of the code sequence.

- **Fast Constant Generation:** In some cases, the processor can leverage the Fast Constant Generation feature if specific coding rules are followed. This reduces the latency and execution bandwidth required for these operations. See [Section 5.5.3.3, “Register Constants”](#).

Recommendation: Use short sequences of ALU operations to construct constants.

[Magnitude: Low | Applicability: Medium] Use short sequences of moves, adds, and shift instructions to create constants, even though this results in a slight increase in code size. The ALU sequences commonly improve latency, reduce pressure on the load execution units, and leverage immediately handling features of the processor.

2.8.3 Long Address Generation

The Arm ISA requires a two-step instruction sequence to generate long distance PC-relative addresses. `ADRP` adds a 21-bit signed immediate value to the page address (sometimes referred to as page number) of the PC while clearing the bottom 12 bits. This allows an address to be generated +/- 4 GiB from top of the current PC page. `ADRP` is often followed by an `ADD` instruction with a page offset as an immediate value to generate a full address. In many cases, the assembly includes a name instead of the immediate value, which is converted to an immediate value by the linker.

```
0000000000000132c  ADRP X0, #0x84          ; 0x84000 + 0x001000 = 0x85000
00000000000001330  ADD  X0, X0, #0xa00      ; 0x85000 + 0x000a00 = 0x85a00
```

2.9 Condition Codes

Branch and conditional instructions test `PSTATE.NZCV` condition flags against condition codes. Condition flags can be set by explicit comparison instructions, such as `CMP` and `FCMP`, or certain arithmetic instructions, such as `SUBS`.

Table 2.3. Condition Codes

Mnemonic extension	Meaning (integer)	Meaning (floating point)	Conditional flags
EQ	Equal	Equal	Z==1
NE	Not equal	Not equal, or unordered [†]	Z==0
CS / HS	Carry set/Unsigned higher or same	Greater than, equal, or unordered [†]	C==1
CC / LO	Carry clear/Unsigned lower	Less than	C==0
MI	Minus, negative	Less than	N==1
PL	Plus, positive or zero	Greater than, equal, or unordered [†]	N==0
VS	Overflow	Unordered [†]	V==1
VC	No overflow	Not unordered [†]	V==0
HI	Unsigned higher	Greater than, or unordered [†]	C==1 and Z==0
LS	Unsigned lower or same	Less than or equal	C==0 or Z==1

Mnemonic extension	Meaning (integer)	Meaning (floating point)	Conditional flags
GE	Signed greater than or equal	Greater than or equal	N==V
LT	Signed less than	Less than, or unordered [†]	N!=V
GT	Signed greater than	Greater than	Z==0 and N==V
LE	Signed less than or equal	Less than, equal, or unordered [†]	Z==1 or N!=V
AL / NV ^{††}	Always (unconditional)	Always (unconditional)	Any

^{††}The `NV` condition code does not appear in the Arm Architecture Reference Manual, however some assemblers accept it. Although the `NV` condition code implies "Never" and is encoded to suggest inverted `AL` "Always", the behavior of `NV` is equivalent as `AL` which is "always taken". See `shared/functions/system/ConditionHolds()` shared pseudocode in the Arm reference manual.

The Arm instruction set offers a number of instructions that modify register values depending on a condition. See [Section 5.4.5, “Conditional Branch Mispredicts and Conditional Instructions”](#) for a list of these instructions and recommendations on when to employ them.

The Arm ISA offers a limited set of *fused* instructions that combine an operation with an add or accumulate. For example, `MADD` (Multiply-Add) multiplies two register values, adds a third register value, and writes the result to the destination register.

Almost all ELO instruction and data bytes reside in Arm "normal" type memory. Normal memory employs a "weakly ordered" memory model. Although a thread's own loads and stores appear ordered from within that thread, both loads and stores may be reordered with respect to other processors and system memory in order to optimize latency and bandwidth. When ordering is needed for specific regions of code, barrier instructions are available to guarantee ordering.

For more information on Arm memory types, weakly ordered memory, and barriers, see [Memory Model](#).

Arm offers further examples of barrier behavior in Appendix K11 "Barrier Litmus Tests" in the [Arm Architecture Reference Manual](#).

Lastly, see topics in [Chapter 7, Asymmetric Multiprocessor \(AMP\) Optimization](#) that discuss APIs for inter-thread communication and thread checking.

Recommendation: Use barrier instructions along with acquire and release semantics to enforce memory ordering due to weakly ordered memory model.

[Magnitude: High | Applicability: Low] Use inter-thread communication APIs where possible. When writing custom code, if any particular memory ordering is required, use barrier functionality to ensure ordering. In addition to DMB and DSB instructions, other instructions may optionally include the same barrier functionality, such as LDREX/STREX, load-acquire, store-release, and atomics.

2.13 Self-Modifying or JIT-Generated Code Deployment

MacOS Only:

Self-modifying code and just-in-time generated code are techniques where software modifies or generates code by writing instruction words as data to memory and then fetches them back as instructions. In the Arm ISA, writes to instruction memory by data stores won't naturally be seen by the Instruction Fetch Unit until the instructions are re-fetched from memory. This typically happens only when the existing bytes are first flushed from the instruction cache due to non-use or due to an explicit flush operation. To facilitate this, the Arm ISA offers data and instruction barriers and other cache maintenance instructions that must be executed in a specific sequence to ensure that bytes have been written to memory before the Instruction Fetch Unit attempts to read them. By their nature, some of these instructions are privileged and cannot be executed from user code.

Apple offers APIs to enable dynamic code generation. See [Porting Just-In-Time Compilers to Apple Silicon](#) guide for more information.

Recommendation: Avoid frequent JIT-generated code and carefully follow prescribed steps.

[Magnitude: Low | Applicability: Low] Self-modifying or JIT-generated code requires executing several steps to ensure the new code is visible to the instruction fetch unit in the processor. One step includes invalidating the instruction cache, which evicts other useful cached instructions. Avoid frequent code deployment, and hence frequent instruction cache invalidations, as much as possible.

2.14 Data-Independent Timing (FEAT_DIT)

Certain instructions in the Arm ISA may take a different amount of time to execute depending on the data values on which they operate. Malicious code running on the chip might use this property to infer information about the data the CPU is processing, such as cryptographic keys, or other sensitive data.

Apple silicon supports Arm's data-independent timing capability in which the processor completes certain instructions in a constant amount of time. With DIT enabled, the processor uses the longer, worst-case amount of time to complete the instruction,

regardless of the input data. The CPU may also disable certain other microarchitectural performance optimizations. Thus DIT should only be used in very specific situations.

See Xcode documentation for details on how to use [DIT](#).

2.15 ISA Characterization

Broad bucket characterization of ISA usage is available through the following metrics.

Table 2.4. Common Instruction Mix Metrics

Name and Formula (Event Definitions: Section 8.1, “CPU Counters With Performance Monitoring Events”)	Description
Branch Density[†]: <Ev INST_BRANCH> / <Ev INST_ALL>	Proportion retired branch instructions (including calls and returns) of all retired instructions
Integer Operation Density: <Ev INST_INT_ALU> / <Ev INST_ALL>	Proportion retired integer execution instructions of all retired instructions, excluding branches and memory operations
Advanced SIMD and FP Operation Density: <Ev INST_SIMD_ALU> / <Ev INST_ALL>	Proportion retired Advanced SIMD and FP execution instructions of all retired instructions, excluding memory operations
Advanced SIMD Operation Density: <Ev INST_SIMD_ALU_VECTOR> / <Ev INST_ALL>	Proportion retired Advanced SIMD execution instructions (including integer and floating point data types) of all retired instructions, excluding memory operations. Note: Available on M2 family of chips and following, and A15 Bionic chip and following
Load and Store Density: <Ev INST_LDST> / <Ev INST_ALL>	Proportion retired load and store instructions of all retired instructions
Integer Load Density: <Ev INST_INT_LD> / <Ev INST_ALL>	Proportion retired integer load instructions of all retired instructions
Integer Store Density: <Ev INST_INT_ST> / <Ev INST_ALL>	Proportion retired integer store instructions of all retired instructions
Advanced SIMD and FP Load Density: <Ev INST_SIMD_LD> / <Ev INST_ALL>	Proportion retired Advanced SIMD and FP load instructions of all retired instructions
Advanced SIMD and FP Store Density: <Ev INST_SIMD_ST> / <Ev INST_ALL>	Proportion retired Advanced SIMD and FP store instructions of all retired instructions
Data Barrier Density: <Ev INST_BARRIER> / <Ev INST_ALL>	Proportion retired data barrier (DSB, DMB) instructions of all retired instructions

[†] Additional detailed branch instruction characterization is available in [Table 5.9: “Common Branch Instruction Mix and Direction Metrics”](#).



Chapter 3. ISA Optimization: Advanced SIMD and FP Unit

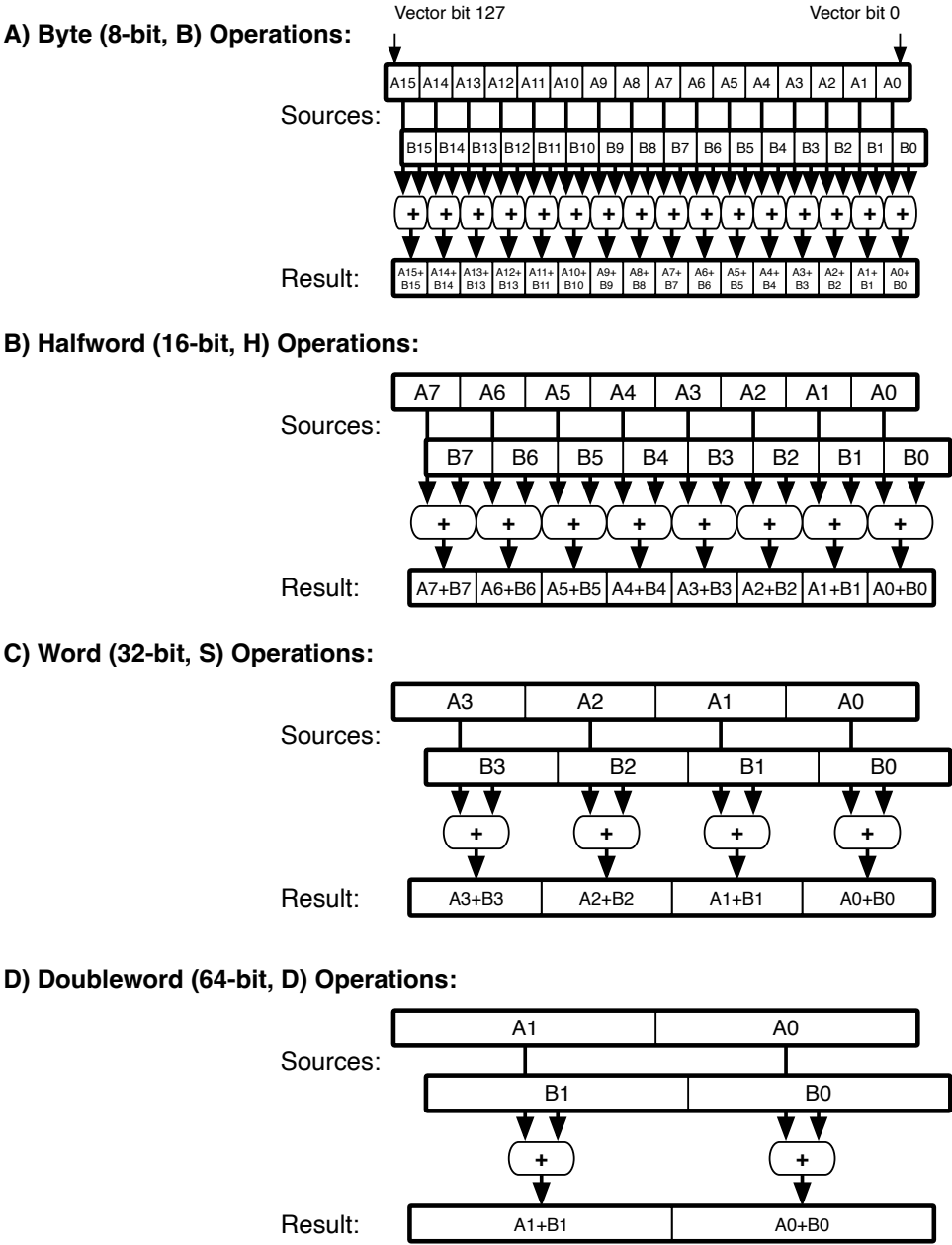
Arm's [Advanced SIMD and FP](#) instructions and registers offer a wide range of operations on scalar and packed data types in vector registers. Note that the Advanced SIMD architecture, its associated implementations, and supporting software, are commonly referred to as NEON technology, although Arm no longer uses that term broadly. This section provides an overview of Advanced SIMD and FP capabilities, including data types, mathematical operations, and specific instructions. It is not exhaustive. It is designed to highlight key capabilities and present the Arm ISA versions of common and important operations. (Details on the Streaming Scalable Vector Extension (SSVE) and Scalable Matrix Extension (SME) can be found in [Chapter 4, ISA Optimization: Scalable Matrix Extension \(SME\)](#), although many of the concepts from this chapter apply to SSVE/SME as well.)

Note: Before writing custom code using assembly or intrinsics, explore the auto-vectorization capabilities of the compiler. Considering applying the `-ffast-math` compiler flag (or use `-ofast`) to allow the compiler to aggressively optimize floating point code. This allows the compiler to restructure the floating point equations to make them more efficient to execute, but the final result may be (typically slightly) different. Commonly, this is due to rounding differences when performing the operations in different orders. Further, [fused multiply-accumulate](#) instructions skip the rounding step between the multiply and the add. Lastly, `fast-math` does not properly support infinity nor Not-a-Number (NaN), however these are only typically used in some domain-specific codes. Apple silicon and the Arm FP ISA are IEEE754 compliant. Follow IEEE754 best practices. For a full list of the transformations, see the [Clang User Manual](#).

Typical operations executing in these pipelines can perform up to 16 parallel operations at once down the length of each vector (with byte-wide operations). Operations with larger data types reduce the number of parallel operations performed in inverse proportion to the size of each data element. These effects are both illustrated in [Figure 3.1: “Standard Advanced SIMD Addition ADD \(vector\)”](#) which shows the operations performed by typical vector addition operations. In addition, these pipelines can perform scalar integer or floating point operations, using only the first (#0) element of each vector instead of the entire vector width. There are only a few scalar integer instructions that execute in the Advanced SIMD and FP pipelines given that Integer Unit pipelines typically execute such instructions more efficiently and with lower latency. Thus, scalar operations in the Advanced SIMD and FP Unit primarily operate on floating point data types..

Because the integer pipelines can already handle most kinds of scalar integer operations quite well, scalar Advanced SIMD instructions are primarily used for floating point calculations.

Figure 3.1. Standard Advanced SIMD Addition ADD (vector)



3.1 Advanced SIMD and FP Data Types

Advanced SIMD and FP registers and operations support the following data types:

- IEEE 754 Floating Point Numbers
 - Half precision (16b)
 - "Brain" float precision (16b), with FEAT_BF16
 - Single precision (32b)
 - Double precision (64b)

- Integer Numbers
 - Byte (8b)
 - Halfword (16b)
 - Word (32b)
 - Doubleword (64b)

3.1.1 Floating Point Data Types

"Brain" float precision (16b), commonly referred to `bf16` or `bfloat16`, is a data type commonly used in machine learning algorithms. The data type is 16 bits wide, the same as the half-precision floating-point data type. However, `bf16` uses the large 8-bit exponent range of the single-precision 32-bit floating-point data type but a significantly smaller mantissa precision than even the half-precision data type, 7 bits plus a sign bit. For comparison, half precision mantissa uses 10 bits plus a sign bit while single precision uses 23 bits plus a sign bit. This feature is available on processors that support the FEAT_BF16 feature. (See [Section 2.2, "Arm AArch64 ISA"](#)).

Table 3.1. FP Types Supported in the Advanced SIMD and FP Unit: By Field

Data Word Size	Name	Exponent					Mantissa	
		Bits	Min	Min Decimal	Max	Max Decimal	Bits*	Decimal Precision (in digits)
16b	half	5	-14	-4.22	+15	+4.51	10+1	3.31
16b	bfloat	8	-126	-37.95	+127	+38.23	7+1	2.40
32b	single	8	-126	-37.95	+127	+38.23	23+1	7.22
64b	double	11	-1022	-307.83	+1023	+307.95	52+1	15.95

Table 3.2. FP Types Supported in the Advanced SIMD and FP Unit: By Range

Data Word Size	Name	Absolute Range Available (binary)			Absolute Range Available (decimal)		
		Min Denormal	Min	Max	Min Denormal	Min	Max
16b	half	2^{-24}	2^{-14}	$2^{+16} - 2^{+5}$	5.960×10^{-8}	6.104×10^{-5}	65504
16b	bfloat	2^{-133}	2^{-126}	$2^{+128} - 2^{+120}$	9.184×10^{-41}	1.175×10^{-38}	$3.390 \times 10^{+38}$
32b	single	2^{-149}	2^{-126}	$2^{+128} - 2^{+104}$	1.401×10^{-45}	1.175×10^{-38}	$3.403 \times 10^{+38}$
64b	double	2^{-1074}	2^{-1022}	$2^{+1024} - 2^{+971}$	4.941×10^{-324}	2.225×10^{-308}	$1.798 \times 10^{+308}$

*For normalized values, there is effectively +1 extra mantissa bit from the "1" most-significant bit that is assumed to be at the beginning of each mantissa, but is not actually stored. This "free" mantissa bit is lost for denormalized numbers, however.

As the floating point word size increases, the available range and precision increases. For the small 16-bit sizes, the *half* type offers a small range with reasonable precision, while the *bfloat16* type trades decreased precision for increased range, putting it on par with 32-bit single-precision ranges. The step size between representable values varies dramatically across the range, from the minimum denormalized amount at the lower end

of the range up to very large steps at the high end ($2^{+104} = 2.02 \times 10^{+31}$ for singles and $2^{+971} = 2.00 \times 10^{+292}$ for doubles, equal to the subtracted part of the maximum binary range in the table). Because the mantissa is a finite number of bits, rounding inevitably occurs, snapping to the steps between representable values. Errors introduced by rounding can gradually build up over time. As a result, many floating point conversion instructions (e.g., various varieties of `FRINT` and `CVT` instructions) explicitly allow the instruction to control how rounding occurs, while `FPCR` (Table 3.3: “FP Rounding, NaN Handling, and Denormal Handling Controls in FPCR”) controls rounding during calculations.

In addition to rounding errors, some calculations can potentially introduce unexpected precision errors, such as during the subtraction of two values that are almost identical (or, equivalently, addition of a positive and negative number with nearly identical absolute values). The two numbers can largely cancel each other out during the subtraction operation, potentially producing a very small result that has much less precision than either of the two inputs. Over the course of long sequences of calculations, rounding errors can gradually accumulate. *Fused* multiply-add/subtract operations typically do not round in between the multiply of two values and subsequent add/subtract with a third. Although skipping rounding usually results in more precise answers, the result can be somewhat different than those calculated using separate multiply and add/subtract operations with a rounding step in between. See Section 3.5.4, “Fused Op with Add/Accumulate Instructions (Integer and ASIMD and FP Types)” for additional notes.

Many algorithms employ a longer floating point word size during computation than they use for inputs and outputs. Generally, more precision (more mantissa bits) is needed to reduce the effects of intermediate rounding, with a final rounding step performed before storing the final data in a smaller size. Choosing the right intermediate computation word size is a complex trade-off between precision and computational bandwidth because larger word sizes have fewer elements in vectors. For example, a single input vector of half-precision input data is often converted to two input vectors of single precision before feeding two computational instructions.

Various rounding, NaN handling, and denormal handling controls are available in the Floating Point Control Register (FPCR). FPCR can be accessed directly through the `MSR/MRS` instructions or preferably through `fcntl` functions declared in `fcntl.h`. Note that settings other than the default are rarely needed beyond specific mathematical algorithms and circumstances.

Table 3.3. FP Rounding, NaN Handling, and Denormal Handling Controls in FPCR

Field Name	Bits	Required Feature	Definition
FIZ	0	FEAT_AFP	Flush inputs to zero. Controls whether single-precision, double-precision, and BFloat16 input operands that are denormalized numbers are flushed to zero.
AH	1	FEAT_AFP	Alternate handling. Controls alternate handling of denormalized floating-point numbers.
NEP	2	FEAT_AFP	Controls how the output elements other than the lowest element of the vector are determined for Advanced SIMD scalar instructions.
FZ16	19	FEAT_FP16 (available in all CPUs in this guide)	Flushing denormalized results to zero control bit on half-precision data-processing instructions.

Table 3.3. FP Rounding, NaN Handling, and Denormal Handling Controls in FPCR (cont.)

Field Name	Bits	Required Feature	Definition
RMode	23:22		Rounding mode control field.
FZ	24		Flushing denormalized results to zero control bit.
DN	25		Default NaN use for NaN propagation.
AHP	26		Alternative half-precision control bit.

See Arm documentation for more details on [FPCR](#).

3.1.2 Integer Data Types

As shown in [Table 3.4](#) below, Advanced SIMD supports integer element types with two additional smaller types than are supported in the General Purpose Registers. All sizes feature both unsigned and two's complement signed forms.

Table 3.4. Integer Types Supported in the Advanced SIMD and FP Unit

Data Word Size	Name	Unsigned Range			Signed Range			
		Min	Max (binary)	Max (decimal)	Min (binary)	Min (decimal)	Max (binary)	Max (decimal)
8b	byte	0	2 ⁸ - 1	255	-2 ⁷	-128	+2 ⁷ - 1	+127
16b	halfword	0	2 ¹⁶ - 1	65,535	-2 ¹⁵	-32,768	+2 ¹⁵ - 1	+32,767
32b	word	0	2 ³² - 1	4,294,967,295	-2 ³¹	-2,147,483,648	+2 ³¹ - 1	+2,147,483,647
64b	doubleword	0	2 ⁶⁴ - 1	1.8447x10 ¹⁹	-2 ⁶³	-9.2234x10 ¹⁸	+2 ⁶³ - 1	+9.2234x10 ¹⁸

Operations normally do not require rounding, so results are exact. Addition operations require an additional output bit to capture a full result (i.e., adding decimal 255 as an unsigned 8b value with itself results in a value of 510, which requires 9 bits to encode). Multiplication operations require twice the number of output bits to capture the full result (i.e., multiplying decimal 255 as an unsigned 8b value with itself results in a value of 65025, which requires 16 bits to encode).

Most ASIMD integer multiplies come in both standard versions (such as `MUL`) that only return the lower n bits of the result and versions that return the full lengthened result (such as `SMULL/UMULL`) with all $2n$ result bits in a data type that is twice the size. Accumulating instructions add to an accumulator that is already either n (for `MLA` and similar) or $2n$ (for `SMLAL/UMLAL` and similar) bits long. Choose the appropriate version on a multiply-by-multiply basis, depending upon whether the high-order bits are required to provide sufficient range. Repeated addition/accumulation and multiplication can grow the maximum range of the overall result by many bits.

Unlike with floating point numbers, there are no special encodings such as NaNs or $\pm\infty$ to indicate that an overflow has occurred during a calculation. When a calculation result overflows the available range, the result typically wraps around to the other end of the range and continues. This sudden and discontinuous change between the maximum and minimum representable values can cause problems with applications like signal processing. ASIMD offers saturating variants of some instructions that clamp overflowing

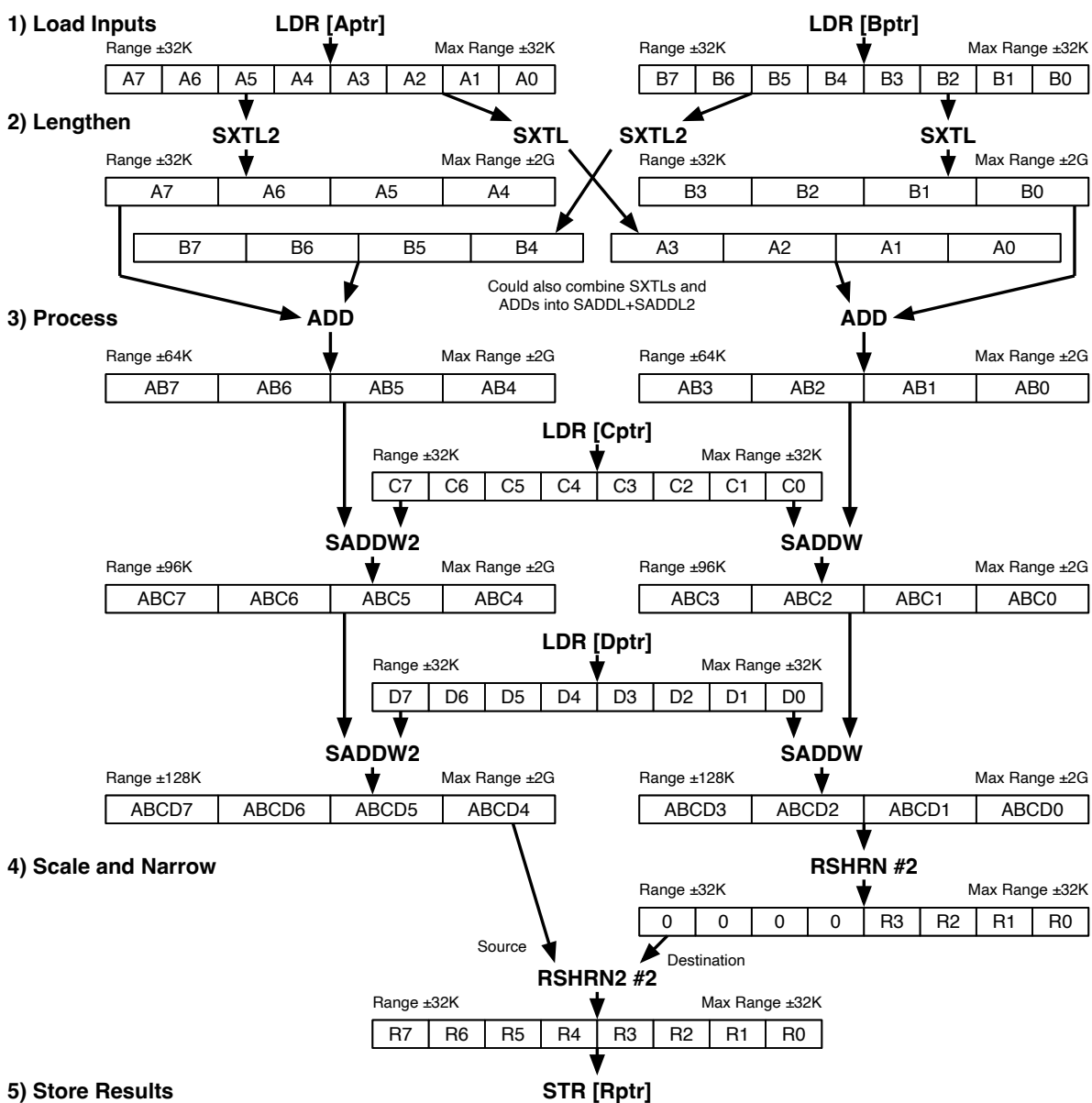
output results at either end of the range instead of letting them wrap around. For example, byte-size unsigned integers, which can represent values 0–255, are commonly used to represent pixel values in image processing. Normal wrap around math that adds values 200 and 60 together exceeds the maximum representable value and wraps around to a result of 4. A saturating add, in contrast, returns the maximum value of 255 instead. In this way, two bright (high-value) pixels can add together without overflowing and wrapping around back to a black (low-value) pixel. Note that all saturating operations *must* be explicitly signed or unsigned because the minimum and maximum saturation points are different.

When the instruction scales down the result to fit into a smaller data type, low-order bits can be lost and the result is truncated, such as with `SHRN`. This is equivalent to a round towards zero policy. Unfortunately, this can cause numerical problems with some algorithms because it inherently rounds down in all cases. As a result, the ISA offers variations that round toward the nearest value where ties round toward $+\infty$, such as `RSHRN`. Using this option is recommended in most cases where it is available because it preserves better numerical accuracy than the non-rounding variations.

Selecting the appropriate intermediate computation data size is a balance between managing potential overflow and element parallelism. For example, ASIMD operations can work with sixteen 8-bit integers at once, but only eight 16-bit integers or just four 32-bit integers per operation.

[Figure 3.2: “ASIMD Integer Processing Example”](#) below shows an example of vector operation $R = (A+B+C+D)/4$, using integers, and including range extension (by lengthening) to prevent overflow from occurring, followed by final scaling at the end. Each vector lists actual possible ranges for values (left) and the ranges that the data types can hold (right).

Figure 3.2. ASIMD Integer Processing Example



- 1. Load Inputs:** Load instructions bring in data from memory as a small type, such as 8b or 16b.
- 2. Lengthen:** Instructions such as `SXTL/UXTL` (signed/unsigned extend-and-lengthen) lengthen these narrow types into larger types that have enough range to avoid overflow during computations. Each pass of lengthening instructions doubles the bit width of every element in the vectors. A sequence of instructions may lengthen multiple times if larger data types are needed, such as 8-bit to 32-bit. Note that some operations perform input lengthening as part of the instruction, such as `SADDL/UADDL` (both inputs are half the size of the output) and `SADDW/UADDW` (only one input is half the size of the output).
- 3. Process:** A sequence of instructions perform the desired mathematical function (a single `ADD` in the example) using the full available range of the larger data type.

- Scale and Narrow:** In preparation for storage, the larger type used for computation is reduced down to the size of the storage type. In ASIMD, the inverse of lengthening operations are narrowing operations; each pass halves the bit width of the data elements in the vectors, keeping only the low-order bits. In the simplest case, the extra high-order bits can just be discarded with an `XTN` instruction (extract-and-narrow). However, often the values accumulated in a larger intermediate format need to be first scaled down to fit into the smaller format. This is generally performed using a variety of `SHRN` (shift right-and-narrow) instructions to both shift the result right (effectively dividing each element by a power of two) and halve output element storage size. Note that there are many varieties of `SHRN` available to control how the instruction handles bits lost both at the low-order end (due to the shifting) and at the high-order end (due to the narrowing, which reduces the representable range).
- Store Results:** Store instructions return the narrowed results to memory, often as the same data type as the input.

3.2 Arm Advanced SIMD and FP Registers

Regardless of element data type, all use the same set of thirty two 128-bit vector registers. These can be used as either scalar values, 64-bit short vectors, or 128-bit full-size vectors, as is depicted in [Table 3.5: “Vector Register Layouts”](#). In assembly language, vector registers are specified as `V<register number>.<number of elements><element size letter>`, while scalar registers consisting of only a single data element are specified as `<element size letter><register number>`. The possible combinations are listed in [Table 3.6: “Vector Register Interpretation”](#). Individual elements (or sometimes “lanes”) within a vector can be selected by some instructions, and are encoded as vector registers with an additional [`<lane number>`] specifier at the end. For example

- v20.4s:** vector register #20, interpreted as a 128-bit vector of four 32-bit words
- v13.8b:** vector register #13, interpreted as a 64-bit vector of 8 bytes
- s20:** a scalar 32-bit word in vector register #20.

This scalar version happens to be equivalent to `v20.4s[0]`, the first element of the same register when interpreted as a vector. When using only a scalar or 64-bit subset of the entire register, the low-order 8, 16, 32, or 64 bits are always active while the high-order bits of any destination registers are usually zeroed. In addition, some Advanced SIMD instructions allow direct access to the `w` or `x` integer registers in order to allow transfers between the separate register files. However, use of these instructions should always be minimized, because these cross-unit accesses are slower than normal register accesses.

Table 3.5. Vector Register Layouts

Element Size	Element Number within Vector															
	white = scalar, white + light gray = 64b vectors, all cells = 128b vectors															
Byte→	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
8b	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
16b	7		6		5		4		3		2		1		0	
32b	3				2				1				0			
64b	1								0							

Table 3.6. Vector Register Interpretation

Element Size	Assembly Notation			Intrinsic Notation				
	Scalar	64b Vector	128b Vector	Signed Integer	Unsigned Integer	Galois Polynomial	Float	Brain Float
8b	Bn	Vn.8B	Vn.16B	_s8	_u8	_p8	N/A	N/A
16b	Hn	Vn.4H	Vn.8H	_s16	_u16	_p16	_f16	_bf16
32b	Sn	Vn.2S	Vn.4S	_s32	_u32	_p32	_f32	N/A
64b	Dn	← Use Scalar	Vn.2D	_s64	_u64	_p64	_f64	N/A

The interpretation of the Advanced SIMD and FP registers is controlled strictly by the choice of instructions and not by anything inherent in the registers themselves. Note that this means that a wide number of "untyped" instructions work well for all data types. Loads, stores, byte transposition, shuffle operations, and bitwise operations like masking all work equally well with elements of any type. Also, no matter the type of data contained in a register, it may be freely used by any other instruction. Although generally not a good idea with mathematical operations, this can advantageous with some kinds of shuffles, for example.

C language intrinsics work with explicitly typed vectors and use a somewhat different naming scheme from assembly that emphasizes the type and size of the data used by each operation, encoded using the notations listed in the right-hand half of [Table 3.6: "Vector Register Interpretation"](#). Explicit casts are needed to shift from one type to another, and if necessary, vector lengths are encoded with additional suffix letters, most commonly `q` for 128-bit vectors (as "quadwords").

3.3 Overview of Arm Advanced SIMD Mnemonics

Advanced SIMD instruction mnemonics are constructed in a systematic manner from a modest list of prefixes and suffixes in conjunction with roughly 60 base instruction mnemonics. Consider `SQRDMLAH`, a common fixed-point instruction:

- `s` = signed
- `Q` = saturating
- `R` = rounding
- `D` = doubling
- `MLA` = multiply-accumulate instruction
- `H` = high half of result

Most Arm Advanced SIMD instruction mnemonics are constructed in a similar manner. The Arm Advanced SIMD instruction prefixes are summarized in [Table 3.7: "Advanced SIMD Instruction Mnemonic Prefixes"](#), base instructions in [Table 3.8: "Advanced SIMD Base Instructions Mnemonics Grouped By General Function"](#), and suffixes in [Table 3.9: "Advanced SIMD Instruction Mnemonic Suffixes"](#). Details are further described in subsequent sections.

Table 3.7. Advanced SIMD Instruction Mnemonic Prefixes

Field Name	Coding	Type	Description
Data Type	(none) / BF / F / S / U	All	Selects between brain floating point (bfloat16), general floating point, signed integer / fixed point, and unsigned integer / fixed point operation types, if needed
Modulo / Saturating	(none) / Q	Integer and fixed point	Controls how operation handles overflow at high-order end (most significant bits)
Truncating / Rounding	(none) / R	All	Controls how operation handles low-order bits (least significant bits) that are lost
Doubling / Halving	D / H	Fixed point	Multiplies the result x2 (D), to align fixed-point binary points, or x0.5 (H), returning an arithmetic mean instead of a sum
Polynomial	P	Galois polynomial	Selects Galois field multiplication
Negated	N	Floating point	Negates the result
Complex Number Arithmetic	C	Floating point	Modifies operations to work with complex numbers arranged as interleaved (real, imaginary) values

In the middle of each mnemonic is the abbreviated name of the actual operation being performed by the instruction. Most are three letters long. About half of the instructions do not use prefixes or suffixes at all and only come in a single form, but a few more common operations such as ADD, SUB, and SHR come in 20–40 variations each.

Table 3.8. Advanced SIMD Base Instructions Mnemonics Grouped By General Function

	Group	Instructions
Mathematical Operations	Sign Control	ABS, NEG
	Arithmetic	ABA, ABD, ADA, ADD, SUB
	Multiply	DOT, MADD, MLA, MLS, MSUB, MUL
	Divide/Sqrt	DIV, RECP, RSQRT
	Comparison	AC<cond>, CCMP, CM<cond>, CMP, CMTST, CSEL, MAX, MIN
	Conversion	CVT, INT
Bitwise Operations	Shift	SHL, SHR, SLI, SRA, SRI, XT
	Logic	AND, BIC, EOR, NOT, ORN, ORR
	Masking	BIF, BIT, BSL
	Cryptography	AES, BCAX, RAX, SHA, XAR
	Misc. Bitwise	CLS, CLZ, CNT, RBIT
Element Operations	Move	DUP, INS, MOV, MVN
	Byte Shuffle	EXT, REV, TBL, TBX, TRN, UZP, ZIP
Memory Operations	Load	LD<n>, LD<n>R, LDP, LDNP, LDR, LDUR
	Store	ST<n>, STP, STNP, STR, STUR

Table 3.9. Advanced SIMD Instruction Mnemonic Suffixes

Field Name	Coding	Types	Description
Cryptographic	Various	N/A	Various AES / SHA / etc. cryptographic instruction specifications
Load / Store Interleaving and Replicating	1 / 2 / 3 / 4 / R	Load and store	Selects how elements are interleaved / replicated as they are loaded from memory
Load / Store Non-Temporal	N	Load and store	For loads and stores, provides a hint to the memory system that the access is non-temporal
Immediate	I	All	For the MOV and MVN instructions, selects the source value from an immediate operand
Comparison Conditions	EQ / GE / GT / HI / HS / LE / LT	All	For compare-and-mask and conditional instructions, chooses a value comparison mode
Special Value Handling Mode	E / NM / X	Floating point	For specific instructions, controls how operation handles NaN and $\pm\infty$ values
FP Rounding	A / I / M / N / P / X / Z	Floating point	For CVT and INT instructions, selects a floating point rounding mode
Output Data Type	(none) / F / S / U	All	Selects an output data type if it is different from the input
Return High Half	H	Typically fixed point	Indicates that the instruction result contains only the high-order half (most significant bits)
Narrow / Long / Wide	N / L / W	All	Indicates that the instruction result elements have smaller or larger bit width than each source element
Lower / Upper Half	(none) / 1 / 2	All	Selects an operation half for instructions that need double-wide input or produce double-wide output
Paired or Horizontal	P / V	All	Indicates operations that work "horizontally" across vectors instead of in lanes
Bottom / Top	B / T	Brain floating point (bfloat16)	For lengthening instructions, selects only odd or only even input elements

3.3.1 ASIMD and FP Overall Instruction Data Type: (none) / BF / F / S / U

Most Advanced SIMD instruction mnemonics start with one of four code letters/sequences, indicating the operation's data class.

- **BF-class:** instructions operate on brain floating point (bfloat16) values
- **F-class:** instructions operate on general floating point values
- **s-class:** instructions operate on signed integer or fixed-point values
- **u-class:** instructions operate on unsigned integer or fixed-point values

Mathematical operation instructions without one of these three classification letters work correctly with both signed or unsigned integer or fixed-point values, but not floating point values.

Non-mathematical operations like loads, stores, bitwise logic/masking, and permutation instructions also do not have a class indicator, and can be used with any data type.

These class distinctions are required based on certain features of various instructions. For example, an `ADD` in modulo arithmetic works correctly with both unsigned and 2's complement signed numbers. However, any `ADD` with saturating arithmetic works differently with unsigned and signed values, because the minimum/maximum saturation values are different. Hence, these instructions always come in separate `s` and `u` versions. The `SUQADD` instruction is a unique mixed-mode case that has one signed input (the destination accumulator) and one unsigned input.

3.3.2 ASIMD Modulo or Saturating Arithmetic: Q

By default, integer Advanced SIMD instructions compute using modulo arithmetic, so any results that overflow "wrap around" from one end of the representable range of values to the other. This is the normal type of math used in all conventional integer Arm operations. Operations with the `q` prefix, however, use saturating math which clamps any overflowing results to the maximum or minimum value instead of wrapping around. Note that all instructions with the `q` prefix must also have an `s` or `u` prefix, since the maximum and minimum values for signed and unsigned numbers are different.

3.3.3 ASIMD Truncating or Rounding: R

By default, operations that lose bits at the right low-order end (least significant bits) of each data word, such as right shifts, simply truncate (drop) the lost bits. However, instructions that specify the `R` rounding prefix instead round up if the last bit to be shifted out — the one effectively at the "minus one" bit position — is a 1 value. Rounding has the same effect as truncation if the last bit shifted out is a 0 value. This behavior is desirable for many signal processing algorithms, as it avoids biasing the results of long calculations toward zero.

3.3.4 ASIMD Doubling or Halving: D / H

The `D` prefix indicates that the integer multiply-accumulate operation multiplies the multiplier output by an additional 2x factor before accumulating, doubling the multiply result. This makes these multiply-accumulate operations compatible with the signed, fixed-point arithmetic used frequently in signal processing that uses a range of $[-1$ to $+1)$. Multiplications of two of these signed integers results in a value twice as wide, but with two "sign" bits at the high end. Doubling the result with a 1-bit left shift corrects this to a number with effectively with one "sign" bit again. These instructions should generally never be used with simple integers.

The `H` prefix indicates that the result of an addition is divided by two before the result is returned. This effectively turns the addition into an "average" operation, which produces the arithmetic mean of the two inputs and cannot ever overflow. Unlike the doubling operations, these operations are useful with both integer and fixed-point types.

3.3.5 ASIMD Polynomial: P

The `P` varieties of multiply instructions perform Galois field polynomial multiplication operations instead of standard ones. These are primarily for use in cryptography.

3.3.6 FP Negated: N

The floating point multiplication operations with the `N` prefix negate their results before returning the value. With this instruction, many explicit `FNEG` instructions can be eliminated when multiplying-and-adding long sequences of terms.

3.3.7 ASIMD Complex Number Arithmetic: C

The floating point addition and multiplication instructions with the `C` prefix are designed to perform mathematical operations on vectors of complex numbers organized as one double or two float pairs of interleaved (real, imaginary) values. Each instruction has an immediate value that specifies the positive or negative orientation of each multiply. See [Section 3.5.5, "Complex Number Instructions"](#) for a description of how these instructions can be used to efficiently perform complex multiplications.

3.3.8 ASIMD Cryptographic: Various

The cryptographic instructions have a large number of customized extensions. Unlike with most extensions, these are not special modes, but actually a large number of highly specialized and unique instructions that are customized for use in specific uses within hand-written cryptographic code. Consult the Arm instruction reference for further information.

3.3.9 ASIMD Load/Store by Element Interleaving and Replicating: 1 / 2 / 3 / 4 / R

A `1 / 2 / 3 / 4` notation is used at the end of `LD<n>` (or middle of `LD<n>R`) instructions to indicate the interleave factor for the elements in memory. (This is the only possible suffix for load/store operations, so they cannot be confused with the other use of `1 / 2` in [Section 3.3.17, "ASIMD Lower-Half / Upper-Half: \(none\) / 1 / 2"](#).) These load and store instructions are used to load interleaved data from memory and de-interleave it as it moves into registers without using separate `UZP` instructions, and then to re-interleave the data as it is stored without extra `ZIP` instructions. The number indicates the number of fields that are in each data element.

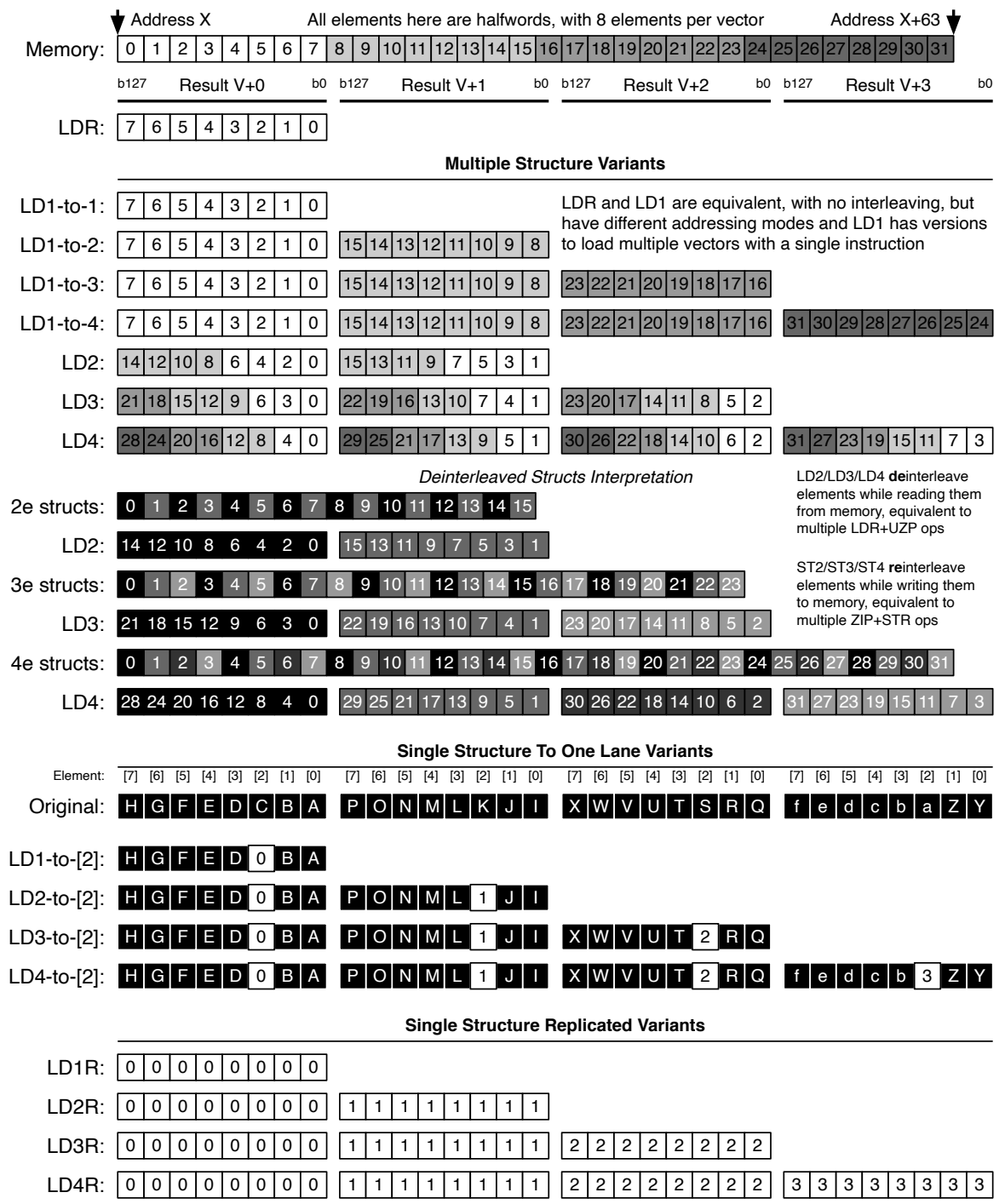
For example, when loading an array of pixel values consisting of interleaved RGB values, a `LD3` instruction can be used to de-interleave these values and bring them in as three separate all-R, all-G, and all-B vectors. A similarly packed array of RGBA pixels (RGB with an alpha channel) would instead be de-interleaved using a `LD4` instruction into separate all-R, all-G, all-B, and all-alpha vectors. Note that `LD1` instructions do not actually perform any de-interleaving, so they are just normal vector loads with different addressing modes. For a code example that uses `LD3` and `ST3` intrinsic functions, see [Section 3.4.4.9, "Load/Store Operations"](#).

`ST<n>` instructions perform the reverse operations, interleaving packed vectors as their contents are written out to memory in interleaved form.

[Figure 3.3: "Advanced SIMD Load/Store Element Interleaving and De-Interleaving"](#) illustrates the interleaving patterns that these instructions perform. Note that these instructions are often microcoded and require additional resources compared with a simple load or store (see [Section 5.4.8, "Multi-μop Instructions"](#)). However, these instructions are

considerably more efficient than equivalent sequences of simpler instructions. Use them when interleaving/de-interleaving long vectors, but do not use them if operating on only a few elements.

Figure 3.3. Advanced SIMD Load/Store Element Interleaving and De-Interleaving



Recommendation: Leverage LD<n> and ST<n> instructions for interleaved memory structures.

[Magnitude: Medium | Applicability: Low] The LD<n> and ST<n> instructions provide methods for interleaving and de-interleaving of memory data. Use them if operating on more than a few elements. Rather, consider a sequence of simple loads, stores, and various logical instructions when only operating on a few elements.

3.3.10 ASIMD Create Immediate: I

This extension of MOV and MVN synthesizes a new vector from an immediate value and loads it into every element lane of the destination vector. Several shift variants are available for distributing the small immediate value into the element. Most are fairly straightforward placements of 8-bit immediate values into ranges of each element specified by the LSL shift indication. However, the MSL shift variants pack in 1 bits below the immediate value to allow any partial masking of 32-bit values to be generated with a single instruction. Note that a MOV with MSL #24 can be generated with a MVN with LSL #24. Similarly, the 64-bit version is designed for generating arbitrary byte-by-byte masks of various kinds. In the following table, assume that the 8-bit immediate value is abcdefgh.

Table 3.10. Advanced SIMD Move Immediate

Element Size	Name	Binary Interpretation With 8-bit Immediate Value abcdefgh
8b	#imm	abcdefgh
16b	#imm{, LSL #0}	00000000 abcdefgh
	#imm, LSL #8	abcdefgh 00000000
32b	#imm{, LSL #0}	00000000 00000000 00000000 abcdefgh
	#imm, LSL #8	00000000 00000000 abcdefgh 00000000
	#imm, LSL #16	00000000 abcdefgh 00000000 00000000
	#imm, LSL #24	abcdefgh 00000000 00000000 00000000
	#imm, MSL #8	00000000 00000000 abcdefgh 11111111
	#imm, MSL #16	00000000 abcdefgh 11111111 11111111
64b	#imm	aaaaaaaa bbbbbbbb cccccccc dddddddd eeeeeeee ffffffff gggggggg hhhhhhhh

Other instructions obtain source values from immediate values such as ORR and BIC, but do not use the I suffix.

Floating point FMOV instructions with immediate values do not use the I suffix, but also have a unique encoding. Instead of being interpreted directly, the 8-bit immediate value is treated as a compact FP format with 1 sign bit, 3 exponent bits, and 4 mantissa bits. These compact floating point values offer a range anywhere from 0.125–31.0, using an exponent range of 2⁻³ to 2⁺⁴, with both positive and negative values representable. In the upper half of this range, the small mantissa limits precision to just an integer-by-integer level, while at the low end much smaller steps of 2⁻⁷ between individual values are possible. This

scheme allows a wide variety of common small constants to be encoded with a single `FMov` instruction. Zero is not representable using this simple scheme, but `FMovI` of `#0.0` can be created by aliasing it to an integer `MOV` of `#0` instead. See "Modified immediate constants in A64 instructions" in the Arm Architecture Reference Manual for additional details.

3.3.11 ASIMD Comparison Conditionals: EQ / GE / GT / HI / HS / LE / LT

The `CM`, `FAC`, and `FCM` instructions create masks in vector registers based on per element comparison operations. The available instruction-comparison combinations are listed in [Table 3.11: "Advanced SIMD Comparison Conditionals"](#). Note that not all conditions can be used with all instructions. For example, "less than" conditions are not available on `CM` instructions that compare two vector registers, because the operand order can be reversed and the "greater than or equal" condition be used. However, all conditions that compare against an immediate value of `#0` are available. For a full list of the condition codes across the Arm ISA, see [Section 2.9, "Condition Codes"](#).

Note that comparison instructions that set flags, like `FCMP`, do not need a condition suffix because consumer instructions includes a condition code that is evaluated against the flags.

Table 3.11. Advanced SIMD Comparison Conditionals

Mnemonic Code	Name	Interpretation	Instructions
EQ	Equal	$A == B, A == \#0$	<code>CM<cond>, FCM<cond></code>
GE	Greater Than or Equal	$A \geq B, A \geq \#0$	<code>CM<cond>, FAC<cond>, FCM<cond></code>
GT	Greater Than	$A > B, A > \#0$	<code>CM<cond>, FAC<cond>, FCM<cond></code>
HI	Higher Than	unsigned $A > B$	<code>CM<cond></code>
HS	Higher Than or Same	unsigned $A \geq B$	<code>CM<cond></code>
LE	Less Than or Equal	$A \leq \#0$	<code>CM<cond>, FCM<cond></code>
		$A \leq B$	All aliased to GT with swapped operands (may not be available on all assemblers): <code>CM<cond>, FAC<cond>, FCM<cond></code>
LT	Less Than	$A < \#0$	<code>CM<cond>, FCM<cond></code>
		$A < B$	All aliased to GE with swapped operands (may not be available on all assemblers): <code>CM<cond>, FAC<cond>, FCM<cond></code>

3.3.12 FP Special Value Control Suffixes: E / NM / X

Three suffixes trigger special handling of IEEE 754 Not-a-Number (NaN) and Infinity values:

- `FCMP` and `FCCMP` instructions with the `E` suffix trigger *exceptions* when comparisons are made with a NaN value, instead of quietly selecting a default result. A signal handler must be registered to handle these exceptions.
- `FMAX` and `FMIN` instruction variants normally use a mode where NaN values always "win" the maximum/minimum comparison and are propagated to the output, so errors

are evident. The **NM** variants use IEEE NaN handling instead, always selecting any actual numerical value over NaN value in any particular lane. In this mode, NaN values are propagated to the result *only* if *both* inputs are NaN values, and can disappear otherwise.

- **FMUL** instructions have a special **x** variant that produces a value of 2.0 for any occurrences of zero times infinity, instead of just zero. The result is negative if only one of the values is negative, otherwise the result is positive. This represents a common limit case for some very specific mathematical operations. Unrelated, the **x** suffix on the **FRECPX** instruction represents "exponent".

These three suffixes are seldom used in general-purpose computations.

3.3.13 FP Rounding Mode: A / I / M / N / P / X / Z

Both **CVT** and **INT** instructions use a variety of suffixes to select different rounding modes for FP-to-integer or FP-to-fixed-point conversions, which often have a fractional part that needs to be rounded off. Variations include round towards: nearest, zero, $+\infty$, $-\infty$, and others. These variants are usually used to determine the range of possible rounding error that may be introduced by conversion operation. See Arm Architecture Reference Manual for full descriptions and options.

There are also two special **INT** instruction rounding modes:

- **I**: The instruction uses the current default **FPSCR** rounding mode. This is also the default option for **CVT** operations that have no explicit rounding mode suffix.
- **X**: The instruction uses the "eXact" mode that triggers an exception when any rounding occurs. Use this mode to guarantee that only integers are being converted. A signal handler must be registered prior to the use of these instructions to catch the resulting exceptions.

Do not confuse the rounding **N** suffix with the narrowing **N** suffix, because both can be used with different kinds of **CVT** instructions. Rounding **N** always includes an output type specifier from [Section 3.3.14, "ASIMD/FP Different Output Type: \(none\) / F / S / U"](#) just after the **N** suffix.

3.3.14 ASIMD/FP Different Output Type: (none) / F / S / U

Most instructions output the same type as the input, from the type prefix described in [Section 3.3.1, "ASIMD and FP Overall Instruction Data Type: \(none\) / BF / F / S / U"](#). However, most **CVT** instructions use explicit type suffixes to indicate the different destination type for the conversion (e.g., **F** for floating point, **S** for signed integer, or **U** for unsigned integer). There are also a few signed-to-unsigned integer/fixed-point **SHR** and **XT** instructions that can produce an unsigned output from a signed input, stripping the sign bit off of the output in order to get an extra bit of range. These are indicated with a **U** suffix.

3.3.15 ASIMD Return High Half: H

Eligible base instructions are either arithmetic instructions (such as multiplies) that can produce results with ranges larger than the input type, or narrowing (precision reduction) operations. Without the **H** suffix, most eligible base instructions return the low-order half of the 2x-wide result and throw out the high bits (or saturate at the maximum representable

value, for `q`-prefix saturating instructions). Instructions with the `H` suffix, however, return the high-order half of these 2x-wide results.

A pair of instructions (one `H` version and one non-`H` version) can be used to obtain the full, 2x-wide operation results. The high-order half of each result is written into one vector, while the low-order half is written into a second. Although this is feasible for some scenarios, usually pairs of `L` suffix lengthening operations (see [Section 3.3.16, "ASIMD Narrow, Long, Wide: N / L / W"](#)) are a better choice. More commonly, however, these versions are used with signal processing algorithms to get the high-order half of fixed-precision results when the low-order bits can be truncated or rounded (the latter using the `R`-prefix).

3.3.16 ASIMD Narrow, Long, Wide: N / L / W

By default, Advanced SIMD operations have the same data type for the input and output. These three suffixes are used to select different bit widths for the input and output:

- **Narrow (`n`):** Destination is half the size of the source type(s): 64→32 bit (`D`→`S`), 32→16 bit (`S`→`H`), and 16→8 bit (`H`→`B`) conversions
- **Long (`L`):** Destination is double the size of the source type(s): 8→16 bit (`B`→`H`), 16→32 bit (`H`→`S`), and 32→64 bit (`S`→`D`) conversions
- **Wide (`w`):** Destination and one source are double the size of the second source: 16+8→16 bit (`H`+`B`→`H`), 32+16→32 (`S`+`H`→`S`), 64+32→64 bit (`D`+`S`→`D`) conversions

These suffixes most commonly apply to integer and logical instructions, but they do apply to a small set of floating-point instructions, primarily converts. With these three modes, integer elements can be converted from narrower, lower-range/precision formats to wider, higher-range/precision formats and back. Most frequently, lengthening or widening versions are used to increase the range or precision (or both) of data values as they come into registers, in order to avoid overflow and rounding during long sequences of calculations. Narrowing versions are then used to pack the results back down to smaller elements before they are stored to memory. See [Section 3.1, "Advanced SIMD and FP Data Types"](#) for details on element data types and their value ranges.

Narrowing versions are used to decrease the range or precision (or both) of data values, such as through saturation or rounding, as the data values must fit into a smaller number of bits. They also sometimes come in high-half versions (see [Section 3.3.15, "ASIMD Return High Half: H"](#)) that return the high half of the result instead of the low half.

These options are always used with either lower/upper-half or paired/horizontal suffix options, described in [Section 3.3.17, "ASIMD Lower-Half / Upper-Half: \(none\) / 1 / 2 "](#) and [Section 3.3.18, "ASIMD Cross-Lane Paired or Horizontal: P / V"](#), in order to specify how lanes from different vectors should be combined.

For instance, consider `RADDHN` and `RADDHN2`, Rounding Add returning High Narrow. These instructions add each vector element in the first source Advanced SIMD and FP register to the corresponding vector element in the second source Advanced SIMD and FP register, and place the most significant half of the result into a vector. `RADDHN` writes the vector to the lower half of the destination Advanced SIMD and FP register, and `RADDHN2` writes the upper half of the destination Advanced SIMD and FP register.

Recommendation: Use mathematical instructions that include type conversions.

[Magnitude: Medium | Applicability: Medium] Instead of using explicit extension instructions, such as `sxtl`, on inputs prior to a calculation that results in a larger-type size, use instructions that include the conversion in with the calculation, such as `saddl` and `saddl2`.

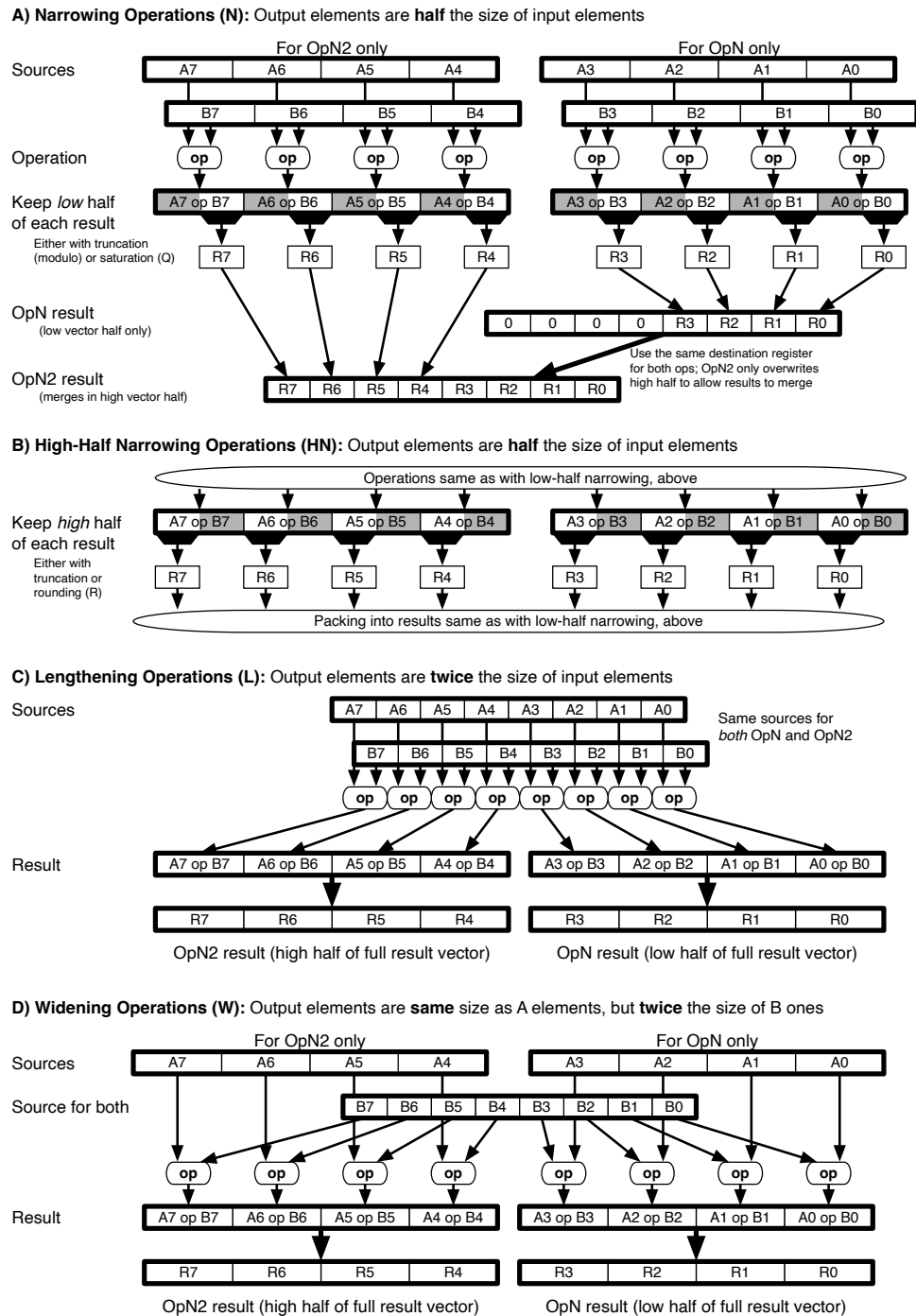
3.3.17 ASIMD Lower-Half / Upper-Half: (none) / 1 / 2

Some Advanced SIMD instructions inherently produce destination vectors that are half or twice the size as the source vectors. The subsequent numerical suffix controls which half of two-part sources or destinations are consumed or produced by any particular instruction. The `N` / `L` / `W` suffixes produce a result that is 2x wider than the inputs, or else require 2x wider sources, such as:

- **Narrow(`n`) + (none)**: Writes the narrowed result to lower half of the destination register, and clears the upper half. For example: `SQRSHRN`.
- **Narrow(`n`) + 2**: Writes the narrowed result to upper half of the destination register, while maintaining the previously-computed lower half in the destination register. For example: `SQRSHRN2`.
- **Long(`L`) + (none)**: Selects the lower half of both sources. For example: `SQDMLAL`.
- **Long(`L`) + 2**: Selects the upper half of both sources. For example: `SQDMLA`.
- **Wide(`w`) + (none)**: Selects the lower half of second source only. For example: `SSUBW`.
- **Wide(`w`) + 2**: Selects the upper half of second source only. For example: `SSUBW2`.

Using the number suffix, pairs of source or destination vectors can be processed in an unambiguous manner, according to the diagrams in [Figure 3.4: “Advanced SIMD Narrow, Lengthen, and Widen”](#).

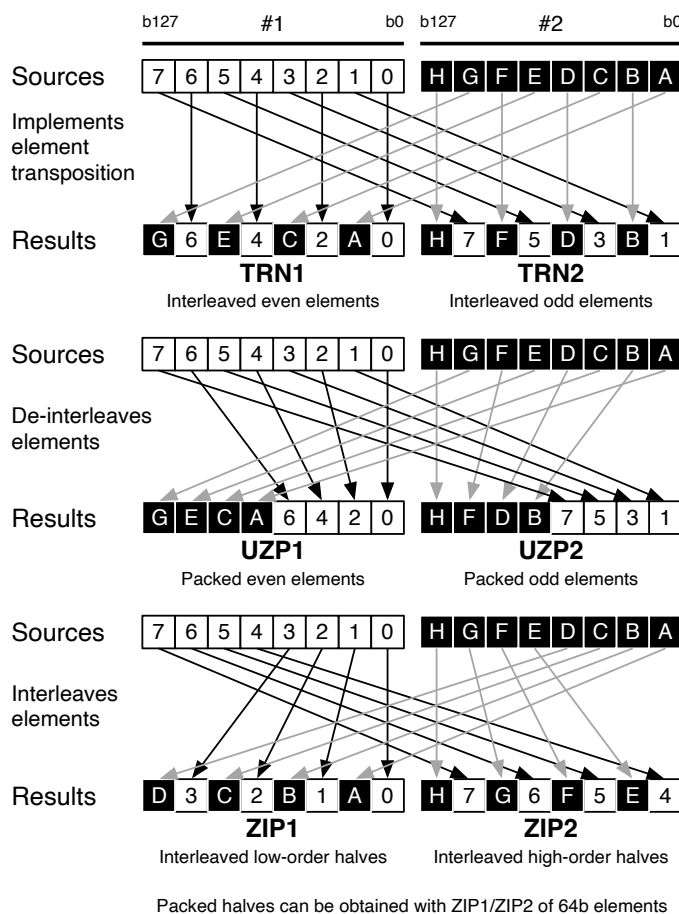
Figure 3.4. Advanced SIMD Narrow, Lengthen, and Widen



Similarly, the 1 / 2 notation is used by the TRN, UZP, and ZIP transformation operations to choose the "primary" lower half or "secondary" upper half of the resulting merger of input words, as depicted in Figure 3.5: "Advanced SIMD Byte Transformations". An explicit 1 notation is required for these instructions; assemblers do not assume that for you. Section 3.5.7, "Shuffle and Permute Instructions" illustrates this idea further and shows how different instruction variants can be used to generate a wide variety of

transformations. Note that while these permutation instructions are often used in 1 / 2 pairs, there are many situations when using only one or the other might still be useful.

Figure 3.5. Advanced SIMD Byte Transformations



Packed halves can be obtained with ZIP1/ZIP2 of 64b elements

3.3.18 ASIMD Cross-Lane Paired or Horizontal: P / V

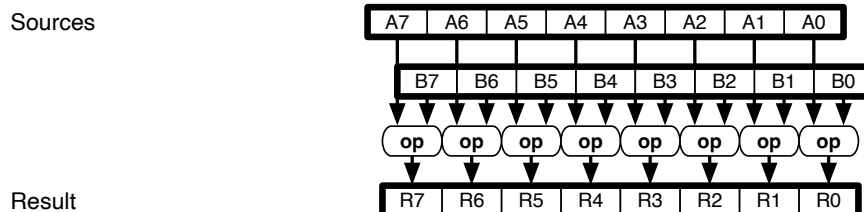
Advanced SIMD has two different types of horizontal reduction operations, which combine results in different vector lanes together. The *p*-suffix *paired* versions combine adjacent vector lanes across two vectors together to produce a single vector as a result. [Figure 3.6: “Advanced SIMD Cross Lane Paired and Horizontal Instructions”](#) (B) illustrates a paired instruction. Trees of repeated *paired* operations gradually combine all lanes of input down to result element #0. These versions are available for any data type. Meanwhile, the powerful *v*-suffix *across vector* versions compute a sum, minimum, or maximum across *all* elements of a vector with a single instruction, as is depicted in [Figure 3.6: “Advanced SIMD Cross Lane Paired and Horizontal Instructions”](#) (C). For complexity reasons, these are not available for all data types, particularly FP types. They can be synthesized with repeated *p*-suffix operations if necessary, however, as is described in [Section 3.5.3, “Horizontal Arithmetic and Test Instructions”](#). For comparison, [Figure 3.6: “Advanced SIMD Cross Lane Paired and Horizontal Instructions”](#) (A) depicts the data flow through normal operations *within* vector lanes.

These suffixes can also be combined with the $\mathbb{L}$ (long) suffix from [Section 3.3.16, “ASIMD Narrow, Long, Wide: N / L / W”](#) to create paired or cross-vector operations that produce

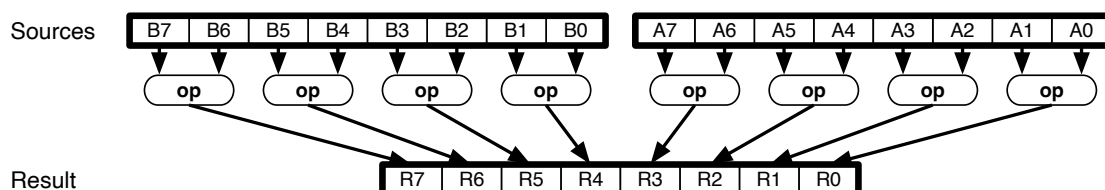
elements twice as long as the original values, as depicted in [Figure 3.6: "Advanced SIMD Cross Lane Paired and Horizontal Instructions"](#) (D-E). These can prevent overflow from occurring, if it might be a concern.

Figure 3.6. Advanced SIMD Cross Lane Paired and Horizontal Instructions

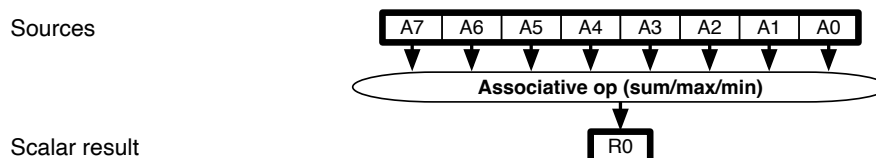
A) Normal Lane-Based Operation



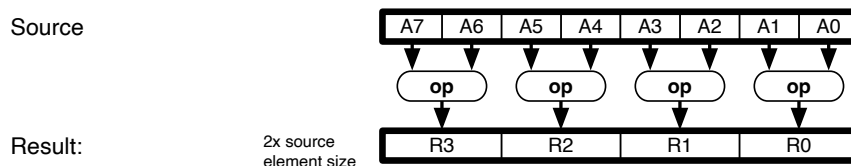
B) Paired Operation (P)



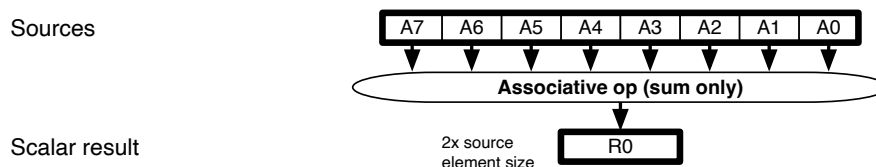
C) Horizontal Operation (V)



D) Long Paired Operation (LP)



E) Long Horizontal Operation (LV)



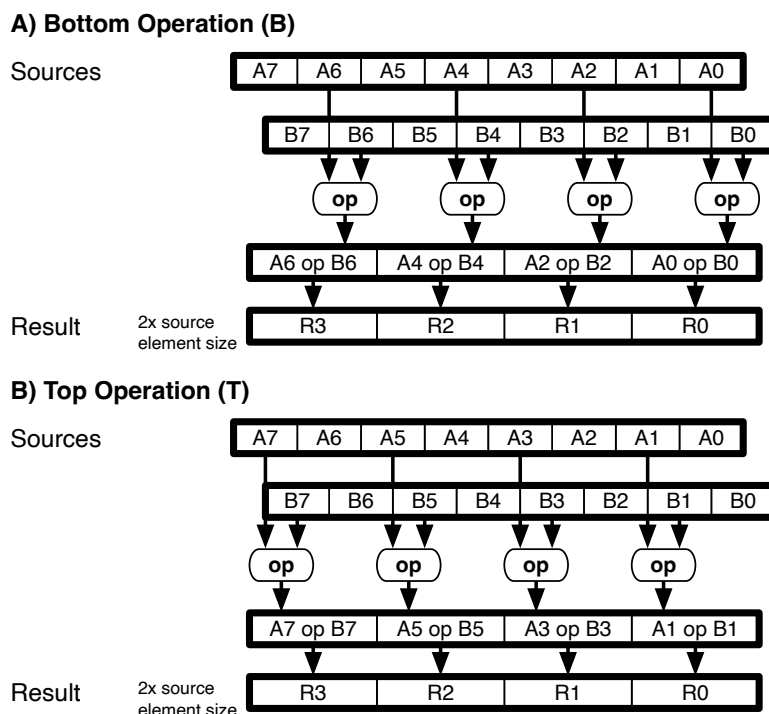
3.3.19 ASIMD Bottom or Top Half of Element Pairs: B / T

Most operations involving destination elements twice as wide as the source elements are encoded as lengthening operations using the lower or upper half of the entire vector, as described in [Section 3.3.17, “ASIMD Lower-Half / Upper-Half: \(none\) / 1 / 2”](#). This approach works well for algorithms where all of the output results of the resulting double-length vector are needed, with the elements in the original order.

However, some algorithms prefer the lengthened output elements in interleaved order or proceed to reduce them afterward. These instructions always come in pairs, with the

bottom (B) version operating on even input elements (0, 2, 4, etc.) and the top (T) version operating on odd input elements (1, 3, 5, etc.), as depicted in [Figure 3.7: “Advanced SIMD Top and Bottom Half of Element Pairs”](#). Only a few brain floating point (bf`loat16`) operations utilize this interleaved approach.

Figure 3.7. Advanced SIMD Top and Bottom Half of Element Pairs



3.4 Compiler Intrinsics for Instructions and Data Types

Advanced SIMD and FP instructions discussed in the following sections can be used directly in assembly code or in compiled code through *intrinsics*, which are functions that translate directly to Arm Advanced SIMD and FP instructions. When routines are written in assembly, the developer must enforce aspects of the Application Binary Interface, such as calling conventions.

The Xcode development environment also offers a standard set of Advanced SIMD and FP intrinsics that allow a developer to code the bulk of their software in a high-level language while embedding specific Arm instructions using a function-like notation. This gives programmers complete control over the selection of Advanced SIMD computation instructions, at a level similar to what's possible with assembly language. At the same time, the compiler handles the routine bookkeeping tasks like loading values from memory, register allocation, loop control, and storing results to memory. Because the instructions required to handle these tasks are often much more than half of the assembly code in any Advanced SIMD routine, leaving them to the compiler greatly reduces the developer workload.

As an example, "Transpose Vectors (Primary)" `TRN1` instruction operating on 128b SIMD variables consisting of two 64b double-precision floating point values can be called using the following intrinsic:

```
#include <arm_neon.h>

// float64x2_t vtrn1q_f64(float64x2_t __p0, float64x2_t __p1);

float64x2_t a0, a1, b0;
...
b0 = vtrn1q_f64(a0, a1);
```

Intrinsics can be used in any C, C++, or Objective-C code simply by adding `#include <arm_neon.h>` at the beginning of any relevant source files and then using the intrinsics where needed.

Arm also offers [reference pages](#) on the intrinsic datatypes and functions.

Recommendation: Use intrinsic functions to manually vectorize the core algorithm in high-level languages.

[Magnitude: High | Applicability: High] Implement the core algorithm using intrinsics to efficiently leverage the Advanced SIMD instruction set. Leave the mundane tasks of obeying the ABI, managing pointers and loop constructs, allocating registers, and general loading and storing to the compiler. The compiler typically implements these tasks efficiently without any developer intervention.

3.4.1 C Vector Types

Before using the intrinsics themselves, the developer must convert variables to the appropriate vector types read or written by the intrinsic. The available types are summarized in [Table 3.12: “Advanced SIMD Intrinsic Data Types”](#). Each unique type comes in versions for use with scalars, 64-bit Advanced SIMD vectors, and 128-bit Advanced SIMD vectors. Each type explicitly contains a value describing its bit width, using notation based on that used in the C `stdint.h` header file to precisely specify integer types. In addition, the “xN” notation at the end of each type name specifies the number of fields of that type present in the vector, and thereby implicitly encodes the length of the vector.

Table 3.12. Advanced SIMD Intrinsic Data Types

Type	Element Size	Scalar	64b Vector	128b Vector
Floating Point	16b	float16_t	float16x4_t	float16x8_t
	32b	float32_t	float32x2_t	float32x4_t
	64b	float64_t	float64x1_t ← Use Scalar	float64x2_t
Brain Floating Point	16b	bfloat16_t	bfloat16x4_t	bfloat16x8_t
Signed Integer or Fixed Point	8b	int8_t	int8x8_t	int8x16_t
	16b	int16_t	int16x4_t	int16x8_t
	32b	int32_t	int32x2_t	int32x4_t
	64b	int64_t	int64x1_t ← Use Scalar	int64x2_t

Table 3.12. Advanced SIMD Intrinsic Data Types (cont.)

Type	Element Size	Scalar	64b Vector	128b Vector
Unsigned Integer or Fixed Point	8b	uint8_t	uint8x8_t	uint8x16_t
	16b	uint16_t	uint16x4_t	uint16x8_t
	32b	uint32_t	uint32x2_t	uint32x4_t
	64b	uint64_t	uint64x1_t ←Use Scalar	uint64x2_t
Galois Field Polynomial	8b	poly8_t	poly8x8_t	poly8x16_t
	16b	poly16_t	poly16x4_t	poly16x8_t
	64b	poly64_t	poly64x1_t	poly64x2_t
	128b	poly128_t	N/A	N/A

In addition to these basic types, there are also "multi-vector" types that are structures of 1–4 vectors together, with an additional "xM" specifier at the end of the type name to specify how many vectors are encoded. An example of this kind of type is `float32x4x4_t`. The various `float32x4_t` sub-vectors inside are defined as `val[<M>]`.

```
float32x4x4_t floatVector;
...
float32x4 v2 = floatVector.val[2];
```

Although they can be used for other purposes, these special types are generally only necessary when using `LD<n>`, `ST<n>`, `TBL<n>`, or `TBX<n>` instructions, which can have multi-vector sources or destinations (destination vectors for `LDn`, vectors to be stored by `ST<n>`, and input tables for `TBL<n>/TBX<n>`). Note that the `M=1` versions of these instructions just use normal vectors (e.g., `float32x4_t`) for input/output instead of `x1` multi-vectors (e.g., `float32x4x1_t`). Those multi-vector types with just one vector are seldom used.

3.4.2 Computation Intrinsics: A General Guide

Most Advanced SIMD intrinsic operations follow a fairly straightforward naming convention that translates the assembly-language instruction to a notation that works well in C. This section describes some of the main rules used to write Advanced SIMD assembly language using the intrinsic notation.

3.4.2.1 Basic Intrinsic Naming

Most of the intrinsics follow a straightforward naming scheme:

```
v<OP>[q|s|d][_high]_<input_type>
```

Each field has a well-defined meaning:

- **<OP>**: The Advanced SIMD assembly language mnemonic. The mnemonic does not include the type specification prefix letter (s/u/f letters from [Section 3.3.1, "ASIMD and FP Overall Instruction Data Type: \(none\) / BF / F / S / U"](#)) and half-select suffix (1/2 from

[Section 3.3.17, “ASIMD Lower-Half / Upper-Half: \(none\) / 1 / 2 ”](#)). Most other prefix and suffix letters are still included.

- **[(none) | q | s | d]**: The length of the vector. No letter here means the intrinsic works with 64-bit vectors, or is used with narrowing/lengthening operations that work with both 64-bit vectors *or* individual halves of 128-bit vectors. A *q* means that the intrinsic works exclusively with 128-bit vectors. The *s* and *d* letters explicitly indicate 32-bit and 64-bit scalar operations in rare cases where there is a scalar intrinsic and so this length indication is needed to avoid a naming conflict with a vector intrinsic.
- **[(none) | _high]**: The half select. *_high* is equivalent to the *2* suffix in assembly (as described in [Section 3.3.17, “ASIMD Lower-Half / Upper-Half: \(none\) / 1 / 2 ”](#)).
- **<input_type>**: The source data type. These indicators are taken from the list in [Table 3.6: “Vector Register Interpretation”](#), and specify both the type and size of the *sources*, which need to be specified with the types from [Table 3.12: “Advanced SIMD Intrinsic Data Types”](#). Note that the destination type, while often identical, may be different, particularly for the various “narrowing” and “lengthening” instructions. For the “widening” instructions that have two different sizes of inputs, this specifies the type of the *second* source, which always has smaller data words; the first source has larger data words. To reduce the need for casting between types, most “untyped” operations like load/store, bitwise logic, and byte rearrangement come in versions for *all* Advanced SIMD types, since these instructions are likely to be used with any type of vectors.

Consider the following variants of addition instructions. The first six show the equivalent intrinsic functions for normal in-lane operations with different vector sizes, types, and opcodes.

<code>int32x2_t</code> (64-bit vector)	<code>ADD</code>	<code>: vadd_s32</code>
<code>int32x4_t</code> (128-bit vector)	<code>ADD</code>	<code>: vaddq_s32</code>
<code>uint8x16_t</code>	<code>ADD</code>	<code>: vaddq_u8</code>
<code>float64x2_t</code>	<code>ADD</code>	<code>: vaddq_f64</code>
<code>int32x4_t</code>	<code>SHADD</code>	<code>: vhaddq_s32</code>
<code>uint8x16_t</code>	<code>UQADD</code>	<code>: vqaddq_u8</code>

The next four show variants that include lengthening and widening operations. These may require casting between 128-bit and 64-bit vectors depending on the specific intrinsic function. Operations that only use the lower half of 128-bit vectors, which is normal for the sources of the base (non-2) versions of lengthening and widening operations, always use 64-bit vector types for their input, instead of full 128-bit vectors. To do the casting, use `vget_low_<type>` intrinsics (see [Section 3.4.4.5, “Vector Length Conversion Operations”](#)).

<code>int32x4_t</code> to <code>int64x2_t</code>	<code>SADDL</code>	<code>: vaddl_s32</code> (cast both inputs to <code>int32x2_t</code>)
<code>int32x4_t</code> to <code>int64x2_t</code>	<code>SADDL2</code>	<code>: vaddl_high_s32</code>
<code>int64x2_t</code> + <code>int32x4_t</code> to <code>int64x2_t</code>	<code>SADDW</code>	<code>: vaddw_s32</code> (cast 32b input to <code>int32x2_t</code>)
<code>int64x2_t</code> + <code>int32x4_t</code> to <code>int64x2_t</code>	<code>SADDW2</code>	<code>: vaddw_high_s32</code>

The next two show variants that include narrowing operations. The result from the base (non-2) versions of narrowing operations is always a 64-bit vector type. It is usually fed immediately into the high (2) version.

```
int16x8_t to int8x16_t RADDHN : vraddhn_s16 (output is 64-bit int8x8_t)
int16x8_t to int8x16_t RADDHN2 : vraddhn_high_s16 (supply base version destination
                                                    as first input to merge with
                                                    high version)
```

Horizontal (v) operations follow normal rules, but take only single vectors as inputs and have scalar outputs.

```
int32x4_t to int32_t SADDV : vaddvq_s32
int32x4_t to int64_t SADDLV : vaddlvq_s32
```

Exceptions to basic naming conventions exist.

3.4.2.2 Intrinsic Functional Notation

Following C protocols, the various intrinsics use a functional notation. Input source variables are supplied like the inputs to C functions, and the destination result is supplied as a functional output:

```
destination = <intrinsic_name>(source1,source2,...)
```

Each of the sources is listed in the same order as the source registers for the matching assembly-language instruction. Unlike raw assembly language, this notation allows for things like nested intrinsics, such as a whole tree of ADDS on one line that reduces 16 signed 32-bit values to a single scalar:

```
sum = vaddvq_s32(vaddq_s32(vaddq_s32(input1,input2),vaddq_s32(input3,input4)));
```

One case that does not map precisely to this model is destructive operations that use their destination as an additional source, such as most of the multiply-accumulate instructions (see [Section 2.7, "Separate Source and Destination Registers"](#) and [Section 3.5.4, "Fused Op with Add/Accumulate Instructions \(Integer and ASIMD and FP Types\)"](#)). In these cases, the destination (also used as a source) is included as the first source operand:

```
dest_as_result = <intrinsic_name>(dest_as_src,source1,source2,...);
```

Another common case is the base-and-'2' sequence of narrowing operations, where the '2' operation merges the result of the base operation in the lower half of its destination register with results of the '2' operation put into the high half. Using the example from the previous section, the base version is often placed in the first parameter field of the high version intrinsic:

```
result = vraddhn_high_s16 (vraddhn_s16(src1Lo,src2Lo),src1Hi,src2Hi);
```

The normal return value for the intrinsic function is used for specifying the destination variable. The RADDHN2 assembly instruction uses a single register that serves as both the first source (dest_as_src) and the destination (dest_as_result), while the intrinsic code does not. However, the compiler allocates and manages registers correctly to match the instruction requirements.

For instructions that require immediate values, the immediate values are specified as integer constants as the last input to the intrinsic. A compilation error occurs if these values do not evaluate to legal integer constants at compile time.

3.4.3 Computation Intrinsics: Unusual Cases

Intrinsics for the majority of computation instructions can be synthesized using the rules in the previous section. However, there are a number of cases among mathematical operations that exhibit exceptions to these rules.

3.4.3.1 Paired Operations

For historical reasons dating back to ARMv7, the `p` for paired operations is applied as a prefix instead of a suffix:

```
vp<op>[q]_<type>
```

For example:

```
2xint32x4_t to int32x4_t SADDP   : vpaddq_s32
int32x4_t to int64x2_t  SADDLP  : vpaddlq_s32
```

3.4.3.2 Comparison Operations

Advanced SIMD comparison instructions have two unusual cases. The first is a reversal of letters from the assembly-language `FAC<cond>` instructions, also due to ARMv7 historical precedent:

```
vca<cond>[q]_<type>
```

Another exceptional case is for the Advanced SIMD instructions that perform a comparison against an immediate value of `#0`. These are handled by adding a `z` to the end of the intrinsic after the `q` suffix:

```
vc<cond>[q]z_<type>
```

Because the zero is embedded in the name of the intrinsic, there is no need to supply an explicit `#0` immediate input to these intrinsics; they therefore only need one input vector.

3.4.3.3 Multiplication by Element or Scalar Operations

Advanced SIMD vector-vector multiplications follow the normal notation. However multiplications between a vector and a single element have a unique notation:

```
v<multiply_name>[q]_lane[q]_<type>
```

The extra `_lane[q]` notation indicates that only a single element from the final (and optionally `q`-size) vector should be used in the multiplication. These multiply instructions require an extra input — an integer constant — to indicate which element to select. For example:

```
vmulq_laneq_f32(int32x4_t, int32x4_t, int32)
```

For lengthening operations, the high half '2' output of the lengthening element multiplication requires a `_high` indicator in the middle of the intrinsic, between the name of the multiply and the lane select:

```
v<multiply_name>_high_lane[q]_<type>
```

These forms are often used in sequences of multiplication operations that pick out successive elements from a vector, for example when performing matrix multiplications.

Instructions that multiply a vector by a constant scalar value also exist:

```
v<multiply_name>[q]_n_<type>
```

For example:

```
float32x4_t v_in = ...
float32_t s_in = 5.0;
float32x4_t v_out = vmulq_n_f32(v_in, s_in);
```

The final input to these operations is not a vector, but a scalar value of the same type as the input vector elements. The compiler typically loads the constant into element 0 of a vector, and uses the "by element" flavor of multiply instruction, but the compiler is free to use alternate optimization solutions.

3.4.3.4 Shift-by-Immediate Operations

Advanced SIMD shift-by-immediate instructions always have a `_n` after the name of the intrinsic:

```
v<shift_name>[q]_n_<type>
```

The immediate shift amount is the final input to the intrinsic. Variable-amount shifts do not have this `_n` specifier, and instead require a vector as their final input. This naming scheme allows clear distinctions between immediate-amount and variable-amount shifts with similar names.

3.4.3.5 Type Conversion Operations

Because conversion (`cvT`) instructions involve multiple data types, they have a unique format:

```
vcvt[a|m|n|p|z][q]_<output_type>_<input_type>
```

The type specifier at the end of the intrinsic specifies the input type, like usual. Because the input type does not define the output type for these instructions, an additional second type specifier is also required. This output specifier is placed just before the input specifier. Optionally, an explicit rounding mode letter (see [Section 3.3.13, "FP Rounding Mode: A / I / M / N / P / X / Z"](#)) may be added as well.

Because the sizes of the two types do not always match, several types of `cvT` instructions come in narrowing (when the destination is half the length) or lengthening (when the destination is twice the length) versions that require an additional high half '2' instruction to complete the operation (from [Section 3.3.17, "ASIMD Lower-Half / Upper-Half: \(none\) / 1 / 2"](#)). Unlike with most intrinsics that perform size changes, the `L` and `N` suffix letters from [Section 3.3.16, "ASIMD Narrow, Long, Wide: N / L / W"](#) are not used in the intrinsics, since the choice of types already imply lengthening or narrowing. The high half '2' versions are selected using `_high` specifiers.

```
vcvt[x]_high_<output_type>_<input_type>
```

Although the notation is somewhat different than most mathematical operations, these are still almost always used in pairs with the base (non-'2') version. In this first example, a pair of instructions is used to perform 64-bit-to-32-bit floating point conversion, packing two vectors with 64-bit elements into one vector with 32-bit elements.

```
v_f32 = vcvt_high_f32_f64(vcvf_f32_f64(v1_f64),v2_f64);
```

Similarly, a pair of instructions can lengthen one vector with 32-bit elements into two vectors of 64-bit elements.

```
v1_f64 = vcvt_f64_f32(v_f32);
v2_f64 = vcvt_high_f64_f32(v_f32);
```

Another variation of `cvT` is the family of conversions between fixed-point and floating point, which include an immediate value to specify the location of the binary point in the fixed-point number. These intrinsics, indicated with an additional `n` in the middle, all implicitly assume the `z` rounding mode, because that is the only one offered for this type of Advanced SIMD conversion.

```
vcvt[q]_n_<output_type>_<input_type>
```

The first input is the vector to be converted, while the second is a constant value specifying the location of the binary point within the fixed-point value, whether that type is used in the source or destination.

Finally, the related `INT` instructions are triggered using an intrinsic with a completely different name, `rnd`:

```
vrnd[a|i|m|n|p|x|z][q]_<type>
```

As with the `cvT` intrinsics, the letters for the various explicit rounding modes can be added if needed. The output type of these instructions is always in the same floating point type as the input, so there is no need for a second type.

3.4.4 Vector Manipulation Intrinsics

Vector element-wise or byte-wise manipulations often have names that do not correspond with the equivalent Advanced SIMD instructions. Further, some do not even match one-to-

one with instructions, but instead just specify the movement that must occur, and the compiler inserts instructions as needed. Sometimes no actual instructions are needed (e.g., most type casts between vector types), so these intrinsics are just for clarity.

3.4.4.1 Element Insertion and Extraction Operations

One of the first steps common to many Advanced SIMD routines is to assemble vectors from a number of scalar elements. The simplest option is to load a scalar into a Advanced SIMD unit register and cast it to a vector with a "create" intrinsic:

```
<vector_dest> = vcreate_<type>(<scalar_src>);
```

For example:

```
int32x2_t    vcreate_s32(uint64_t a);    // VMOV d0,r0,r0
```

The input could even be a `UINT64_C(...)` constant. With these intrinsics, the resulting vector has the scalar input in lane #0 while the other lanes are zeroed out. Note that there is no 128-bit variant of these instructions; the result is always a 64-bit vector. This is quick and easy, since it does not require an actual instruction.

To set values in other lanes or insert a value into the middle of an existing vector, use a "set lane" intrinsic:

```
<vector_dest> = vset[q]_lane_<type>(<scalar_src>,<vector_src>,<dest_lane#>);
```

This intrinsic inserts a scalar value into the supplied existing vector at the requested lane number, using an `INS` instruction. A sequence of these intrinsics can be used to populate an entire vector from an existing group of scalar values. Alternately, if the source values are already part of existing vectors, then a "copy" intrinsic is needed:

```
<vector_dest> = vcopy[q]_lane[q]_<type>(<vector_dest>,<dest_lane#>,<vector_src>,<src_lane#>);
```

This performs the same operation as the "set" intrinsic, also using a type of `INS` instruction, but takes a vector and lane number as its source. Note the two separate optional `q` specifiers. The first is used to indicate that the *destination* vector is 128 bits, while the second indicates that the *source* vector is 128 bits. Unlike with most vector instructions, they do not need to be the same.

Finally, when vector computation is complete and a scalar value needs to be extracted from a vector, the "get lane" intrinsic can be used:

```
<scalar_dest> = vget[q]_lane_<type>(<vector_src>,<src_lane#>);
```

This straightforward instruction translates into a vector-to-scalar `MOV` operation. There are also synonyms for these operations that use the "dup" intrinsic name, instead:

```
<scalar_dest> = vdup[b|h|s|d]_lane_<type>(<vector_src>,<src_lane#>);
```

Although these work, there should be little or no reason to use them. Note that the *b/h/s/d* letter is *not* optional, and is needed to differentiate this usage from the more traditional "dup" operations described in the next section.

3.4.4.2 Element Duplication Operations

An alternate way to create a vector from a scalar value is to duplicate a single scalar value across all element lanes within the vector, using a single `DUP` instruction. If the original value is in a scalar variable, there are two equivalent intrinsics to perform the task:

```
<vector_dest> = vmov[q]_n_<type>(<scalar_src>);
<vector_dest> = vdup[q]_n_<type>(<scalar_src>);
```

All lanes are set to the input scalar value. There is no difference between the two, so use whichever seems more appropriate. The "dup" variety is recommended, since it makes the function clearer. The "mov" variant may be more appropriate if used in a manner similar to the "create" intrinsic (but populating the upper lanes instead of zeroing them).

There is also a version that takes a vector input, but only in the "dup" variety:

```
<vector_dest> = vdup[q]_lane[q]_<type>(<vector_src>, <src_lane#>);
```

Analogous to the aforementioned "copy" intrinsic, this acts just like a `vdup_n`, but takes a vector and lane number as its source instead of a scalar. Note the two separate optional *q* specifiers. The first is used to indicate that the *destination* vector is 128 bits, while the second indicates that the *source* vector is 128 bits. As with the "copy" intrinsic, these do not need to match in code that uses both vector sizes.

3.4.4.3 Narrowing and Lengthening Element Operations

Variants of `MOV` intrinsics are used in place of the assembly instructions for narrowing and widening vector elements. For example, the narrowing `XTN[2]`, `SQXTN[2]`, `UQXTN[2]`, and `SQXTUN[2]` become the following:

```
<vector_dest> = v[q]mov[u]n_<type>(<vector_src>);
<vector_dest> = v[q]mov[u]n_high_<type>(<vector_dest>, <vector_src>);
```

The first intrinsic takes all elements of the vector source and packs them down into the lower half of the destination vector, leaving the upper half zeroed. The second intrinsic can then be used to pack all elements of a second vector source into the upper half of the destination vector.

The optional *q* intrinsic prefix specifies a saturating conversion, and indicates one of `SQXTN[2]`, `UQXTN[2]`, and `SQXTUN[2]`.

Without the *u* intrinsic suffix (prior to the 'n'), the saturating conversion retains its signed or unsigned type from source to destination, and therefore maps to either `SQXTN` or `UQXTN`. With the 'u' suffix, the intrinsic maps to `SQXTUN`, which reads signed sources and saturates them to unsigned destinations.

Mirroring this, there are similar `MOV` macros to use lengthening `SSHLL[2]` #0 (alias `SXTL[2]`) and `USHLL[2]` #0 (alias `UXTL[2]`) instructions, which do the opposite:

```
<vector_dest> = vmovl_<type>(<vector_src>);
<vector_dest> = vmovl_high_<type>(<vector_src>);
```

These intrinsics just widen each element, zero-extending or sign-extending from the upper bit as appropriate for the supplied input type.

Note that these intrinsics perform simple narrowing and lengthening, with no shifting involved. In order to do things like maintain the position of binary points, it is often necessary to use `SHLL` and `SHRN` computation instructions to shift the bits of each data element as the element is narrowed or lengthened.

3.4.4.4 Type Casting Operations

The "reinterpret" intrinsics perform a typecast from one vector type to another:

```
<vector_dest> = vreinterpret[q]_<dest_type>_<src_type>(<vector_src>);
```

Note that these intrinsics require type indicators for both the source *and* destination. No actual instructions are required to do this "operation", it is purely a C typecast. In some cases, traditional typecast semantics may also be supported:

```
<vector_dest> = (<dest_type_name>)<vector_src>;
```

When supported, type casts require the full destination type (e.g., `int16x8_t`) instead of just the short type indicator (e.g., `_s16`).

3.4.4.5 Vector Length Conversion Operations

Another type of vector casting is movement between 64-bit and 128-bit vector types. These types of manipulations sometimes require explicit instructions, but sometimes do not. The intrinsics abstract away the internal implementations of these transformations, so that programmers only need to specify what vector conversions are desired, but not necessarily how.

Going from 64-bit to 128-bit, the operation involves taking two 64-bit vectors and "combining" them into a 128-bit vector:

```
<128b_dest> = vcombine_<type>(<64b_src_low>, <64b_src_high>);
```

This operation just merges the two supplied 64-bit vectors together into a vector that is twice the length, with one original vector occupying the low-order half of the destination while the other occupies the high-order half.

In the reverse direction, a pair of intrinsics is needed to extract out the low-order and high-order 64-bit halves from a 128-bit vector:

```
<64b_dest_low> = vget_low_<type>(<128b_src>);
<64b_dest_high> = vget_high_<type>(<128b_src>);
```

Each of these intrinsics just extracts the requested half of the 128-bit vector as a 64-bit vector, using the type on the same row of [Table 3.12: "Advanced SIMD Intrinsic Data Types"](#).

3.4.4.6 Byte Rotations Operations

The "extract" operation allows a vector to be extracted from any bitwise rotation of a pair of vectors. This instruction is most often used for realigning vectors that have been read in an unaligned fashion, but can also be used for a wide variety of vector rotations. The format of the operation follows standard guidelines:

```
<vector_dest> = vext[q]<type>(<vector_src_low>, <vector_src_high>, <shift_amt>);
```

The result is the high-order elements of <vector_src_low> rotated down to the low-order end of the destination vector, and the low-order elements of <vector_src_high> rotated in to the high-order end of the destination. The overall rotation is controlled by <shift_amt>. These operations all use the EXT Advanced SIMD instructions, which only come in byte-type versions with a shift amount of 0-15 bytes. The *intrinsics* automatically scale this amount to match the type:

- **Half precision:** shift amounts of 0-7 halfwords for 128-bit vectors (0-3 for 64-bit vectors)
- **Single precision:** shift amounts of 0-3 words for 128-bit vectors (0-1 for 64-bit vectors)
- **Double precision:** shift amounts of 0-1 doublewords for 128-bit vectors (doubleword operations with 64-bit vectors do not perform any operation).

Note that due to the allowed range of immediate values, the result can be the entire <vector_src_low> vector (effectively making this a NOP when the shift amount is 0), but can never be the entire <vector_src_high> vector.

3.4.4.7 Data Transposition Operations

Three main data transposition instructions are available in the Advanced SIMD instruction set:

- **Transpose (TRN):** Selects and packs even (or odd) numbered elements from two vectors. These are designed to facilitate matrix transposition (see [Section 3.5.7.1, "Byte Shuffle Example: Transposing Matrices"](#)).
- **Unzip (UZP):** De-interleaves two vectors of interleaved data.
- **Zip (ZIP):** Interleaves two vectors of de-interleaved data.

These operations are usually used in "primary" low half ('1') and "secondary" high half ('2') pairs to mix two vectors together. However, there are also other ways to use these instructions for more arbitrary data transformations (see [Section 3.3.17, "ASIMD Lower-Half / Upper-Half: \(none\) / 1 / 2 "](#) and [Section 3.5.7, "Shuffle and Permute Instructions"](#)). The intrinsics or these instructions is straightforward:

```
<vec_dest_low> = v[trn|uzp|zip]1[q]<type>(<vec_src_low>, <vec_src_high>);  
<vec_dest_high> = v[trn|uzp|zip]2[q]<type>(<vec_src_low>, <vec_src_high>);
```

There are also older versions of these intrinsics that produce multi-vector type output:

```
<vec_dest_x2> = v[trn|uzp|zip][q]<type>(<vec_src_low>, <vec_src_high>);
```

These just compile into pairs of equivalent instructions, so using the explicit '1'/'2' instructions is generally recommended to avoid the hassle of dealing with multi-vector types (see [Section 3.4.1, "C Vector Types"](#)) at the instruction output. Otherwise, performance is no different.

3.4.4.8 Table Operations

The most complex vector permutation operations in the Advanced SIMD instruction set are the two "table vector lookup" operations. These allow a vector to be arbitrarily assembled from a "table" of bytes in 1–4 source vectors, based on the byte positions selected in a "control" vector. The table vector lookup `TBL` instruction assembles the output just using the input table, while the table vector lookup extension `TBX` instruction can also merge in existing bytes from the destination register, which is a separate input to the intrinsic. The invocations of these are very similar:

```
<vector_dest>          = vqtbl<#>[q]_<type>(<multi_vector_table>,<vector_control>);
<vec_dest_as_output> = vqtbx<#>[q]_<type>(<vec_dest_as_input>,<multi_vector_table>,<vector_control>);
```

Unlike most untyped operations which have intrinsics available in all types, only the three "byte" types are supported for these instructions (i.e. `_p8`, `_s8`, `_u8`). Hence, the destination and table inputs should always be in `poly8x8_t`, `poly8x16_t`, `int8x8_t`, `int8x16_t`, `uint8x8_t`, or `uint8x16_t` types. Intrinsics do not exist for other types, so casting is necessary when using non-byte type vectors with `TBL` or `TBX`. Meanwhile, the control vector is `uint8x8_t` or `uint8x16_t` for both the `_p8` and `_u8` cases, or `int8x8_t` or `int8x16_t` for the `_s8` case.

The table type can vary in its structure. If it only contains 1 vector, it is just a standard vector type. However, if it contains 2–4 vectors, then they are encapsulated in a "multi-vector" type (see [Section 3.4.1, "C Vector Types"](#)), and passed in together as a structure. The number of vectors in the table is always specified by the `<#>` field in the intrinsic.

The use of 64-bit vectors and 128-bit vectors is particularly complex with these intrinsics. The `q` suffix determines whether or not the destination and control vectors are 64-bit or 128-bit. Meanwhile, the special `q` field at the beginning of the intrinsic name indicates that the 1–4 vectors making up the input table are always 128-bit. It is required to avoid a collision with old intrinsics for the equivalent ARM ISA v7 operations without this `q`, which used 64-bit vectors for their tables. Although still defined, intrinsics for these old operations should generally *not* be used in ARM ISA v8/v9 code.

3.4.4.9 Load/Store Operations

When using intrinsics, it generally is not necessary to insert any explicit load or store operations. Write the actual computation with intrinsics using properly typed variables in the language. The compiler automatically inserts loads or stores to move data between memory and registers when it performs register allocation. Load and store intrinsics force the moment of data to and from memory that compiler may have otherwise been able to optimize.

However, loads and stores that interleave data are the primary exception (see [Section 3.3.9, "ASIMD Load/Store by Element Interleaving and Replicating: 1 / 2 / 3 / 4 / R"](#)):

```
<vector_dest> = vld<#>[q]_<type>(<vector_ptr>);
(no ret type)   vst<#>[q]_<type>(<vector_ptr>, <vector_src>);
```

These operations load or store interleaved data from/to the address indicated by `vector_ptr`. As with table operations, the nature of the `vector_dest` and `vector_src` operands used with these intrinsics varies depending upon the `<#>` factor in the instruction. With a value of 1, these are just standard vectors, but with values of 2–4 "multi-vector" types of the specified length are used instead (see [Section 3.4.1, "C Vector Types"](#)). In addition to specifying the length of the multi-vectors, the `<#>` factor also specifies the de-interleaving pattern of elements during loading or interleaving pattern of elements during storing. For example, `LD3/ST3` loads 16 interleaved RGB pixels from memory into three separate red/green/blue registers:

```
uint8x16x3_t pixels = vld3q_u8(srcPixelPtr);
uint8x16_t   redVec  = pixels.val[0];
uint8x16_t   greenVec = pixels.val[1];
uint8x16_t   blueVec  = pixels.val[2];
... compute with colors separately here ...
pixels.val[0] = redVec;
pixels.val[1] = greenVec;
pixels.val[2] = blueVec;
vst3q_u8(destPixelPtr, pixels);
```

Interleaved RGBa pixel information can be processed with analogous `LD4/ST4` combinations. Although complex numbers are typically loaded in interleaved form in order to leverage the complex number instructions (see [Section 3.3.7, "ASIMD Complex Number Arithmetic: C"](#)), they can be de-interleaved and re-interleaved using the `LD2/ST2` instructions (likely with `_f32` or `_f64` intrinsic types).

Note that the de-interleave load and interleave store instructions can be complex and somewhat slow and should only be used when necessary. However, they are usually more efficient than custom de-interleaving and re-interleaving code sequences, especially for the `LD3/ST3` variant.

The second varieties of element load/stores are the "lane" versions that load a scalar and insert it into a supplied vector at the requested location, or extract a single vector element and store it to memory. These are the equivalent of a scalar `LDR` followed by an `INS` operation to the selected lane, or the reverse `MOV-STR` sequence:

```
<vec_dest_as_output> = vld<#>[q]_lane_<type>(<scalar_ptr>, <vec_dest_as_input>,
                                                <dest_lane#>);
(no ret type)         vst<#>[q]_lane_<type>(<scalar_ptr>, <vector_src>,
                                                <src_lane#>);
```

The load operation inserts the loaded element into `vec_dest_as_input` at the requested lane, while the store extracts the element from the requested lane of `vector_src` and stores it. These operations can be a very convenient way to encode these scalar load/stores in a fairly efficient way, although using separate instructions is often equally efficient.

Finally, one can use `LD<#>R` operations that effectively combine a scalar load with a series of `DUP` instructions afterwards:

```
<vector_dest> = vld<#>[q]_dup_<type>(<scalar_ptr>);
```

These operations just load the scalar indicated by `scalar_ptr`, and then duplicate it across the `vector_dest`. The `<#>` value in this case is just a number of vectors to copy the element across. The `vector_dest` operand is either a standard vector, with a `<#>` value of 1, or a "multi-vector" type with values of 2-4 (see [Section 3.4.1, "C Vector Types"](#)). As with the "lane" versions, these are basically equivalent to just loading the value as a scalar and using one or more `DUP` instructions.

3.5 Advanced SIMD Coding Recommendations

This section describes a variety of useful coding tips for use with vectorized code written using Arm Advanced SIMD instructions.

3.5.1 Manage Architectural Register Limits

Proper architectural register allocation is an important part of maximizing performance for any software routine, but it is particularly important in small, tight loops consisting of high-density code that executes many instructions per cycle. Typically, at any point in execution of such a loop, there are many variables that are "live", that is, they contain values that are needed by subsequent instructions. The compiler, or the developer in the case of loops written in assembly language, must map these variables into registers. When there are insufficient registers to hold all of the live values, some values must be spilled to memory and filled back into registers before they are needed. However, the round trip time for a spill-fill sequence can be significant. The spill portion isn't always needed if the value can be simply reread from memory (e.g., a constant) when needed. Although this is faster than a spill-fill sequence, the additional loads still consume bandwidth and potentially add undesired latency.

Advanced SIMD routines tend to push register allocation to the limit, to the point where these spill-fill delays can sometimes set the critical path through the loop. This is because Advanced SIMD code tends to have a few key characteristics that combine to increase register pressure when performance is crucial:

- **Tight inner loops:** The performance-critical inner loops of Advanced SIMD code often tend to be rather short. As a result, if any registers are spilled out to memory in the loop body, the loads to read them back in inevitably arrives soon thereafter. There is little room within the loop body to move the spills and fills a "safe" distance apart.
- **High instructions per cycle (IPC) code:** For maximum performance, inner loops of Advanced SIMD programs must usually contain very dense code that is able to sustain an execution rate of 5–8 instructions every cycle, mostly across the Advanced SIMD and load/store execution units.
- **High latency instructions:** Advanced SIMD and load instructions usually take 2–5 cycles each to perform, so many parallel chains of independent instructions are usually needed to supply enough independent instructions to meet the execution rate of 5–8 instructions per cycle. In the worst case, it may be necessary for software to keep on the order of 20 parallel streams of instructions active in order to keep the Advanced SIMD and load/store units fully utilized. Out-of-order execution hardware identifies some parallelism automatically as it runs ahead and decodes multiple parallel loop

iterations. However, run ahead does not fill the instruction window with independent instructions immediately. Further, the code structure may contain inadvertent structural hazards such as loop carry chains that prevent automatic parallel execution. For more information on how the CPU executes instructions, see [Section 5.3, "Core Processor Pipeline Fundamentals"](#).

- **Extensive loop unrolling and interleaving:** In order to provide enough additional inter-instruction parallelism, software often contains unrolled loops. The unrolling process creates multiple copies of the loop body that can then be interleaved with each other to provide enough independent instructions at any point in time. Although this technique can be helpful at exposing useful parallelism, each copy of the loop body also requires its own copy of the "live" register state. The number of "live" registers thereby grows proportionally with the degree of unrolling.
- **Constants:** Another factor that can reduce the number of vector registers available is the need to devote registers to holding constant values such as zeros, filter weights, byte masks, permutation tables (for `TBL` and `TBX` instructions), and the like. Although these registers do not need to be replicated as loops are unrolled because they can be shared across all loop iterations, they can still occupy a significant number of registers that cannot be used for "live" values within each iteration.

Combining these factors, it is possible (and even likely) that the number of live values and constants exceeds the number of architectural registers available in the ISA. And, that using spill stores and fill loads to alleviate register pressure may consume instruction bandwidth while possibly adding in new critical paths through memory.

There is no universal solution for this problem. However, when coding the inner loops of Advanced SIMD algorithms, keep register constraints in mind. Try to minimize the number of live variables likely to need register allocation at any point, and keep in mind how loop unrolling may increase the number of live variables. In some cases, rearranging code within the loop to shorten the lifetime of live values may help. But ultimately loop unrolling may need to be limited to what the register file can support. In some cases, the maximum number of unrolls allowed by architectural register limits may be less than what is truly desired to attain maximum parallelism, but if loop unrolling triggers register spill-fill activity, it usually is counterproductive. Try to keep the number of live variables limited to ~25 (out of 32 available) at one time in order to maintain maximum performance, because a few registers are likely needed for transient values.

Recommendation: Balance exposed parallelism with architectural register limits.

[Magnitude: Medium | Applicability: Medium] Unrolling and interleaving of iterations creates instruction-level parallelism that is easy for the processor to exploit to achieve high performance. However, these techniques often require the use of many simultaneous variables. When the number of variables exceeds the architectural limit, the compiler spills variables to memory and fills them back when needed, resulting in reduced performance due to high pressure on the load and store unit. Maximize unrolling and interleaving while minimizing spills and fills. Trial-and-error may be required to find the best balance point.

3.5.2 Minimize Moves Between Integer and Vector Registers

Movement of data between integer and vector registers requires several cycles. Load directly into vector registers and use Advanced SIMD integer operations for very short

sequences. See [Section 5.5.1, “Movement of Data from General Purpose Registers to Vector Registers”](#) for additional details.

3.5.3 Horizontal Arithmetic and Test Instructions

When performing "horizontal" reductions across vectors, the Arm ISA offers a variety of options. For background, see [Section 3.3.18, "ASIMD Cross-Lane Paired or Horizontal: P / V"](#).

There are a basic set of "paired" operations that operate on adjacent vector elements:

- **Scalar:** `dest_el[0] = src_el[0] op src_el[1]`
- **Vector:** `dest_el[n] = src_el[2n] op src_el[2n+1]`, from two concatenated source registers.

Instructions include:

- `ADDP`
- `SADALP, SADDLP, UADALP, UADDLP`
- `SMAXP, SMINP, UMAXP, UMINP`
- `FADDP`
- `FMAXNMP, FMAXP, FMINNMP, FMINP`

There are also versions of most of these operations that work "across all" elements (lanes) of a vector:

- ADDV
- SADDLV, UADDLV
- SMAXV, SMINV, UMAXV, UMINV
- FMAXNMV, FMAXV, FMINNMV, FMINV

Note that it is possible to synthesize unofficial "horizontal" versions of the three remaining "paired" instructions with one- or two-instruction sequences:

Synthesized FADDV:

FADDV.S/2S:	FADDP.2S	Vx,	Vx,	Vany
FADDV.S/4S:	FADDP.4S	Vx,	Vx,	Vany
	FADDP.2S	Vx,	Vx,	Vany
FADDV.D/2D:	FADDP.2D	Vx,	Vx,	Vany

Synthesized SADALV:

```
SADDLV.H/S/D    H/S/Dtemp, Vx.8B/16B/4H/8H/4S
ADD             Ddest, Ddest, Dtemp
; must interpret signed result as H/S/D size to match SADDLV
```

Synthesized UADALV:

UADDLV.H/S/D	H/S/Dtemp, Vx.8B/16B/4H/8H/4S
ADD	Ddest, Ddest, Dtemp

The "any" registers can contain anything, because those vector lanes are ignored in the final result. Use registers with constant zeros to avoid accidentally including subnormals or NaN values or creating any unintended incoming register dependencies. The signed result for the synthesized `SADALV` and `UADALV` must be interpreted using the H/S/D size of the `SADDLV/UADDLV` result, and are not actually D-size. In fact, the `SADALV` output can only be interpreted using the `SADDLV` H/S/D size unless one explicitly sign-extends the result out to longer sizes.

With these operations, Arm ISA cores can perform tests on vectors fairly efficiently. Often in vector code, it is necessary to branch if any lane of a vector holds a special value, or all lanes hold a special value. For example:

Branch if any lane of V0 is zero:

```
UMINV.16B  B1, V0
FMOV      W0, S1      ; UMOV may be more "natural" but FMOV is faster because
                     ; it does not perform any conversion
CBZ       W0, target
```

Branch if any lane of V0 is non-zero:

```
UMAXV.16B  B1, V0
FMOV      W0, S1
CBNZ     W0, target
```

Branch if all lanes of V0 are zero:

```
UMAXV.16B  B1, V0
FMOV      W0, S1
CBZ       W0, target
```

Branch if all lanes of V0 are non-zero:

```
UMINV.16B  B1, V0
FMOV      W0, S1
CBNZ     W0, target
```

Similarly efficient code can be constructed for reductions and other cross-vector operations.

The integer and brain floating point (`bf16`) dot product instructions also sum across the results of the lane multiplies. See [Section 3.5.6, "Dot Product and Matrix Multiply Instructions"](#).

Recommendation: Use horizontal instructions for common vector tests.

[Magnitude: Medium | Applicability: Medium] Common any- and all-lane tests can be accomplished through the `UMINV` and `UMAXV` instructions. The result can be moved to the integer registers where it can be used as a branch condition.

3.5.4 Fused Op with Add/Accumulate Instructions (Integer and ASIMD and FP Types)

The Arm ISA offers a limited set of *fused* instructions that combine an operation with an add or accumulate (e.g., floating point fused multiply-add to accumulator (FMLA) instruction). These instructions offer both functional and microarchitectural benefits. Although microarchitectural topics are discussed in [Chapter 5, Core Microarchitecture Optimization](#), the following includes a discussion of the microarchitectural fundamentals of these fused instructions.

Some fused two-operation instructions use three source registers and a separate destination register. This style of instruction is used for fused instructions that execute on the Integer Execution Unit. For example:

- **MADD:** Multiply-Add multiplies two integer register values, adds a third integer register value, and writes the result to the integer destination register.

Others, as noted in [Section 2.7, "Separate Source and Destination Registers"](#), use 2 source registers and a third combined source+destination register. This style of instruction is used for Advanced SIMD and FP instructions, including those that operate on integer typed-elements. For example:

- **MLA:** Multiply-Add to Accumulator multiplies corresponding integer elements in the vectors of the two source Advanced SIMD and FP registers, and accumulates the results with the vector elements of the destination Advanced SIMD and FP register.

In this section, "op-add" refers generically to all relevant instructions, whether they add to a separate destination or into an accumulator.

There are several benefits to op-add instructions:

- **Increased throughput:** All op-add instructions effectively combine two operations into one. Combining them cuts in half the resources required in various parts of the CPU.
 - All benefit from instruction reduction compared with separate multiplies and adds, which improves Instruction Fetch and Instruction Caching performance.
 - Advanced SIMD and FP op-adds process as single μ ops throughout the remainder of the CPU.
 - Integer op-adds on many chips process as single μ ops; however, the P cores starting on the M4 and A18 families of chips [crack](#) these into two μ ops. This cracking strategy maintains the single-instruction fetch benefit but allows the operations to use additional multiply execution units, thereby alleviating the execution unit bottleneck.
- **Lower overall latency:** In many cases, the op-add instruction executes with the same latency as an instruction that only executes the base op.
 - For FP op-adds, the latency through the multiply inputs is the same as that of just a multiply. However, the latency through the add input may be longer than with individual op and add instructions. For instance, the FMLA (4 cycle latency) executes the equivalent of an FMUL (4 cycle latency) followed by an FADD (3 cycle latency). (See note about accuracy and rounding below.) Although the overall latency is 4 cycles instead of 7, the latency through the add input is 4 instead of 3. This may impact the throughput if the critical path is through the add input. However, the bandwidth savings are usually significant for using FMLA, and the critical path can often be

Apple Silicon CPU Optimization Guide: 4.0

ISA Optimization: Advanced SIMD and FP Unit Fused Op with Add/Accumulate Instructions (Integer and ASIMD and FP Types)

alleviated by unrolling and restructuring the algorithm to have 4 independent chains of FMLAS that are combined at the end. See [Appendix A, Instruction Latency and Bandwidth](#). Integer Advanced SIMD op-adds have similar behavior.

- For integer unit op-adds on all P cores on the M-series family of chips up through M3, the A-series family of chips up through A17, and on all E cores, the latency through the multiply inputs is the same as that of just a multiply. The op-add instruction may be issued early, while the add input is still being computed by a previous instruction. That input is only needed for the final addition step, after the multiplication has completed and produced its result. This can allow sequences of dependent adds/accumulations to be chained together so that only one cycle per additional fused operation is required. Normally, each add/accumulation in the chain would require three cycles each because they would not begin executing until all of their source inputs are available.
- For integer unit op-adds on the P cores of the M4 and A18 families of chips, the latency through the multiply inputs is of a separate multiply and add. However, this separation allows the CPU to schedule the components independently and use the full set of add and multiply execution resources. The critical one-cycle latency through the accumulation input is maintained.
- **Increased accuracy for FP operations:** The floating point flavors of op-adds are all multiply-accumulates. The processor performs no rounding step on the output of the multiply prior to performing the addition. Because of this, the final result may be slightly different than if the computation were performed with a discrete multiply followed by a discrete addition. See note below about use in high-level languages.

Fused two operation instructions:

- **Integer Execution Instructions:**
 - Multiply-add/subtract (separate destination): MADD, MSUB
 - Multiply-add/subtract long (separate destination): SMADDL, UMADDL, SMSUBL, UMSUBL
- **Advanced SIMD and FP Execution Instructions:**
 - Integer absolute difference and accumulate (accumulator): SABA, UABA
 - Integer absolute difference and accumulate long (accumulator): SABAL(2), UABAL(2)
 - Integer add and accumulate long pairwise (accumulator): SADALP, UADALP
 - Integer shift right and accumulate (accumulator): SSRA, USRA, SRSRA, URSRA
 - Integer multiply-add/subtract to accumulator (accumulator): MLA, MLS
 - Integer signed saturating rounding doubling multiply accumulate (accumulator): SQRDMLAH, SQRDMLSH
 - Integer dot product (accumulator): SDOT, UDOT, SUDOT, USDOT
 - Integer multiply-add/subtract long (accumulator): SMLAL(2), SMLSL(2), UMLAL(2), UMLSL(2)
 - Integer saturating multiply-add/subtract long (accumulator): SQDMLAL(2), SQDMLSL(2)

- Integer matrix multiply-accumulate (accumulator): `SMMLA`, `UMMLA`, `USMMLA`
- Floating point multiply-add/subtract to accumulator (accumulator): `FMLA`, `FMLS`
- Floating point complex multiply accumulate (accumulator): `FCMLA`
- Floating point multiply-accumulate long to accumulator (accumulator): `FMLAL(2)`, `FMLSL(2)`
- Brain floating point (bfloating16) dot product (accumulator): `BFDOT`
- Brain floating point (bfloating16) widening multiply-add (accumulator): `BFMLALB`, `BFMLALT`
- Brain floating point (bfloating16) matrix multiply-accumulate (accumulator): `BFMMLA`

High-level languages typically do not include native language support for multiply-accumulate floating point mathematical operations. Instead, they require the generated code to properly round the result after each floating point operation. Therefore, compilers typically do not automatically synthesize *fused* instructions because of the eliminated intermediate rounding step. However, many compilers including those in the Xcode environment support the "`-ffast-math`" flag that allows the compiler to generate these *fused* mathematical operations, as well as generally reorder floating point mathematical operations. For a broader discussion of fast-math, see note in the introduction of [Chapter 3, ISA Optimization: Advanced SIMD and FP Unit](#).

Recommendation: Use `op-add` (e.g., multiply-accumulate) to increase code density and reduce latency.

[Magnitude: High | Applicability: Medium] Many algorithms rely on the common code construct that performs an operation on two values and adds the result in with the result of previous operations. For example: floating point multiply-accumulate (e.g., `FMLA`). The Arm ISA features a number of these instructions beyond typical multiply-accumulate including absolute difference-accumulate and shift-accumulate. Use of these instructions reduces instruction count and latency compared with performing the `op` and the `accumulate` each with dedicated instructions. In the case of floating point multiply accumulate, increased accuracy is also achieved by not rounding the multiply result prior to the accumulation.

3.5.5 Complex Number Instructions

The `FCADD` and `FCMLA` instructions allow Advanced SIMD computation of the real and imaginary components of complex numbers in parallel. Complex numbers are represented as pairs of elements inside of vector registers, where a pair consists of the real and imaginary components of the complex number (in other words, every even element is a real component and every odd element is an imaginary component).

Without these instructions, most complex operations require de-interleaving and re-interleaving the values into all-real and all-imaginary vectors using `UZP` and `ZIP` instructions. These instructions are much like the non-complex `FADD` and `FMLA` instructions, but they have an extra immediate field to select a "rotation" of the vector's sign bits.

Here are a few sample operations involving multiply-accumulate operations with dense complex vectors and their complex conjugates:

Table 3.13. Example Complex Number Operations

Operation	Instruction Sequence
$A += B \times C$	FCMLA.<T> Va, Vb, Vc, #0 FCMLA.<T> Va, Vb, Vc, #90
$A += \text{conj}(B) \times C$	FCMLA.<T> Va, Vb, Vc, #0 FCMLA.<T> Va, Vb, Vc, #270
$A -= B \times C$	FCMLA.<T> Va, Vb, Vc, #180 FCMLA.<T> Va, Vb, Vc, #270
$A -= \text{conj}(B) \times C$	FCMLA.<T> Va, Vb, Vc, #180 FCMLA.<T> Va, Vb, Vc, #90

Other combinations of FCADD and FCMLA instructions with various rotations allow the handling of any complex arithmetic.

3.5.6 Dot Product and Matrix Multiply Instructions

The dot product and matrix multiply instructions perform a large number of multiplications with either 8-bit integers or 16-bit brain floating point `BFLOAT16` inputs. The instructions then accumulate the results into accumulators arranged as four 32-bit values, either integer or single-precision floating point, respectively.

The M1 and A14 Bionic feature the signed and unsigned dot product instructions `SDOT` and `UDOT`. With `FEAT_I8MM`, `SUDOT` and `USDOT` instructions were added in subsequent families in order to allow the use of mixed signed and unsigned values.

These instructions take two vectors of 8-bit inputs, multiply all of the corresponding elements together, and then accumulate all of the results, unrounded, with the corresponding elements of a vector of 32-bit integers. In other words, each instruction performs 16 separate 8-bit x 8-bit multiplications followed by four parallel 32-bit summations of five values each. Here is the operation expressed mathematically, where *A* and *B* are vectors of 8-bit values and *C* is a vector of 32-bit values:

$$\begin{aligned} C[0] &+= A[0] \cdot B[0] + A[1] \cdot B[1] + A[2] \cdot B[2] + A[3] \cdot B[3] \\ C[1] &+= A[4] \cdot B[4] + A[5] \cdot B[5] + A[6] \cdot B[6] + A[7] \cdot B[7] \\ C[2] &+= A[8] \cdot B[8] + A[9] \cdot B[9] + A[10] \cdot B[10] + A[11] \cdot B[11] \\ C[3] &+= A[12] \cdot B[12] + A[13] \cdot B[13] + A[14] \cdot B[14] + A[15] \cdot B[15] \end{aligned}$$

These operations are particularly useful during filtering of 8-bit pixel values, since each instruction performs many multiplies and adds in parallel without losing precision (no rounding of intermediate values). The newer versions allow additional use cases, such as when unsigned pixel values must be multiplied by signed filter weight values.

Alternately, use the *per element* version of these instructions to reuse a subset of the second vector:

$$\begin{aligned} C[0] &+= A[0] \cdot B[4i+0] + A[1] \cdot B[4i+1] + A[2] \cdot B[4i+2] + A[3] \cdot B[4i+3] \\ C[1] &+= A[4] \cdot B[4i+0] + A[5] \cdot B[4i+1] + A[6] \cdot B[4i+2] + A[7] \cdot B[4i+3] \\ C[2] &+= A[8] \cdot B[4i+0] + A[9] \cdot B[4i+1] + A[10] \cdot B[4i+2] + A[11] \cdot B[4i+3] \\ C[3] &+= A[12] \cdot B[4i+0] + A[13] \cdot B[4i+1] + A[14] \cdot B[4i+2] + A[15] \cdot B[4i+3] \end{aligned}$$

The variable *i* is in the range of 0-3, specifying one of the four 32-bit lanes within the second vector from which to select a group of four byte-size elements. This variant can be

useful if computing a dot product against a number of word-wide patterns, since four of these patterns can be packed into a single vector to save registers.

FEAT_I8MM provides the *matrix* multiply-accumulate instructions, SMMLA, UMMLA, and USMMLA. These instructions perform a matrix multiply operation of a vector containing a 2 x 8 matrix of 8-bit integers by another containing an 8 x 2 matrix of 8-bit integers, with the individual integers either signed, unsigned, or both (in the latter case, the 2 x 8 is unsigned, the 8 x 2 signed). Much like with the dot product instructions, these results are then accumulated into a vector containing four 32-bit accumulators, arranged as a 2 x 2 grid.

```

C[0,0] += A[0,0]•B[0,0] + A[0,1]•B[1,0] + A[0,2]•B[2,0] + A[0,3]•B[3,0]
          + A[0,4]•B[4,0] + A[0,5]•B[5,0] + A[0,6]•B[6,0] + A[0,7]•B[7,0]
C[0,1] += A[0,0]•B[0,1] + A[0,1]•B[1,1] + A[0,2]•B[2,1] + A[0,3]•B[3,1]
          + A[0,4]•B[4,1] + A[0,5]•B[5,1] + A[0,6]•B[6,1] + A[0,7]•B[7,1]
C[1,0] += A[1,0]•B[0,0] + A[1,1]•B[1,0] + A[1,2]•B[2,0] + A[1,3]•B[3,0]
          + A[1,4]•B[4,0] + A[1,5]•B[5,0] + A[1,6]•B[6,0] + A[1,7]•B[7,0]
C[1,1] += A[1,0]•B[0,1] + A[1,1]•B[1,1] + A[1,2]•B[2,1] + A[1,3]•B[3,1]
          + A[1,4]•B[4,1] + A[1,5]•B[5,1] + A[1,6]•B[6,1] + A[1,7]•B[7,1]

```

Effectively, the matrix multiply instructions are the dot products of all four possible combinations of the high and low halves of the A & B input vectors:

```

C[0] += A[0]•B[0]   + A[1]•B[1]   + A[2]•B[2]   + A[3]•B[3]
          + A[4]•B[4]   + A[5]•B[5]   + A[6]•B[6]   + A[7]•B[7]
C[1] += A[0]•B[8]   + A[1]•B[9]   + A[2]•B[10]  + A[3]•B[11]
          + A[4]•B[12]  + A[5]•B[13]  + A[6]•B[14]  + A[7]•B[15]
C[2] += A[8]•B[0]   + A[9]•B[1]   + A[10]•B[2]  + A[11]•B[3]
          + A[12]•B[4]  + A[13]•B[5]  + A[14]•B[6]  + A[15]•B[7]
C[3] += A[8]•B[8]   + A[9]•B[9]   + A[10]•B[10] + A[11]•B[11]
          + A[12]•B[12] + A[13]•B[13] + A[14]•B[14] + A[15]•B[15]

```

Because each of the vectors are encoding entire matrices, there are no *per element* versions of these instructions.

Brain floating point (BFLOAT16) dot product and matrix multiply instructions available with FEAT_BF16 operate similarly, but with fewer elements due to the larger element size.

See [Section 3.5.8, "Brain Float Data Type \(BFLOAT16\) Operations"](#) for a complete list of available instructions.

3.5.7 Shuffle and Permute Instructions

The Advanced SIMD instruction set contains about a dozen byte-shuffling operations. The TBL instruction can perform any combination of byte rearrangement using up to four independent input vectors as input. The TBX performs the same operation, but can insert select bytes arbitrarily into an existing destination vector. However, these instructions are generally microcoded and therefore not necessarily fast. Also, the algorithm must dedicate a register to contain the "mapping" control vector that specifies the input byte to select for each output byte position. For any scenario where a number of different arrangements are used regularly in quick succession, several registers may have to be dedicated to holding these mapping control vectors.

As a result, it is better to use one of the *dedicated* byte-rearranging instructions (DUP, EXT, INS, REV, TRN, UZP, and ZIP) if possible. [Table 3.14: “Advanced SIMD Shuffling Operations”](#) lists these various instructions, along with brief descriptions and a summary of their primary intended uses.

Table 3.14. Advanced SIMD Shuffling Operations

Instruction	Description	Primary Use
DUP	Copies one vector lane of input to all lanes of output	Copying any single value across all vector lanes
EXT #n	Appends 2 full vectors together, rotates n bytes, and extracts 1 vector from the center	Aligning unaligned input; shifting upper-lane elements down to lower lanes
INS	Inserts a value from one vector lane of input into one vector lane of output	Moving of one value around within a vector
REV16	Reverses operand order, within halfwords	Switching endian-ness of half word data
REV32	Reverses operand order, within words	Switching endian-ness of word data
REV64	Reverses operand order, within doublewords	Switching endian-ness of doubleword data
TBL	Selects bytes from 1–4 input vectors to assemble an output vector, based on a selection mask vector	Assembling a vector, byte-by-byte
TBX	Selects bytes from 1–4 input vectors to insert into an output vector, based on a selection mask vector	Inserting bytes into an existing vector
TRN	Interleaves the even or odd numbered elements from two input vectors	Transposing blocks of vectors
UZP	De-interleaves the even or odd numbered elements from two input vectors	De-interleaving packed objects like pixels into separate buffers
ZIP	Interleaves the low or high half elements from two input vectors	Interleaving separated values together, like merging per-color buffers into pixels

The various possible output permutations of the input bytes from each of the Advanced SIMD byte-shuffling instructions are listed in the following tables. Input #1 bytes are indicated using hexadecimal (with lower case letters), while input #2 bytes are indicated using alphabetic capital letters and are shaded gray. Note that TRN/UZP/ZIP all behave identically to each other when the inputs are each comprised of two 64-bit operands and can be used interchangeably.

In many cases, algorithms require specific well-structured byte rearrangement. It may be possible to achieve the desired rearrangement through a series of steps through these dedicated shuffling instructions. Two or four neighboring elements may be moved together by specifying a larger element size than the elements themselves. See the [Section 3.5.7.1, “Byte Shuffle Example: Transposing Matrices”](#) for an example.

Table 3.15. Input-Output Patterns for the Advanced SIMD Shuffling Operations: Byte-size Elements

Format	Op	Vector Byte Lanes															
		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Byte (B)	Input 1	f	e	d	c	b	a	9	8	7	6	5	4	3	2	1	0
	Input 2	P	O	N	M	L	K	J	I	H	G	F	E	D	C	B	A
	EXT #1	A	f	e	d	c	b	a	9	8	7	6	5	4	3	2	1
	EXT #2	B	A	f	e	d	c	b	a	9	8	7	6	5	4	3	2
	EXT #3	C	B	A	f	e	d	c	b	a	9	8	7	6	5	4	3
	EXT #4	D	C	B	A	f	e	d	c	b	a	9	8	7	6	5	4
	EXT #5	E	D	C	B	A	f	e	d	c	b	a	9	8	7	6	5
	EXT #6	F	E	D	C	B	A	f	e	d	c	b	a	9	8	7	6
	EXT #7	G	F	E	D	C	B	A	f	e	d	c	b	a	9	8	7
	EXT #8	H	G	F	E	D	C	B	A	f	e	d	c	b	a	9	8
	EXT #9	I	H	G	F	E	D	C	B	A	f	e	d	c	b	a	9
	EXT #10	J	I	H	G	F	E	D	C	B	A	f	e	d	c	b	a
	EXT #11	K	J	I	H	G	F	E	D	C	B	A	f	e	d	c	b
	EXT #12	L	K	J	I	H	G	F	E	D	C	B	A	f	e	d	c
	EXT #13	M	L	K	J	I	H	G	F	E	D	C	B	A	f	e	d
	EXT #14	N	M	L	K	J	I	H	G	F	E	D	C	B	A	f	e
	EXT #15	O	N	M	L	K	J	I	H	G	F	E	D	C	B	A	f
	REV16	e	f	c	d	a	b	8	9	6	7	4	5	2	3	0	1
	REV32	c	d	e	f	8	9	a	b	4	5	6	7	0	1	2	3
	REV64	8	9	a	b	c	d	e	f	0	1	2	3	4	5	6	7
	TRN1	O	e	M	c	K	a	I	8	G	6	E	4	C	2	A	0
	TRN2	P	f	N	d	L	b	J	9	H	7	F	5	D	3	B	1
	UZP1	O	M	K	I	G	E	C	A	e	c	a	8	6	4	2	0
	UZP2	P	N	L	J	H	F	D	B	f	d	b	9	7	5	3	1
	ZIP1	H	7	G	6	F	5	E	4	D	3	C	2	B	1	A	0
	ZIP2	P	f	O	e	N	d	M	c	L	b	K	a	J	9	I	8

Table 3.16. Input-Output Patterns for the Advanced SIMD Shuffling Operations: Half-size Elements

Format	Op	Vector Byte Lanes							
		15,14	13,12	11,10	9,8	7,6	5,4	3,2	1,0
Half (H)	Input 1	7	6	5	4	3	2	1	0
	Input 2	H	G	F	E	D	C	B	A
	EXT #2	A	7	6	5	4	3	2	1
	EXT #4	B	A	7	6	5	4	3	2
	EXT #6	C	B	A	7	6	5	4	3

Table 3.16. Input-Output Patterns for the Advanced SIMD Shuffling Operations: Half-size Elements

Format	Op	Vector Byte Lanes							
		15,14	13,12	11,10	9,8	7,6	5,4	3,2	1,0
	EXT #8	D	C	B	A	7	6	5	4
	EXT #10	E	D	C	B	A	7	6	5
	EXT #12	F	E	D	C	B	A	7	6
	EXT #14	G	F	E	D	C	B	A	7
	REV32	6	7	4	5	2	3	0	1
	REV64	4	5	6	7	0	1	2	3
	TRN1	G	6	E	4	C	2	A	0
	TRN2	H	7	F	5	D	3	B	1
	UZP1	G	E	C	A	6	4	2	0
	UZP2	H	F	D	B	7	5	3	1
	ZIP1	D	3	C	2	B	1	A	0
	ZIP2	H	7	G	6	F	5	E	4

Table 3.17. Input-Output Patterns for the Advanced SIMD Shuffling Operations: Word-size Elements

Format	Op	Vector Byte Lanes			
		15,14,13,12	11,10,9,8	7,6,5,4	3,2,1,0
Word (S)	Input 1	3	2	1	0
	Input 2	D	C	B	A
	EXT #4	A	3	2	1
	EXT #8	B	A	3	2
	EXT #12	C	B	A	3
	REV64	2	3	0	1
	TRN1	C	2	A	0
	TRN2	D	3	B	1
	UZP1	C	A	2	0
	UZP2	D	B	3	1
	ZIP1	B	1	A	0
	ZIP2	D	3	C	2

Table 3.18. Input-Output Patterns for the Advanced SIMD Shuffling Operations: Double-size Elements

Format	Op	Vector Byte Lanes	
		15,14,13,12,11,10,9,8	7,6,5,4,3,2,1,0
Double (D)	Input 1	1	0
	Input 2	B	A

Table 3.18. Input-Output Patterns for the Advanced SIMD Shuffling Operations: Double-size Elements

Format	Op	Vector Byte Lanes	
		15,14,13,12,11,10,9,8	7,6,5,4,3,2,1,0
	EXT #8	A	1
	TRN1/ UZP1/ ZIP1	A	0
	TRN2/ UZP2/ ZIP2	B	1

The DUP and INS instructions can be used to construct some patterns that cannot be created with any of the other element shuffling instructions. Although both select any arbitrary element from the input vector, DUP copies that element into all elements of the output, whereas INS inserts it into any arbitrary element of the output, leaving the contents of the other elements alone. Note that this insertion property makes INS a destructive, read-modify-write operation.

Finally, as noted at the beginning of this section, TBL and TBX allow creation of any arbitrary vector from one to four input vectors based on a byte selection vector. Because of their flexibility, these instructions require considerable resources to execute. Use them only when the other shuffle and permute instructions do not provide the necessary patterns.

Recommendation: Explore the EXT, REV, TRN, UZP, and ZIP instructions to perform a wide variety of shuffle and permute operations.

[Magnitude: Medium | Applicability: Medium] The EXT, REV, TRN, UZP, and ZIP instructions perform a variety of shuffle and permute operations that can be used alone or in combinations to create useful patterns. Patterns may be available with clever application of the instructions that might not be obvious and may require study of the tables.

3.5.7.1 Byte Shuffle Example: Transposing Matrices

One of the more common byte-shuffling actions is transposition of matrices in memory. The Advanced SIMD TRN instructions can be very helpful with this, but it is not always obvious how to put them to optimal use. They allow one to read a square block of data from the original matrix, where the size of each block is the number of elements in an Advanced SIMD vector, and transpose it in registers. This is most easily observed in a transpose of a matrix of 64-bit operands, which works with 2 x 2 blocks:

```
// Loop down input columns / across output rows
#define BLOCK_SIZE 2
for (k=0; k < INPUT_ROWS; k += BLOCK_SIZE) {

    // Loop across input rows / down output columns
    for (l=0; l < INPUT_COLUMNS; l += BLOCK_SIZE) {
```

Apple Silicon CPU Optimization Guide: 4.0

ISA Optimization: Advanced SIMD and FP Unit

Shuffle and Permute Instructions

```
// -- Read block
a0 = in[k*BLOCK_SIZE+0][1*BLOCK_SIZE];
a1 = in[k*BLOCK_SIZE+1][1*BLOCK_SIZE];

// 2x2 block transpose (C intrinsics)
b0 = vtrn1q_<suf>64(a0, a1);    // <suf> may be 'u' or 'f' or ...
b1 = vtrn2q_<suf>64(a0, a1);

// -- Write block out, transposed
in_t[1*BLOCK_SIZE+0][k*BLOCK_SIZE] = b0;
in_t[1*BLOCK_SIZE+1][k*BLOCK_SIZE] = b1;
}
}
```

Each block only requires two loads, two stores, and two TRN instructions, instead of four scalar loads and four scalar stores. Although this only has 25% fewer instructions/block, there are a full 50% fewer load and store instructions. Because transposes are usually bottlenecked on load and store bandwidth, this can result in a 2x speedup.

For 32-bit operands, a pair of word-size TRN instructions alone does not completely transpose a vector containing four word-size values. Instead, 32-bit TRN instructions need to be mixed with 64-bit TRN instructions in a 2-pass sequence on a 4 x 4 block of operands:

```
// Loop down input columns / across output rows
#define BLOCK_SIZE 4
for (k=0; k < INPUT_ROWS; k += BLOCK_SIZE) {

    // Loop across input rows / down output columns
    for (l=0; l < INPUT_COLUMNS; l += BLOCK_SIZE) {

        // -- Read block
        a0 = in[k*BLOCK_SIZE+0][1*BLOCK_SIZE];
        a1 = in[k*BLOCK_SIZE+1][1*BLOCK_SIZE];
        a2 = in[k*BLOCK_SIZE+2][1*BLOCK_SIZE];
        a3 = in[k*BLOCK_SIZE+3][1*BLOCK_SIZE];

        // 4x4 block transpose (C intrinsics)
        // -- Pass 1: 64b transpose
        b0 = vtrn1q_<suf>64(a0, a2);
        b1 = vtrn1q_<suf>64(a1, a3);
        b2 = vtrn2q_<suf>64(a0, a2);
        b3 = vtrn2q_<suf>64(a1, a3);

        // -- Pass 2: 32b transpose
        c0 = vtrn1q_<suf>32(b0, b1);
        c1 = vtrn2q_<suf>32(b0, b1);
        c2 = vtrn1q_<suf>32(b2, b3);
        c3 = vtrn2q_<suf>32(b2, b3);

        // -- Write block out, transposed
        in_t[1*BLOCK_SIZE+0][k*BLOCK_SIZE] = c0;
        in_t[1*BLOCK_SIZE+1][k*BLOCK_SIZE] = c1;
        in_t[1*BLOCK_SIZE+2][k*BLOCK_SIZE] = c2;
        in_t[1*BLOCK_SIZE+3][k*BLOCK_SIZE] = c3;
    }
}
```

The first pass transposes entire 64-bit halves of the vector, while the second pass transposes the 32-bit values within each of those halves. Note the ordering of TRN1/TRN2 instructions and mixture of inputs at each step. Naively, this transposition would require 16 scalar loads and 16 scalar stores, but can be accomplished with four loads, four stores, and eight TRN instructions. This TRN-based version requires 50% fewer instructions and 75% fewer loads and stores.

This same pattern can be extended to transpose matrices of halfword-size elements using 8 x 8 blocks, or even matrices of byte-size elements using 16 x 16 blocks. For smaller element sizes, additional passes are required, moving from transposing entire halves of each vector down to transposing individual elements: 8 x 8 uses three passes of TRN instructions per block (64-bit / 32-bit / 16-bit), while 16 x 16 uses four passes of TRN instructions (64-bit / 32-bit / 16-bit / 8-bit). Because each vector load and store can handle so many elements at once for these small element sizes, the savings are even more significant. For 16-bit elements, a block that takes 128 scalar load and stores requires only 16 vector load and stores with 24 TRN instructions. For 8-bit elements, a block that originally required 512 scalar load and stores now requires only 32 vector load and stores with 64 TRN instructions. Needless to say, the performance increase in these latter scenarios can be quite significant as the bottleneck shifts from the load and store units to the vector units.

3.5.8 Brain Float Data Type (BFloat16) Operations

The FEAT_BF16 feature parameter indicates that the processor supports both the BFloat16 data type and a limited selection of related instructions:

- **Multiply-Accumulate and Lengthen:** BFMLALB, BFMLALT (by element and by vector)
- **Dot Product:** BFDOT (by element and by vector)
- **Matrix Multiply:** BFMLLA
- **Conversion:** BVCVT, BFCVTN, BFCVTN2

These operations accumulate multiplications of BFloat16 inputs, in a variety of patterns, into Float32 destinations. The BFMLAL operations use the bottom/top input pattern (see [Section 3.3.19, "ASIMD Bottom or Top Half of Element Pairs: B / T"](#)). BFDOT and BFMLLA use other specific patterns custom to their operation.

The arithmetic operations are similar to their counterparts. See [Section 3.5.6, "Dot Product and Matrix Multiply Instructions"](#), [Section 3.5.4, "Fused Op with Add/Accumulate Instructions \(Integer and ASIMD and FP Types\)"](#), and the Arm Architecture Reference Manual for more information.

This small selection of instructions covers the majority of algorithms targeted by BFloat16. To perform other operations, convert the values to conventional Float32 values, do any necessary arithmetic with that higher-precision data type, and then convert back to BFloat16 before writing the values to memory. Note that there is no explicit BFloat16-to-Float32 conversion instruction. Instead, use the following sequence:

SHLL	Vd1.4S, Vn.8H, #16	; vector BFloat16-to-Float32, lower half
SHLL2	Vd2.4S, Vn.8H, #16	; vector BFloat16-to-Float32, upper half

Because the upper 16 bits of both types are identical (sign, exponent, and upper bits of mantissa), these shifts always result in an equivalent `Float32` with its low 16 mantissa bits equal to zero. Of course, since all of the math operations noted above also perform implicit lengthening to `Float32`, it is usually best to just perform lengthening using those operations.

There are, however, explicit instructions for conversion back to `BFloat16`:

<code>BFCVTN</code>	<code>Vd.4H, Vn1.4S</code>	<code>; vector Float32-to-BFloat16, lower half</code>
<code>BFCVTN2</code>	<code>Vd.8H, Vn2.4S</code>	<code>; vector Float32-to-BFloat16, upper half</code>

Use these dedicated instructions instead of an `SHRN` because they apply the proper floating-point rounding of the mantissa during the conversion. These instructions also have a scalar form (`BFCVT Hd, Sn`).

3.5.9 Int8 Matrix Multiplication and Additional Dot Product Operations

The `FEAT_I8MM` feature parameter indicates the availability of the 8-bit integer matrix multiply-accumulate instructions and some additional mixed sign dot product instructions.

- **Matrix Multiply-Accumulate:** `SMMLA`, `UMMLA`, `USMMLA` (by vector)
- **Dot Product:** `SUDOT` (by element), `USDOT` (by element and by vector)

3.6 Advanced SIMD Coding Examples

To illustrate how to use intrinsics, several examples are included in this section. However, these algorithms are common and some of them are already available in the [Accelerate.framework](#) in a highly optimized form. Use the framework capabilities when available instead of writing custom code. In order to focus on the core transformations, these examples have not been maximally optimized and do not handle many critical corner or edge cases. Still, they can provide an example of how Advanced SIMD code should be used to accelerate mathematically intense software.

Note that some of the inner loops in these examples *must* be unrolled in any source code, as they use intrinsics that require compile-time constant-value lane or byte specifiers. These loops, which are indicated with comments, are nevertheless written as such to avoid repetition of code that only differs by a single constant value per line.

3.6.1 Matrix Operations

Advanced SIMD is often used to accelerate dense matrix computations. The structure and regularity of these algorithms makes them well suited for the parallel operations offered by the Advanced SIMD instructions. Nevertheless, there are a number of subtle details about Advanced SIMD functionality that can be learned by looking at these examples. The remainder of this section discusses how 32-bit floating point matrix-matrix and matrix-vector multiplies can be constructed.

3.6.1.1 Matrix-Matrix Multiply

Dense matrix-matrix multiplies are commonly performed using vector instructions. The following block of code shows how a simple matrix-matrix multiply can be performed with $M \times P \times N = M \times N$ row-major matrices:

```
// Simple macros to read/write vectors from/to scalar arrays
#define LD_F32V(float32Ref)  (*((float32x4_t *)&(float32Ref)))
#define ST_F32V(float32Ref)  (*((float32x4_t *)&(float32Ref)))
#define VEC_SIZE_F32V 4
int i, j, k;          // Induction variables
float32x4_t a, c;     // Temporaries
float32_t *A;         // MxP source matrix #1
float32_t *B;         // PxN source matrix #2
float32_t *C;         // MxN destination matrix
float32x4_t zeroVec = vmovq_n_f32(0.0);

// Loop over columns of destination matrix C (rows of A)
for (i=0; i < M; i++) {
    // Loop across rows of destination matrix C (columns of B)
    // -- Calculate VEC_SIZE_F32V columns at once
    for (j=0; j < N; j+=VEC_SIZE_F32V) {
        // Loop over the "inner" P cols/rows of A/B matrices
        c = zeroVec;
        for (k=0; k < P; k+=VEC_SIZE_F32V) {
            // Read a vector full of A elements
            a = LD_F32V(A[i*P + k]);
            // Use multiply-by-element to scan down the "a" vector
            // -- First column of A (element #0) x first row of B
            c = vfmaq_laneq_f32(c, LD_F32V(B[(k+0)*N + j]), a, 0);
            // -- Second column of A (element #1) x second row of B
            c = vfmaq_laneq_f32(c, LD_F32V(B[(k+1)*N + j]), a, 1);
            // -- Third column of A (element #2) x third row of B
            c = vfmaq_laneq_f32(c, LD_F32V(B[(k+2)*N + j]), a, 2);
            // -- Fourth column of A (element #3) x fourth row of B
            c = vfmaq_laneq_f32(c, LD_F32V(B[(k+3)*N + j]), a, 3);
        }
        ST_F32V(C[i*N + j]) = c;
    }
}
```

The resulting assembly code from Clang intermingles the intrinsic instructions (5 LDR and 4 FMLA) with loop control and pointer management C code. Only the innermost loop is shown. For instance, the compiler simplifies the " $B[(k+\{0..3\}) * N + j]$ " indices in the LD_F32V macro to be a common base register x6 that increases once per loop iteration with offsets in x16, x17, and x19 that account for scaling of k by N and element size.

```
.LBB0_9:                                // Parent Loop BB0_3 Depth=1
    LDR    q1, [x5], #16
    LDR    q2, [x6]
    LDR    q3, [x6, x17]
    LDR    q4, [x6, x16]
    LDR    q5, [x6, x19]
    FMLA   v0.4s, v2.4s, v1.s[0]
    FMLA   v0.4s, v3.4s, v1.s[1]
    ADD    x4, x4, #4                    // =4
    FMLA   v0.4s, v4.4s, v1.s[2]
```

```

CMP      x4, x11
FMLA     v0.4s, v5.4s, v1.s[3]
ADD      x6, x6, x15
B.LT     .LBB0_9
B        .LBB0_6
  
```

Although this implementation is simplified, it demonstrates a few key common practices. The routine uses a fixed accumulator, `c`, and accumulates a vector's worth of outputs in parallel into that accumulator before storing its results. Although the code reads entire vectors at a time from the `B` matrix, the overall access pattern is down columns, which requires long strides. Meanwhile, the `A` array is read along rows, but instead of using the entire vectors of data at once, the algorithm uses the *lane* variants of the `vfmaq` intrinsics to pick out individual elements from each vector. This choice of operation arranges elements in such a way that the operation may be performed without transposing one of the input matrices to column-major form. Nevertheless, that form may still be advantageous for performance in some cases.

3.6.1.2 Matrix-Vector Multiply

Matrix-vector operations are often needed along with matrix-matrix operations. In the case of a vector-matrix multiply, with $1 \times P \times P \times N = 1 \times N$ (1-row) vector operation, the matrix-matrix code described previously can be used with $M=1$, so only a single pass through the outer loop occurs. However, for the more common matrix-vector multiply, with $M \times P \times P \times 1 = M \times 1$ (1-column) vector operation, the `B` matrix access pattern used previously does not lend itself well to vector access. As a result, the loop must be modified as shown below (the introductory code is identical to the previous example):

```

// Loop down column of destination matrix C (also rows of A)
for (i=0; i < M; i++) {
    // Scan through the B vector, VEC_SIZE elements at once
    c = zeroVec;
    for (k=0; k < P; k+=VEC_SIZE_F32V) {
        c = vfmaq_f32(c, LD_F32V(B[k]), LD_F32V(A[i*P + k]));
    }
    // Final reduction to a scalar, across lanes
    C[i] = vgetq_lane_f32(vpaddq_f32(vpaddq_f32(c, zeroVec), zeroVec), 0);
}
  
```

Instead of calculating a vector's worth of outputs in parallel, this only calculates a single scalar output value at a time. However, all of the vector lanes are used in parallel to perform computations, so only one fourth as many multiply-accumulate instructions are required when compared with the scalar version. At the end, a *tree* of paired adds accumulates the results down to a single scalar (see [Section 3.5.3, "Horizontal Arithmetic and Test Instructions"](#)).

3.6.1.3 Matrix Operation Improvements

The previously described simple routines produce the correct results and serve to demonstrate the core algorithms. For small input matrices that fit into the L1 data cache, they are sufficient. However, using these simple algorithms on larger matrices creates significant bottlenecks. As a result, real world implementations tend to have a few enhancements:

- **Loop Ordering:** The matrix-matrix example has loops arranged so that accesses to the A matrix are primarily in a row-major manner while accesses to the B matrix occurs in a largely column-major direction. The loops can be rearranged to swap these access patterns. The resulting patterns may be better for some matrix sizes or cache configurations.
- **Blocking:** The matrix-matrix example code reads the entire B matrix over the course of computing each row of the C matrix, using each data value only once during each pass through the matrix. As a result, the CPU tends to stream through the matrix and experience constant cache misses for larger sizes. Thus often the most important enhancement is to perform the multiply in a blocked manner, with an extra pair of outer loops that localize accesses to a subset of the rows/columns of the input matrices at a time. The selected rows/columns can then be cached and used repeatedly before moving to the next block. This process works the same way with vectors as it does with standard scalar operations.
- **Prefetching:** The hardware prefetcher works well with the access patterns running down contiguous rows in the A and C matrices (and B as well, in the matrix-vector case), but the large strides needed to access the B matrix down columns in the matrix-matrix case may not be identified as a single, regular pattern. As a result, software prefetching may help (see [Section 5.6.12.2, "Software Prefetch Instructions"](#)). Note though that the prefetch stream must run sufficiently far ahead of the access stream to hide the latency, and that tuning is likely required. If done well, software prefetches can virtually eliminate access-time cache misses.
- **Column Grouping:** Another problem with access to the B matrix is that reading a single vector's width of 4 columns at a time only uses a fraction of each cache line that is read into the L1 data cache. As a result, significant speedup can be gained simply by grouping larger numbers of columns together and computing 16-32 results (or more) in parallel instead of just four. This allows each cache line of B to be used in its entirety.
- **Transpose Caching:** With a blocked matrix multiply, B matrix access can be optimized further. Software can load the active block of the B matrix into a buffer, sized to fit into the L1 data cache, in the order that it needs to be read. The resulting buffer contents are not a simple scalar transpose, but a combination of the original order within each vector and the transpose between vectors. This buffer can then be re-read repeatedly, with few cache misses. This technique is only useful with very large matrix-matrix multiplies, since the buffer must be reused enough times to amortize the cost of filling it in the transport order in the first place, but it can be quite helpful for very large matrices.

With such enhancements, even relatively simple implementations can achieve good performance, especially with careful loop unrolling to maximize the utilization of vector registers.

3.6.2 Image Filtering

Filtering techniques are used regularly in the field of image processing. Many of these filters are two-dimensional versions of FIR filters, modifying values of pixels based on their current value and those of their nearest neighbors. This section illustrates a few ways that such filters can be constructed, and provides additional examples on how the ISA can be utilized through intrinsic functions.

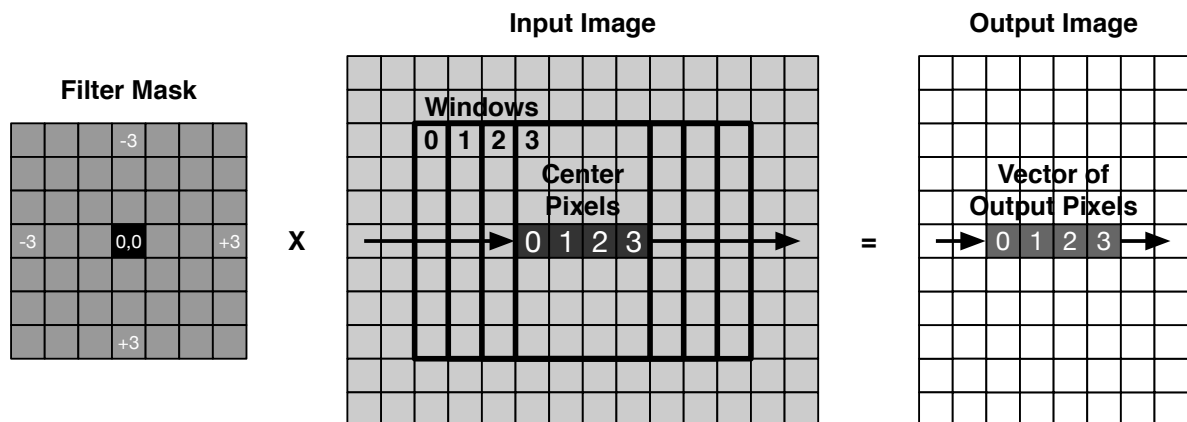
These filters operate on a grayscale buffer, which consists of 8-bit integer values (0-255) for each pixel. The concepts illustrated here can be applied to color images, too. Those

Images contain multiple color channels (RGB or RGBA), each of which is equivalent to a grayscale channel. They can be filtered using grayscale code by de-interleaving the combined buffer into separate R/G/B/a buffers, filtering the buffers separately, and then re-interleaving them afterwards. Another technique to handle color is to leave the buffers interleaved and process all of the colors in each pixel at the same time in different parallel vector lanes. Both approaches are viable; the parallel-colors technique is usually easiest if the same operation is always being applied to all colors, or if the colors within each pixel can interact. In contrast, the de-interleave/compute/re-interleave technique is usually best when different colors require different code sequences, such as for the alpha channel in RGBA images. Using LD3/ST3 or LD4/ST4 instructions to perform the de-interleaving/re-interleaving as a part of the load/store process is recommended in this scenario. See [Section 3.4.4.9, "Load/Store Operations"](#).

3.6.2.1 Window Filtering

Moving across each row of an image, software can calculate the final value for entire vectors of pixels at once. A similar *moving window* scheme is used to select groups of pixels from the area around the output vector to use as inputs while the algorithm scans down each row. Because an image is two dimensional, the algorithms reads input from several rows at once. The basic idea is illustrated in [Figure 3.8: “Advanced SIMD Pixel Filter”](#).

Figure 3.8. Advanced SIMD Pixel Filter



To compute N output pixels in parallel, each filter mask element is multiplied by the pixel from the same coordinate of N moving windows at once ($N=4$, in this case).

The following code processes a figure of COLS x ROWS, using a filter with a radius of EDGE pixels around each target pixel. This simple version can only handle cases with EDGE values up to 3, which results in a 7x7 filter around each pixel, maximum.

```
// Simple macro to access a scalar uint8_t array as vectors, instead
#define LD_U8V(uint8Ref)      (*(uint8_t *)(&(uint8Ref)))
#define ST_U8V(uint8Ref)      (*(uint8_t *)(&(uint8Ref)))
#define VEC_SIZE_U8V         16
#define MASKSIZE              (2*EDGE+1)
#define SCALEDN_PIXEL         8      // need to cancel out gain from filter weights

int i, j, k, v;
```

```
// Image buffers for input and output
uint8_t *imageIn, *imageOut;
// Filter weights, in vector/row (vector length limits MASKSIZE to 8)
int16x8_t filters[MASKSIZE];
int32x4_t zeroVec = vmovq_n_s32(0);

// Temporary buffers
uint8x16_t inBlock, input, nextInputs[MASKSIZE];
int16x8_t inWindow, filter;
int32x4_t acc0, acc1, acc2, acc3;

// Loop through rows of image
for (i=EDGE; i < ROWS - EDGE; i++) {

    // Load in initial block of 16 inputs for this row
    for (v = 0; v < MASKSIZE; v++) {
        nextInputs[v] = LD_U8V(imageIn[(i+v-EDGE)*COLS]);
    }

    // Loop over columns, stepping by vectors
    for (j=EDGE; j < COLS - EDGE; j+=VEC_SIZE_U8V) {

        // Clear accumulators for next block of pixels
        acc0 = acc1 = acc2 = acc3 = zeroVec;

        // Loop over rows of filter window
        for (k = 0; k < MASKSIZE; k++) {
            // Load in next block of inputs
            input = nextInputs[k];
            nextInputs[k] = LD_U8V(imageIn[(i+k-EDGE)*COLS +
                (j-EDGE+VEC_SIZE_U8V)]);
            filter = filters[k];
            // Loop over columns of filter window
            // -- NOTE: MUST be unrolled to support vextq & vmlal!
            for (v = 0; v < MASKSIZE; v++) {
                // Align and zero-extend 16 columns of input
                inBlock = vextq_u8(input, nextInputs[k], v);
                // -- Accumulate 1 weight for left half of vector
                inWindow = (int16x8_t)vmovl_u8(vget_low_u8(inBlock));
                acc0 = vmlal_laneq_s16(acc0,
                    vget_low_s16(inWindow), filter, v);
                acc1 = vmlal_high_laneq_s16(acc1, inWindow, filter, v);
                // -- Accumulate 1 weight for right half of vector
                inWindow = (int16x8_t)vmovl_high_u8(inBlock);
                acc2 = vmlal_laneq_s16(acc0,
                    vget_low_s16(inWindow), filter, v);
                acc3 = vmlal_high_laneq_s16(acc1, inWindow, filter, v);
            }
        }

        // Pack 4 _s32 accumulators back into one _u8 vector for output
        // -- Scale down during 32b->16b conversion,
        // then just narrow & remove sign during 16b->8b
        ST_U8V(imageOut[i*COLS + j]) =
            vqmovun_high_s16( // 4. #2-3 -> u8
            vqmovun_s16( // 3. #0-1 -> u8
            vqrshrn_high_n_s32( // 2a. #1 -> s16
            vqrshrn_n_s32(acc0, SCALEDN_PIXEL), // 1a. #0 -> s16
            acc1, SCALEDN_PIXEL)),
```

```

        vqrshrn_high_n_s32(                // 2b. #3 -> s16
        vqrshrn_n_s32(acc2, SCALEDN_PIXEL), // 1b. #2 -> s16
        acc3, SCALEDN_PIXEL));
    }
}

```

This code leverages 32-bit accumulators, 4 times the size of the 8-bit input elements, to provide extended range for the integer multiplications. However, since it is an image filter it must scan over pixels in both X and Y directions. Although some FIR audio filters tend to have hundreds of taps down a single dimension, the MASKSIZE of image filters tends to be much smaller. Nevertheless, there are still a significant number of taps, MASKSIZE² taps total. This simple implementation allows for mask sizes up to 7x7. However, this limit is only caused by the limited length of the filter weights for each row (single int16x8_t vectors) and the finite size of the input buffer (just 2 uint8x16_t vectors).

One issue that needs to be considered with these filters is the number of variables that need to be register allocated. This example is small enough that all live variables, including the entire filters and nextInputs arrays, can be register-allocated, so the only memory references that need to be made are the LDS and STs noted in the code. However, if the MASKSIZE were to get much larger then these arrays might start to spill out of registers. When that occurs, the compiler inserts additional loads and stores to re-load values from these arrays, which may cause performance to drop off noticeably. See [Section 3.5.1, “Manage Architectural Register Limits”](#) for more commentary on this effect.

3.6.2.2 Per-Pixel Filtering

An alternate strategy for performing the same filter is to calculate a single output pixel at a time. In this scheme, the code uses parallel vector lanes to calculate an entire row’s worth of filter taps for one output pixel at once, instead of calculating the same tap across an entire vector of separate outputs at once. The difference between this and the previous filters is quite analogous to the difference between the matrix-matrix and matrix-vector computations discussed previously in [Section 3.6.1, “Matrix Operations”](#).

```

int32x2_t s32Extension = vmov_n_s32(0);
int16x4_t s16Extension = vmov_n_s16(0);

// Temporary buffers
uint8x16_t inputs[MASKSIZE], nextInputs[MASKSIZE];
int16x8_t inWindow, filter;
int32x4_t acc;

// Loop through rows of image
for (i=EDGE; i < ROWS - EDGE; i++) {

    // Load in initial block of inputs (sufficient for 16-wide filter)
    for (v = 0; v < MASKSIZE; v++) {
        nextInputs[v] = LD_U8V(imageIn[(i+v-EDGE)*COLS]);
    }

    // Loop over columns, stepping by vectors
    for (j=EDGE; j < COLS - EDGE; j+=VEC_SIZE_U8V) {

        for (k = 0; k < MASKSIZE; k++) {
            // Load in next block of inputs (will overrun a bit)

```

```

        inputs[k] = nextInputs[k];
        nextInputs[k] = LD_U8V(imageIn[(i+k-EDGE)*COLS +
            (j-EDGE+VEC_SIZE_U8V)]);
    }

    // Process a vector-size block of output pixels, one per iter.
    // -- NOTE: MUST be unrolled for vextq_u8!
    for (v=0; v < VEC_SIZE_U8V; v++) {
        // Clear vector of accumulators for one pixel
        acc = zeroVec;
        // Loop over rows of filter window
        for (k = 0; k < MASKSIZE; k++) {
            filter = filters[k];
            // Align and 0-extend 8 columns (for MASKSIZE of <=8)
            inWindow = (int16x8_t) vmovl_u8(vget_low_u8(
                vextq_u8(inputs[k],nextInputs[k],v)));
            // Calculate cols -EDGE to -EDGE+3
            acc = vmlal_s16(acc, vget_low_s16(filter),
                vget_low_s16(inWindow));
            // Calculate cols -EDGE+4 to -EDGE+7
            acc = vmlal_high_s16(acc, filter, inWindow);
        }
        // Reduce, narrow, and store out individual pixels
        imageOut[i*COLS + (j+v)] = vget_lane_u8(          // 3. Store pixel
            vqmovun_s16(vcombine_s16(vqshr_n_s32(          // 2. Narrow
                vcombine_s32(vcreate_s32(
                    vaddvq_s32(acc),s32Extension),          // 1. Horiz +
                    SCALEDN_PIXEL),s16Extension)), 0));      // Constants
    }
}

```

This code is somewhat less efficient than the parallel-pixel filtering, for a few reasons:

- With `MASKSIZE = 7`, only seven of eight parallel lanes are in use in each row, wasting 12.5% of each vector. Although reasonable with this configuration, smaller values of `MASKSIZE` result in lower efficiency: 37.5% waste with `MASKSIZE=5`, and 62.5% waste with `MASKSIZE=3`. Hence, use this strategy when the `MASKSIZE` is almost equal to a vector size. Note that the waste becomes less significant for very large `MASKSIZE` values.
- Because all of the filter weights and input buffers must be held in registers, running out of registers is more likely. (`MASKSIZE=7` is about the limit of what might be register-allocatable.) Alternately, if these are not register allocated, there is more pressure on the load/store unit because the filter weights need to be reloaded into registers for every input pixel instead of being shared across a vector's worth of outputs.
- Instructions are needed at the end to perform all of the horizontal adds needed to complete processing of each pixel. Afterwards, a store is required for each output, instead of only one store for every vector's worth of outputs.

Because of these reasons, parallel-output filtering is often preferable, but per-output filtering may be better in some cases. For example, it is easier to use near the edges of images, where there are not enough pixels available to compute an entire vector's worth of output at once.

3.6.2.3 Per-Pixel Filtering Using DOT Instructions

The `SDOT` and `UDOT` instructions described in [Section 3.5.6, “Dot Product and Matrix Multiply Instructions”](#) allow optimized dot product-and-summation operations of all the pixels in every 32-bit lane of an output vector at once. Because of the design of these instructions, they are more easily applicable to per-pixel filter implementations than parallel-pixel implementations. In fact, only the inner loop of the previous example needs to be replaced:

```
... (previous identical) ...
// Loop over rows of filter window
for (k = 0; k < MASKSIZE; k++) {
    // Process an entire 16-pixel-wide mask at once
    sumVector = vdotq_u32(sumVector, filters[k],
        vextq_u8(inputs[k], nextInputs[k], v));
}
... (rest identical) ...
```

This replaces four vector operations (`EXT`, `SHLL`, and two `MLALS`) with just two (`EXT` and `UDOT`), doubling the maximum possible performance. (Doubling is possible if inputs and filter weights are register-allocated. Otherwise, the load/store unit is likely to become the critical bottleneck first.) The only disadvantage of this routine is that the `UDOT` instruction only allows unsigned filter weights, so filters like edge detection that rely upon signed weights do not work. However, the `USDOT` and `SUDOT` instructions added with `FEAT_I8MM` eliminate this restriction, allowing unsigned pixels to be used with signed weights, for code that only needs to run on more recent chips.



Chapter 4. ISA Optimization: Scalable Matrix Extension (SME)

The SME ISA features instructions that operate on a two-dimensional element array for highly parallel matrix computation or alternatively on multiple concatenated rows of the array for highly parallel vector computation.

SME includes the Streaming Scalable Vector Extension (SSVE), which is a set of long standard vector registers ([longer than those in ASIMD](#), but shorter than those offered in the SME 2D array), predicate registers, and related instructions designed to support SME computation. These SSVE capabilities originated in a somewhat larger feature called the Scalable Vector Extension (SVE), and the essential subset of SVE is included in SME as SSVE. When SME/SSVE is engaged by switching into Streaming SVE mode, SSVE *replaces* ASIMD instructions and registers. Algorithms that benefit from vectors need to be rewritten to use similar instructions in SSVE, or software must exit Streaming SVE mode to re-enable ASIMD. Existing scalar floating point instructions are supported in Streaming SVE mode, but execute in SSVE registers as Streaming Scalar Floating Point (SSFP). And finally, predication serves to activate only specific elements within vectors for various computation and memory instructions.

Apple silicon chips that support Arm FEAT_SME also support the Arm FEAT_SME2 enhancements. SME support debuted in the M4 and A18 families of chips, and the list of specific Arm features support by chip can be found in [Section 2.2, “Arm AArch64 ISA”](#). The FEAT_SME feature flag also indicates support for SSVE (the streaming version of SVE), which as noted above, comes along with SME. Apple silicon does not support non-streaming SVE (FEAT_SVE, FEAT_SVE2, and any further extensions) and those feature flags are not set.

SME also includes additional instructions that operate solely on the SSVE register set but are not available in a non-streaming SVE implementation. Some publications refer to these as SME because they are included with FEAT_SME even though they do not operate on the 2D element array.

Note: For the purpose of this guide, SME refers to instructions that interact with the 2D element array as tiles or vectors, and SSVE (and where appropriate, SSFP) refers to instructions that interact with standard long vector and predicate registers, regardless of whether the instructions originated with non-streaming SVE ISA or were added with FEAT_SME.

Recommendation: Looking ahead to [Chapter 6, SME Engine Microarchitecture Optimization](#), design algorithms to use SME instructions targeting ZA storage and the SME Processing Grid. Use SSVE instructions targeting Z registers and the SSVE Execution Unit in a supporting role. The SME Processing Grid offers considerably higher operation bandwidth. Algorithms that can't utilize the SME engine are typically best executed on the core in the Advanced SIMD and FP execution units.

4.1 SSVE/SME Registers

SSVE and SME offer Z vector registers, P predicate registers, and 2D ZA (Z Array) storage.

The Arm ISA allows CPU implementations to support a range of different Z/P/ZA register lengths called the Streaming Vector Lengths (SVLs). Wherever possible, write SVL-agnostic SSVE/SME code such that it is not dependent on any particular SVL. When needed, use the Read SVL instruction (`RDSVL`) to obtain the vector length in bytes. This instruction can be executed even when not in [Streaming SVE mode](#), but only on chips that offer [FEAT_SME](#). Alternatively, use the element count instruction (`CNT{B,H,W,D} ALL`) to obtain the vector length in terms of element count. The ISA also offers instructions that can adjust pointers and indices according to the SVL, which broadly eliminates the need for hard-coded increment constants.

This guide takes a practical approach, referring to the implementation in Apple silicon by describing sizes based on what is implemented. However, other SVLs may be supported in non-Apple CPUs that support Arm's SME ISA, and sizes scale accordingly, with a minimum SVL of 128b/16B and maximum SVL of 2048b/256B. Likewise, Apple silicon may implement other allowed SVLs in the future.

Recommendation: Apple silicon only supports a SVL of 512b (64B) with a SME 2D array size of 4KiB. However, portable, scalable code should avoid assuming the value of SVL.

4.1.1 SSVE Z Vector Registers (z_n)

The 32 SSVE vector registers (0-31) hold scalar floating point as well as both integer and floating point SIMD values. These are similar to the [Advanced SIMD and FP registers \(\$v_n\$ \)](#) but are 4x longer in Apple silicon at 512 bits (64 bytes). These registers act as a register file for the SSVE vector computations and a staging area for data moving into and out of the [SME 2D ZA storage](#).

Register naming includes the *size of the element*, where the specific data type for that size of element is determined in combination with the instruction.

Like with the Advanced SIMD and FP vector registers, scalar floating point instructions that operate on scalar data elements only access the lower bits of a vector register, ignoring the high bits on a read and clearing them to 0 on a write.

Scalar names for SSVE vector registers are:

- B_n : 8b scalar subset name of z_n (limited use)
- H_n : 16b scalar subset name of z_n
- S_n : 32b scalar subset name of z_n
- D_n : 64b scalar subset name of z_n

There is no native *half register width* capability as there is with ASIMD vector registers (8B versus 16B lengths). Because the SVL defines the length of the SSVE vector registers, the register names simply identify the element size. Rather, *predication* provides an ability to precisely activate or ignore specified elements in a vector (see [Section 4.1.2, "SSVE P and PN Predicate Registers \(\$P\(N\)_n\$ \)"](#) below). In most cases, these element sizes can

apply to both integer and floating point data, where the specific data type is determined in combination with the instruction.

SSVE vector register names by element type are:

- `zn.B`: 8b elements in the 512-bit `zn` vector register
- `zn.H`: 16b elements in the 512-bit `zn` vector register
- `zn.S`: 32b elements in the 512-bit `zn` vector register
- `zn.D`: 64b elements in the 512-bit `zn` vector register
- `zn.Q`: 128b elements in the 512-bit `zn` vector register (limited use)

The vector element data types in the SSVE vector registers are the same as in Advanced SIMD and FP (see [Advanced SIMD and FP Data Types: "Advanced SIMD and FP Data Types"](#)), but some types have more limited use. For types with smaller numbers of bits, instructions more commonly lengthen results to types with larger numbers of bits. In summary:

- **"Brain" float precision (`bfloat16`):** These types are primarily used to reduce memory footprint, where `bfloat16` elements are typically lengthened before consumption by multiply-accumulate operations that result in single precision (`float32`) accumulated elements. The ISA also supports conversions in and out of `bfloat16`, but there are no computation instructions that both read and write `bfloat16`.
- **Half precision (`float16`):** For SSVE, the ISA includes a full selection of all-half-precision SSVE operations, plus lengthening multiply-accumulates with single precision (`float32`) accumulators. SME multiply-accumulate operations only come in lengthening versions.
- **Single precision (`float32`) and double precision (`float64`):** For these base floating point types, the ISA features a broad selection of operations.
- **Integers with SSVE:** The ISA generally offers full support for 8/16/32/64-bit signed and unsigned integers across SSVE vector instructions. Multiplication instructions, though, often omit some 8-bit and 64-bit forms. Lengthening operations typically extend 8/16/32-bit inputs up by one step (sometimes two) to 16/32/64-bit results, while narrowing operations do the reverse.
- **Complex numbers:** SSVE supports both floating-point and integer versions of complex number vectors with interleaved real and imaginary components, across a range of types. SME generally does not.

Like Advanced SIMD and FP, some instructions perform operations on groups of two or four Z vector registers, and are referred to as *multi-vectors*. However, the individual Z vector registers that comprise a multi-vector are often different than with ASIMD. In Arm notation, `<z1> ... <z4>` refers to the first through fourth Z register in the multi-vector, which must be populated with actual Z registers according to a specific pattern. Check each instruction in the Arm ISA reference manual for the pattern used. In summary:

- **Consecutive arbitrary { `<z1>.<T>`, `<z2>.<T>`, `<z3>.<T>`, `<z4>.<T>` }:** These follow the ASIMD pattern of four consecutive registers, starting anywhere in the register file, and wrapping around to register 0, as applicable. For example, SSVE instruction `LD4H` (scalar plus immediate), and SME instruction `ADD` (array results,

multiple and single vector) use this pattern. The assembler may also accept the range notation below.

- Consecutive aligned** { <Z1>.<T>-<Z4>.<T> }: This notation often (but not always) indicates that the registers are required to be *aligned*, that is, the first register is a multiple of 2 for the 2 register variant and a multiple of 4 for the 4 register variant. For example, SME instruction MOV (array to vector, four registers) uses this pattern. The assembler may also accept a list of comma-separated registers that follow this pattern, like the syntax of the consecutive arbitrary pattern. The allowed starting register is as follows:
 - 2 register variant: Z0, Z2, Z4, etc.
 - 4 register variant: Z0, Z4, Z8, etc.
- Strided** { <Z1>.<T>, <Z2>.<T>, <Z3>.<T>, <Z4>.<T> }: These have a limited starting register number that is increased by 8 for the second register in the 2 register variant, and by 4 for each of the subsequent registers in the 4 register variant. For example, SSVE instruction LD1H (scalar plus immediate, strided registers) uses this pattern. The allowed starting register numbers are as follows:
 - 2 register variant: Z0-Z7 and Z16-Z23
 - 4 register variant: Z0-Z3 and Z16-Z19

When used in intrinsics in high level languages, the Z registers are represented by a data type that indicates the element type, such as `svfloat32_t` for a SSVE vector of 32b single-precision floating point elements. The overall size of this intrinsic data type is not fixed at compile time, but is rather a function of the SVL. In Apple silicon, the SVL is 512b/64B and thus, this register contains 16 elements of the type `float32_t`.

The Vector Register File also includes the special ZT0 register that is used for the [Lookup Table Instructions](#).

See notes in [Section 4.2, “SVE Streaming and ZA Modes”](#) on how to determine which SVE instructions are available in SSVE.

4.1.2 SSVE P and PN Predicate Registers (P(N)n)

The 16 SSVE predicate registers (0-15) activate (or govern) elements in a vector instruction. The length of each predicate register scales by SVL, one bit per byte of SVL, and are thus 64b on Apple silicon. There are a number of different uses for predicates, but they all prevent an operation from performing the action on an element where its predicate is `FALSE`. Commonly, governing predicates activate individual elements in parallel if-then-else flows as well as manage the effective length of a vector. Consider the following instruction:

```
ABS <Zd>.<T>, <Pg>/M, <Zn>.<T>
```

This instruction takes the integer absolute value of signed elements in the <Zn> source register with elements of type <T>, but only performs the operation on elements for which <Pg> is `TRUE`, writing them into destination register <Zd>.

The /M or /Z notation following the governing predicate controls the behavior of the destination register in vector lanes that are disabled by predication, that is where the

predicate associated with the element is `FALSE`. Instructions using `/Z` predication clear (write 0) the disabled destination vector elements, while instructions using `/M` predication preserve (or merge) the existing destination vector elements.

Note that all `/M` instructions must read their destination register, even if the instruction would not otherwise need to, in order to get the existing contents for disabled elements, while `/Z` instructions do not. Predicated load instructions always use `/z` operation, while predicated computation instructions usually use `/M`. Predicated stores often do not require this notation because they do not touch memory for vector lanes associated with a `FALSE` predicate. (Hence, this is effectively a form of `/M` behavior.)

Instructions that manipulate (or alter) predicate registers have sources and destinations that follow the normal naming patterns of `<Pn>` and `<Pd>` for sources and destinations, etc. A few predicate manipulation instructions have a governing predicate of their own, such that operations on the input predicate only apply to bits where the governing predicate is `TRUE`, and the other bits are merged or zeroed.

To support these and other uses, there are two different types of predicates that map onto the same register file:

- **Bit mask predicate (`Pn`):** This style is the conventional way of using predicates. The 64-bit P registers have one predicate bit per byte in the 64-byte Z vector registers, with each bit activating the corresponding byte in a byte-element operation. For Z vector registers with elements larger than one byte, the predicate bit associated with the *lowest byte* governs the operation on the associated element. See [Figure 4.1: “Bit Mask Predicate \(`Pn`\)”](#) for an illustration of the register layout with various data element sizes. Because this is the original implementation of predication in the Arm ISA, Arm does not have a particular name for this style of predication. To clearly differentiate from the other style (counted predicate, described below), this guide uses the term *bit mask predicate*.

The advantage of this style is that predication can govern each vector element independently, allowing algorithms to use predication to implement parallel if-then-else sequences, as shown in [Section 4.4.2, “Element Activation Control”](#), in addition to algorithms that need merely a vector length. However, the 64-bit length of the P registers limits this to operations that consume just single vectors (not multi-vectors). Most instructions that use P registers in this mode can only specify registers `P0-P7` (due to opcode encoding limitations, using the lower half of the register file), but a few can access any P register in this style.

- **Counted predicate (`PNn`):** For longer, 128B (two-vector) or 256B (four-vector) instructions, there are insufficient bits in the 64-bit predicate registers to map some element types 1:1 with predicate bits. As a result, multi-vector instructions use P registers in an alternate *predicate-as-counter* or PN configuration that can only specify masks of N contiguous `TRUE` elements from one end of the vector or the other. The PN encoding includes both an element count and an element size which, internally, is converted into a P bit mask predicate when consumed according to the *predicate's* encoded element size. The full list of possible options is shown in [Table 4.1: “Counted Predicate \(`PNn`\)”](#). Although Arm uses the term *predicate-as-counter* for this style, this guide uses the term *counted predicate* to simplify the terminology.

This style is sufficient for masking off unneeded vector lanes, for example, during a last loop iteration while processing array rows or columns. Most instructions that use

Apple Silicon CPU Optimization Guide: 4.0

ISA Optimization: Scalable Matrix Extension (SME)

SSVE P and PN Predicate Registers (P (N) n)

P registers in this mode can only specify registers `PN8–PN15` (due to opcode encoding limitations, using the upper half of the register file to avoid conflicts with bit mask predicates). When generating a counted predicate from a `WHILE` instruction, `VLx2` or `VLx4` must be specified to indicate whether the consumers are two-vector or four-vector instructions. Finally, regardless of SVL, PN predicates always encode the element count and element size in the bottom 16 bits of the predicate register, regardless of the register's actual length.

Note that “all FALSE” has the same representation for both types of predicates: the value 0. However, “all TRUE” is represented differently between the two types, and the values are not interchangeable. And, consumer operations do not check that the governing predicate value was generated for an element size consistent with the consuming operation (most notably for counted predicates where the element size is present in the encoding). For example, `ST1B { <Zt1>.B, <Zt2>.B }, <PNg>, [<Xn|SP>{, #<imm>, MUL VL}]`, which uses byte elements does not check that the governing counted predicate `<PNg>` value has bit 0 equal to 1.

Figure 4.1. Bit Mask Predicate (Pn)

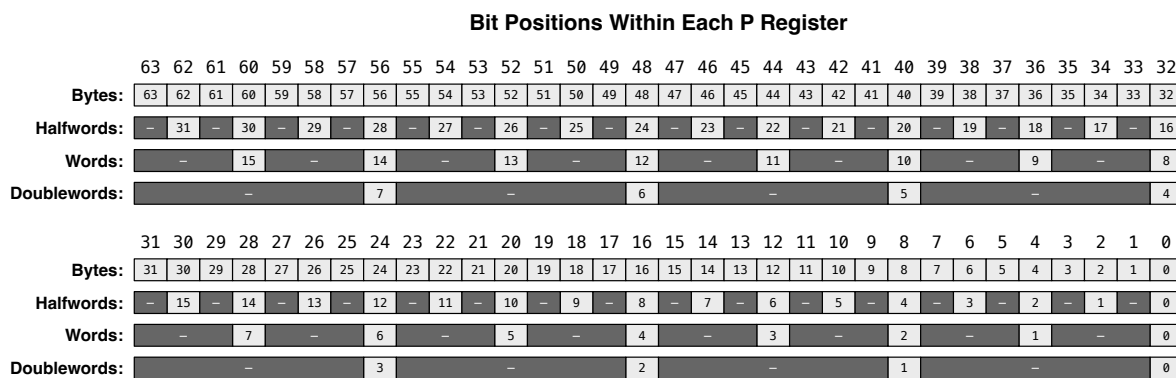


Table 4.1. Counted Predicate (PNn)

Type	b15	b14:4	b3	b2	b1	b0	Description
Clear	0	0	0	0	0	0	All vector lanes off
Byte	0	count				1	Bytes 0 to (count–1) active, remainder inactive
	1	count				1	Bytes 0 to (count–1) inactive, remainder active
Half	0	count			1	0	Halfwords 0 to (count–1) active, remainder inactive
	1	count			1	0	Halfwords 0 to (count–1) inactive, remainder active
Word	0	count		1	0	0	Words 0 to (count–1) active , remainder inactive
	1	count		1	0	0	Words 0 to (count–1) inactive , remainder active

Table 4.1. Counted Predicate (PNn) (cont.)

Type	b15	b14:4	b3	b2	b1	b0	Description
Double	0	count	1	0	0	0	Doublewords 0 to (count-1) active, remainder inactive
	1	count	1	0	0	0	Doublewords 0 to (count-1) inactive, remainder active

Section 4.4, “[Predication and Loop Control](#)” presents an overview of the instructions that manipulate predicates.

When used in intrinsics in high level languages, the predicate registers are represented by types `svbool_t` and `svcount_t`. Their size is dependent on the SVL, but is 64b in Apple silicon.

4.1.3 SME ZA Storage

The SME 2D Z Array (ZA) storage is a square array that is SVL(bytes) x SVL(bytes). Thus the total array storage is SVL(bytes)². In Apple silicon, with an SVL of 64B, ZA storage is 4KiB in size. Fundamentally, it supports data types of two-dimensional tiles or one-dimensional long vectors in the form of sets of concatenated rows. Data can be moved into or out of tiles as horizontal or vertical sets of rows. SME storage and instructions are centered around implementing and optimizing forms of multiply-accumulate, and offer the same element data types as SSVE.

4.1.3.1 ZA Tile Registers (`zA<n>.<size>`)

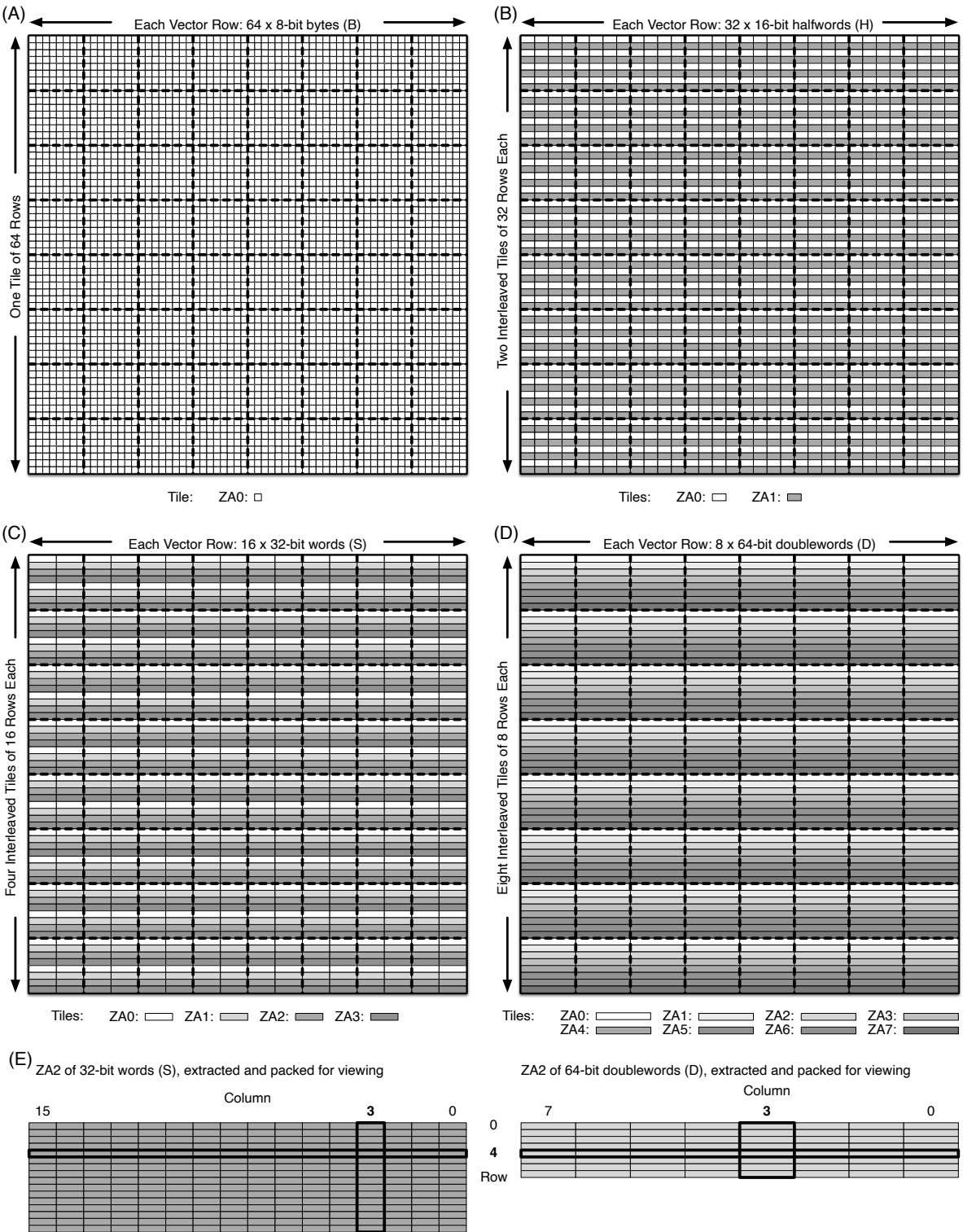
The ZA storage is commonly accessed as a set of two-dimensional tile registers, where the number of available registers depends on the element size. The size names follow the same pattern as for the Z vectors, where the specific data type is determined in combination with the instruction. The tile registers largely act as accumulators for 2D multiply instructions. For 8-bit (byte, B) element sizes, there is a single tile, 64 rows of 64 x 1-byte (B) elements, as depicted in [Figure 4.2: “Layout of the ZA storage”](#) (A). For 16-bit (halfword, H) element sizes, a square tile consists of 32 x 2-byte (H) elements per row requiring 32 rows. Because the full tile register file has 64 total rows, there is room for two tiles (selected by <n>), as depicted in (B). Individual tiles are *interleaved* by rows in the 64 rows x 64B full array, rather than each tile being packed contiguously one tile after another. Similarly, for 32-bit (single, S) element sizes, a square tile consists of 16 elements per row requiring 16 rows, and thus there is room for four interleaved tiles, as depicted in (C). And, for 64-bit (double, D) element sizes, there is room for eight interleaved tiles, as depicted in (D). Last (not shown), the ISA supports 16 tiles of 128-bit (quadword, Q) elements; however, there are no computation instructions that operate on such elements. They are used only for special purposes beyond the scope of this guide and not discussed further. Note that the number of tiles for a particular element size does not change for different values of SVL.

In [Figure 4.2: “Layout of the ZA storage”](#) (E), the `zA2` tile from (C) (32-bit words (S)) and the `zA2` tile from (D) (64-bit doublewords (D)) are extracted and packed together for convenient viewing. Horizontal vector *slice* (row) number 4 (noted as `zA2H[4]`, counted from the top) is highlighted in each. `zA2H[4].s` maps to row 18 of the byte-oriented view of the ZA storage (A), while `zA2H[4].d` maps to row 34 of the byte-oriented view

of the ZA storage. Vertical slice (column, or element) number 3 (noted as `zA2V[3]`, counted from the right) is also highlighted in each. `zA2V[3].s` maps to bytes 12 to 15, are `zA2V[3].D` maps to bytes 24 to 31. Lastly, because of the row aliasing between element types, software that uses `zA3.D` (byte rows 3, 11, 19, 27, 35, 43, 51, 59) or `zA7.D` (byte rows 7, 15, 23, 31, 39, 47, 55, 63) could not simultaneously use `zA3.s` (byte rows 3, 7, 11, 15, 19, 23, 27, 31, 35, 39, 43, 47, 51, 55, 59, 63). Horizontal access is intrinsic to most instructions. Vertical access is always embedded in either the instruction mnemonic (for the few computations that support vertical operands) or the ZA operand (for `mov` instructions).

The dark dashed lines in the figure depict 8 row x 8B portions of the array, and are helpful for understanding how these bytes are interpreted differently depending on the data element type. Looking ahead to the microarchitecture ([Chapter 6, SME Engine Microarchitecture Optimization](#)), the interleaving of the tiles in this manner ensures that all tiles (regardless of element type) are distributed across an 8 x 8 grid of compute processing elements.

Figure 4.2. Layout of the ZA storage



4.1.3.2 ZA Vector Registers (ZA.<type>[row(s)])

Beyond 2D array tiles, ZA storage can also serve as a set of vector registers. Because the number of rows in the ZA storage is dependent on SVL, the number of vectors is

also scalable with SVL. When used with the capabilities described below, ZA vectors offer more parallelism than Z vectors, and execute with significantly higher bandwidth (See [Section 6.3, “SME Processing Grid Optimization”](#) for more information). Vector row selection is performed through an index register (from the core's general purpose register file) and an immediate offset: `<Wv>, <offs>`. The row is selected based on contents of the `Wv` integer register plus the small immediate offset modulo the number of registers. This access methodology allows access to the ZA register file without unrolled loops with immediate-only row selection. For some instructions, as shown below, `<offs>` can be a range of offsets. In Apple silicon, with an SVL of 64B, there are 64 vectors.

Writing portable, scalable, and performant code for ZA vectors is more complex than for ZA tiles. Truly portable code must assume the minimum SVL (16 bytes), which translates to only 16 vectors. Because of the modulo operation in the indexing operation, indices 0 and 16 alias to the same row for the smallest SVL but separate rows for larger SVLs. The highest performing code on Apple silicon uses all 64 rows to expose the most parallelism.

Note: In ZA vector code used on multiple platforms, verify that the number of rows (the SVL) in the code is less than or equal to that provided by the hardware implementation.

Few instructions, primarily loads and store, access ZA storage as 64 x 64B vectors.

```
LDR ZA[<Wv>, <offs>], [<Xn|SP>{, #<offs>, MUL VL}] // Load vector to ZA storage
STR ZA[<Wv>, <offs>], [<Xn|SP>{, #<offs>, MUL VL}] // Store vector from ZA storage
```

When computation operations must work with just 64B inputs and outputs, the SSVE versions of the operations that operate on the 64B Z registers are more efficient. The SME vector computation instructions using ZA vectors can accelerate the computation by accessing up to 16 x 64B rows of ZA with each instruction, spreading computation across rows in two distinct ways.

First, some instructions consume Z vector source registers, lengthen (or double lengthen) the element size, and produce two (or four) consecutive ZA vectors. These are referred to as "double vectors" (or "quad vectors"). For example, the `FMLSL` instruction below consumes vectors `zn` and `zm` with 16b half-precision floating point "H" elements and accumulates them into 32b floating point single-precision "S" elements. Likewise, the `SMLSLL` instruction lengthens the elements by a factor of four, byte "B" to 4-byte "S" elements. (Understanding the specific operation in the following examples is not needed to understand the ZA storage access patterns.)

```
// Double-vector 16b floating-point multiply-sub long by vector (H-to-S)
FMLSL ZA.S[<Wv>, <offsf>:<offsl>], <zn>.H, <zm>.H
// Quad-vector signed 16b int multiply-sub long long by vector (H-to-D)
SMLSLL ZA.D[<Wv>, <offsf>:<offsl>], <zn>.H, <zm>.H
```

Looking at the `FMLSL` above, the two output ZA vectors are determined by adding the contents of the `Wv` register with the two offsets, first (`offsetf`) and last (`offsetl`), where they are +1 apart. Note that the ISA requires the pair to start on an even row number. If the pair starts on an odd row number, 1 is subtracted from the computed row number to obey the property (`access_row = row - (row % 2)`). When there are four output ZA vectors, such as with the `SMLSLL` above, the first and last offset must be +3 apart,

and the ISA requires the quad to start on a row number that is a multiple of 4, adjusting down if necessary ($\text{access_row} = \text{row} - (\text{row} \% 4)$). [Figure 4.3: "ZA Tile and Vector Register Mapping for Computation Instructions"](#) depicts the mapping of rows into the array: "Lengthening (double-vector) / Single" and "Double Lengthening (quad-vector) / Single" columns compared with the "Normal / Single" column.

Second, there are operations that combine vector rows to make 2x or 4x longer vectors (more elements) called *vector groups*: VGx2 and VGx4 . With these versions, the number of indexable ZA vector registers is cut in half or cut by a factor of 4, offering just 32 or 16 vectors. To support the longer vectors, for VGx2 , the ZA storage is correspondingly cut in half horizontally, where the indexed row contains the lower 64B section of the vector and the indexed row +32 contains the higher 64B section of the vector. Similarly for VGx4 , the ZA storage is cut into four contiguous pieces horizontally, where the indexed row contains the lower 64B section, and the next three sections of 64B are at +16, +32, and +48 rows. Stated another way, for VGx2 , the lower 64B is stored in $\text{access_row} = (\text{Wv} + \text{offs}) \% 32$, and similarly $\% 16$ for VGx4 and even simply $\% 64$ for the baseline with no vector grouping. See "Normal / VGx2 " and "Normal / VGx4 " columns compared with the base "Normal / Single" column in [Figure 4.3: "ZA Tile and Vector Register Mapping for Computation Instructions"](#). The following instructions perform a multiply of a two (or four) first source Z vector with the corresponding elements of a second source Z vector, subtracting that result from two (or four) ZA single-vector groups (VGx2 , or VGx4).

```
// Two Single-vector Group 32b floating-point multiply-sub by vector (S-to-S)
FMLS ZA.S[<Wv>, <offs>{, VGx2}], { <Zn1>.S-<Zn2>.S }, <Zm>.S
// Four Single-vector Group 32b floating-point multiply-sub by vector (S-to-S)
FMLS ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.S-<Zn4>.S }, <Zm>.S
```

And lastly, both methods can be combined to generate lengthened pairs of results in two (long) and four (long long) consecutive registers that are also 2x (VGx2) or 4x (VGx4) longer in number of elements: one, two, or four ZA [single]-, double-, or quad-vector groups. See additional columns in [Figure 4.3: "ZA Tile and Vector Register Mapping for Computation Instructions"](#).

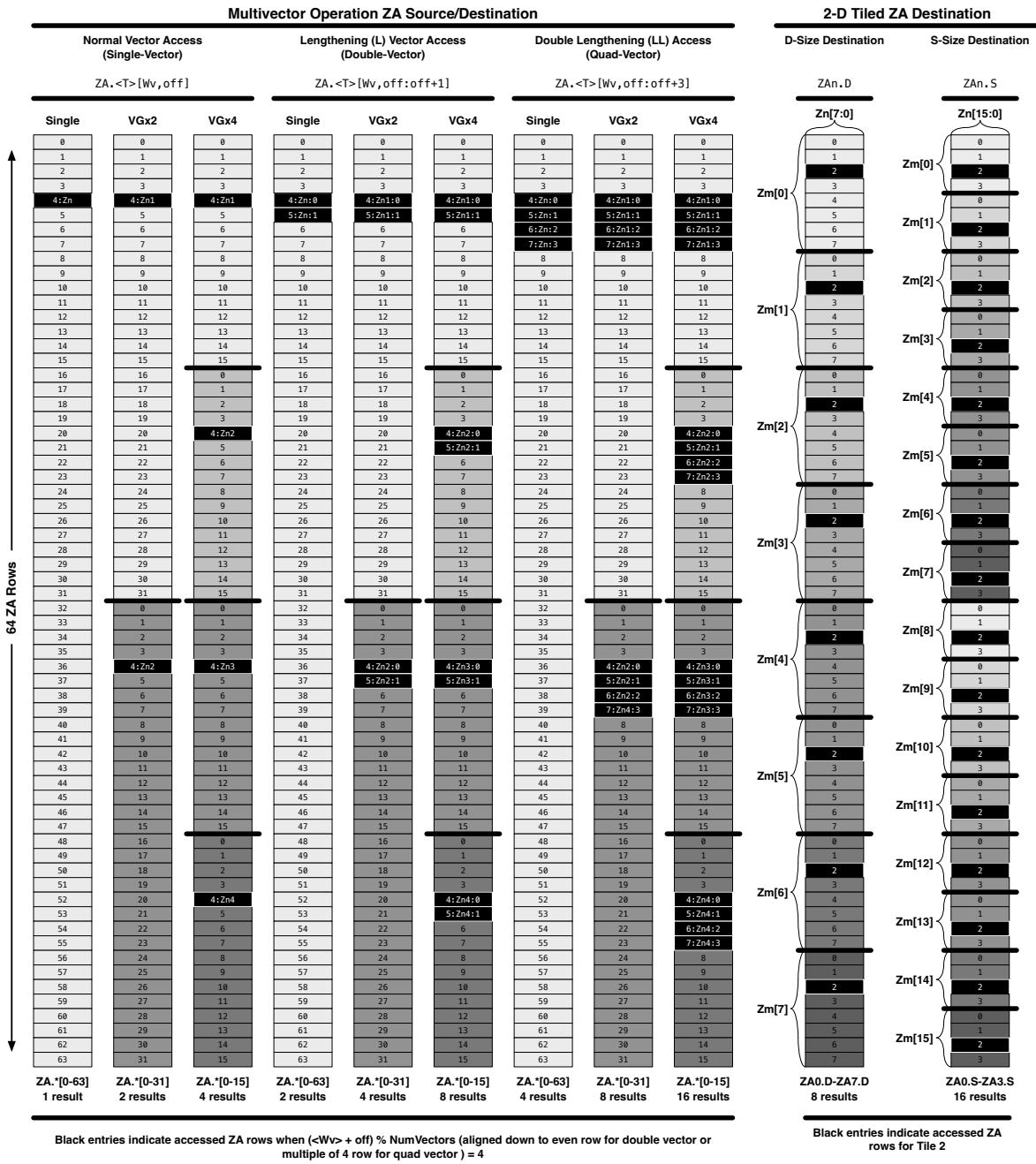
```
// Two Double-vector Group 16b floating-point multiply-sub long by vector (H-to-S)
FMLS ZA.S[<Wv>, <offs>f:<offs1>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H
// Four Double-vector Group 16b floating-point multiply-sub long by vector (H-to-S)
FMLS ZA.S[<Wv>, <offs>f:<offs1>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H
// Two Quad-vector Group signed 16b int multiply-sub long long by vector (H-to-D)
SMLS ZA.D[<Wv>, <offs>f:<offs1>{, VGx2}], { <Zn1>.H-<Zn2>.H }, <Zm>.H
// Four Quad-vector Group signed 16b int multiply-sub long long by vector (H-to-D)
SMLS ZA.D[<Wv>, <offs>f:<offs1>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H
```

As is more visible in upcoming [Figure 4.3: "ZA Tile and Vector Register Mapping for Computation Instructions"](#) and [Figure 4.4: "ZA Tile and Vector Register Mapping for Data Movement Instructions"](#), ZA tiles and ZA vectors all alias to the same ZA storage rows. Understanding this aliasing is important for both code correctness and for managing row reuse hazards. See [Section 6.3, "SME Processing Grid Optimization"](#) for more information.

4.1.4 ZA Storage Addressing Modes

ZA storage access for *computation* instructions is shown below in Figure 4.3: “ZA Tile and Vector Register Mapping for Computation Instructions”. The black cells are highlighted according to the specific access described at the bottom of the figure.

Figure 4.3. ZA Tile and Vector Register Mapping for Computation Instructions



Data *computational* instructions use the memory addressing modes described in Table 4.2: “Addressing Modes for ZA Storage for Computational Instructions”.

Table 4.2. Addressing Modes for ZA Storage for Computational Instructions

Addressing	Description	Notes	Base Instructions
$ZA.<T>[Wv, <imm>\{, VGx2/4\}]$	1 ZA single vector as 1, 2, or 4 rows, depending on the length of the group	$<Wv>=W8-W11$ $<imm>=0-7$	ADD SUB DOT FADD FSUB FMLA FMLS
$ZA.<T>[Wv, <imm>:<imm+1>\{, VGx2/4\}]$	1 ZA double vector as 2, 4, or 8 rows, depending on the length of the group	$<Wv>=W8-W11$ $<imm>=0-\{14,6,6\}$ in multiples of 2, depending on number of groups	MLAL MLSL
$ZA.<T>[Wv, <imm>:<imm+3>\{, VGx2/4\}]$	1 ZA quad vector as 4, 8, or 16 rows, depending on the length of the group	$<Wv>=W8-W11$ $<imm>=0-\{12,4,4\}$ in multiples of 4, depending on number of groups	MLALL MLSLL
$ZAn.<T>$	1 tile	$<T>=S: n=0-3$ $<T>=D: n=0-7$	ADDHA ADDVA MOP
$\{ZAn.D, \dots\}$ list or $\{ZA\}$	List of tiles up to full array	Tile list up to full array	ZERO

Figure 4.4: “ZA Tile and Vector Register Mapping for Data Movement Instructions” shows the ZA storage accesses for *movement* instructions. The black cells are highlighted per the access described at the bottom of the figure.

Figure 4.4. ZA Tile and Vector Register Mapping for Data Movement Instructions

LDR/STR	Multivector Src/Dst		Tiled ZA Source/Destination				
Single	VGx2	VGx4	Q-Size Tiles	D-Size Tiles	S-Size Tiles	H-Size Tiles	B-Size Tiles
ZA [Wv, off]	ZA.D [Wv, off]		ZAn.Q [Ws, off]	ZAn.D [Ws, off]	ZAn.S [Ws, off]	ZAn.H [Ws, off]	ZA0.B [Ws, off]
0	0	0	0:0	0:0	0:0	0:0	0:0
1	1	1	1:0	1:0	1:0	1:0	0:1
2:Zn	2:Zn1	2:Zn1	2:0	2:0	2:0	0:1	0:2
3	3	3	3:0	3:0	3:0	1:1	0:3
4	4	4	4:0	4:0	0:1	0:2	0:4
5	5	5	5:0	5:0	1:1	1:2	0:5
6	6	6	6:0	6:0	2:1	0:3	0:6
7	7	7	7:0	7:0	3:1	1:3	0:7
8	8	8	8:0	0:1	0:2	0:4	0:8
9	9	9	9:0	1:1	1:2	1:4	0:9
10	10	10	10:0:Zn1	2:1	2:2:Zn1	0:5	0:10:Zn1
11	11	11	11:0	3:1	3:2	1:5	0:11:Zn2
12	12	12	12:0	4:1	0:3	0:6	0:12:Zn3
13	13	13	13:0	5:1	1:3	1:6	0:13:Zn4
14	14	14	14:0	6:1	2:3:Zn2	0:7	0:14
15	15	15	15:0	7:1	3:3	1:7	0:15
16	16	0	0:1	0:2	0:4	0:8	0:16
17	17	1	1:1	1:2	1:4	1:8	0:17
18	18	2:Zn2	2:1	2:2:Zn1	2:4:Zn3	0:9	0:18
19	19	3	3:1	3:2	3:4	1:9	0:19
20	20	4	4:1	4:2	0:5	0:10:Zn1	0:20
21	21	5	5:1	5:2	1:5	1:10	0:21
22	22	6	6:1	6:2	2:5:Zn4	0:11:Zn2	0:22
23	23	7	7:1	7:2	3:5	1:11	0:23
24	24	8	8:1	0:3	0:6	0:12:Zn3	0:24
25	25	9	9:1	1:3	1:6	1:12	0:25
26	26	10	10:1:Zn2	2:3:Zn2	2:6	0:13:Zn4	0:26
27	27	11	11:1	3:3	3:6	1:13	0:27
28	28	12	12:1	4:3	0:7	0:14	0:28
29	29	13	13:1	5:3	1:7	1:14	0:29
30	30	14	14:1	6:3	2:7	0:15	0:30
31	31	15	15:1	7:3	3:7	1:15	0:31
32	0	0	0:2	0:4	0:8	0:16	0:32
33	1	1	1:2	1:4	1:8	1:16	0:33
34	2:Zn2	2:Zn3	2:2	2:4:Zn3	2:8	0:17	0:34
35	3	3	3:2	3:4	3:8	1:17	0:35
36	4	4	4:2	4:4	0:9	0:18	0:36
37	5	5	5:2	5:4	1:9	1:18	0:37
38	6	6	6:2	6:4	2:9	0:19	0:38
39	7	7	7:2	7:4	3:9	1:19	0:39
40	8	8	8:2	0:5	0:10	0:20	0:40
41	9	9	9:2	1:5	1:10	1:20	0:41
42	10	10	10:2:Zn3	2:5:Zn4	2:10	0:21	0:42
43	11	11	11:2	3:5	3:10	1:21	0:43
44	12	12	12:2	4:5	0:11	0:22	0:44
45	13	13	13:2	5:5	1:11	1:22	0:45
46	14	14	14:2	6:5	2:11	0:23	0:46
47	15	15	15:2	7:5	3:11	1:23	0:47
48	16	0	0:3	0:6	0:12	0:24	0:48
49	17	1	1:3	1:6	1:12	1:24	0:49
50	18	2:Zn4	2:3	2:6	2:12	0:25	0:50
51	19	3	3:3	3:6	3:12	1:25	0:51
52	20	4	4:3	4:6	0:13	0:26	0:52
53	21	5	5:3	5:6	1:13	1:26	0:53
54	22	6	6:3	6:6	2:13	0:27	0:54
55	23	7	7:3	7:6	3:13	1:27	0:55
56	24	8	8:3	0:7	0:14	0:28	0:56
57	25	9	9:3	1:7	1:14	1:28	0:57
58	26	10	10:3:Zn4	2:7	2:14	0:29	0:58
59	27	11	11:3	3:7	3:14	1:29	0:59
60	28	12	12:3	4:7	0:15	0:30	0:60
61	29	13	13:3	5:7	1:15	1:30	0:61
62	30	14	14:3	6:7	2:15	0:31	0:62
63	31	15	15:3	7	3:15	1:31	0:63
ZA [0-63] 1 vector dest only	ZA.D [0-31] 2 vector dest only	ZA.D [0-15] 4 vector dest only	ZA0.Q-ZA15.Q [0-3] 1/2/4 vectors	ZA0.D-ZA7.D [0-7] 1/2/4 vectors	ZA0.S-ZA3.S [0-15] 1/2/4 vectors	ZA0.H-ZA1.H [0-31] 1/2/4 vectors	ZA0.B [0-63] 1/2/4 vectors
Black entries indicate accessed ZA rows when (<Wv> + off) % NumVectors = 2			Black entries indicate accessed ZA rows for 4 vector dest case when (<Wv> + off) % NumRowsInTile: Q-sized: Tile 10 starting at row 0 D- and S-sized: Tile 2 starting at row 2 H- and B-sized: Tile 0 starting at row 10				

Data movement instructions use the addressing modes described in Table 4.3: "Addressing Modes for ZA Storage for Data Movement Instructions". Load, store, and

movement instructions use different versions of the addressing modes because they only operate on small groups of 64B vectors or 64B slices of tiles. Put another way, the full data parallelism available to individual compute instructions across the long vectors and tiles in the ZA storage is not accessible to individual load, store, and movement instructions. First, rows in the ZA storage can be loaded and stored directly by row number. For tiles, one, two, or four contiguous rows (or columns) can be accessed from a specified tile. For vector groups (long vectors), data can be moved between one, two, or four Z vector registers accordingly by indexing the row number associated with the first 64B of the long vector. There is no support for accessing the multiple 64B vectors in double and quad vectors as a group (the result of lengthening), rather these are indexed as if they are single vectors of the larger type. However, `ZA0H.B` *tile* access variations work well to read out the results of lengthening and double-lengthening *vector* operations because the tile accesses operate on groups of adjacent vector rows in the ZA storage.

Most of these instructions use `w12–w15` for their offset register, so they do not conflict with the `w8–w11` commonly used for the computation instructions. As a result, software algorithms can usually use eight integer registers to hold ZA storage positions, with distinct sets of computation and movement offsets instead of just four.

Table 4.3. Addressing Modes for ZA Storage for Data Movement Instructions

Addressing	Description	Notes	Base Insts.
<code>ZA[Wv, <imm>]</code>	1 row	<code><Wv>=W12–W15</code> <code><imm>=0–15</code>	LDR STR
<code>ZA.D[Wv, <imm>{, VGx2/4}]</code>	1 ZA vector as 1, 2, or 4 rows, depending on the length of the group	<code><Wv>=W8–W11</code> <code><imm>=0–7</code>	MOV MOVA
<code>ZAn<HV>.<T>[Ws, <imm>]</code>	1 slice of a tile	<code><Wv>=W12–W15</code> <code><T>=B: n=0, <imm>=0–15</code> <code><T>=H: n=0–1, <imm>=0–7</code> <code><T>=S: n=0–3, <imm>=0–3</code> <code><T>=D: n=0–7, <imm>=0–1</code> <code><T>=Q: n=0–15, <imm>=0</code>	LD1* ST1* MOV MOVA
<code>ZAn<HV>.<T>[Ws, <imm>:<imm+1>]</code>	2 contiguous slices of a tile	<code><Wv>=W12–W15</code> <code><T>=B: n=0, <imm>=0–14 (mult. of 2)</code> <code><T>=H: n=0–1, <imm>=0–6 (mult. of 2)</code> <code><T>=S: n=0–3, <imm>=0–2 (mult. of 2)</code> <code><T>=D: n=0–7, <imm>=0</code>	MOV MOVA
<code>ZAn<HV>.<T>[Ws, <imm>:<imm+3>]</code>	4 contiguous slices of a tile	<code><Wv>=W12–W15</code> <code><T>=B: n=0, <imm>=0–12 (mult. of 4)</code> <code><T>=H: n=0–1, <imm>=0–4 (mult. of 4)</code> <code><T>=S: n=0–3, <imm>=0</code> <code><T>=D: n=0–7, <imm>=0</code>	MOV MOVA

The `MOVA` instructions move Z vectors in and out of the ZA storage in either a horizontal orientation (normal vector rows) or vertical orientation (vector columns). The vertical `MOVA` instructions allow software to read out matrices in transposed form, as is demonstrated in [Section 4.7.2, “SME Matrix Transposition Example”](#). Because many inputs to SME matrix calculations must be in transposed form instead of normal form, reading out a result matrix directly into transposed form can save having to perform a dedicated transposition operation later. Meanwhile, the vertical `VDOT` instructions allow certain computations to be performed on data that is only available in transposed form.

There are no instructions that access more than one ZA tile or ZA vector in one instruction, thus there are no arithmetic, logic, or copy instructions between ZA tiles or between ZA vectors.

4.2 SVE Streaming and ZA Modes

Engaging SSVE/SME requires switching into Streaming SVE mode before executing SSVE/SME code. When in Streaming SVE mode:

- Advanced SIMD instruction set and v_n registers are unavailable. The contents of the v_n registers are zeroed when entering or exiting Streaming SVE mode. Executing Advanced SIMD instructions result in an illegal instruction exception.
- SSVE vector z_n registers and predicate p_n registers, along with the SSVE instructions, are available. They are zeroed when entering or exiting Streaming SVE mode.
 - Not all Arm ISA instructions are available in Streaming SVE mode. Advanced SIMD is not supported, but many ASIMD instructions have counterparts in non-streaming SVE with their own opcode encodings, many of which are also available in SSVE. The Arm ISA reference manual annotates which SVE instructions are not supported in Streaming SVE mode with the following: "This instruction is illegal when executed in Streaming SVE mode, unless FEAT_SME_FA64 is implemented and enabled." Apple silicon does not support FEAT_SME_FA64 and thus [sysctl query](#) for FEAT_SME_FA64 (Full Streaming SVE mode instructions) returns -1 or a queried value of 0.
- Scalar Floating Point is available as Streaming Scalar Floating Point (SSFP). The lowest numbered bits of the SSVE vector registers, z_n , also hold the scalar registers. SSFP opcode encodings are the same between non-Streaming SVE mode and Streaming SVE mode.
- Instructions that operate on SME ZA storage are also available when ZA mode is enabled (see below).

Executing SSVE and SME instructions outside of Streaming SVE mode results in an illegal instruction exception.

Because of the size and inherent cost to save and restore the SME two-dimensional ZA storage, the ISA features a separate mode control called ZA mode for enabling or disabling the array. ZA storage is zeroed when exiting ZA mode. Note that only two instructions can operate on ZA storage when in ZA mode but not in Streaming SVE mode: `LDR (vector)` to load into the storage and `STR (vector)` to store out from storage.

Table 4.4. ISA and Register Availability in Non-Streaming Mode and Streaming Mode

	Class	Non-Streaming Mode [†] (PSTATE.SM=0)	Streaming Mode (PSTATE.SM=1)
ISA	Scalar FP	Yes	Yes, as Streaming Scalar FP (SSFP)
	Advanced SIMD	Yes	No
	Streaming SVE (SSVE)	No	Yes
	SME	No	Yes, when also in ZA mode
State	Scalar FP Registers	Yes, as subsets of the V registers	Yes, as subsets of the Z registers (SSFP)
	Advanced SIMD Registers	Yes	No
	SME/SSVE Z/P Registers	No	Yes
	SME ZA storage	According to ZA mode (PSTATE.ZA)	

[†]Note that Apple silicon does not support SVE in non-Streaming SVE mode.

Applications can switch dynamically between non-Streaming and Streaming SVE modes, and can dynamically enable and disable ZA storage, both via PSTATE.{SM, ZA}:

- SM: Enable and disable Streaming SVE mode
- ZA: Enable and disable ZA mode

Use the MSR instruction to modify the PSTATE.{SM, ZA} bits in the Streaming Vector Control Register (SVCR) as follows:

- MSR SVCRSM, #<imm>; #0 to disable Streaming SVE mode, #1 to enable
- MSR SVCRZA, #<imm>; #0 to disable ZA mode, #1 to enable
- MSR SVCRSMZA, #<imm>; #0 to disable both, #1 to enable

The SMSTART instruction is an alias of the MSR instruction that can set PSTATE.SM and PSTATE.ZA:

- SMSTART: Enable both Streaming SVE mode and ZA mode
- SMSTART SM: Enable Streaming SVE mode
- SMSTART ZA: Enable ZA mode

The SMSTOP instruction is an alias of the MSR instruction that can clear PSTATE.SM and PSTATE.ZA:

- SMSTOP: Disable both Streaming SVE mode and ZA mode
- SMSTOP SM: Disable Streaming SVE mode
- SMSTOP ZA: Disable ZA mode

For most code, both modes are enabled or disabled together, but see [Section 6.11, “Multithreading and Mode Management”](#) for performance implications of each mode.

The compiler supports defining entire functions to execute in Streaming SVE/ZA mode, where SSVE/SME code is generated instead of Advanced SIMD and FP code. When these

functions are called from non-Streaming SVE/non-ZA mode code, the function bodies are wrapped with `SMSTART`/`SMSTOP` instructions, and vice-versa. There is no notion of *nested* `SMSTART`s and `SMSTOP`s. Although it is safe to execute additional `SMSTART`s when already in Streaming SVE/ZA mode, the first `SMSTOP` exits the mode. Use the following keywords on the function definition to indicate Streaming SVE/ZA mode:

```
// "n" stands for "non-streaming"
// "s" stands for "streaming"
// "sc" stands for "streaming-compatible"

// The program is in non-streaming statements when calling this function
void n_callee(void);

// The program is in streaming statements when calling this function
void s_callee(void) __arm_streaming;

// The program is in streaming-compatible statements when calling this function
void sc_callee(void) __arm_streaming_compatible;
```

Caller- or callee-saving of register state can be expensive due to the large register sizes. It may be beneficial to inline all function calls to form a single block of Streaming SVE/ZA mode code to avoid these concerns. Or, in custom code, assume all SSVE/SME registers are simply live across function boundaries. In some scenarios, it might be useful to have both non-Streaming SVE/non-ZA mode and Streaming SVE/ZA mode versions of some functions to use the proper ISA.

For more information on SME and Clang, including ongoing enhancements beyond the basic annotation above, see [AArch64 SME Attributes](#).

4.3 Memory Consistency Model

SSVE/SME memory accesses obey the same rules as for non-streaming SVE memory accesses, as defined in the *ArmArchitecture Reference Manual for A-profile architecture* section B2.1.3, "SVE memory model", including those relaxations described in section B2.3.9, "SVE memory ordering relaxations". In short, from the perspective of a single-threaded application, all loads and stores appear to occur in program order. That is, stored values from older Advanced SIMD and FP or integer stores are visible to younger SSVE/SME loads, and vice-versa. SSVE/SME also largely follow the overall ARM ISA "weak" memory consistency model, requiring barriers to enforce ordering between threads. Lastly, inactive elements due to [predication](#) do not cause a read from device memory or signal a fault, and are set to zero in the destination vector. See the ARM reference manuals for more details on the memory consistency model.

4.4 Predication and Loop Control

SSVE and some SME instructions utilize predicates to govern an instruction's behavior over the various elements in the vectors. See [Section 4.1.2, "SSVE P and PN Predicate Registers \(P\(N\)n\)"](#) for a description of the register file and the meaning of the bits in the predicate registers. Other instructions work alongside predication to control loop indices and bounds.

4.4.1 Predicate and Loop Control Operations

SSVE vectors are designed to scale with the implementation, so that vector-length agnostic software runs on any core that supports the Arm ISA regardless of the implemented vector length(s). The ISA offers specialized instructions that generate and manipulate predicate register contents, adjust pointers to point to the next vectors in memory, and increment loop induction variables by the number of elements (or active elements, factoring in predication) in a vector. These instructions largely replace the loop management instructions performed in normal scalar loops with vector equivalents that automatically scale to match the vector length and predication status.

Some predicate manipulation instructions also use a governing predicate, using /z zeroing behavior, such that `FALSE` bits in the governing predicate create `FALSE` bits in the destination predicate, regardless of the operation of the instruction.

Important microarchitectural note: On Apple silicon, predicate manipulation and loop control occurs inside of the CPU core, while comparison-based generation of predicates, as well as use of governing predicates, occurs inside of the SME engine. Code sequences with a critical path that frequently *crosses back and forth* experience the latency of these traversals, and are not recommended. See more in [Chapter 6, SME Engine Microarchitecture Optimization](#).

Some of the predicate manipulation instructions and some vector comparison instructions write NZCV flags, with the following interpretation:

- **N:** First (set if the first active element is `TRUE`)
- **Z:** None (cleared if *any* active element was `TRUE`)
- **C:** Not last (cleared if the last active element was `TRUE`)
- **V:** Unused (always set to 0 by predicate operations)

The following instructions generate and manipulate predicate values:

- **MOV[S], LDR, STR predicates:** Predicate register-to-register moves, along with predicate loads and stores. The `MOV`s versions set the NZCV flags.
- **ADDSPL:** Adjusts a pointer by a multiple of the size of the predicate registers in bytes.
- **CM<cc>/FAC<cc>/FCM<cc>:** Comparison-based predicate generation. These instructions perform vector element-by-element comparison to produce new predicate registers: integer, floating point absolute value, and floating point. Some but not all, such as `CMP<cc>`, set the NZCV flags. Per the microarchitectural note above, comparison results often feed loop control or predicate manipulation instructions, and thus may incur a long latency.
- **PSEL:** Conditional select of a predicate. This instruction selects between a predicate and the all-`FALSE` predicate, depending on the selected bit from an input predicate.
- **PTRUE/PFALSE:** Predicate initialization. These instructions create all-`TRUE` or mostly `TRUE` and all-`FALSE` predicates for the given element size. Among other specific uses, `PTRUE` creates a predicate that can activate all elements in a predicated instruction (making it effectively non-predicated). `PTRUE` comes in both `Pn` bit mask and `Pnn` counted style predicates, because an all-`TRUE` predicate is not the same in each case. The `PTRUE`s

version sets the flags accordingly (see below). Lastly, the bit mask version can also create specific patterns of `TRUE` predicates:

- `POW2`: The largest power of two number of elements that fit into the vector of the given type, setting `FALSE` predicates for any remainder elements.
- `VL<num>`: A specific number of elements that fit into the vector of the given type. Returns the `FALSE` predicate if the requested number does not fit into the vector.
- `MUL[ 3, 4 ]`: The maximum number of triple or quadruple sets of elements, setting `FALSE` predicates for the remainder of elements in the vector.
- `ALL`: The maximum number of elements.
- **WHILE<condition>**: "While" loop control. These instructions allow for the generation of end-of-vector/row/column predication masks from loop induction variables (from [core GPRs](#)). In addition, the `RW` and `WR` versions check two contiguous address ranges for a conflict or overlap that could result in a loop-carried dependency through memory within the same iteration of a loop. Most of these instructions come in versions for both `Pn` bit mask and `PNn` counted predicate styles.
- **BRK**: "Break" loop control. These instructions take the outputs from parallel vector compare instructions and converts them into predicates that implement a traditional loop "break," where elements before the first match are marked `TRUE` and elements after that first match are marked `FALSE`, with an option of `TRUE` versus `FALSE` for that first match. There are flavors that propagate the condition to additional predicate registers, as well as flavors that set the flags (see below).
- **PFIRST/PNEXT**: Predicate iteration. These instructions allow an algorithm to step through the `TRUE` predicate bits typically created by vector compares for scalar processing. Read the ISA instruction definition carefully to determine how to use the governing and source predicates correctly in these instructions.
- **PTEST**: Condition flag generation. This instruction converts a predicate into `NZCV` flags, so that the state of the predicate can control branches.
- **AND/BIC/EOR/NAND/NOR/NOT/ORN/ORR[S]**: Logical predicate manipulation. These instructions are analogs of normal integer logic instructions, but applied to predicate registers. With bit mask `Pn` predicates, they can be used to combine the bit masks produced by vector compares to evaluate complicated logic expressions across all vector elements in parallel. These instructions should *not* be used with counted predicates. The "s" versions set the `NZCV` flags per below.
- **PUNPK[HI, LO]/REV/TRN[1, 2]/UZP[1, 2]/ZIP[1, 2]**: Predicate rearrangement. These instructions are all analogs of the various SSVE instructions normally used to combine vectors together, and should be used in parallel with those instructions to rearrange the bits of any bit mask predicates to match the rearrangement of the elements themselves.
- **PEXT**: This instruction converts a `PNn` counted predicate into four `P` bit mask predicates, writing one or two of the resulting four `P` bit mask predicates into predicate registers. Extracting all four `P` bit mask predicates requires execution of two `PEXT` instructions.

The following instructions adjust loop induction variables of loops by vector lengths. They are largely replacements for the `ADD` or `SUB` operations that would normally be used to implement induction variable adjustments such as `i++` or `j--`:

- INC<size>, DEC<size> (scalar):** Adjusts loop induction variables by number of elements multiplied by an immediate. When the immediate is 1 (often omitted from disassembly), these update by element count to feed WHILE<condition> instructions. When the immediate is element sized, these update by total byte count. Some of these instructions come in signed and unsigned saturating versions, and can also count by the same element pattern constraints (ALL, MUL3, etc.) described under PTRUE above. The "P" version increments a scalar register by the number of TRUE elements.
- CNT<size>:** Set a register to the number of elements multiplied by an immediate. This instruction is typically used to create the increment value for a loop induction variable. These instructions perform the same as INC/DEC above, but simply return the adjustment amount into a general purpose register.
- CTERM<EQ/NE>:** Termination test. These instructions set flags when serially processing through elements of a vector, detecting when the specified condition is TRUE or when the end of the vector is reached. See below for how these instructions use the flag bits.

These instructions use an NZCV flag interpretation that is unique to them:

- Z, C = **unmodified**
- N, V = **continue loop** (00), **terminate loop, last element selected** (01), **terminate loop, compare succeeded** (10)

The following instructions adjust or return the streaming vector length in bytes.

- ADDSVL:** Adjusts loop induction variables by number of bytes of a streaming vector register. This instruction increments a pointer by a multiple of the length of an SSVE vector.
- RDSVL:** Sets register value to the current vector length in number of bytes. This instruction writes a multiple of the vector length used by ADDSVL into a register, which can then be used for more complex pointer adjustments.

Most Arm assemblers accept the SSVE/SME branch condition mnemonics listed in [Table 4.5: “SSVE/SME Variant Conditional Branch Mnemonics”](#), although branches with integer names also work.

Table 4.5. SSVE/SME Variant Conditional Branch Mnemonics

Mnemonic Extension	Meaning (SSVE/SME)	Condition Flags	Reference Meaning (Integer)
NONE	All active elements are FALSE or there are no active elements	Z==1	EQ
ANY	An active element is TRUE	Z==0	NE
NLAST	The last active element is FALSE or there are no active elements	C==1	CS/HS
LAST	The last active element is TRUE	C==0	CC/LO
FIRST	The first active element is TRUE	N==1	MI
NFRST	The first active element is FALSE or there are no active elements	N==0	PL

Table 4.5. SSVE/SME Variant Conditional Branch Mnemonics (cont.)

Mnemonic Extension	Meaning (SSVE/SME)	Condition Flags	Reference Meaning (Integer)
PMORE	An active element is TRUE, but the last active element is FALSE	C==1 && Z==0	HI
PLAST	The last active element is TRUE, or all active elements are FALSE, or there are no active elements	C==0 Z==1	LS
-	CTERM comparison failed, but end of partition reached	V==1	VS
-	CTERM comparison succeeded, or end of partition not reached	V==0	VC
TCONT	CTERM termination condition not detected	N==V	GE
TSTOP	CTERM termination condition detected	N≠V	LT

4.4.2 Element Activation Control

Bit mask predicates can be used to allow vector operations to emulate simple if/then/else control flow within a vector loop, potentially allowing a loop that otherwise could not be vectorized to become a viable vector target. However, per the microarchitectural note in [Section 4.4.1, “Predicate and Loop Control Operations”](#), predicate manipulation and loop control instructions that depend on element comparisons may perform considerably slower than expected, and are not generally recommended on Apple silicon chips.

Consider this sample loop that adds elements when either element is zero or greater, setting to zero when both are less than zero:

```

for (i=0; i < bufferLength; i++) {
    if ((a[i] < 0) && (b[i] < 0)) {
        c[i] = 0;
    } else {
        c[i] = a[i] + b[i];
    }
}

```

The above loop can be accomplished with the following instruction sequence designed to demonstrate the ISA:

```

// Add elements when either element is zero or greater

int bufferLength;
int32_t *aPtr, *bPtr, *cPtr;
svint32_t aV, bV, cV, zeroV;                                     // Actual length depends on SVL
svbool_t Pvec, Palt, Pblt, PifThen;

loop:
    WHILELT Pvec.S, i, bufferLength                               // Buffer elem-valid mask
    LD1W { aV.S }, Pvec/Z, [aPtr, i, LSL #2]
    LD1W { bV.S }, Pvec/Z, [bPtr, i, LSL #2]
    CMPLT Palt, Pvec/Z, aV.S, zeroV.S                             // Calculate a[i] < 0
    CMPLT Pblt, Pvec/Z, bV.S, zeroV.S                             // Calculate b[i] < 0
    AND PifThen, Pvec/Z, Palt, Pblt                               // Calculate ((a[i]<0)&&(b[i]<0))

```

```

ADD cV.S, aV.S, bV.S           // Calculate A + B for all elems
SEL cV.S, PifThen, zeroV.S, cV.S // Sel elems: ADD result vs 0 Vector
ST1W { cV.S }, Pvec, [cPtr, i, LSL #2]
INCW i, all
CMP i, bufferLength
B.LT loop

```

Much more complex conditional statements can be implemented by combining predicates, and leveraging the governing predicate in predicate manipulation instructions.

For another example, considering searching for the terminating character '\0' in a string.

```

// String length

char *strPtr;
int bufferLength;
svbool_t Pvec, Pend, Pall0;
svuint8_t v, zeroV, vIndex, Vpos;           // Bpos = Vpos as Bn register

// Loop to find vector with end '\0' character
MOV i, #0
PFALSE Pend.B
B loop_entry
loop:
    INCB i, all           // Increment count and pointer
    ADDVL strPtr, strPtr, #1
loop_entry:
    WHILELT Pvec.B, i, bufferLength           // Buffer elem mask + overrun test
    B.NONE no_end_character
    LD1B { v.B }, Pvec/Z, [strPtr]           // Load only bytes within the buffer
    CMPEQ Pall0.B, Pvec/Z, v.B, zeroV.B       // Find any '\0' chars in the buffer
    PFIRST Pend.B, Pall0, Pend.B           // 1st '\0' match ONLY + found test
    B.NONE loop

    // Found the end character . . .
    INDEX vIndex.B, #0, #1           // Create 0, 1, 2, 3, ... vector
    UMAXV Bpos, Pend, vIndex.B       // Get position of '\0'
    UMOV x0, Vpos.B[0]           // Extract Bpos scalar as vector elem 0
    ADD x0, x0, i           // Add start point of current vector
    RET

    // Ran out of buffer . . .
no_end_character:
    MOV x0, #0           // Returns 0, but could also return bufferLength, etc.
    RET

```

Note that the INDEX instruction produces a vector of {0, 1, 2, 3, ... 63} values in successive byte elements. There are a number of other SSVE/SME control instructions such as BRK and LAST with flavors that can implement the algorithm.

4.4.3 Data Structure Edge Control

Vector instructions are used to process many elements in parallel, iterating over large data structures. However, many large data structures do not map neatly to an integer number of iterations. There may be a number of unneeded elements in the vector register when reaching the edge of the data structure. Some data structures may be padded with extra

unnneeded elements to match the size of the vector, but this can be wasteful. In such scenarios, other implementations may execute a remainder loop of scalar instructions to complete the loop. However, using scalar operations can dramatically reduce the overall achieved parallelism. Instead, SSVE/SME offers predication support that allows the vector instructions to process the entire data structure, automatically gating off the effects of the instruction for elements beyond the data structure's bounds.

Many SSVE/SME instructions read one predicate input to control which vector lanes are enabled, while SME 2D operations read two (one for masking rows, and a second for masking columns). Virtually all load and store operations are predicated, in order to avoid the need for padding out the ends of rows or columns to whole vector lengths, but only some computation instructions are predicated. Although it depends on the specific instruction sequence, it is generally safe to employ predication only on loads and stores: load zeros for elements beyond the edges of the data structure, compute over those zeros, and then employ predication again on stores in order to prevent writes to those elements in memory. Thus many computation instructions do not have a governing predicate.

To illustrate, the following example uses bit mask predication with SSVE vectors to simply scale all of the elements in a matrix by a single scalar value:

```
// Scale the A matrix

float32_t      A[M*P];           // M rows x P columns
float32_t      *scalarPtr;
float32_t      *aPtr,
float32_t      xIn, yIn;         // Registers
svbool_t       Prow, Ptrue;
int            i, j, numElements;

PTRUE Ptrue.S, ALL
CNTW numElements, ALL           // numElements = 16 on Apple silicon

// -- Load in the scaling constant, replicating down length of the vector
LD1RW { yIn.S }, Ptrue/Z, [scalarPtr]

// -- Loop down matrix, over each row
for (i=0; i < M; i++) {
    // -- Loop across each row, 16 float32 elements per vector
    aPtr = &(A[i*P]);
    for (j=0; j < P; j+= numElements) {
        // -- Calculate predicate mask for this iteration
        //   Prow.S has FALSE predicates for elems "off the end of the row"
        WHILELT Prow.S, j, P
        // -- Perform the scaling for one vector per iteration
        LD1W { xIn.S }, Prow/Z, [(svfloat32_t *) aPtr]
        FMUL xIn.S, xIn.S, yIn.S
        ST1W { xIn.S }, Prow, [(svfloat32_t *) aPtr]
        aPtr += numElements;
    }
}
```

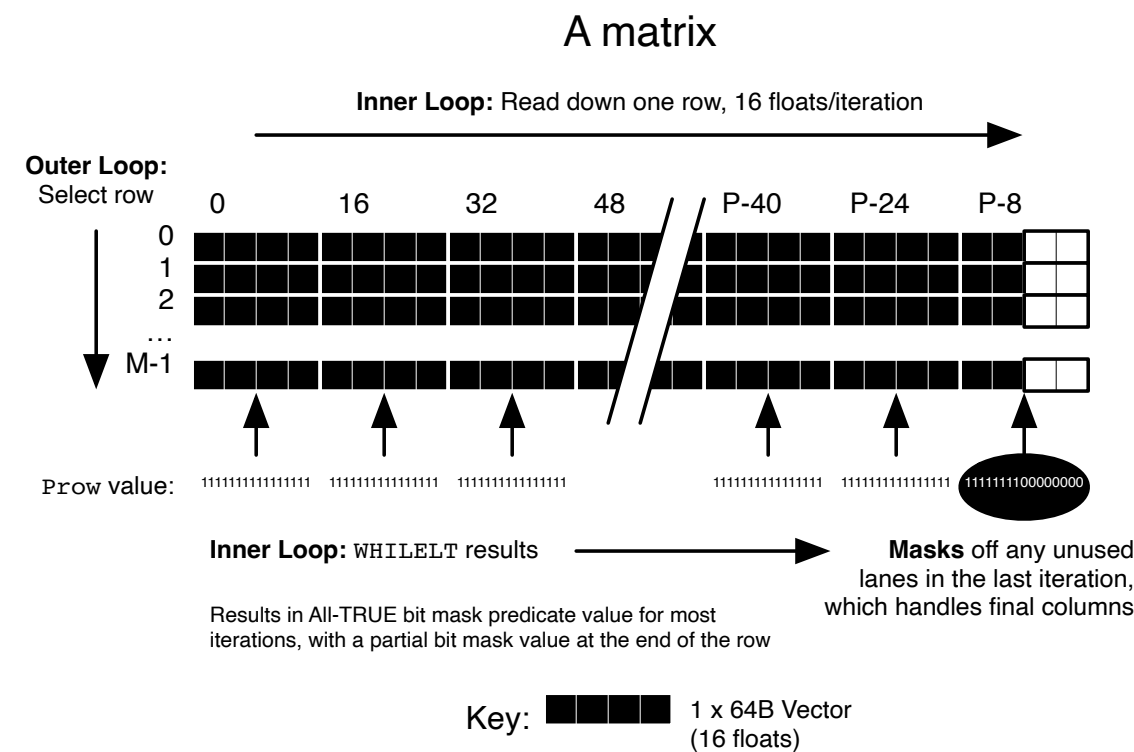
At the start, the code sequence executes a LD1RW instruction to read one float32_t scalar value from memory and replicate it down the length of the yIn vector, so all 16 float32_t lanes contain the same scaling value. Note the earlier PTRUE instruction, which sets the

predicate register `Ptrue` to activate all elements, so that *all* elements in the `yIn` vector receive copies of the scaling value during the subsequent `LD1RW` instruction.

The core of this routine is the `LD1W/FMUL/ST1W` sequence in the inner loop, which is analogous to equivalent `a *= scale` scalar code. Each execution of the inner loop loads in a vector of 16 `float32_t` values from memory, multiplies all values in the vector element-by-element with the scaling vector `yIn`, and then stores the resulting vector back out, overwriting the original vector in memory. In the loop body, these three instructions scale 16 `float32_t` values in parallel. Around this, the two-level nested loop iterates over the elements of the matrix, the same as how an equivalent scalar loop iterates, except that the inner loop increments by 16 elements (one vector length) each iteration.

The `WHAILELT` instruction takes the current value of the induction variable and the length of the row, and generates an appropriate predicate mask (`Prow`) for this iteration along the row. Because the test clause of the `FOR` loop uses a less-than (`<`) evaluation, this code uses a `WHAILELT` to match. (There are varieties of `WHILE<type>` instructions to match most different types of termination tests.) As depicted in [Figure 4.5: "Example of Edge Control Predication"](#), for most iterations, the instruction simply synthesizes an all-`TRUE` predicate. However, at the end of each row, the instruction creates a mask that deactivates any unneeded vector lanes that would have extended beyond the end of the row, so no extra elements are either loaded or stored. In the example, the final eight lanes are unused during the last iteration of each inner loop, so each load and store during these iterations only loads or stores 32 bytes, instead of the usual 64. This primarily avoids the hazard of overwriting bytes off the end of the row, but it also prevents the core from accidentally triggering superfluous page faults caused by any "off the edge" accesses.

Figure 4.5. Example of Edge Control Predication



Predication is integral to load and store operations to avoid polluting memory or taking unnecessary page faults. In contrast, computation instructions come in both predicated and non-predicated versions. Predication is usually not necessary in computation instructions because arbitrary values can be computed in unneeded lanes that are simply masked out on the final store. For example, the code above uses the non-predicated version of `FMUL`, because the `LD1W` and `ST1W` handle all of the necessary predication. Specifically, the load instruction fills vector elements with zero for any elements beyond the end of the row. Then, the `FMUL` simply calculates `#0.0 x scale factor = 0.0`, which does no harm, and the predicated `ST1W` only writes out result values for elements in the actual row.

A predicated version of `FMUL` can also be used:

```
FMUL xIn.S, Ptrue/M, xIn.S, yIn.S
```

This predicated version of `FMUL` generates the same result when using the `Ptrue` predicate as the non-predicated version, both of which compute all lanes. However, the predicated version is a destructive operation, with the first input and result sharing the same register, while the non-predicated version allows a distinct nondestructive result register. Of course, the predicated version could have also been used with the `Prow` predicate used by the load and store:

```
FMUL xIn.S, Prow/M, xIn.S, yIn.S
```

Instead of performing any calculation in the unneeded lanes at the end, this version just leaves the unneeded lanes holding their original contents, which in this case is `0.0` due to the predicated load. As a result, any of the three different versions of `FMUL` compute exactly the same result, but in other situations there may be a reason to choose one variant over another.

Although the above loop used bit mask predicates, counted predicates work equally well. Below is the same loop, where each iteration does `4x FMUL` (compared with above) using `Z` quad-vector load/stores and counted predicates. The destination predicate type notation `PN` selects the counted predicate version of the `WHILELT` instruction, and the compiler uses `PN8-PN15`, per the instruction definition. The `LD1W` four-vector version requires strided vector registers, and only accepts counted predicates.

```
// Scale the A matrix

float32_t      A[M*P];                      // M rows x P columns
float32_t      *scalarPtr;
float32_t      *aPtr,
svfloat32_t    xIn1, xIn2, xIn3, xIn4, yIn;  // registers
svbool_t       Ptrue;
svcount_t      PNrow;
int            i, j, numElements;

PTRUE Ptrue.S, ALL
CNTW numElements, ALL, MUL #4  // numElements = 16 elems * 4 vecs on Apple silicon

// -- Load in the scaling constant, replicating down length of the vector
LD1RW { yIn.S }, Ptrue/Z, [scalarPtr]
```

```
// -- Loop down matrix, over each row
for (i=0; i < M; i++) {
    // -- Loop across each row, 16 float32 elements per vector
    aPtr = &(A[i*P]);
    for (j=0; j < P; j+= numElements) {
        // -- Calculate predicate mask for this iteration (counted predicate).
        //     PNrow contains the count of active elements along with S size where
        //     the VLx4 indicates the predicate is used with four-vector consumers
        WHILELT PNrow.S, j, P, VLx4
        // -- Perform the scaling for 4 vectors/iteration
        LD1W { xIn1.S, xIn2.S, xIn3.S, xIn4.S }, PNrow/Z, [(svfloat32_t *) aPtr]
        FMUL xIn1.S, xIn1.S, yIn.S
        FMUL xIn2.S, xIn2.S, yIn.S
        FMUL xIn3.S, xIn3.S, yIn.S
        FMUL xIn4.S, xIn4.S, yIn.S
        ST1W { xIn1.S, xIn2.S, xIn3.S, xIn4.S }, PNrow, [(svfloat32_t *) aPtr]
        aPtr += numElements; // Jump to next quad vector
    }
}
```

Although less important in this example, the ISA contains flavors of the LD1W and ST1W that use consecutive aligned vector registers instead of strided, which use the following notation:

```
LD1W { xIn1.S-xIn4.S }, PNrow/Z, [aPtr]
ST1W { xIn1.S-xIn4.S }, PNrow, [aPtr]
```

Hence, when used for edge control, counted predicates and bit mask predicates may be used almost interchangeably, with bit mask predicates are used when working with single-vector load/stores and operations, whereas counted predicates are used with multi-vector load/stores and operations.

4.5 SSVE/SME Instruction Fundamentals

The following subsections describe instruction characteristics or operations that are notably different than ASIMD.

4.5.1 Register Operand Limitations

SSVE and SME instructions can involve many registers. With only 32 bits to encode the full instruction, some restrictions are placed on register operand selection. As a result of this practical limitation, many SSVE instructions are *destructive*, and overwrite one of their inputs. Although many Advanced SIMD instructions are also destructive, the shared source and destination is typically an accumulator. In contrast, many SSVE instructions without accumulators are still destructive simply due to these encoding limitations. The ISA employs a number of strategies to increase flexibility:

- **Predicated and non-predicated versions of instructions:** A number of the most commonly used instructions come in two different forms: a predicated version that is destructive and a second unpredicated version that is not destructive. By not encoding a governing predicate, the bits can be use to separate the source and destination registers. For example:

```
ADD <Zdn>.<T>, <Pg>/M, <Zdn>.<T>, <Zm>.<T> // Predicated: <Zdn> is both dest & src
ADD <Zd>.<T>, <Zn>.<T>, <Zm>.<T> // Unpredicated: Separate dest <Zd>
// and src <Zn>
```

- **Synthesized nondestructive operations (MOVPRFX):** Although some operations have both destructive and nondestructive versions, there are still many that are only destructive. In order to offer more flexibility, the ISA offers a special register move instruction, MOVPRFX, that can be placed before any destructive SSVE operation to synthesize a nondestructive operation using two instructions. The MOVPRFX simply copies a source register into the combined source and destination of the computation instruction located immediately after it. The hardware may be able to internally optimize the execution of the pair over a standard vector-vector move that would serve the same logical purpose.

```
// This pair of instructions in the source code...
MOVPRFX Zd, Pg/[ZM], Zs1
ADD Zd, Pg/M, Zd, Zs2 // Zd: destructive src & dest

// ...effectively become this single instruction
ADD Zd, Pg/[ZM], Zs1, Zs2
```

- **Reversed input destructive operations:** Some operations are non-commutative with two distinct input operands. When encoding limitations require a destructive operand, the ISA often offers a second "reversed" version of the operation that swaps the meaning of the operands, which effectively allows for either of the sources to be overwritten. See [Section 4.6.8, "SSVE Reversed Inputs: R"](#).
- **Limited operand register options:** Some operations can only access a limited portion of a register file or access specific sets of registers in a register file. Instead of being able to access any of the 32 Z registers, which requires a full 5-bit specifier, or access any of the 16 P registers, which requires a 4-bit specifier, a particular register specifier might be limited to only 8 registers (a 3-bit specifier) or even just 4 registers (a 2-bit specifier). This is particularly common with predicate registers, which are usually limited to just half or even a quarter of the predicate register file. ([Section 4.4.1, "Predicate and Loop Control Operations"](#)) These limitations may require copying of operands from one portion of the register file to another, in addition to careful register allocation. For example:

```
FDOT ZA.S[<Wv>, <offs>{, VGx4}], { <Zn1>.H-<Zn4>.H }, <Zm>.H[<index>]
```

- *wv* is a general purpose register in the range of w12-w15.
- *Zn1-Zn4* must be aligned so that *Zn1* is a register number that is a multiple of four.
- *Zm* must be Z0-Z15.

4.5.2 SSVE Horizontal (Pairwise) Instructions

Long vectors are challenging for the implementation of horizontal (pairwise) instructions where individual operations span multiple vector lanes. [Figure 4.6: "ADDP Instruction in Advanced SIMD and SSVE"](#) depicts the ADDP instruction on byte-size elements for both Advanced SIMD and SSVE (other element sizes are similar). With the Advanced SIMD

```
DUP yIndex.Q, yIn.Q[1]
  < USE yIndex.<size>[index] for operations, index = 0,1,...,(vec_elements - 1) >
// Replicate third-lowest 16B of yIn across yIndex, and use in computation
DUP yIndex.Q, yIn.Q[2]
  < USE yIndex.<size>[index] for operations, index = 0,1,...,(vec_elements - 1) >
// Replicate last 16B of yIn across yIndex, and use in computation
DUP yIndex.Q, yIn.Q[3]
  < USE yIndex.<size>[index] for operations, index = 0,1,...,(vec_elements - 1) >
```

Rather than rotating through the subvectors (segments) via hardcoded immediate values, the subvectors themselves could be rotated by 16 bytes while always replicating the bottom 16 bytes. This approach is used in [Section 4.7.5, “Matrix-Vector Multiply”](#) and eliminates the unrolling:

```
LD1<size> { yIn.<size> }, Pvec/Z, [ptr]
// -- Loop over each scalar value from Y vector for (k=0; k < 4; k++) {
// Replicate lowest 16 bytes of yIn across yIndex
DUP yIndex.Q, yIn.Q[0]
  < USE yIndex.<size>[index] for ops, index = 0,1,...,(vec_elements - 1) >
// Rotate yIn by 16 bytes for next iteration
EXT yIn.B, yIn.B, yIn.B, #16
```

The choice between the above approaches may depend on algorithm characteristics or microarchitectural bottlenecks discussed in [Chapter 6, SME Engine Microarchitecture Optimization](#) and [Section A.4, “SSFP/SSVE Execution Unit”](#).

SME ZA vector operations with two- or four-vector (VGx2, VGx4) destinations have three varieties. The *multiple and indexed vector* element variety is similar to SSVE, where an element is selected per 16B subvector and applied to each element in the corresponding 16B subvector but to each multiple source vector. The *multiple and single vector* variety applies each element in the single vector to each corresponding element in each multiple source vector. And lastly, the *multiple vectors* variety applies each element individually across the two sets of multiple source vectors.

FMLA (multiple and indexed vector): Multi-vector floating-point fused multiply-add
FMLA ZA.<T>[<Wv>, <offs>{, VGx2}], { <Zn1>.<T>-<Zn2>.<T> }, <Zm>.<T>[<index>]

FMLA (multiple and single vector): Multi-vector floating-point fused multiply-add
FMLA ZA.<T>[<Wv>, <offs>{, VGx2}], { <Zn1>.<T>-<Zn2>.<T> }, <Zm>.<T>

FMLA (multiple vectors): Multi-vector floating-point fused multiply-add
FMLA ZA.<T>[<Wv>, <offs>{, VGx2}], { <Zn1>.<T>-<Zn2>.<T> }, { <Zm1>.<T>-<Zm2>.<T> }

4.6 Overview of Arm SSVE/SME Mnemonics

SSVE/SME instruction mnemonics are constructed in a systematic manner from a modest-size number of prefixes, suffixes, and about 120 base instruction mnemonics. Prefix and suffix letters generally mean the same between SSVE/SME and Advanced SIMD.

The Arm SSVE/SME instruction prefixes are summarized in [Table 4.6: “SSVE/SME Instruction Mnemonic Prefixes”](#), base instructions in [Table 4.7: “SSVE/SME Base Instructions Grouped By General Function”](#), and suffixes in [Table 4.8: “SSVE/SME Instruction Mnemonic Suffixes”](#).

Where the prefix or suffix definition is substantially the same as for Advanced SIMD, the field name link refers to the explanations in the Advanced SIMD chapter.

Table 4.6. SSVE/SME Instruction Mnemonic Prefixes

Field Name	Coding	Type	Description
Data Type	(none) / BF / F / S / U	All	Selects between brain floating point (bfloat16), general floating point, signed integer / fixed point, and unsigned integer / fixed point operation types, if needed
Modulo / Saturating	(none) / Q	Integer and fixed point	Controls how operation handles overflow at high-order end (most significant bits)
Truncating / Rounding	(none) / R	All	Controls how operation handles low-order bits (least significant bits) that are lost
Doubling / Halving	D / H	Fixed point	Multiplies the result 2 (D), to align fixed-point binary points, or ½ (H), returning an arithmetic mean instead of a sum
Polynomial	P	Galois polynomial	Selects Galois field multiplication
Vertical	V	SME vector vertical DOT product	Specifies that the input elements to the DOT product are read <i>across</i> all of the two or four individual vectors instead of horizontally within the vectors (neighboring elements)
Negate	N	All	Negates the result
Complex Number Arithmetic	C	All	Modifies operations to work with complex numbers arranged as interleaved (real, imaginary) values

Table 4.7. SSVE/SME Base Instructions Grouped By General Function

	Group	Instructions
Mathematical Operations	Sign Control	ABS, NEG
	Arithmetic	ABA, ABD, ADA, ADC, ADD, SBC, SUB
	Multiply	DOT, MAD, MLA, MLS, MOP, MSUB, MUL
	Divide/Sqrt	DIV, RECP, RSQRT, SQRT
	Transcendentals	LOGB, SCALE
	Comparison	AC<cond>, CLAMP, CM<cond>, CMP<cond>, CNOT, MAX, MIN
	Conversion	CVT, INT
	Loop Indices	DEC, INC
Bitwise Operations	Shift	ASR, ASRD, LSL, SLR, SHL, SHR, SLI, SRA, SRI, XT
	Logic	AND, BIC, EON, EOR, NOT, ORN, ORR, ORV
	Masking	BCAX, BSL, DUPM
	Misc. Bitwise	BMOP, CLS, CLZ, CNT, RBIT
Element Operations	Move	CPY, DUP, INSR, MOV, MOVPRFX
	Predicate-Based Shuffle	SEL, SPLICE

Table 4.7. SSVE/SME Base Instructions Grouped By General Function (cont.)

	Group	Instructions
	Element Shuffle	EXT, REV, TBL, TBX, TRN, UNPK, UZP, ZIP
	Lookup Table	LUTI
	Element Extract	CLAST, LAST
Misc.	Clear ZA Tiles	ZERO
	Vector Position	INDEX
Memory Operations	Load	LD<n>, LDR
	Store	ST<n>, STR
	Prefetch	PRF
Pointer Operations	Arithmetic / Constants	ADD, RDVL, RDSVL
Predication	Logic	AND, BIC, EOR, NAND, NOR, NOT, ORN, ORR
	Loop Control	BRK, CTERM, WHILE
	Loop Index Adjustments	DEC, INC
	Movement / Load / Store	LDR, MOV, STR
	Misc	CNT, PEXT, PFALSE, PFIRST, PNEXT, PSEL, PTEST, PTRUE
	Shuffles	PUNPK, REV, SEL, TRN, UZP, ZIP

Table 4.8. SSVE/SME Instruction Mnemonic Suffixes

Field Name	Coding	Types	Description
Cryptographic	Various	N/A	Various AES/SH4 cryptographic instruction specifications
Load / Store Interleaving	1 / 2 / 3 / 4	Load and store	Selects how elements are interleaved as they are loaded from memory
Load Replicating / Broadcasting	R / RQ	LD1	Causes one element or element block to be replicated (broadcast) across the destination vector
Load / Store Non-Temporal	NT	LD1/ST1	For loads and stores, provides a hint to the memory system that the access is non-temporal
Load / Store Element Size	B / H / W / D / Q	Load and store	Selects the size of elements used for some load/store operations
Comparison Conditions	EQ / GE / GT / HI / HS / LE / LO / LS / LT / NE / UO	All	Chooses comparison mode for compare-and-mask or conditional operations
Special Value Handling Mode	NM / X	Floating point	For specific instructions, controls how operation handles NaN and $\pm\infty$ values
FP Rounding	A / I / M / N / P / X / Z	Floating point	For CVT and INT instructions, selects a floating point rounding mode

Table 4.8. SSVE/SME Instruction Mnemonic Suffixes (cont.)

Field Name	Coding	Types	Description
Output Data Type	(none) / F / S / U	All	Selects an output data type if it is different from the input
Return High Half	H	Typically fixed point	Indicates that the instruction result contains only the high-order half (most significant bits)
Narrow-Interleave / Long / Long-Long / Wide	N / L / LL / W	All	Indicates that the instruction result elements have smaller or larger bit width than each source element
Lower / Upper Half	(none) / 1 / 2 / LO / HI	TRN/UNPK/ UZP/ZIP	Selects a destination half for some byte shuffling operations that produce double- wide output
Paired or Horizontal	P / V	All	Indicates operations that work horizontally across vectors instead of in lanes
Bottom / Top	B / T	All	For lengthening instructions, selects only odd or only even input elements in an interleaved manner
Reversed Input	R	Non-commutative	Switches order of the inputs, for destructive and non-commutative operations like subtraction or division
Horizontal / Vertical	H / V	SME vertical tile	Explicitly applies SME tile operation across either ZA rows or columns
SME Target	A / S / T	MOV/ADDHA/ ADDVA/MOP/ MOVT	Moves to/from ZA storage, applies a vector to each slice of a ZA tile, multiplies and then adds/ subtracts with ZA storage, or moves to/from ZT0 (lookup table source register)

4.6.1 SSVE/SME Narrow, Long, Long Long, Wide: N / L / LL / W

SSVE/SME builds upon the Advanced SIMD $N/L/W$ suffixes (described in [Section 3.3.16](#), “ASIMD Narrow, Long, Wide: N / L / W”) that convert between bit widths of various data types. See [Section 3.1](#), “Advanced SIMD and FP Data Types” for details on element data types and their value ranges.

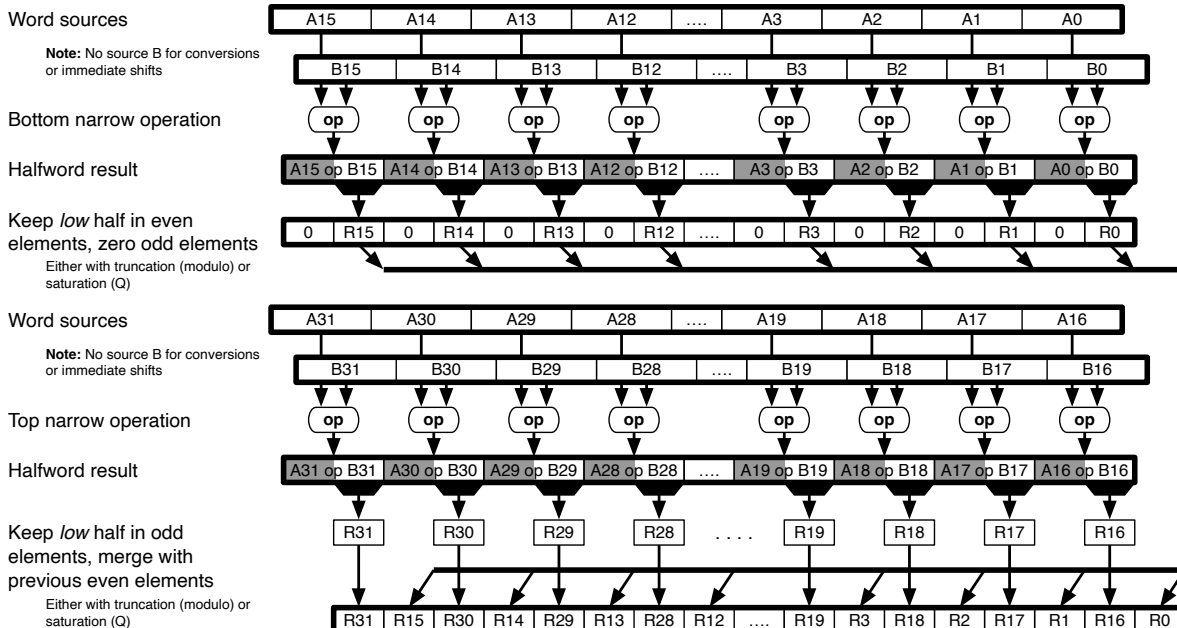
Because of the length of SSVE vectors, and the implementation cost of moving bytes great distances inside the vectors, the instructions tend to perform operations within lanes, where the relative byte position of the source elements is more or less the same as that of the resulting destination elements. The examples in this section depict a specific source and destination data type, but the pattern applies to other data type combinations for the same instructions as well.

Figure 4.8: “SSVE: Narrowing Instructions” and **Figure 4.9: “SME: Narrowing Instructions”** depict narrowing operations for SSVE ISA and SME ISA (instructions included with SME that are not available in non-streaming SVE), respectively. For SSVE, these instructions use bottom(B)/top(T) operations to fill in the even/odd elements of the result vector using two instructions, performing similar operations as ASIMD 1/2 instructions (**Figure 3.4: “Advanced SIMD Narrow, Lengthen, and Widen”**). However, they produce interleaved outputs similar to ASIMD B/T (**Section 3.3.19, “ASIMD Bottom or Top Half of Element Pairs: B / T”**) instead of deinterleaved outputs.

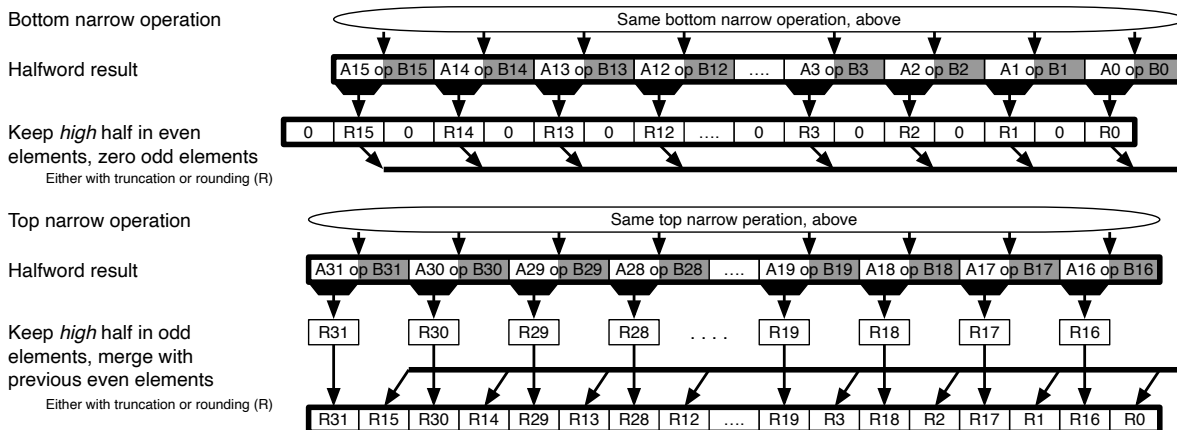
Figure 4.8. SSVE: Narrowing Instructions

A) SSVE Bottom and Top Narrow Operations (NB* and NT)

*Any narrow FCVT without an "NT" suffix acts as an "NB" operation



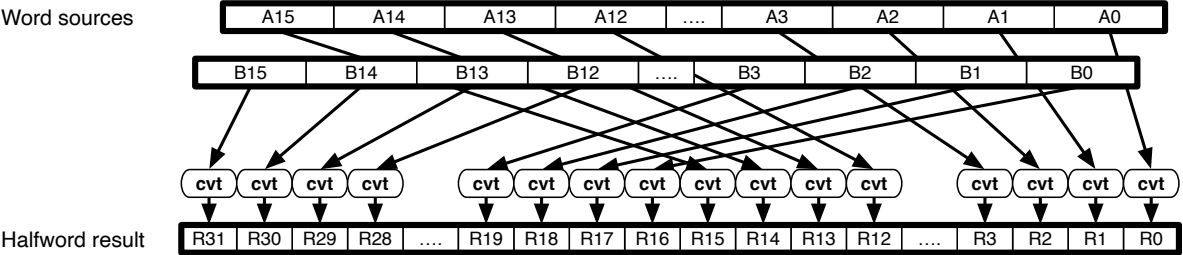
B) SSVE Bottom and Top High-Half Narrow Operations (HNB and HNT)



To better support SME, several Z vector conversion-only narrowing instructions are available that do not use the bottom-even/top-odd source interleaving, but rather convert with deinterleaved elements (convert with packed elements) or convert with interleaved elements. These are often used immediately after extracting data from ZA storage into Z vectors.

Figure 4.9. SME: Narrowing Instructions

A) SME Narrow Conversion/Shift, Deinterleaved (e.g. FCVT, SQCVT, or SQRSHR S-to-H)



B) SME Narrow Conversion/Shift, Interleaved (e.g. FCVTN, SQCVTN, or SQRSHRN S-to-H)

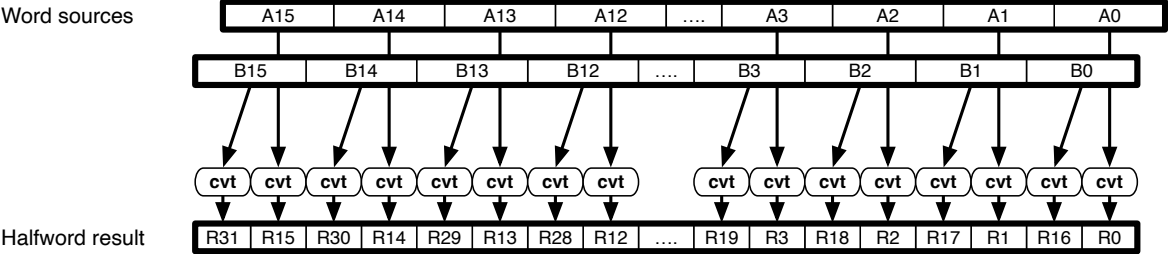
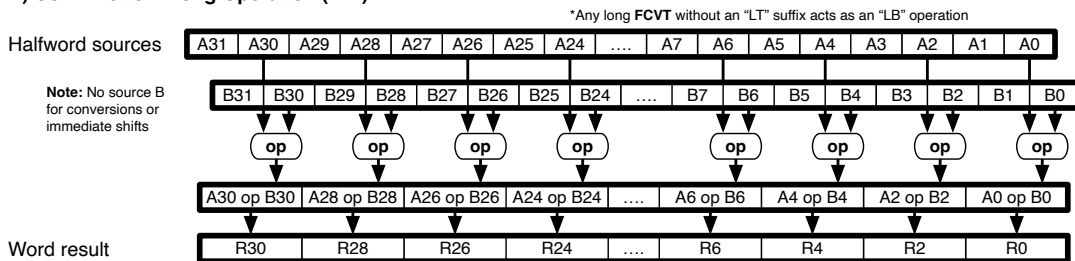


Figure 4.10: “SSVE: Widening/Lengthening Instructions” and Figure 4.11: “SME: Widening/Lengthening Instructions” depict widening/lengthening operations for SSVE ISA and SME ISA, respectively.

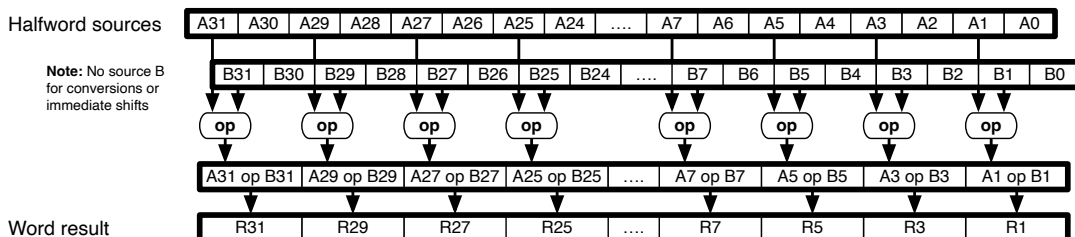
SSVE uses B/T/BT suffixes to select even/odd elements of the source vectors, where the operation itself is like the similar ASIMD 1/2 instructions (Figure 3.4: “Advanced SIMD Narrow, Lengthen, and Widen”). However, they read interleaved sources similar to ASIMD B/T (Section 3.3.19, “ASIMD Bottom or Top Half of Element Pairs: B / T”) instead of deinterleaved sources.

Figure 4.10. SSVE: Widening/Lengthening Instructions

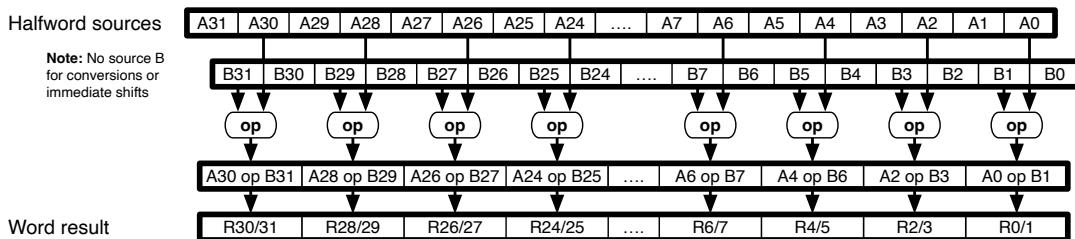
A) SSVE Bottom Long Operation (LB*)



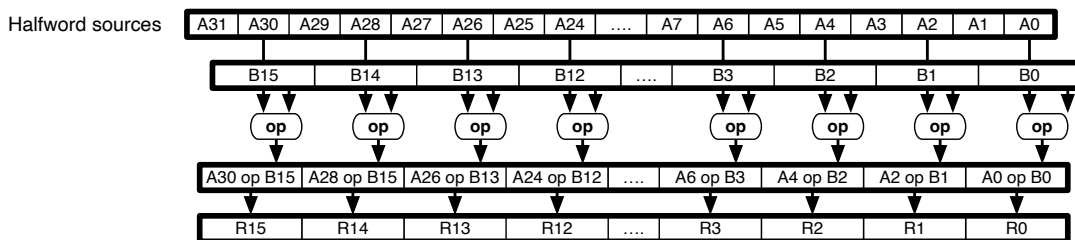
B) SSVE Top Long Operation (LT)



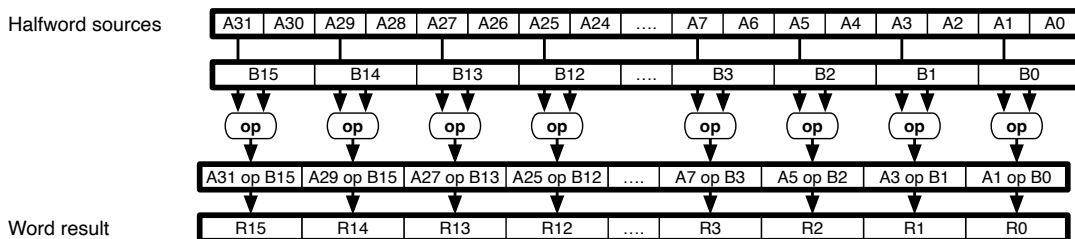
C) SSVE Bottom-Top Long Operation (LBT)



D) SSVE Bottom Wide Operation (WB)



E) SSVE Top Wide Operation (WT)

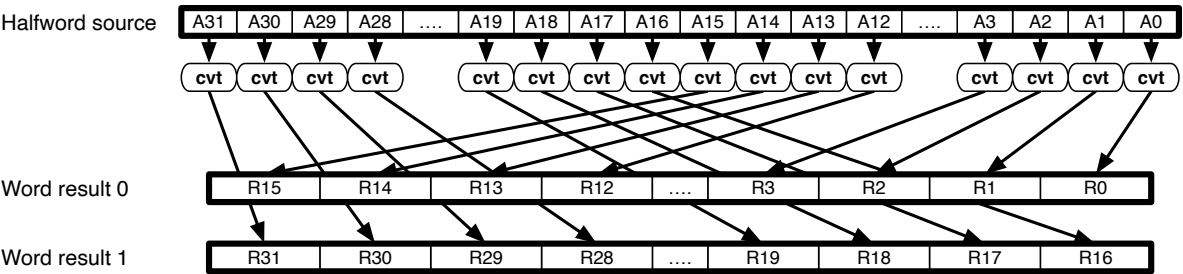


To better support SME, several conversion-only lengthening/widening instructions are available that do not use the bottom-even/top-odd source interleaving, but rather convert with deinterleaved elements (convert with in-order elements) or convert with interleaved

elements. These instructions read a single Z register and produce two Z registers containing lengthened/widened elements. They are referred to as SME ISA to distinguish from those included in the SSVE ISA.

Figure 4.11. SME: Widening/Lengthening Instructions

A) SME Long “Widening” Conversion (e.g. FCVT H-to-S), Deinterleaved



B) SME Long “Lengthening” Conversion (e.g. FCVTL H-to-S)

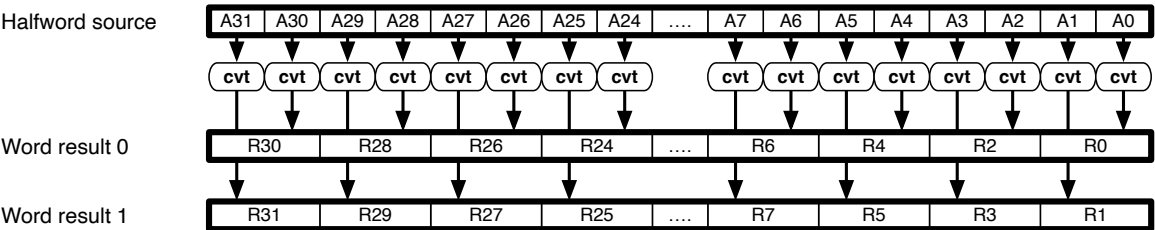
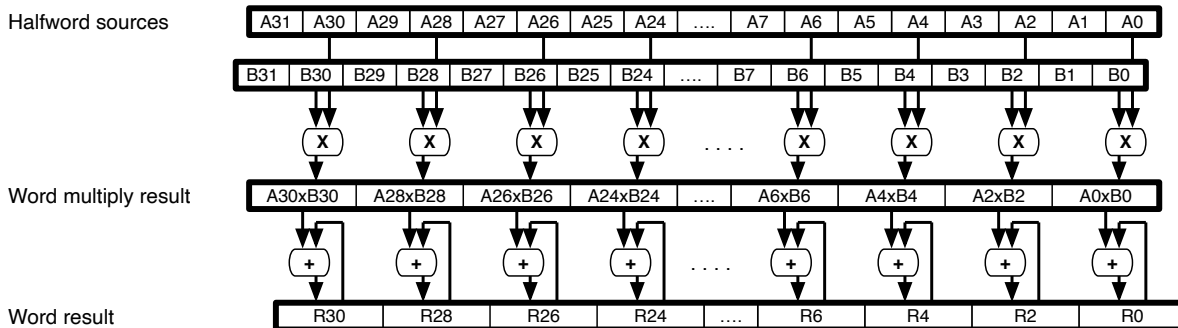


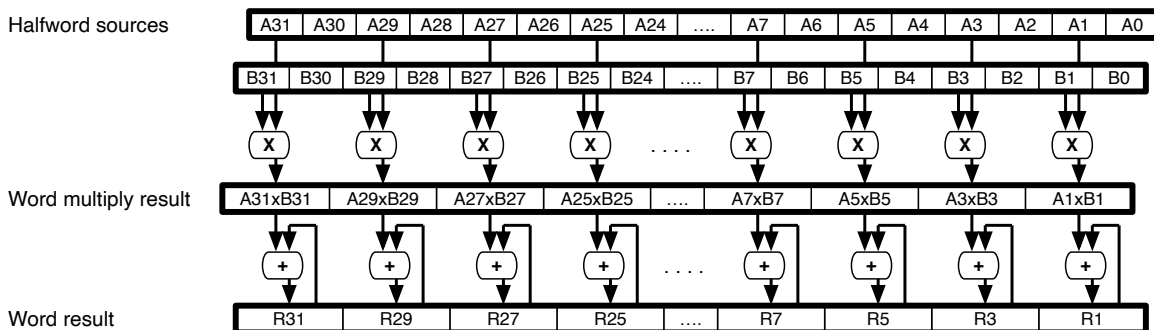
Figure 4.12: “SSVE: Lengthening Multiply-Accumulate Instructions” and Figure 4.13: “SME: Lengthening Multiply-Accumulate Instructions” depict the widening/lengthening multiply-accumulate operations for SSVE ISA and SME ISA, respectively. They follow the same patterns as above. `LL` is a new suffix that is only used with some integer multiply-accumulate SME ISA operations. As the “Long Long” name implies, the output from these instructions does not just double the length of the input operands like `L` instructions, but instead quadruples the length of the input operands. This allows room for both the doubling of the length of the input operations as they are multiplied plus another doubling to allow room for accumulation of many multiplication results without having any intermediate rounding. This saves a preliminary input-lengthening step from the sequence used in similar Advanced SIMD or SSVE calculations, as can be seen in Figure 3.2: “ASIMD Integer Processing Example”.

Figure 4.12. SSVE: Lengthening Multiply-Accumulate Instructions

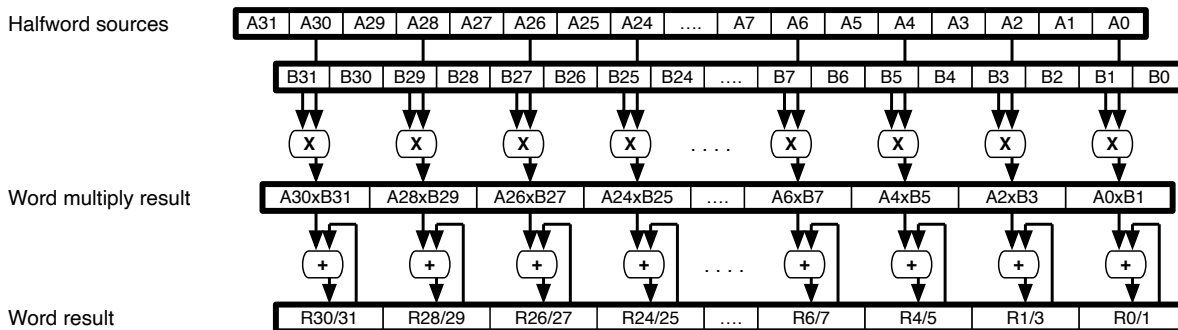
A) SSVE Long Bottom Multiply-Accumulates (LB, H-to-S)



B) SSVE Long Top Multiply-Accumulates (LT, H-to-S)



C) SSVE Long Bottom-Top Multiply-Accumulates (LBT, H-to-S)



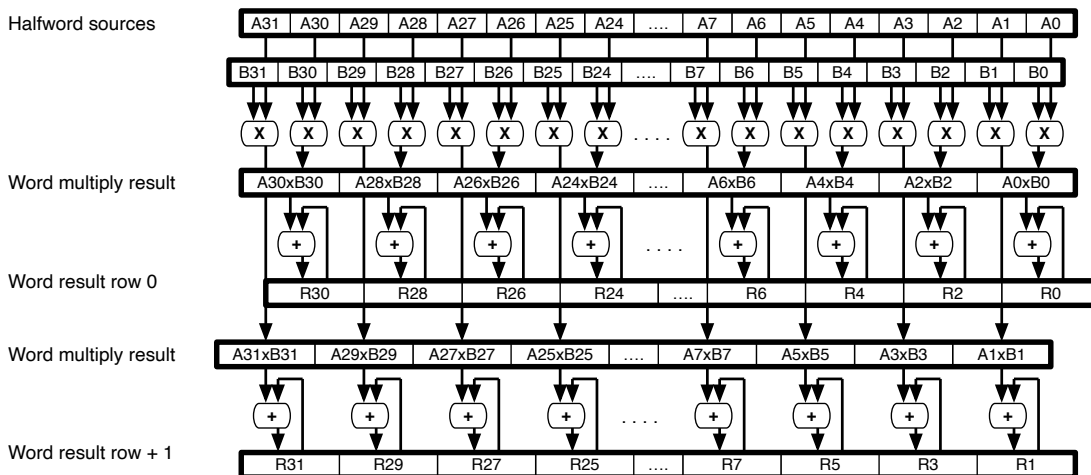
These SME instructions operate on ZA storage.

Apple Silicon CPU Optimization Guide: 4.0

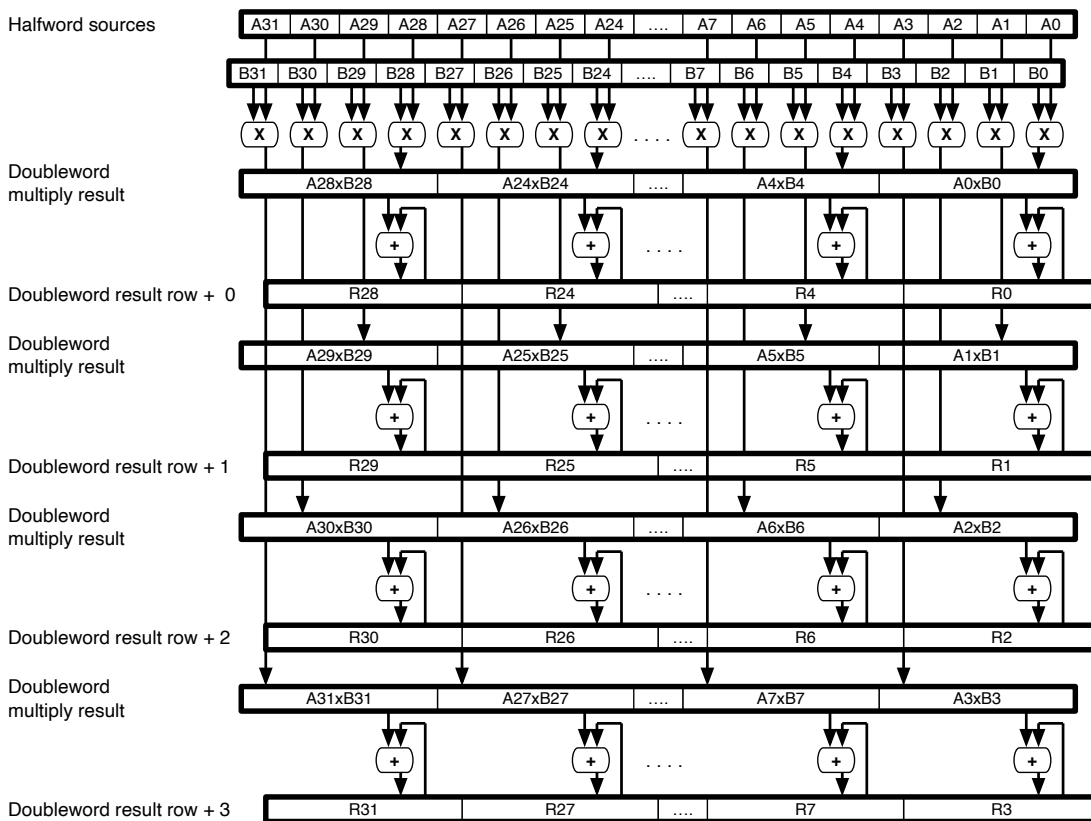
ISA Optimization: Scalable Matrix Extension (SME) SSVE/SME Lower / Upper Half + Bottom / Top: (none) / 1 / 2 / LO / HI / B / T

Figure 4.13. SME: Lengthening Multiply-Accumulate Instructions

A) SME Long Vector Multiply-Accumulates to ZA (L, H-to-S)



B) SME Double Long Vector Multiply-Accumulates to ZA (LL, H-to-D)



4.6.2 SSVE/SME Lower / Upper Half + Bottom / Top: (none) / 1 / 2 / LO / HI / B / T

There are a large number of Advanced SIMD instructions, such as lengthening operations that produce double-length outputs or narrowing operations with double-length inputs, which must be able to specify which half of the input or output is used by the instruction.

These are almost always handled with pairs of instructions that include a 1/2 suffix to specify whether the instruction should consume/produce the lower (1 or omitted) or upper (2) part of the input/output. (See [Section 3.3.17, “ASIMD Lower-Half / Upper-Half: \(none\) / 1 / 2”](#)) However, this technique is only used in a few cases with SSVE/SME, particularly for the TRN, UNPK (uses HI/LO instead of 1/2, but the effects are similar), UZP, and ZIP families of element shuffle instructions.

Instead, most lengthening and narrowing instructions in SSVE/SME whose Advanced SIMD equivalents use 1/2 behavior instead use bottom (B) / top (T) *interleaving* control because the much longer SSVE vectors require long horizontal shuffles to pack elements together or spread inputs out down the length of the vector. In contrast, interleaved operations only need to perform horizontal shuffles within each adjacent even/odd pair of element lanes. Within these even/odd lane pairs, the B/T suffixes note which lane (B=even, T=odd) should be used for input (for lengthening operations) or output (for narrowing operations). These suffixes were previously used only for a handful of Advanced SIMD operations, as noted in [Section 3.3.19, “ASIMD Bottom or Top Half of Element Pairs: B / T”](#), but are widespread in SSVE/SME. See [Section 4.6.1, “SSVE/SME Narrow, Long, Long Long, Wide: N / L / LL / W”](#) for examples of how they are used in conjunction with narrowing and lengthening.

Porting code that uses many 1/2-suffix Advanced SIMD instructions to SSVE/SME code that uses B/T operations is conceptually straightforward, but requires significant adjustments to the byte arrangement code. Individual narrowing operations only produce either even B elements or odd T elements in their output. B operations produce zeros in the odd lanes. T operations *merge* in the even elements presumably produced by a previous B operation. UZP instructions (UZP1 to pack two B result vectors into one, or UZP2 to pack two T result vectors into one) can be used to remove the zero elements from single source conversions. Similarly, lengthening operations select from their available inputs using B (to select even input lanes), T (to select odd input lanes), or occasionally even BT/TB (to select even lanes from one input and odd lanes from the other). A comprehensive guide to porting, however, is beyond the scope of this document.

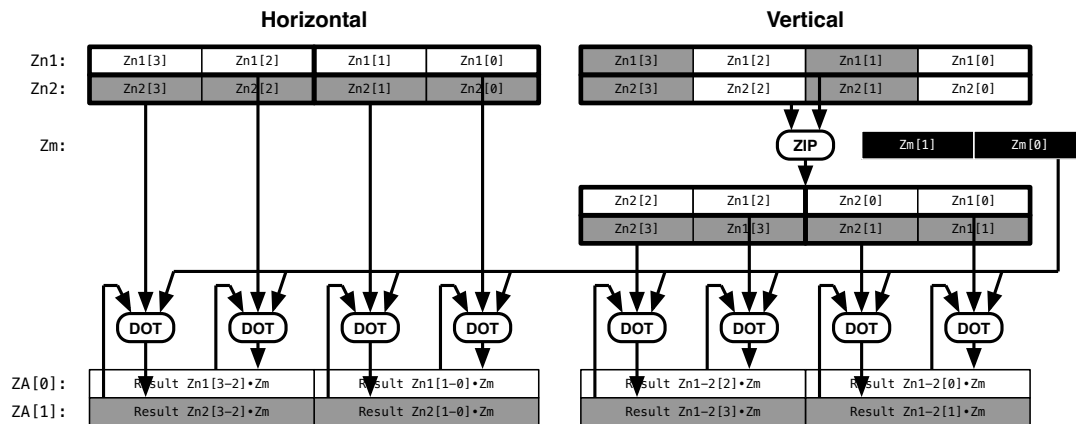
The SME ISA lengthening/widening/narrowing instructions that operate on ZA storage produce a different element ordering pattern that may be advantageous when considering the rest of the algorithm.

4.6.3 SME Horizontal / Vertical in Vector: H / V (Prefix)

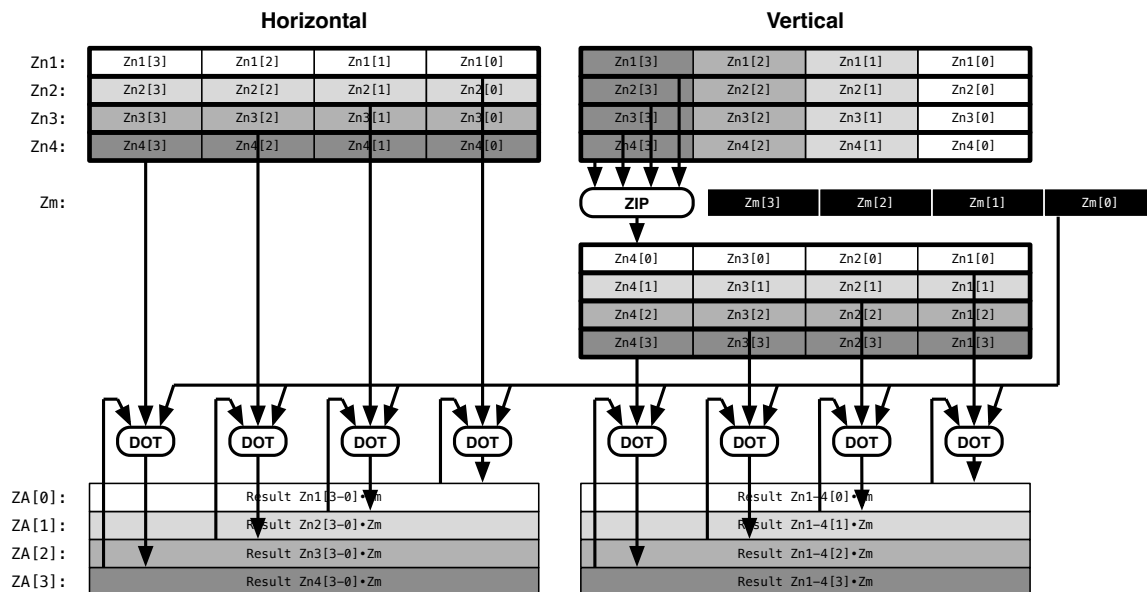
SME vector lengthening dot product (DOT) operations can interpret their first operand in either the standard *horizontal* way, along each vector, or in a *vertical* manner, with elements coming from *across* all of the two or four individual vectors that make up the input multi-vector. DOT operations using this latter addressing pattern are marked with a “v” prefix to differentiate them from the equivalent horizontal versions. See [Figure 4.14: “SME Vector: Horizontal and Vertical DOT Product Instructions”](#) for a visualization of the access patterns at each output element (the B-to-S element form is available but not shown). Effectively, the vertical operations are the equivalent of executing ZIP of first operand onto the beginning of a conventional DOT operation (which is detailed in [Section 4.6.11.2, “SME Dot Product \(DOT\) Vector Instructions”](#)). Note that the vDOT instructions only come in the quad-word indexed register form for Z_M (see [Section 4.5.3, “Element Indexing”](#)).

Figure 4.14. SME Vector: Horizontal and Vertical DOT Product Instructions

A) 2-Way Lengthening DOT Product (H-to-S)



B) 4-Way Double-Lengthening DOT Product (H-to-D)



4.6.4 SME Horizontal / Vertical in Tile: H / V (Suffix)

A few SME tile instructions (such as ADDHA/ADDVA) can apply their vector operand in either the standard row-wise direction (horizontal, H) or in a column-wise direction (vertical, V). Similarly, some SME LD1, ST1, and MOVA instructions can move data to/from a row or a column of the tile. However, these specify the orientation in the ZA operand notation instead of the instruction mnemonic (such as ZA2H[4].D compared with ZA2V[4].D, described in Section 4.1.3.1, "ZA Tile Registers (ZA<n>.<size>)").

Because all other unmarked instructions work exclusively with one or more rows, the vertical tile operations are only used in special circumstances, including [matrix transposition](#).

4.6.5 SSVE Load Replicating / Broadcasting: R / RQ

The replicating/broadcasting instructions copy an element or group of elements across the vector. These are useful when preparing for a computation that applies that element or group of elements to a full vector. See [Section 4.5.3, “Element Indexing”](#).

The LD1R<type> instructions broadcast a single loaded element of the specified type, widening as necessary to the destination element size, replicating to fill all elements of the destination register.

The LD1RQ<type> instructions replicate a 128-bit (quadword) vector of the specified type, replicating to fill all quadwords of the destination register.

4.6.6 SSVE Comparison Conditionals: EQ / GE / GT / HI / HS / LE / LO / LS / LT / NE / UO

SSVE comparison operations support the same conditions as those in Advanced SIMD, as described in [Section 3.3.11, “ASIMD Comparison Conditionals: EQ / GE / GT / HI / HS / LE / LT ”](#), plus a few additional. The full set of SSVE comparison conditionals is listed in [Table 4.9: “SSVE Comparison Conditionals”](#). Unlike the Advanced SIMD conditional operations which produce masking vectors, the SSVE conditional operations generate predicates. Complex conditional expressions can then be evaluated for all vector lanes in parallel using [predicate logic instructions](#), and the final result can be used to control conditional execution down the length of the vector. Most come in three versions:

- **Vector (annotated as "v." in the table):** Each element in the first source vector is compared with the corresponding vector in the second source vector.
- **Wide Elements (annotated as "w.e." in the table):** Each element in the first vector is compared with the corresponding overlapping 64-bit doubleword element in the second source vector. For instance, CMPEQ on a 8b first source vector compares each of the 8b elements 0-7 with the same 32b element 0 of the second source vector, and each of the 8b elements 8-15 with the 32b element 1, and so on.
- **Immediate (annotated as "#imm" in the table):** Each element in the first source vector is compared with an immediate value. For floating point comparisons, the immediate can only be #0.0.

Comparisons in the table below refer to both the vector version and the wide elements version, unless specified.

Table 4.9. SSVE Comparison Conditionals

Mnemonic Code	Name	Interpretation	Instructions
EQ	Equal	$A == B, A == \#imm$	CMP<cond>, FCM<cond>
GE	Greater Than or Equal	$A \geq B, A \geq \#imm$	CMP<cond>, FAC<cond>, FCM<cond>
GT	Greater Than	$A > B, A > \#imm$	CMP<cond>, FAC<cond>, FCM<cond>
HI	Higher Than	unsigned $A > B$, unsigned $A > \#imm$	CMP<cond>

Table 4.9. SSVE Comparison Conditionals (cont.)

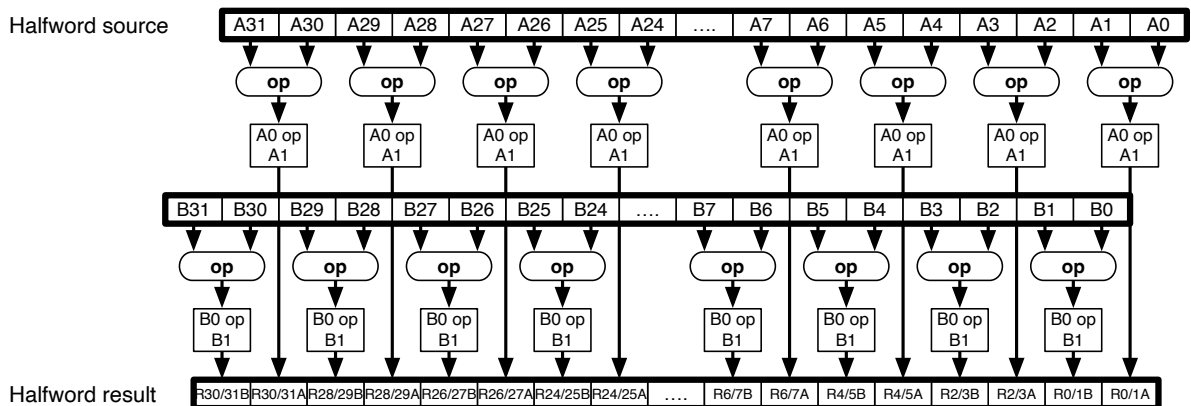
Mnemonic Code	Name	Interpretation	Instructions
HS	Higher Than or Same	unsigned $A \geq B$, unsigned $A \geq \#imm$	CMP<cond>
LE	Less Than or Equal	$A \leq B$ (v. only)	All aliased to GT with swapped operands (may not be available on all assemblers): CMP<cond>, FAC<cond>, FCM<cond>
		$A \leq B$ (w.e. only), $A \leq \#imm$	CMP<cond>, FAC<cond>, FCM<cond>
LO	Lower Than	unsigned $A < B$ (v. only)	All aliased to HS with swapped operands (may not be available on all assemblers): CMP<cond>
		unsigned $A < B$ (w.e. only), unsigned $A < \#imm$	CMP<cond>
LS	Lower Than or Same	unsigned $A \leq B$ (v. only)	All aliased to HI with swapped operands (may not be available on all assemblers): CMP<cond>
		unsigned $A \leq B$ (w.e. only), unsigned $A \leq \#imm$	CMP<cond>
LT	Less Than	$A < B$ (v. only)	All aliased to GE with swapped operands (may not be available on all assemblers): CMP<cond>, FAC<cond>, FCM<cond>
		$A < B$ (w.e. only), $A < \#imm$	CMP<cond>, FAC<cond>, FCM<cond>
NE	Not Equal	$A \neq B$, $A \neq \#imm$	CMP<cond>, FCM<cond>
UO	Unordered (only checks for NaN)	$A ? B$	FCM<cond>

4.6.7 SSVE Cross-Lane Paired or Horizontal: P / V

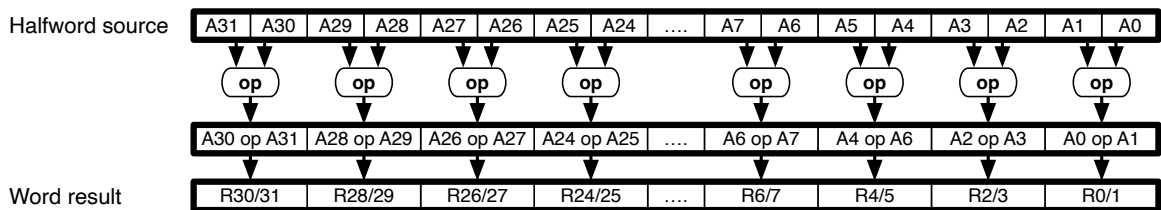
Paired or horizontal instructions operate similarly to their ASIMD counterparts (see [Section 3.3.18, "ASIMD Cross-Lane Paired or Horizontal: P / V"](#)) but the pattern of the output elements differs, keeping the destination lanes for element operations near the source lanes. [Figure 4.15: "SSVE: Horizontal \(Pairwise\) Instructions"](#) (A) depicts the destination interleaving pattern between the source operands for ADDP. Lengthening element operations shown in (B), such as for SADALP, behave like their ASIMD counterparts, where the destination occupies the same byte positions as the paired sources. The v versions combine all elements in the vector into a scalar, the same as ASIMD.

Figure 4.15. SSVE: Horizontal (Pairwise) Instructions

A) Non-Lengthening Pairwise Operation (P)



B) Lengthening Pairwise Operation (LP)



4.6.8 SSVE Reversed Inputs: R

Some instructions are destructive where the destination register is the same as one of the sources. For instructions that are commutative such as `ADD` or `MUL`, the operands can be swapped freely to choose which of the operands is overwritten. For non-commutative instructions such as `SUB` or `DIV`, the operands cannot be swapped while preserving desired functionality. However, the `R` version of the instruction reverses the order of the operands, thus offering an option as to which of the operands (e.g., dividend or divisor) is overwritten by the destination. Other than the swapped inputs, `R` operations are identical to their non-`R` equivalents.

4.6.9 SSVE/SME Target Specification: A / S / T

A few suffixes have assorted meanings:

- **A in `MOV/MOVB` (SME):** These instructions move data in and out of ZA storage. The ISA aliases non-A names to the A names, but there is no functional difference.
- **A in `ADDHA/ADDVA` (SME):** These instructions apply a single vector to each horizontal/vertical slice of a ZA tile.
- **A/s in `MOP` instructions (SME):** These instructions perform the desired multiplication and then either add to (accumulate) or subtract from the elements in the ZA tile.
- **T in `MOVMT` (SSVE):** These `MOV` instructions read and write the dedicated [Table Source \(ZT0\)](#) vector register instead of normal Z registers. They operate on Z registers, and are hence referred to here as SSVE, but are a portion of the SME ISA feature extension not available in the non-streaming SVE ISA.

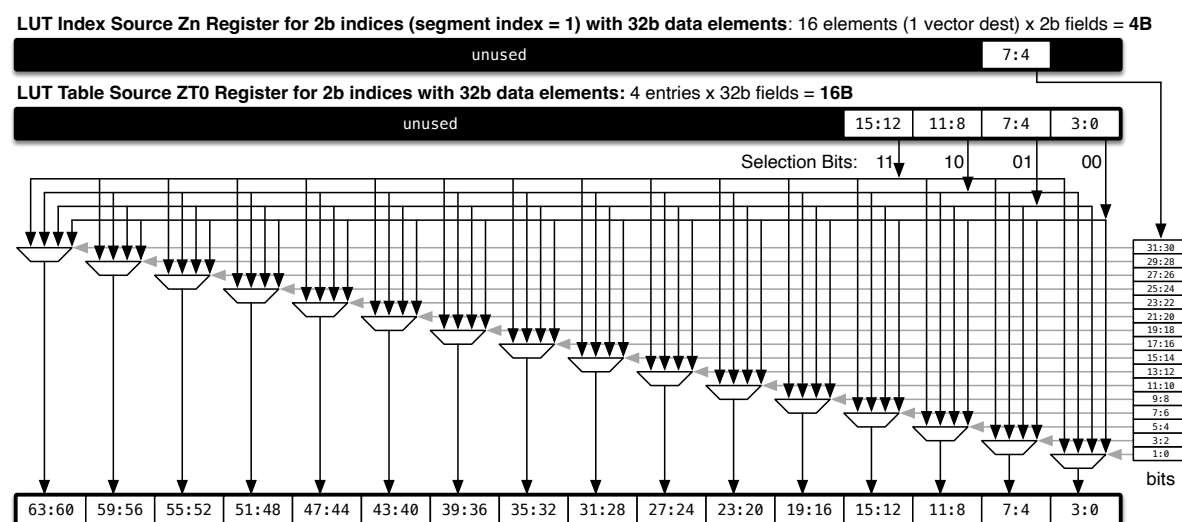
4.6.10 SSVE Lookup Table Instructions

The SSVE vector *Lookup Table* (LUT) instructions use narrow 2b or 4b indices to select elements from a dedicated Table Source (ZT0) vector register to populate each of the lanes in the destination register(s). They operate on Z registers, and are hence referred to here as SSVE, but are a portion of the SME ISA feature extension not available in the non-streaming SVE ISA. These instructions are commonly used in machine learning applications, where values such as input weights have been quantized down to a small number of bits and need to be expanded back to the size used in the overall computation. For example:

```
LUT12 <Zd>.<T>, ZT0, <Zn>[<index>] // Lookup table read with 2-bit indexes
```

First, the instruction reads a Z register to supply the necessary lookup indices, arranged into a packed table of two or four bits per output. The *segment* (portion) of the input index vector is chosen by the `<index>` value encoded in the instruction. In [Figure 4.16: “SSVE Lookup Table Instruction LUTI2 on 4B Elements with Single Vector Destination”](#), the input register with the LUT indices is shown in the top row, 2 bits for each of the 16 destination elements for a total of 4 bytes (16 x 2b) of indices. In this example, the indices come from segment 1 (`<index>`=1, bytes 7:4) of the index vector. Then, each 2-bit index in the segment (down the right side of the figure) is used to select one of the 4 packed 4-byte elements from the ZT0 register. LUTI2 uses 2-bit indices to select from 4 elements, and LUTI4 uses 4-bit indices to select from 16 elements, and each has 1-/2-/4-destination vector varieties (except for LUTI4 with 8-bit elements and 4 destination vectors, which is not supported).

Figure 4.16. SSVE Lookup Table Instruction `LUTI2` on 4B Elements with Single Vector Destination



The size of a segment in the input vector (and thus the number of segments) is dependent on the index bit length, the output element size, and the number of destination vectors. [Figure 4.17: “SSVE Lookup Table Register Usage”](#) depicts how the registers are used in various cases.

Figure 4.17. SSVE Lookup Table Register Usage

A) LUT Index Source Zn Register for 2-Bit Indices

8b (B) data elements: 64 x 2b fields = 16 bytes of indices per vector destination

1 vector dest	3 (63:48)	2 (47:32)	1 (31:16)	0 (15:0)
2 vector dest	1 (63:32)		0 (31:0)	
4 vector dest	0 (63:0)			

Key: Segment number (byte range)

16b (H) data elements: 32 x 2b fields = 8 bytes of indices per vector destination

1 vector dest	7 (63:56)	6 (55:48)	5 (47:40)	4 (39:32)	3 (31:24)	2 (23:16)	1 (15:8)	0 (7:0)
2 vector dest	3 (63:48)		2 (47:32)		1 (31:16)		0 (15:0)	
4 vector dest	1 (63:32)				0 (31:0)			

32b (S) data elements: 16 x 2b fields = 4 bytes of indices per vector destination range

1 vector dest	15 (63:60)	14 (59:56)	13 (55:52)	12 (51:48)	11 (47:44)	10 (43:40)	9 (39:36)	8 (35:32)	7 (31:28)	6 (27:24)	5 (23:20)	4 (19:16)	3 (15:12)	2 (11:8)	1 (7:4)	0 (3:0)
2 vector dest	7 (63:56)		6 (55:48)		5 (47:40)		4 (39:32)		3 (31:24)		2 (23:16)		1 (15:8)		0 (7:0)	
4 vector dest	3 (63:48)				2 (47:32)				1 (31:16)				0 (15:0)			

LUT Table Source ZT0 Register for 2b Indices: Byte range of 4 source elements

8b													1	2	8	4	0
16b													13:12	9:8	5:4	1:0	
32b													15:12	11:8	7:4	3:0	

B) LUT Index Source Zn Register for 4-Bit Indices

8b (B) data elements: 64 x 4b fields = 32 bytes of indices per vector destination

1 vector dest	1 (63:32)		0 (31:0)	
2 vector dest	0 (63:0)			
4 vector dest	Not supported			

16b (H) data elements: 32 x 4b fields = 16 bytes of indices per vector destination

1 vector dest	3 (63:48)		2 (47:32)		1 (31:16)		0 (15:0)	
2 vector dest	1 (63:32)				0 (31:0)			
4 vector dest	0 (63:0)							

32b (S) data elements: 16 x 4b fields = 8 bytes of indices per vector destination

1 vector dest	7 (63:56)		6 (55:48)		5 (47:40)		4 (39:32)		3 (31:24)		2 (23:16)		1 (15:8)		0 (7:0)	
2 vector dest	3 (63:48)				2 (47:32)				1 (31:16)				0 (15:0)			
4 vector dest	1 (63:32)								0 (31:0)							

LUT Table Source ZT0 Register for 4b Indices: Byte range of 16 source elements

8b	6	5	5	4	4	4	3	3	2	2	2	1	1	8	4	0
16b	61:60	57:56	53:52	49:48	45:44	41:40	37:36	33:32	29:28	25:24	21:20	17:16	13:12	9:8	5:4	1:0
32b	63:60	59:56	55:52	51:48	47:44	43:40	39:36	35:32	31:28	27:24	23:20	19:16	15:12	11:8	7:4	3:0

The Arm SME ISA includes a few basic operations on the dedicated Table Source (zT0) vector register:

```
LDR ZT0, [<Xn|SP>]           // Load ZT0 register
MOVT ZT0[<offs>], <Xt>        // Move 8 bytes from general-purpose register to ZT0
MOVT <Xt>, ZT0[<offs>]        // Move 8 bytes from ZT0 to general-purpose register
STR ZT0, [<Xn|SP>]           // Store ZT0 register
ZERO { ZT0 }                  // Zero ZT0
```

4.6.11 SME ZA Tile and Vector Computation Instructions

Most computational instructions that write the SME ZA storage use all or part of a multiply-accumulate arithmetic operation per output element. For instance, when ZA storage is used as 2D tiles, the outer product operation can be computed from the two input Z vectors, multiplying each input element of the first vector with each element of the second vector, and accumulating those with the existing values in a 2D tile. Yet other instructions may only perform addition (or subtraction), for example.

At the beginning of every grid computation, the ZA tile registers are typically initialized to zero. Software may clear out the entire ZA tile register file at once, or selectively clear out just a subset of the tile registers. Clearing the entire ZA storage is recommended, due to microarchitectural benefits.

```
ZERO { <mask> }
```

<mask> can be any tile or the whole tile register file such as ZA0.B, ZA6.D, list ZA0.S, ZA1.S, or simply ZA for the entire array. Note that the preferred disassembly for <mask> uses the shortest list of tile names that represent the encoded mask.

ZA tile registers can also be pre-loaded with existing data using MOV operations from Z registers or through loads directly from memory. For example:

```
LD1H { <ZAt><HV>.H[<Ws>, <offs>] }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #1}]
// Contiguous load of halfwords to 16-bit element ZA tile slice of tile <t>
// into a horizontal or vertical slice chosen by register <Ws> plus the
// immediate <offs>.

MOVA ZA.D[<Wv>, <offs>{, VGx2}], { <Zn1>.D-<Zn2>.D }
// Move two 'aligned contiguous' vector registers to two ZA single-vector groups,
// where <Zn1> is an even numbered register and <Zn2> is <Zn1>+1.

MOVA <ZAd><HV>.S[<Ws>, <offsf>:<offsl>], { <Zn1>.S-<Zn2>.S }
// Move two 'aligned contiguous' vector registers to two ZA tile slices,
// where <Zn1> is an even numbered register and <Zn2> is <Zn1>+1.
```

The SME engine supports various forms of the *outer product* operation, which performs multiply-accumulate (MOPA) or multiply-subtract (MOPS) instructions for each tile element based on two Z register vector inputs, with z_n distributed across all of the tile columns while z_m is distributed across tile rows. As a result, any column/row intersection point inherently selects an element from z_n (based on the column) and from z_m (based on the row). This instruction is designed to act as the core instruction for matrix multiply operations, as can be seen in [Matrix-Matrix Multiply \(Same Element Size\): “Matrix-Matrix Multiply \(Same Element Size\)”](#).

There are varieties of MOP instructions to handle several different element types, as listed in [Table 4.10: “ZA Tile Operation Type Support”](#). Selecting a balanced intermediate

element data type to cover the range of values while enabling the densest element count is critical, as it was for ASIMD (see [Advanced SIMD and FP Data Types: "Advanced SIMD and FP Data Types"](#)). Instructions with input types smaller than 32 bits always use lengthening (L) or double-lengthening (LL) in order to allow the ZA tile accumulators to use extended precision and therefore minimize rounding for floating point elements or provide sufficient range for integer elements. Note that MOP instructions support predication in *both* dimensions, using predicate registers associated with both the z_n and z_m input vectors. (The net predicate at each point of the tile is the logical-AND of the row and column predicates, so both must be predicated to true in order to generate a result for that element.) Although this is usually used to avoid computing off of the edge of matrices, it can also be used to do things such as interleave computations using different vectors into "checkerboard patterns" of various kinds, all under the control of predicates.

The ISA also features full-tile ADD instructions, ADDHA and ADDVA, which spread elements of a Z vector to tile elements across all of the rows/columns at once and add them to a ZA tile register.

Table 4.10. ZA Tile Operation Type Support

Group	Source Type(s)	Destination Type(s)	Instructions
Floating Point	bfloat16	float32	BFMOPA/BFMOPS
	float16		FMOPA/FMOPS
	float32		FMOPA/FMOPS
	float64	float64	
Integer	sint8/uint8	sint32/uint32	SMOPA/SMOPS
	sint16/uint16	sint32/uint32, sint64/uint64	SUMOPA/SUMOPS
			UMOPA/UMOPS
			USMOPA/USMOPS
	sint32/uint32	sint32/uint32	ADDHA/ADDVA
			BMOPA/BMOPS
	sint64/uint64	sint64/uint64	ADDHA/ADDVA

The ISA offers a similar set of instructions for ZA vectors, as listed in [Table 4.11: "ZA Vector Operation Type Support"](#). Unlike MOP operations, these instructions generally do *not* support predication; instead, "extra" vector lanes can simply be predicated off during loads before and stores after the ZA vector operations. This zeros the lanes beforehand and ignores them when storing out the results. If element (de)interleaving or other processing is needed, it must be done before computation or after extraction from ZA using SSVE operations.

Table 4.11. ZA Vector Operation Type Support

Group	Source Type(s)	Destination Type(s)	Instructions
Floating Point	bfloat16	float32	BFDOT/BFVDOT
	float16		BFMLAL/BFMLS
			FDOT/FVDOT

Table 4.11. ZA Vector Operation Type Support

Group	Source Type(s)	Destination Type(s)	Instructions
			FMLAL/FMLSL
	float32		FADD/FSUB
	float64	float64	FMLA/FMLS
Signed and Unsigned Integer	sint8/uint8	sint32/uint32	SDOT, SMLALL/SMLSLL, SUDOT, SUMLALL, SUVDOT, SVDOT, UDOT, UMLALL/UMLSLL, USDOT, USMLALL, USVDOT, UVDOT
	sint16/uint16	sint32/uint32	SDOT, SMLAL/SMLSL, SVDOT, UDOT, UMLAL, UMLSL, UVDOT
		sint64/uint64	SDOT, SMLALL/SMLSLL, SVDOT, UDOT, UMLALL/UMLSLL, UVDOT
	sint32/uint32	sint32/uint32	ADD/SUB
	sint64/uint64	sint64/uint64	

Both ZA tile and vector instructions have a number of features that can be selected using their various different mnemonics and operand types:

- Source vectors of 8-/16-bit integer values and 16-/32-/64-bit floating point values (including bfloat16)
- Integer computations in signed, unsigned, and some mixed data types
- Multi-row vector multiplication by a replicated scalar, replicated single 64-byte vector, or another entire multi-vector, based on the format of the second operand (see [Section 4.5.3, "Element Indexing"](#))
- Double-length (L) and quad-length (LL) accumulation to ZA storage for all 8-/16-bit data types, allowing these smaller data types to be used with 32-bit or 64-bit accumulators in order to avoid accumulator overflow/saturation (for integer operations) or excess rounding (for floating-point operations)

For both ZA tile and ZA vector floating point instructions, the ISA offers full support for denormal values.

After calculations are performed, the contents of ZA storage must be extracted into Z registers or stored to memory. For example:

```

ST1H { <ZAt><HV>.H[<Ws>, <offs>] }, <Pg>, [<Xn|SP>{, <Xm>, LSL #1}]
// Contiguous store of halfwords from 16-bit element ZA tile slice of tile <t>
// from a horizontal or vertical slice chosen by register <Ws> plus the
// immediate <offs>.

MOVA { <Zd1>.D-<Zd2>.D }, ZA.D[<Wv>, <offs>{, VGx2}]
// Move two ZA single-vector groups to two aligned contiguous vector registers,
// where <Zn1> is an even numbered register and <Zn2> is <Zn1>+1.

MOVA { <Zd1>.H-<Zd2>.H }, <ZAn><HV>.H[<Ws>, <offs1>:<offs2>]
```

```
// Move two consecutive ZA tile slices to two aligned contiguous vector registers,
// where <Zn1> is an even numbered register and <Zn2> is <Zn1>+1.
```

In many cases, result data in ZA storage must be down-converted to smaller data types or interleaved/deinterleaved before being stored to memory, which requires extraction to Z vector registers and additional SSVE Z vector operations. The microarchitecture can optimize this phase with fused instruction pairs (see [Section 6.8, "SSVE/SME Instruction Operation Fusion"](#)).

4.6.11.1 SME Outer Product (MOP) Tile Instructions

Outer product MOP instructions are computed across the entire accumulator tile in *cells* that can each execute mathematical operations independently. For 64-bit elements, each computational cell processes the element at the intersection of one row and one column, while selecting from eight available 64-bit tile registers in its portion of ZA. In contrast for 32-bit elements, each computation cell processes a two-row by two-column portion of the computation covering four elements. Because each cell produces 4 x 32b = 128b worth of results with each instruction, its portion of ZA is only divided across four tiles.

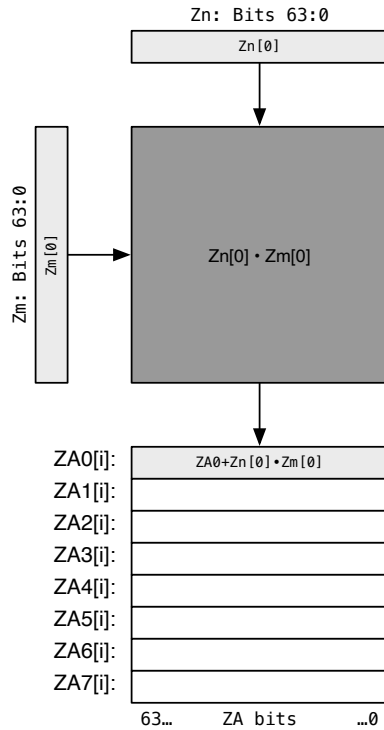
[Figure 4.18: "ZA Tile Element Operation Patterns for 16/32/64-bit Input MOP Instructions"](#) and [Figure 4.19: "ZA Tile Element Operation Patterns for 8-bit Input MOP Instructions"](#) depict the operations performed in each computational cell. For the 32- or 64-bit forms shown in subfigures (A) and (B), the operations simply multiply the z_n and z_m elements at each grid intersection and then adds those results into an accumulator element (or four elements) located in the cell of the ZA tile at that position. As was noted in [Section 4.1.3.1, "ZA Tile Registers \(\$zA<n>.<size>\$ \)"](#), the actual rows in ZA storage are spaced eight rows apart for 64-bit operations, or four rows apart for 32-bit operations, corresponding to the numbered registers.

For 8- or 16-bit operands shown in subfigures (C)-(E), only the sub-elements down the main diagonals within each surrounding 32- or 64-bit cell are multiplied with each other. Arm refers to these as "4-way" (for 8b) "2-way" (for 16b) and are essentially four-input and two-input dot product operations across the neighboring elements. The results of these four or two multiplications are then summed together before being added to the 32- or 64-bit accumulator for that part of the tile. (As previously discussed, the larger accumulator data elements provide additional precision or range.)

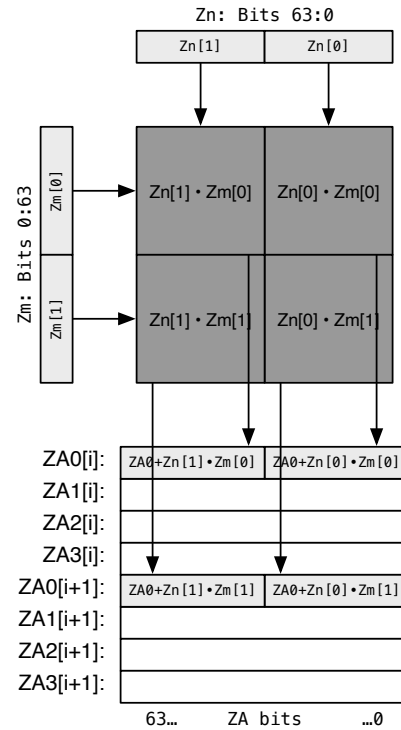
With properly interleaved operands, this operation allows each MOP to execute one, two, or four iterations of the matrix multiply inner loop at once with each instruction. [Section 4.7, "SSVE/SME Coding Examples"](#) provide some basic examples that use these instructions.

Figure 4.18. ZA Tile Element Operation Patterns for 16/32/64-bit Input MOP Instructions

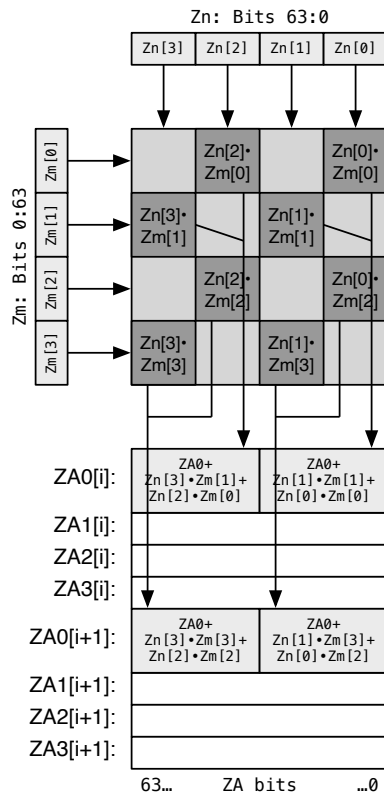
A) MOP D-to-D



B) MOP S-to-S



C) MOP H-to-S



D) MOP H-to-D

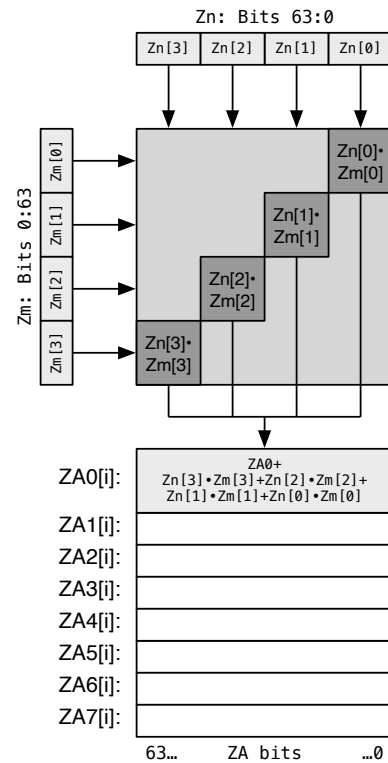
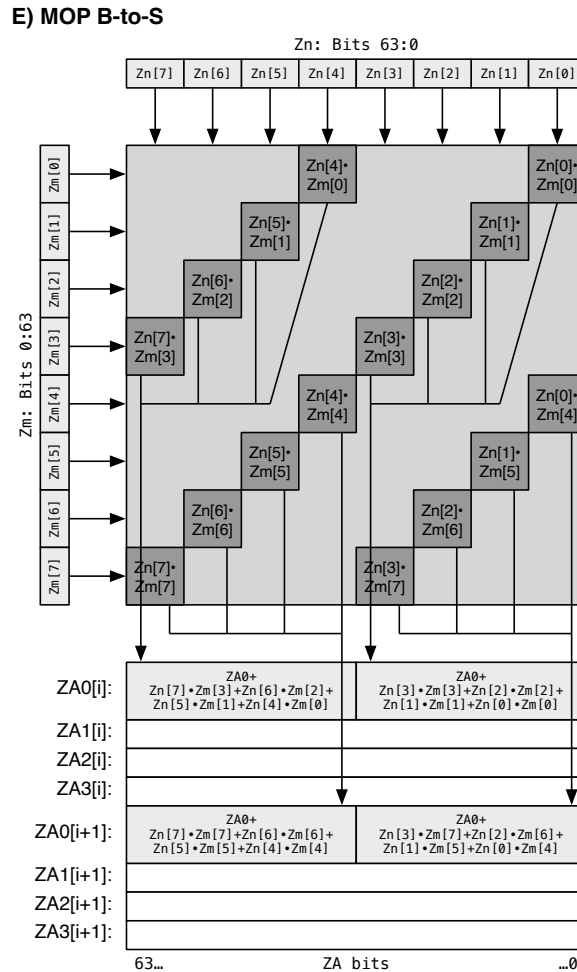


Figure 4.19. ZA Tile Element Operation Patterns for 8-bit Input MOP Instructions



4.6.11.2 SME Dot Product (DOT) Vector Instructions

The DOT operations are the vector equivalents of the MOP tile operations working with 8- or 16-bit operands, and are illustrated in [Figure 4.20: “ZA Vector Element Patterns for DOT Instructions”](#). They perform multiplies of each pair of input operands and then sum all of the results together in each 32- or 64-bit group with the ZA vector destination register contents for that group, all in one unified instruction. The vertical versions of the DOT instructions collect their Z_n input elements across the multi-vector instead of from neighboring elements, as described in [Section 4.6.3, “SME Horizontal / Vertical in Vector: H / V \(Prefix\)”](#)

the ZA vector accumulators in the same order as the input operands, the two or four result sub-vectors need to be re-interleaved together with SSVE `ZIP` instructions after they are extracted from the ZA vector registers to Z registers.

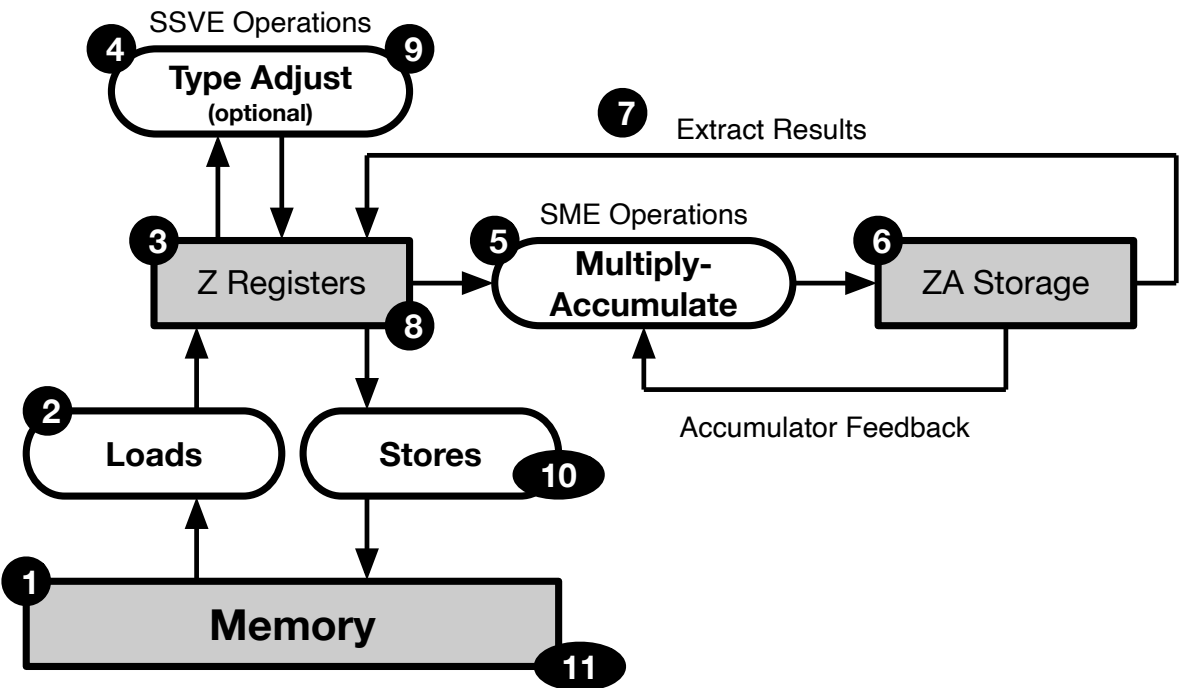
4.7 SSVE/SME Coding Examples

The following subsections provide some basic examples of SSVE and SME algorithms. These examples demonstrate the fundamentals of the ISA and include some of the important microarchitectural optimizations described in [Chapter 6, SME Engine Microarchitecture Optimization](#). They have not been maximally optimized and do not handle some corner or edge cases. Still, they can provide an example of how SSVE/SME code could be used to accelerate mathematically intense software.

4.7.1 General Flow of an SSVE/SME Routine

The general flow of most SSVE/SME routines follows the pattern described in [Figure 4.22: “SSVE/SME Data Flow Overview”](#). Matrix and vector data stored in memory (1) is loaded (2) into the Z registers (3), which act as a staging area for incoming data. If the incoming data needs to have its type adjusted, upsized, downsized, rearranged, or decompressed in some way to prepare it for computation, SSVE vector operations can be invoked to make the necessary adjustments (4). Multiply-accumulate instructions then multiply streams of incoming data (5) and accumulate the results with existing contents of a section of the ZA storage (6). Most of the execution time in a typical SSVE/SME routine is occupied performing long streams of this (1–6) process. When an entire stage of computation is complete, results are then extracted (7) back out to the Z registers (8). If necessary, the extracted results can be adjusted, upsized, downsized, rearranged, compressed prior to storage in memory using SSVE operations (9). These can then be used again for further computations, looping through (5–8), or stored (10) back to memory (11). Most SSVE/SME routines use some variation of this basic data flow, just varying the particular adjustment instructions used, multiply-accumulate operation type, and the data access patterns to perform different kinds of operations.

Figure 4.22. SSVE/SME Data Flow Overview



4.7.2 SME Matrix Transposition Example

Matrix transposition is a fairly straightforward operation using SME tile operations. The matrix can be read from memory as rows and filled into the tile by rows, and then read out of the tile by columns and written into memory as rows.

```
// Transpose the A matrix
// -- Only needed if transpose is not already available from a previous calculation

// -- SVL-agnostic; operates properly for any allowed SVL

float32_t A[M*P];           // M rows x P columns
float32_t AT[P*M];          // A, transposed: P rows x M columns
svfloat32_t ZrowBuffer;     // Z register.  svfloat32_t is equivalent to
                             // float32x16_t on Apple silicon, but varies by SVL
                             // on other platforms

float32_t *aPtr, *atPtr;
svbool_t Pcol, Prow;

CNTW NumEls, ALL            // Number of float32 elements in SSVE vector

// -- Loop down sets of rows
for (i=0; i < M; i += NumEls) {
    WHILELT Pcol.S, i, M
    // -- Loop across rows
    for (j=0; j < P; j+= NumEls) {
        WHILELT Prow.S, j, P
        // -- Read in block to ZA, as rows
        aPtr = &(A[i*P + j]);
        for (k=0; k < NumEls && (i+k < M); k++) {    // Avoid reading extra rows
                                                         // (which are 0s in Pcol)
            LD1W { ZrowBuffer.S }, Prow/Z, [(svfloat32_t *) aPtr]
            MOVA ZA0H.S[k, 0], Prow/M, ZrowBuffer.S    // Load into tile via ZA0H
                                                         // (horizontal) writes
            aPtr += P;                                  // Jump to next row
        }
        // -- Write out block from ZA, as columns
        atPtr = &(AT[j*M + i]);
        for (k=0; k < NumEls && (j+k < P); k++) {
            MOVA ZrowBuffer.S, Pcol/M, ZA0V.S[k, 0]    // Extract from tile via ZA0V
                                                         // (vertical) reads
            ST1W { ZrowBuffer.S }, Pcol, [(svfloat32_t *) atPtr]
            atPtr += M;                                  // Jump to next row
        }
    }
}
}
```

For comparison, [Section 3.5.7.1, "Byte Shuffle Example: Transposing Matrices"](#) discusses matrix transposition with ASIMD.

This routine involves a full load-insert-extract-store cycle, so this rearrangement operation takes nearly as long as a simple matrix computation, even though it is not actually computing anything. At the cost of extra memory footprint, a routine that produces the input matrix might extract it out of ZA storage into memory directly in transposed form.

For more on how predication is used (Prow and Pcol in this example), see [Section 4.4.3, "Data Structure Edge Control"](#).

From a microarchitectural performance standpoint, there are other enhancements that could be applied to this routine. See [Chapter 6, SME Engine Microarchitecture Optimization](#) to learn about the underlying hardware implementation. In this example, software pipelining techniques that use additional ZA tiles are likely to improve performance. Although the ASIMD version of transpose requires more instructions, it may be faster for small matrices that fit in the L1D caches, which SSVE/SME cannot use (see [Section 6.5, “SME Engine Load/Store Execution Unit Optimization”](#)).

4.7.3 Matrix-Matrix Multiply (Same Element Size)

The SME tile instruction set has been designed to efficiently execute matrix-matrix multiply operations. In this example, the elements are 32b single-precision floating point (S-size) values. The inner loop body below computes the outer product of two 16-element input row vectors by another two 16-element input column vectors (for SVL=64B). Each of the four MOP instructions performs 256 parallel single-precision multiply-accumulate operations across all combinations of the 16-element input vectors. The two outer loops scan over the entire result matrix, block-by-block, while the inner loops perform the necessary computation (in the first inner loop) and extraction (in the second) needed within each block. [Figure 4.23: “SME Matrix-Matrix Multiply Memory Access Pattern”](#) illustrates the memory access patterns. This code sequence also writes out both the normal and transposed forms of the result, which may facilitate faster consumption of the matrix later at the expense of additional memory usage.

```
// Simple SME Matrix Multiply
// A X B = C, MxP X PxN = MxN matrix-matrix multiply

// -- SVL-agnostic; operates properly for any allowed SVL

float32_t AT[P*M];           // A, transposed: P rows x M columns
float32_t B[P*N];           // P rows x N columns
float32_t C[M*N];           // M rows x N columns
float32_t CT[N*M];          // C, transposed (optional): N rows x M columns
svfloat32_t xIn1, xIn2;
svfloat32_t yIn1, yIn2;
float32_t *atPtr, *bPtr;    // Source pointers
float32_t *cPtr, *ctPtr;    // Destination pointers
int i, j, k, numElements;
svbool_t Pcol0, Prow0, Pcol1, Prow1;
svcount_t PNcol, PNrow      // Counted predicates

CNTW numElements, ALL, MUL #2 // numElements = 32 = 2*16e in Apple silicon

// Loop over Z-blocks of destination matrix C
// -- Loop down columns
for (i=0; i < M; i += numElements) {
    WHILELT PNcol.S, i, M, VLx2
    PEXT { Pcol0.S, Pcol1.S }, PNcol[0]
    // -- Loop across rows
    for (j=0; j < N; j += numElements) {
        WHILELT PNrow.S, j, N, VLx2
        PEXT { Prow0.S, Prow1.S }, PNrow[0]

        // Calculate this 2xSVL x 2xSVL block
        // -- Reset input pointers to tops of next set of columns
        atPtr = &(AT[i]);
```

Apple Silicon CPU Optimization Guide: 4.0

ISA Optimization: Scalable Matrix Extension (SME)

Matrix-Matrix Multiply (Same Element Size)

```

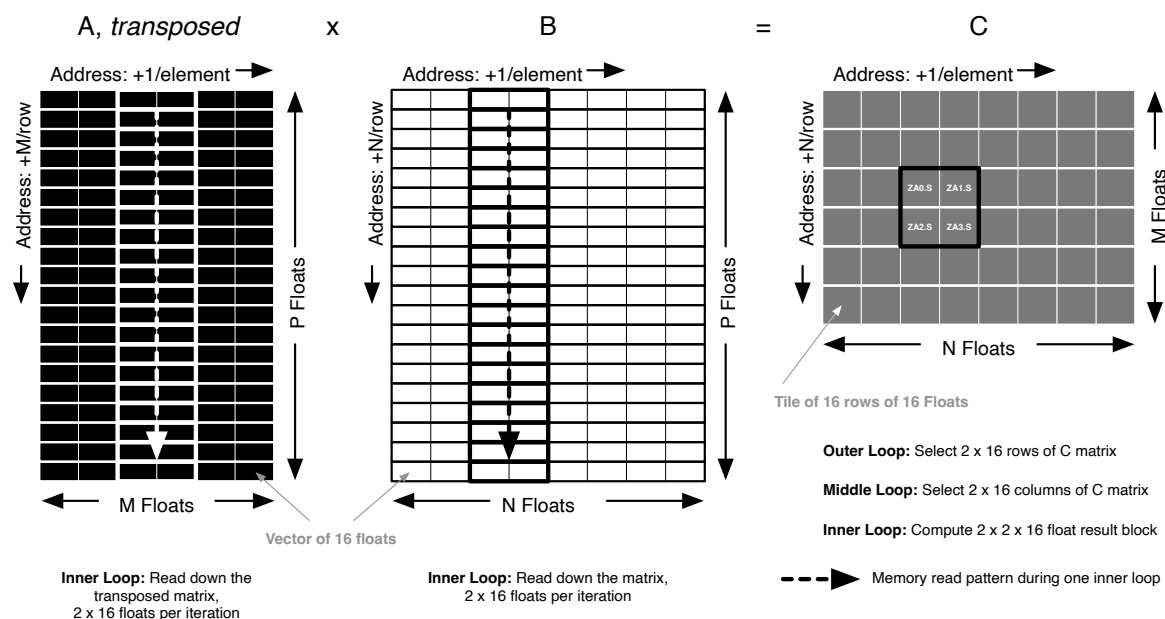
bPtr = &(B[j]);
// -- Clear out accumulator before next block
ZERO { ZA }

// -- Loop over the "inner" P rows of AT/B matrices
// loading two consecutive row vectors and column vectors
for (k=0; k < P; k++) {
    LD1W { yIn1.S, yIn2.S }, PNcol/Z, [(svfloat32_t *) atPtr]
    atPtr += M; // Jump to next column (row in transpose)
    LD1W { xIn1.S, xIn2.S }, PNrow/Z, [(svfloat32_t *) bPtr]
    bPtr += N; // Jump to next row
    // -- MOP a full 2x2 "square" at once
    FMOPA ZA0.S, Prow0/M, Pcol0/M, xIn1.S, yIn1.S
    FMOPA ZA1.S, Prow1/M, Pcol0/M, xIn2.S, yIn1.S
    FMOPA ZA2.S, Prow0/M, Pcol1/M, xIn1.S, yIn2.S
    FMOPA ZA3.S, Prow1/M, Pcol1/M, xIn2.S, yIn2.S
}

// Extract and store the 2xSVL x 2xSVL block (normal and transpose)
// -- Loop over columns/rows
// -- Store normal or transposed (or both) versions based on algorithm
cPtr = &(C[i*N + j]);
ctPtr = &(CT[j*M + i]);
// -- C (normal)
for (k=0; (k < numElements/2) && (i+k < M); k++) {
    // -- write out two rows of "normal" C matrix as rows
    MOVA zOut1.S, Prow0/M, ZA0H.S[k, 0] // ZA0H horizontal read-out
    MOVA zOut2.S, Prow1/M, ZA1H.S[k, 0] // ZA1H horizontal read-out
    ST1W { zOut1.S, zOut2.S }, PNrow, [(svfloat32_t *) cPtr]
    cPtr += N; // Jump to next row
}
for (k=0; (k < numElements/2) && (i+numElements/2+k < M); k++) {
    MOVA zOut1.S, Prow0/M, ZA2H.S[k, 0] // ZA2H horizontal read-out
    MOVA zOut2.S, Prow1/M, ZA3H.S[k, 0] // ZA3H horizontal read-out
    ST1W { zOut1.S, zOut2.S }, PNrow, [(svfloat32_t *) cPtr]
    cPtr += N; // Jump to next row
}
// -- CT (transposed)
for (k=0; (k < numElements/2) && (j+k < N); k++) {
    MOVA zOut1.S, Pcol0/M, ZA0V.S[k, 0] // ZA0V vertical read-out
    MOVA zOut2.S, Pcol1/M, ZA1V.S[k, 0] // ZA1V vertical read-out
    ST1W { zOut1.S, zOut2.S }, PNcol, [(svfloat32_t *) ctPtr]
    ctPtr += M; // Jump to next row
}
for (k=0; (k < numElements/2) && (j+numElements/2+k < N); k++) {
    MOVA zOut1.S, Pcol0/M, ZA2V.S[k, 0] // ZA2V vertical read-out
    MOVA zOut2.S, Pcol1/M, ZA3V.S[k, 0] // ZA3V vertical read-out
    ST1W { zOut1.S, zOut2.S }, PNcol, [(svfloat32_t *) ctPtr]
    ctPtr += M; // Jump to next row
}
}
}

```

Figure 4.23. SME Matrix-Matrix Multiply Memory Access Pattern



For comparison, [Section 3.6.1.1, “Matrix-Matrix Multiply”](#) provides a similar matrix-matrix multiply but using ASIMD instructions.

From a microarchitectural performance standpoint, several enhancements were employed to improve performance. See [Chapter 6, SME Engine Microarchitecture Optimization](#) to learn about the underlying hardware implementation. To keep the larger implementation of the SME engine close to full utilization, four `FMOP` operations operate on four ZA tiles to cover the four-cycle latency of S-type `FMOP`. To accomplish this, the inner `k` loop computes a 2×2 block of 16×16 S-size elements (each a tile), with each `FMOP` working on one of the four quadrants. The algorithm also uses two-vector loads (`LD1W`) and stores (`ST1W`), which in this case don't necessarily improve memory bandwidth, but they do reduce the number of issued instructions. Lastly, when starting the computation for the new output 2×2 block, the algorithm begins with a `ZERO { ZA }` instruction. This leverages the limited coarse-grained microarchitectural renaming of the ZA storage (see [Section 6.3, "SME Processing Grid Optimization"](#)), which allows computation to begin on the next 2×2 block while the values from the previous block are extracted and stored. Note that the overall algorithm may benefit from additional memory management optimizations, such as blocking and prefetching.

For small matrices, the ASIMD version may be faster than the SME equivalent described in this section, but as the matrices get larger, SME's ability to perform 256 multiplies of this type per cycle in parallel becomes significant.

4.7.4 Matrix-Matrix Multiply (Double Lengthening Element Size)

The matrix-matrix multiply example in [Section 4.7.3, “Matrix-Matrix Multiply \(Same Element Size\)”](#) was performed with floating point values where the accumulator element size was the same as the input element size. However, in integer code, the accumulator element size is often larger (2x or 4x bit length) than the input element size. This is often important to avoid overflowing the accumulator element with the result of the

multiplication and repeated summations. See discussions in [Section 3.1, “Advanced SIMD and FP Data Types”](#) and [Section 3.3.16, “ASIMD Narrow, Long, Wide: N / L / W”](#).

As shown in [Section 4.6.11.1, “SME Outer Product \(MOP\) Tile Instructions”](#), for these double lengthening cases, the MOP instruction performs a dot product on subvectors of two or four elements. The following example shows the innermost loop body of a matrix-matrix multiply of 8b integer input elements to 32b integer output elements using the B-to-S MOP instruction shown in [Figure 4.19: “ZA Tile Element Operation Patterns for 8-bit Input MOP Instructions”](#). The memory access pattern and vector manipulations are shown in [Figure 4.24: “SME Matrix-Matrix Multiply with Double Lengthening Memory Access Pattern”](#). This example provides a starting point for using these double lengthening instructions, showing the inner loop body for this type of computation. The remainder of the code, including the looping, is similar to the simple matrix multiply example in the previous section.

For each output element, the SMOPA instruction accumulates four per-element multiplications. In the previous matrix-matrix multiply example, for a particular output element, the inner loop body accumulated the multiplication of just one element from the input column of A-transpose with just one element from the input column of B, where the inner loop sweeps down the full length of those columns one by one. In this example, for a particular output element, the inner loop body accumulates the multiplications from four successive elements from the input column of matrix A-transpose with the four successive elements from the input column of matrix B, which is the dot product of those elements. To feed the SMOPA instruction, four rows of the columns of matrix A-transpose and matrix B are read into vector registers. Then the elements must be rearranged (zipped) such that elements of a particular single-element column are adjacent in the vectors. (See ZIP in [Section 3.5.7, “Shuffle and Permute Instructions”](#); the interleaving pattern is the same between ASIMD and SSVE, but can be done in one SSVE instruction instead of requiring 1 and 2 instructions.) To complete a particular output element, the inner loop sweeps down the columns in steps of four rows.

Note, however, that four SMOPAs (using the four S-size output tiles) only consume some of the loaded/zipped values; zipped `yIn{3,4}.B` and zipped `xIn{3,4}.B` are not used. 16 S-size output tiles would be needed to fully utilize all of the loaded/zipped elements, but the architecture only supports four. They are essentially by-products of the minimum load size of 64B. (There would be no advantage in attempting to mask off those loads with predication.) Although it might be tempting to save/restore the ZA tiles in order to consume the extra elements, the implementation is optimized for successive accumulations and it is best to proceed down the columns to complete the particular output elements.

```
// Simple SME Matrix Multiply w/ Double Lengthening
// — PARTIAL example illustrating the inner loop body

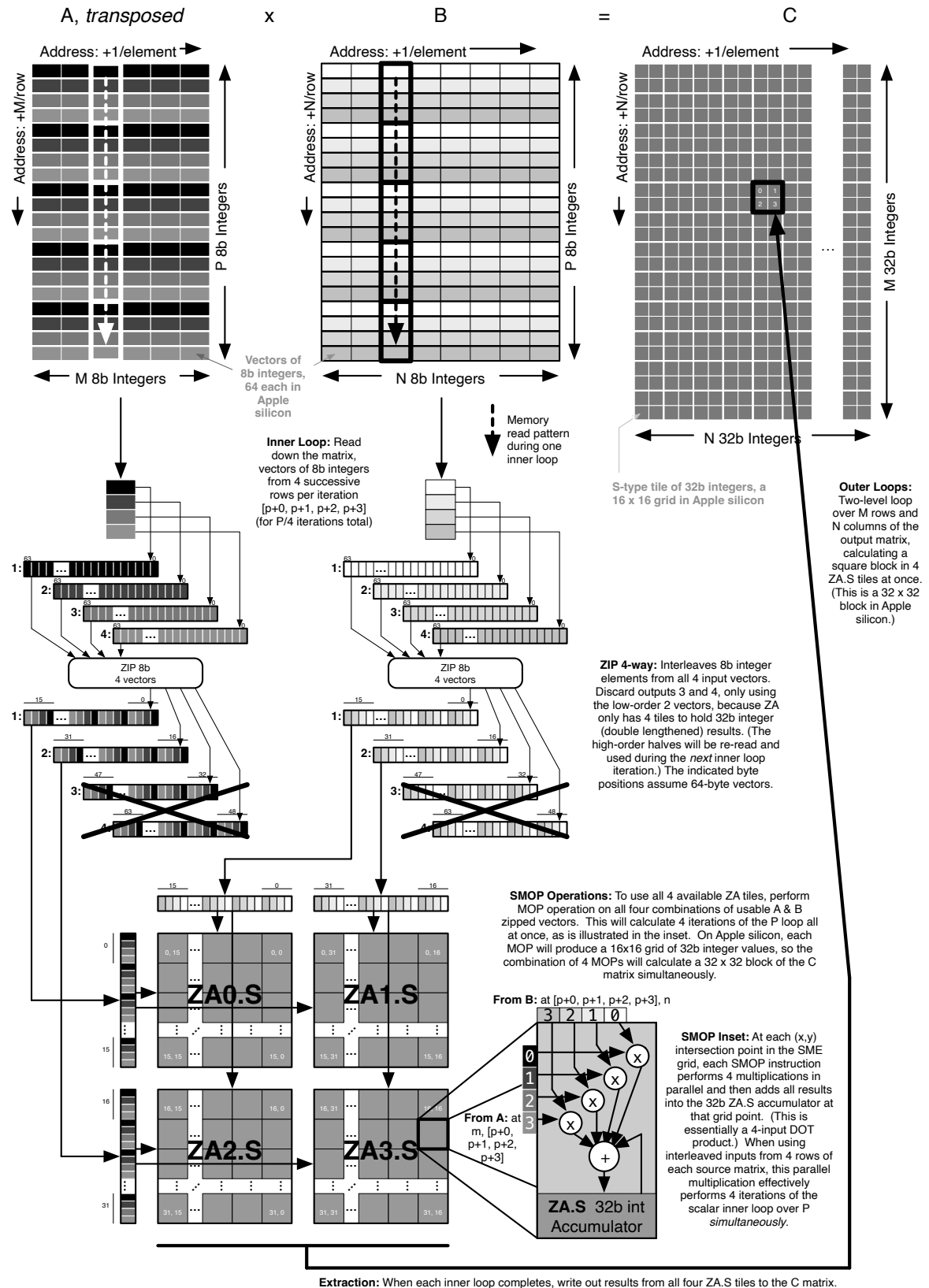
// — SVL-agnostic; operates properly for any allowed SVL

svint8_t xIn1, xIn2, xIn3, xIn4;
svint8_t yIn1, yIn2, yIn3, yIn4;
svint8_t *aIPtr, *bIPtr;                                // int8_t input matrices

// Inner loop body
```

161

Figure 4.24. SME Matrix-Matrix Multiply with Double Lengthening Memory Access Pattern



4.7.5 Matrix-Vector Multiply

In this next example, the code uses sequences of vector multiplications to calculate the output vector (one column, in this case) from a matrix-vector multiply operation. Instead of using SME ZA tile instructions, SME ZA vector instructions consume four Z vector inputs that operate on an SME four single-vector group (VGx4). The actual computation is performed with a vector FMLA instruction that multiplies all values in a multi-vector with a series of scalar values selected from another Z register. In this way, the *entire* input vector D is used in the calculation of every element of the output vector E. [Figure 4.25: "SME Matrix-Vector Multiply Memory Access Pattern"](#) shows the memory access patterns used by the following code:

```
// Simple SME Matrix-Vector Multiply
// A X D = E, MxP X Px1 = Mx1 matrix-vector multiply

// -- SVL-agnostic; operates properly for any allowed SVL
// -- Core loop requires that P be an integer multiple of 4

float32_t AT[P*M];           // A, transposed: P rows x M columns
float32_t D[P];              // P rows x 1 columns
float32_t E[M];              // M rows x 1 column
svfloat32_t xIn1, xIn2, xIn3, xIn4;
svfloat32_t yIn, yIndex;
float32_t *atPtr, *dPtr;     // Source pointers
float32_t *ePtr;             // Destination pointers
int i, j, k, numElements;
svbool_t Prow;
svcount_t PNcol;             // Counted predicates

CNTW numElements, ALL        // numElements = 16 in Apple silicon
ePtr = E;

// Loop over Z-vectors of destination matrix E
// -- Loop down vector segments
for (i=0; i < M; i += numElements*4) {
    WHILELT PNcol.S, i, M, VLx4

    // Calculate this 4*numElements elements of output
    // -- Clear accumulator contents
    ZERO { ZA }
    // -- Reset input pointers to top of next set of columns
    atPtr = &(AT[i]);
    dPtr = D;
    // -- Loop over the "inner" P rows of AT matrix/D vector
    for (j=0; j < P; j += numElements) {
        WHILELT Prow.S, j, P
        // Load next segment of the D vector
        LD1W { yIn.S }, Prow/Z, [(svfloat32_t *) dPtr]
        dPtr += numElements; // Jump to next vector segment
        // -- Multiply by each scalar value from D vector, 4 elements per loop body
        // -- Core loop requires that P be an integer multiple of 4!
        for (k=0; (k < numElements) && (j+k < P); k += 4) {
            // Replicate lowest 16 bytes of yIn across yIndex
            DUP yIndex.Q, yIn.Q[0]
            // Process four rows from matrix using 4 scalar values from vector
            LD1W { xIn1.S, xIn2.S, xIn3.S, xIn4.S }, PNcol/Z, [(svfloat32_t*) atPtr]
            atPtr += M; // Jump to next row
        }
    }
}
```

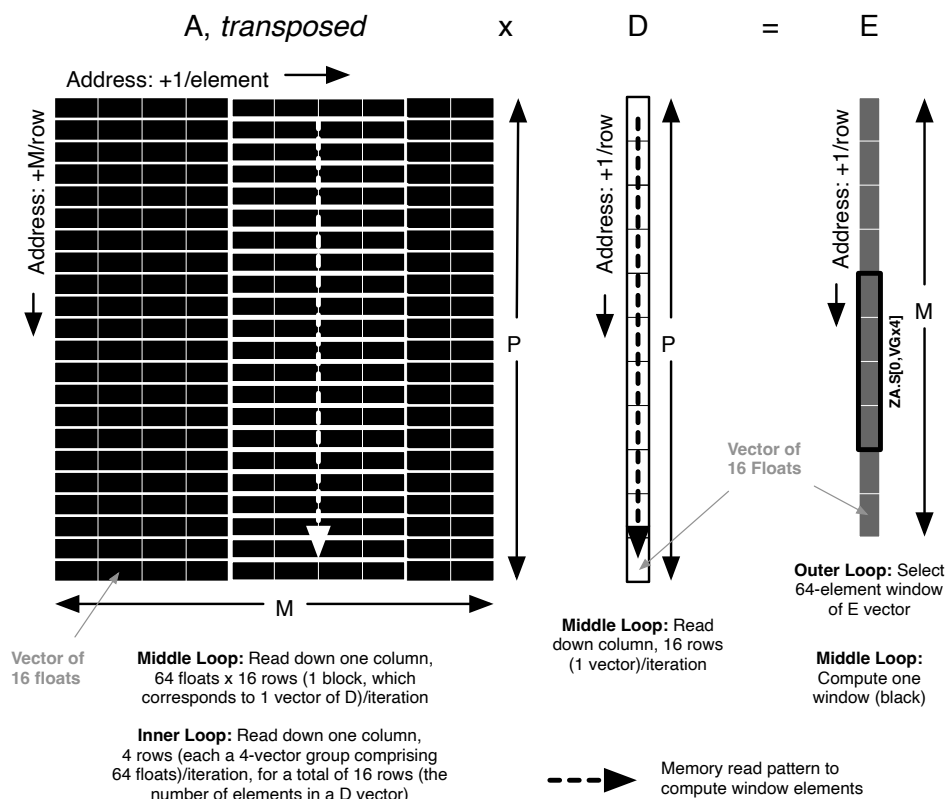
```

        FMLA ZA.S[0, 0, VGx4], { xIn1.S, xIn2.S, xIn3.S, xIn4.S }, yIndex.S[0]
        LD1W { xIn1.S, xIn2.S, xIn3.S, xIn4.S }, PNcol/Z, [(svfloat32_t*) atPtr]
        atPtr += M; // Jump to next row
        FMLA ZA.S[0, 0, VGx4], { xIn1.S, xIn2.S, xIn3.S, xIn4.S }, yIndex.S[1]
        LD1W { xIn1.S, xIn2.S, xIn3.S, xIn4.S }, PNcol/Z, [(svfloat32_t*) atPtr]
        atPtr += M; // Jump to next row
        FMLA ZA.S[0, 0, VGx4], { xIn1.S, xIn2.S, xIn3.S, xIn4.S }, yIndex.S[2]
        LD1W { xIn1.S, xIn2.S, xIn3.S, xIn4.S }, PNcol/Z, [(svfloat32_t*) atPtr]
        atPtr += M; // Jump to next row
        FMLA ZA.S[0, 0, VGx4], { xIn1.S, xIn2.S, xIn3.S, xIn4.S }, yIndex.S[3]
        // Rotate yIn by 16 bytes for next iteration
        EXT yIn.B, yIn.B, yIn.B, #16
    }
}

// Extract and store the vector segment
MOVA { zOut1.D,zOut2.D,zOut3.D,zOut4.D }, ZA.D[0,0]
ST1W { zOut1.S,zOut2.S,zOut3.S,zOut4.S }, PNcol, [(svfloat32_t*) ePtr]
ePtr += 4*numElements; // Jump to next output vector segment
}

```

Figure 4.25. SME Matrix-Vector Multiply Memory Access Pattern



Note that the example in [Section 3.6.1.2, “Matrix-Vector Multiply”](#) is roughly the equivalent code using the ASIMD instruction set.

This example also uses the vector indexing techniques described in [Section 4.5.3, “Element Indexing”](#) to select the individual elements from the D vector.

From a microarchitectural performance standpoint, several enhancements were employed to improve performance. See [Chapter 6, SME Engine Microarchitecture Optimization](#) to learn about the underlying hardware implementation. This code can be optimized further by using *all* of the ZA storage. In this example, the FMLA instruction uses VGx4 single vectors groups that are 512B long, of which there are 16. But this example code only uses one such vector (`ZA.S[0]`), whereas the matrix-matrix multiply example uses all four of the S-size ZA tiles that are available. Spreading the FMLA destinations across the ZA vector registers could improve performance by allowing more overlap between the series of dependent FMLA instructions. Options include using multiple vectors each with partial accumulations that are summed at the end, to unrolling and computing a taller section of E (from a wider section of A-transpose).

If the matrix and vector being multiplied are both small, the ASIMD routine may be faster.



Chapter 5. Core Microarchitecture Optimization

Apple silicon general purpose CPUs consist of groups of cores called clusters, where each cluster also contains shared caches and, in some chips, a shared execution engine for accelerated vector and matrix operations. This chapter covers the overview, the cores, and shared cache. [Chapter 6, SME Engine Microarchitecture Optimization](#) covers the shared execution engine. Instruction latencies and bandwidths are covered in [Appendix A, Instruction Latency and Bandwidth](#).

The details covered in this chapter and [Appendix A, Instruction Latency and Bandwidth](#) represent the most important information to consider when optimizing software for the microarchitecture. The CPUs employ additional techniques to improve performance beyond what is described here. These may change across chip families, and it may be difficult or counter-productive to try to transform software to match the heuristics. The guidelines in this chapter represent the optimization opportunities with the most significant impact across P cores and E cores, and across chip series and families.

5.1 Apple Silicon CPU and Chip Overview

Apple silicon M-series and recent A-series chips feature two types of general purpose CPU cores, performance cores (P cores) and efficiency cores (E cores):

- **P cores:** Designed to achieve maximum performance. They are aggressive, out-of-order, superscalar, pipelined microprocessors that feature advanced forms of dynamic branch prediction, register renaming, out-of-order speculative execution, memory dependence prediction, and many other state-of-the-art features.
- **E cores:** Designed to achieve maximum efficiency, thus saving power and increasing battery life while reducing heat generation and fan noise (where applicable). They are based on a similar microarchitecture to the P cores, and still provide compelling high performance. E cores are the most efficient place to run lighter-weight and everyday tasks, allowing the P cores to be used for the most demanding workflows. Use the E cores to meaningfully increase throughput in heavily threaded applications.

Microarchitectural recommendations broadly apply to both core types, but with a focus on the P cores. In some cases, a recommendation may apply to only one CPU core type, but it does not adversely affect performance of the other type.

On Apple silicon, the system decides whether to schedule work on the P cores or E cores based on many factors. The system does not offer any direct controls to software or to end users regarding task placement. However, software can provide hints to the system about where to schedule tasks via the Quality of Service controls. See [Section 7.1, “Prioritizing Work”](#).

Several cores of the same type are grouped together and share a second level cache and are collectively known as a cluster.

In some chips, each cluster also features a shared SME engine for higher throughput 2D tile and long vector computations. Similar to cores, Apple silicon chips feature both a performance version (P SME engine) and an efficiency version (E SME engine). Both are generally the same, except that the E SME engine is smaller and has lower throughput than the P SME engine. Otherwise the recommendations apply to both versions, except where noted. Microarchitectural optimization of SSVE/SME code that executes on the SME engine is covered in [Chapter 6, SME Engine Microarchitecture Optimization](#).

The overall parametrized topology is shown in [Figure 5.1: “General Purpose Compute Complex and Related Components”](#). For instance, the M1 chip consists of two clusters. The first cluster contains 4 P cores that share a second level cache. The second cluster contains 4 E cores that share their own second level cache. Both clusters are connected to each other and to other system components via the fabric. Also connected to the fabric is a high performance memory controller featuring a memory cache. Many other components including GPU and neural engine (not shown) are also connected to the fabric. Other chips contain similar topologies, but the number of cores per cluster and the number of clusters may vary, as shown in [Table 5.1: “Topology Configurations: M-Series Chips”](#). In Ultra chips, the two dies are connected via low-latency high-bandwidth Ultra Fusion interconnect.

Figure 5.1. General Purpose Compute Complex and Related Components

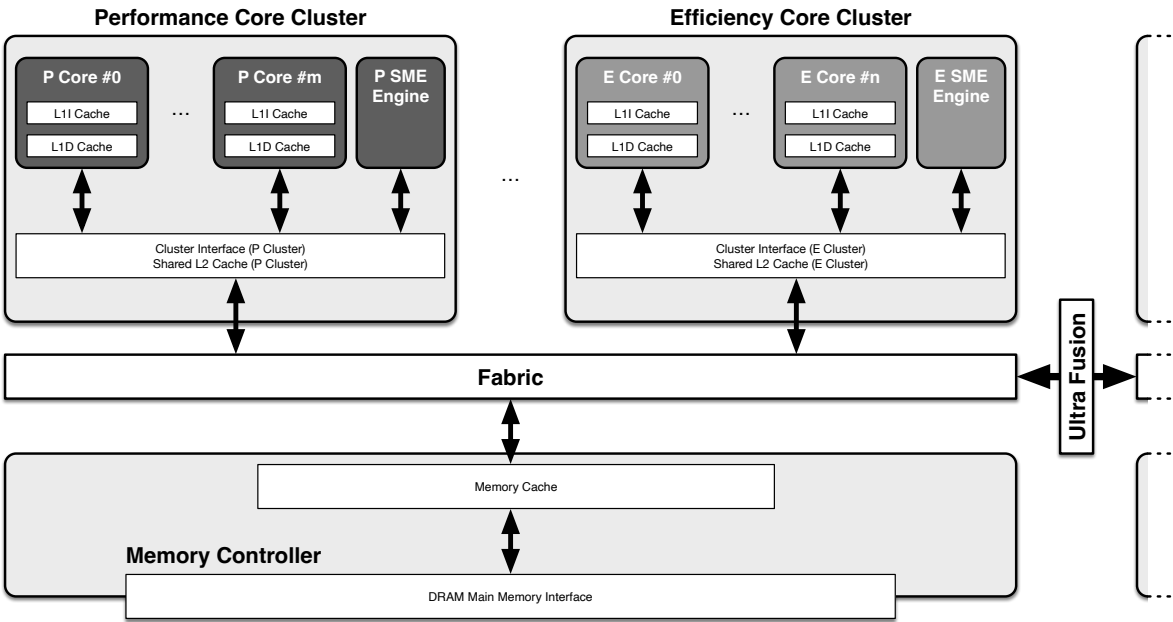


Table 5.1. Topology Configurations: M-Series Chips

Chip	P Cluster			E Cluster			Shorthand
	Cores per Cluster	SME Engine	Instances	Cores per Cluster	SME Engine	Instances	NumDie x (Cluster + ... + Cluster)
M1	4	No	1	4	No	1	1 x (4P + 4E)
M1 Pro	3 or 4	No	2	2	No	1	1 x (3P + 3P + 2E) 1 x (4P + 4P + 2E)
M1 Max	4	No	2	2	No	1	1 x (4P + 4P + 2E)

Table 5.1. Topology Configurations: M-Series Chips (cont.)

Chip	P Cluster			E Cluster			Shorthand
	Cores per Cluster	SME Engine	Instances	Cores per Cluster	SME Engine	Instances	NumDie x (Cluster + ... + Cluster)
M1 Ultra	4	No	4	2	No	2	2 x (4P + 4P + 2E)
M2	4	No	1	4	No	1	1 x (4P + 4E)
M2 Pro	3 or 4	No	2	4	No	1	1 x (3P + 3P + 4E) 1 x (4P + 4P + 4E)
M2 Max	4	No	2	4	No	1	1 x (4P + 4P + 4E)
M2 Ultra	4	No	4	4	No	2	2 x (4P + 4P + 4E)
M3	4	No	1	4	No	1	1 x (4P + 4E)
M3 Pro	5 or 6	No	1	6	No	1	1 x (5P + 6E) 1 x (6P + 6E)
M3 Max	5 or 6	No	2	4	No	1	1 x (5P + 5P + 4E) 1 x (6P + 6P + 4E)
M3 Ultra	6	No	4	4	No	2	2 x (6P + 6P + 4E)
M4	3 or 4	Yes	1	4 or 6	Yes	1	1 x (3P + 6E) 1 x (4P + 4E) 1 x (4P + 6E),
M4 Pro	4 or 5	Yes	2	4	Yes	1	1 x (4P + 4P + 4E) 1 x (5P + 5P + 4E)
M4 Max	5 or 6	Yes	2	4	Yes	1	1 x (5P + 5P + 4E) 1 x (6P + 6P + 4E)

Table 5.2. Topology Configurations: A-Series Chips

Chip	P Cluster			E Cluster			Shorthand
	Cores per Cluster	SME Engine	Instances	Cores per Cluster	SME Engine	Instances	NumDie x (Cluster + ... + Cluster)
A14 Bionic	2	No	1	4	No	1	1 x (2P + 4E)
A15 Bionic	2	No	1	4	No	1	1 x (2P + 4E)
A16 Bionic	2	No	1	4	No	1	1 x (2P + 4E)
A17 Pro	2	No	1	4	No	1	1 x (2P + 4E)
A18 Family	2	Yes	1	4	Yes	1	1 x (2P + 4E)

Note: Core and cluster topology specifications are provided for reference. Software should check the appropriate `sysctl` parameter to dynamically adjust to chip configuration. See [Appendix B, Dynamic Determination of Chip-Specific Capabilities](#) for more information.

5.2 Microarchitectural Characteristics

The cores share fundamental microarchitectural characteristics with other modern processors. The P cores in particular are multi-GHz, pipelined, superscalar, out-of-

order, speculative microprocessors with deep instruction windows. Thus many common strategies used for improving performance are applicable to these CPUs.

However, there are a number of differences to be aware of when optimizing for performance, including:

- **Asymmetric multiprocessing:** Apple silicon chips feature two different classes of general purpose computing cores, one targeting high performance and one targeting efficiency. The ISA is identical between the two core types. However, because threads may run slower when assigned to the E cores, multithreaded software should be constructed with this in mind.
- **Single threaded cores:** Each core executes only one thread at a time, allowing the full resources of the core to be applied to the executing thread. A core's caches and TLBs are not shared with any other simultaneously executing thread.
- **Cache hierarchy:** The cores in each cluster contain their own private L1 instruction and data caches, and share a common L2 shared cache with other cores in the same cluster. The memory system maintains a Memory Cache in the memory controller shared by many agents across the chip. Some algorithms benefit from blocking to the specific L1D and L2 cache sizes. The cores also apply non-temporal memory hints in a way that best fits this particular cache hierarchy, which may require additional tuning.
- **Instruction latency and bandwidth:** The latency and bandwidth characteristics of cores differ from other processors. High performance algorithms developed for other processors may need to be retuned using alternate sequences of instructions to best match the available instruction execution characteristics.
- **Preferred instructions:** The cores optimize performance of specific instructions and operands used for specific tasks. These instructions include those used for moving (copying) registers and zeroing registers (sometimes referred to as idioms).
- **Store-to-load data dependencies:** The cores optimize the performance of store-to-load data flow through data forwarding and prediction algorithms that may require different tuning.
- **Hardware prefetcher:** Common straightforward access patterns are prefetched without issue, however applications may need to be tuned differently in the presence of more unusual patterns.

5.3 Core Processor Pipeline Fundamentals

The P cores are multi-GHz, pipelined, out-of-order, speculative, superscalar microprocessors with deep instruction windows.

- **Multi-GHz, pipelined:** Processing of an individual instruction requires, at a minimum, several nanoseconds to complete. This includes fetching the instruction from memory, decoding the operation, fetching any required input data, and performing the operations. To improve throughput, the processor breaks the full lifetime into steps, and uses specialized hardware to perform each step.

When an instruction completes a step, it moves to the next piece of specialized hardware to complete the next step. Simultaneously, the next instruction executing in a prior pipeline step can now move into the vacated specialized hardware. The time allotted for completion of each step is a clock cycle. The processor can vary the

frequency, in clock cycles per second, from 100s of MHz to a few GHz, depending on need. Faster clock cycles often result in higher performance, but also consume more power.

- **Out-of-order execution with deep instruction window:** Short sequences of instructions often consist of chains of instructions that the processor must execute serially. For example, to increment a value in memory: a load instruction reads a value from memory and feeds that into an add instruction, the result of which is written to memory via a store instruction. These serial sequences prevent processors from utilizing all of the hardware simultaneously: the load must complete, then the add, and then the store. For this sequence, when the processor is executing the add, there are no other instructions to keep the load and store hardware busy.

Therefore, the processor keeps a large buffer of upcoming instructions. Instructions are read from memory and inserted into the buffer in program order to ensure inter-instruction dependences are properly understood. From that buffer, the processor attempts to find independent instructions (those whose inputs are available) that it can execute out of order and simultaneously with other instructions. The instruction window is how far ahead the processor looks for independent work. The P cores are capable of looking ahead up to several hundred instructions. Age-based terminology (*older* and *younger*) is often used with reference to the instruction window, where younger instructions are further along in the instruction stream compared to older instructions.

The out-of-order mechanism stashes results of completed work and only commits those results to registers and memory when they become the oldest in the instruction window. This is known as *in-order* retirement. The retirement process ensures that instructions appear to have executed in the proper sequential order, the order in which a simple single instruction processor would execute them. Put another way, if execution stops after a particular instruction, all instructions older than the stopping point have completed execution and have had their results reflected in register and memory state, while no younger results have affected register and memory state.

- **Speculative:** In order to keep the instruction window as full as possible, the processor routinely makes predictions regarding control flow branches. The processor fetches instructions following the predicted paths, and feeds those instructions into the instruction window. When the branches actually execute, the predicted direction or target is compared against the result. Occasionally the actual result of the branch does not match the prediction, and a misprediction occurs. Thus, the instructions in the instruction window that are younger than the branch are not the proper instructions. At this point, the processor flushes the younger instructions from the instruction window, and new instructions are fetched from memory at the target of the branch.

Similarly, the processor may speculate past memory dependences. That is, the processor may guess which, if any, older stores in the instruction window may write values to memory that need to be read by younger loads in the window. In cases where it guesses wrong and misses a dependence, the load read incorrect data and potentially fed that to subsequent younger instructions. In such cases, the processor must flush those instructions and re-execute them.

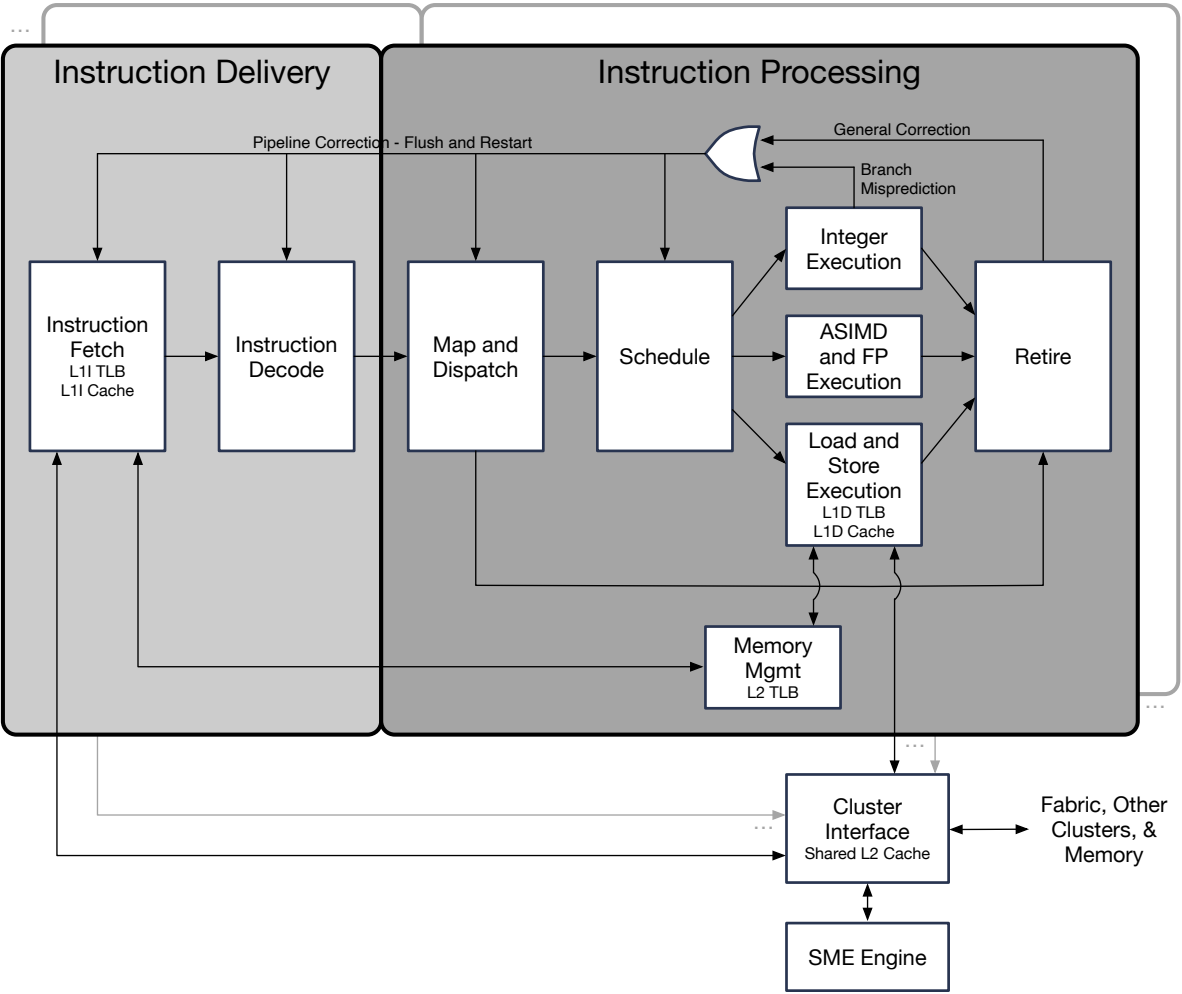
- **Superscalar:** To further increase throughput, the processor employs duplicated, or parallel, specialized hardware components to operate on more than one instruction per cycle. For example, the M1 P cores are capable of decoding 8 instructions each

clock cycle, and performing 6 integer add operations per cycle (plus SIMD and memory operations as well).

At the highest level, the processor consists of two main components, Instruction Delivery and Instruction Processing. Instruction Delivery fetches and decodes instructions while Instruction Processing manages the instruction window and executes the instructions. Instruction Delivery predicts branches and fetches instructions as fast as possible to keep the instruction window in Instruction Processing as full as possible. Instruction Processing searches through all of the instructions in the window identifying and executing independent work, while retiring the oldest instructions. The Cluster Interface Unit is outside of the core boundary and serves both Instruction Delivery and Instruction Processing.

Instruction Delivery and Instruction Processing each consist of several units and connect to the Cluster Interface Unit.

Figure 5.2. Abstract View of the Core Processor Pipeline.



Conceptually, the processor operates on ISA instructions, as previously described. More technically, however, the processor decodes (translates) ISA instruction bytes into executable units called microoperations (μ ops). These μ ops are highly tailored to the

microarchitecture. Most instructions are translated into a single μop , but some complex instructions require multiple μops . In many cases, the distinction is not particularly important, but the term *instruction* is used when specifying a particular instruction at the source code level and μop when referring to specific hardware.

Instruction Delivery consists of the following units:

- **Instruction Fetch Unit (Fetch)** reads instruction bytes from the memory system based on various prediction mechanisms. Contains the Instruction Translation Lookaside Buffer (L1 TLB), which supports [virtual memory](#) by obtaining physical addresses from virtual addresses. Contains the L1 Instruction Cache to improve performance of reads of instruction bytes from memory.
- **Instruction Decode Unit (Decode)** translates instructions bytes into executable microoperations (μops).

Instruction Processing consists of the following units:

- **Map and Dispatch Unit (Map)** inserts new μops into the instruction window. It obtains the necessary resources from the Execution and Retirement portions of the processor to implement out-of-order execution of the μops .
- **Schedule Unit (Schedule)** selects ready μops and issues them to the appropriate execution units.
- **Integer Execution Units (Integer Execution)** sometimes referred as the General Purpose Execution Units. Performs the desired operation on data associated with the General Purpose Registers.
- **Advanced SIMD and FP Execution Units (ASIMD and FP Execution)** perform the desired operation on data associated with the Advanced SIMD and FP (Vector) Registers.
- **Load and Store Execution Units (Load and Store Execution)** read data from and writes data to the memory system. Contains the Data Translation Lookaside Buffer (L1D TLB) to obtain physical addresses from virtual addresses to support virtual memory. Contains the L1 Data Cache to improve performance of reads and writes data bytes from and to memory.
- **Memory Management Unit (MMU)** performs page table walks and caches the virtual to physical address translations in the L2 TLB.
- **Retirement Unit (Retire)** collects completed μops and commits the oldest results to registers and memory to ensure the appearance of sequential execution.

The Cluster Interface Unit consists of the following units:

- **Shared L2 Cache** caches bytes from the memory system to improve performance of Instruction Fetch and Load and Store Execution Units. It is shared with other cores in the cluster.
- **Fabric Interface** moves cache lines to and from the fabric. The fabric transports cache lines between other clusters, memory, and other devices and chip resources.

This guide uses the following additional microarchitectural definitions:

- **Dispatch** sends μops from the Map and Dispatch Unit to the Schedule Unit.
- **Issue** sends μops from the Schedule Unit to an Execution Unit.

- **Slots** are parallel lanes in which μ ops travel, usually in in-order portions of the processor.
- **Demand accesses** are required accesses to a cache structure (including TLB) according to the predicted flow of instructions, as opposed to prefetch accesses that are an educated guess of future need.
- **Unique misses** are the initial misses in a caching structure (including TLB) that initiate fills. Subsequent misses while a fill is in progress are not unique. There may be many unique misses for a particular cache line over the course of execution because the line may be evicted from the cache and need to be re-filled later.
- **Datapath bypassing (forwarding) paths** are data buses that feed the result of an operation from the output of an execution unit into the input of another execution unit to eliminate the latency of a write and subsequent read of the register file.

5.4 Instruction Delivery Optimization

The following sections describe optimization opportunities for identifying and improving Fetch and Decode Unit bottlenecks in Instruction Delivery.

[Bottleneck Analysis](#) in the Instruments tool offers empirical measurements about the relative impact of Instruction Delivery and its components, as well as preconfigured sets of performance monitoring events for deep analysis. The analysis also provides empirical measurements about the relative impact of the effects of branch mispredictions.

5.4.1 L1 Instruction (L1I) TLB

The Instruction Translation Lookaside Buffer (TLB) caches virtual-to-physical address mappings. Application pages are 16KiB.

The L1I TLB is organized as follows. The L1D TLB and the L1I TLB are backed by a much larger Shared L2 TLB as well as a Page Table Walker. See discussion in [Section 5.6.2, “L1 Data \(L1D\) TLB and Shared L2 TLB”](#).

Table 5.3. L1I 16KiB-Page TLB Organization: M-Series Chips

Chip	Entries (Coverage)	
	Performance Cores	Efficiency Cores
M1 Family	192 entries (3MiB)	128 entries (2MiB)
M2 Family	192 entries (3MiB)	192 entries (3MiB)
M3 Family	192 entries (3MiB)	192 entries (3MiB)
M4 Family	192 entries (3MiB)	192 entries (3MiB)

Table 5.4. L1I 16KiB-Page TLB Organization: A-Series Chips

Chip	Entries (Coverage)	
	Performance Cores	Efficiency Cores
A14 Bionic	192 entries (3MiB)	128 entries (2MiB)
A15 Bionic	192 entries (3MiB)	192 entries (3MiB)
A16 Bionic	192 entries (3MiB)	192 entries (3MiB)

Table 5.4. L1I 16KiB-Page TLB Organization: A-Series Chips (cont.)

Chip	Entries (Coverage)	
	Performance Cores	Efficiency Cores
A17 Pro	192 entries (3MiB)	192 entries (3MiB)
A18 Family	192 entries (3MiB)	192 entries (3MiB)

Many applications and libraries contain a moderately sized set of commonly used functions and a larger set of less commonly used functions. To minimize the number of TLB entries required, and thus the number of TLB misses encountered, place the commonly used functions into a small set of pages. Alternatively, restructure the algorithm to improve the temporal locality of function calls, where possible. That is, organize work to heavily use a function in a phase of execution, and then move on to another phase with different functions, rather than interleaving many functions.

Fetches that miss the L1I TLB are delayed by several cycles or more. These delays can result in no μ ops being delivered to Map until the translation is available.

Table 5.5. Common L1 Instruction TLB Metrics

Name and Formula (Event Definitions: Section 8.1, "CPU Counters With Performance Monitoring Events")	Description
L1I TLB Miss Density[†]: <Ev L1I_TLB_MISS_DEMAND> / <Ev INST_ALL>	Frequency of speculative L1 instruction TLB misses due to demand fetches compared to the count of all retired instructions
L1I TLB Fill Density[†]: <Ev L1I_TLB_Fill> / <Ev INST_ALL>	Frequency of speculative L1 instruction TLB fills for any reason compared to the count of all retired instructions

[†]Additional TLB and Address Translation Metrics are available in [Table 5.23: "Common L1D TLB, Shared L2 TLB, and Address Translation Metrics"](#).

Recommendation: Collect commonly used (hot) functions into a small set of pages or improve temporal access locality to reduce virtual address translation overhead.

[Magnitude: Medium | Applicability: Medium] Separate functions into commonly used (hot) and uncommonly used (cold). Place the commonly used functions into a small set of pages. Or, where possible structure algorithms to heavily use a small set of functions and then move on to another set.

5.4.2 L1 Instruction (L1I) Cache

Each core uses its own private L1 Instruction Cache. The L1I Cache is backed by the Shared L2 Cache that is shared between instruction and data. See [Section 5.6.4, "Shared L2 Cache"](#).

Table 5.6. L1I Cache Organization: M-Series Chips

Chip	Capacity, Associativity, and Line Size	
	Performance Cores	Efficiency Cores
M1 Family	192KiB, 6-way, 64B lines	128KiB, 8-way, 64B lines
M2 Family	192KiB, 6-way, 64B lines	128KiB, 4-way, 64B lines (reduced associativity)
M3 Family	192KiB, 6-way, 64B lines	128KiB, 4-way, 64B lines
M4 Family	192KiB, 6-way, 64B lines	128KiB, 4-way, 64B lines

Table 5.7. L1I Cache Organization: A-Series Chips

Chip	Capacity, Associativity, and Line Size	
	Performance Cores	Efficiency Cores
A14 Bionic	192KiB, 6-way, 64B lines	128KiB, 8-way, 64B lines
A15 Bionic	192KiB, 6-way, 64B lines	128KiB, 4-way, 64B lines (reduced associativity)
A16 Bionic	192KiB, 6-way, 64B lines	128KiB, 4-way, 64B lines
A17 Pro	192KiB, 6-way, 64B lines	128KiB, 4-way, 64B lines
A18 Family	192KiB, 6-way, 64B lines	128KiB, 4-way, 64B lines

Note: Capacity and associativity specifications are provided for reference. Software should check the appropriate `sysctl` parameter to dynamically adjust to CPU and chip configurations. See [Appendix B, Dynamic Determination of Chip-Specific Capabilities](#) for more information.

Fetches that miss the L1 Instruction Cache are delayed by several cycles or more. These delays can result in no μops being delivered to Map until the instruction bytes are available.

Often, only portions of functions are actually commonly used. For example, functions often have clauses to handle error conditions or unlikely cases. Because these clauses are rare or only occur when the application terminates, there is little benefit to holding them in the instruction cache instead of other commonly executed code. Collect the commonly executed portions of the function together to reduce the number of cache lines that occupy space in the cache for common execution patterns. Collect uncommonly executed portions and place them together, typically at the end.

Table 5.8. Common L1 Instruction Cache Metrics

Name and Formula (Event Definitions: Section 8.1, "CPU Counters With Performance Monitoring Events")	Description
L1I Cache Miss Density: <code><Ev L1I_CACHE_MISS_DEMAND> / <Ev INST_ALL></code>	Frequency of speculative L1 instruction demand cache misses compared to the count of all retired instructions

Recommendation: Collect commonly used (hot) portions of functions into a small set of cache lines.

[Magnitude: Medium | Applicability: Medium] Separate portions of functions into commonly used (hot) and uncommonly used (cold). Place the commonly used portions of functions into a small set of cache lines to reduce the L1I Cache occupancy. That is, reduce the space in the L1I Cache required for this function for typical execution. More specifically, collect error conditions or uncommon then/else statements and place them together at the end. These cache lines only occupy space in the instruction cache in rare circumstances.

Recommendation: Avoid L1I Cache set conflicts.

[Magnitude: Low | Applicability: Low] Although unlikely to be a significant problem, L1I Cache misses may result from set conflicts. Apple silicon uses 16KB memory pages that may make this even more likely to occur, since independent libraries tend to be aligned to page boundaries when they are loaded.

5.4.3 Branch Target Alignment

The Instruction Fetch Unit features flexible alignment of reads from the L1 instruction cache coupled with aggressive alignment of instruction bytes into the Instruction Decode Unit. In other words, the processor efficiently fills Decode no matter the position within the instruction memory.

Some microarchitectures benefit from alignment of branch targets to the beginning of an instruction cache line. Thus, when the processor fetches a cache line at the target of a branch, the Instruction Fetch Unit provides a full line of bytes to Decode. To accomplish this, software typically contains padding `nop` instructions prior to the branch target to shift the target to the beginning of a cache line, while allowing fall through execution to the target to execute properly.

Because of the alignment capabilities, software alignment of branch targets is generally unnecessary and sometimes detrimental. Rather, software should retain unaligned branch targets in favor of reduced code size.

Recommendation: Refrain from aligning branch targets through added unnecessary instructions.

[Magnitude: Low | Applicability: Medium] Avoid alignment of branch targets to cache line boundaries through the addition of extraneous `nop` instructions. These alignment instructions do not typically improve performance and adversely affect code size and instruction cache performance.

5.4.4 Taken Branch Reduction

Taken branches cause an interruption in the fetch sequence. As described in [Section 5.4.3, "Branch Target Alignment"](#), the Instruction Fetch Unit reads groups of instructions even when they span a cache line boundary. However, the Fetch Unit must discard instructions that are fetched with the group that appear after a predicted taken branch. (For an in-depth discussion of branch terminology, see [Section 1.4, "Branch Terminology"](#).) When the Fetch Unit frequently delivers only partial groups of instructions

to Decode, the processor may suffer from a bandwidth bottleneck. This bottleneck is most often encountered when executing short loop bodies with high iteration counts.

Some common taken branch reduction optimizations include:

- **Function inlining:** Eliminates both a call-type taken branch and a return-type taken branch (and often allows for optimization across the caller and callee).
- **If-conversion:** Eliminates branches by converting them into conditional instructions, such as conditional move. (See [Section 5.4.5, “Conditional Branch Mispredicts and Conditional Instructions”](#))
- **Hot path straightlining (code layout optimization):** Reduces taken branches by placing the commonly executed target instructions along the fall-through path from the branch. When the taken path is swapped with the fall-through path, the branch condition must be inverted. It is especially important to ensure that error-handling code is reached via a taken branch so that normal operation falls through.
 - In C++20, use the `[[likely]]` and `[[unlikely]]` attributes to empower the compiler to optimize for the common path. See [likely and unlikely attributes](#) in the LLVM reference manual for more information.
 - In C and older versions of C++, use `__builtin_expect()` or `__builtin_expect_with_probability()` to empower the compiler. See [__builtin_expect](#) discussion in the LLVM reference manual for more information.

```
#define likely(x)      __builtin_expect(!!(x),1)
#define unlikely(x)    __builtin_expect(!!(x),0)

int idiv(int a, int b)
{
    if (unlikely(b == 0)) err(DIV_BY_ZERO);
    return a/b;
}
```

- **Loop unrolling:** Reduces taken loop-back branches by placing additional copies of the loop body along the fall-through path, thus converting taken loop-back branches to fall-through branches.
 - For *counted* loops (iteration count is known when the loop begins execution), unrolling is particularly useful because the iteration count comparison and backward branch can be eliminated for all but the last unrolled iteration. This reduces code size and uses fewer execution resources. Should the unrolled amount not be evenly divisible into the total iteration count (ex., loop unrolled to 4 copies for 30 iterations), a remainder loop can be included to iterate over the remaining iterations (ex., 7 iterations through the main loop body to execute 28 original iterations, and 2 iterations through the remainder loop).

Table 5.9. Common Branch Instruction Mix and Direction Metrics

Name and Formula (Event Definitions: Section 8.1, "CPU Counters With Performance Monitoring Events")	Description
Branch Density: <Ev INST_BRANCH> / <Ev INST_ALL>	Proportion retired branch instructions (including calls and returns) of all retired instructions
Call Density: <Ev INST_BRANCH_CALL> / <Ev INST_ALL>	Proportion retired call (direct and indirect) instructions of all retired instructions
Return Density: <Ev INST_BRANCH_RET> / <Ev INST_ALL>	Proportion retired return instructions of all retired instructions
Indirect Branch Density: <Ev INST_BRANCH_INDIR> / <Ev INST_ALL>	Proportion retired indirect branch (including call) instructions of all retired instructions
Taken Branch Density (of instructions): <Ev INST_BRANCH_TAKEN> / <Ev INST_ALL>	Proportion retired taken branch instructions of all retired instructions
Taken Branch Density (of branches): <Ev INST_BRANCH_TAKEN> / <Ev INST_BRANCH>	Proportion retired taken branch instructions of all retired branches

Recommendation: Reduce taken branch density to improve instruction fetch bandwidth.

[Magnitude: Medium | Applicability: High] When taken branch densities approach 1 in 8 instructions, unroll loops, optimize conditionals to commonly fall through (increase straight line code), and apply other general branch reduction techniques.

5.4.5 Conditional Branch Mispredicts and Conditional Instructions

The Arm instruction set provides both conditional branches and a limited set of conditional (predicated) instructions. The conditional instructions include three basic types:

- Conditional moves:** For example, `CSEL` that writes the destination with one of two input values based on the condition.
- Conditional sets:** For example, `CSET` that sets the destination to one of two fixed values based on the condition.
- Conditional logic or arithmetic operations:** For example, `CINC` that writes the destination with either the input value or the input value plus one according to the condition.

When writing a code sequence, a developer has a choice between a conditional branch and a conditional sequence. Conversion of a control dependency (branch) into a data dependency (conditional instruction) is referred to as "if-conversion." Deciding when to use a conditional instruction sequence depends on the expected execution cost.

When the processor correctly predicts conditional branches, the application executes with optimal latency. Only the required instructions are executed and the out-of-order instruction window remains full. More generally, the cost of an instruction sequence including a branch is determined by four factors:

1. The probability that one path is taken rather than another
2. The execution cost of each path
3. The probability of a branch misprediction
4. The cost of a branch misprediction

Note that the overall cost of a branch misprediction includes several components:

- The cost of instructions executed on the wrong path
- The cost to reset the Instruction Fetch Unit and Map and Dispatch Unit to the correct target
- The cost of lost opportunity to speculatively execute younger instructions along the correct path

These parameters may or may not be known at compile time, so heuristics may be required.

The cost of a conditional instruction sequence is relatively simple to calculate, since the outcome of the condition does not change the cost of the instruction sequence. The overall latency includes that of a conditional instruction sequence whether the result of the sequence is used or discarded. For an if-then-else construct, the overall latency is the maximum of both paths, the correct and wrong paths.

General guidelines for selecting branches versus conditional instructions:

- **Use branches when the condition is highly predictable.** The cost of mispredicts is low, and the code executes with optimal latency.
 - Strongly biased branches are typically highly predictable. For further optimization, place the commonly executed code at the fall through path of the branch. See [Section 5.4.4, “Taken Branch Reduction”](#) for more information.
- **Use branches when the cost of executing the wrong path is higher than the cost of mispredicts** — that is, high path costs or low mispredict probability. High path costs are a particular problem with conditional blocks that include the following types of instructions:
 - Long chains of dependent operations, particularly chains with multicycle latencies such as FP calculations
 - Complex, long-latency instructions such as divide, load with de-interleave, and table lookup
 - Loads that are likely to stall due to cache or TLB misses (use performance monitoring to identify these memory operations)
- **Use conditional instructions when the cost of mispredicts is higher than the cost of executing the “wrong” path**, i.e. low path costs and very unpredictable branch outcomes. For example, use conditionals if the cost of executing both paths simultaneously is similar to executing just the correct path. Likewise, use conditionals when the mispredict rate is high. The extra cost of always executing the wrong path along with the correct path may be less than the cost of the mispredictions.

Broadly, the instruction set includes a number of instructions that conditionally modify data. Depending on the algorithm, use these instructions instead of branching to or around code that unconditionally performs similar functions.

- **CINC:** Conditional Increment returns, in the destination register, the value of the source register incremented by 1 if the condition is true, and otherwise returns the value of the source register.
- **CINV:** Conditional Invert returns, in the destination register, the bitwise inversion of the value of the source register if the condition is true, and otherwise returns the value of the source register.
- **CNEG:** Conditional Negate returns, in the destination register, the negated value of the source register if the condition is true, and otherwise returns the value of the source register.
- **CSEL:** If the condition is true, Conditional Select writes the value of the first source register to the destination register. If the condition is false, it writes the value of the second source register to the destination register.
- **CSET:** Conditional Set sets the destination register to 1 if the condition is true, and otherwise sets it to 0.
- **CSETM:** Conditional Set Mask sets all bits of the destination register to 1 if the condition is true, and otherwise sets all bits to 0.
- **CSINC:** Conditional Select Increment returns, in the destination register, the value of the first source register if the condition is true, and otherwise returns the value of the second source register incremented by 1.
- **CSINV:** Conditional Select Invert returns, in the destination register, the value of the first source register if the condition is true, and otherwise returns the bitwise inversion value of the second source register.
- **CSNEG:** Conditional Select Negation returns, in the destination register, the value of the first source register if the condition is true, and otherwise returns the negated value of the second source register.

In this example, the code uses the `CNEG` instruction to compute the absolute value. The added cost of executing the conditional negation, regardless of the condition, is small compared to potential effects of mispredicting a branch based on the comparison.

```
int my_abs(int y)
{
    if (y < 0) return -y;
    return y;
}

my_abs(int):
    CMP     W0, #0
    CNEG    W0, W0, MI
    RET
```

In this second example, the code uses a `FCSEL` to select one input versus the other. (Note that `FMAX` instruction handles Not-a-Number (NaN) values differently than the C source code specifies.)

```
float my_fmax(float a, float b)
{
    if (a > b) return a;
```

```

        return b;
}

my_fmax(float, float):
    FCMP     S0, S1
    FCSEL    S0, S0, S1, GT
    RET

```

In this third example, the compiler would not perform an if-conversion on function `light_att()` because the memory store would not occur on the false path. In function `light_att2()`, however, the compiler performed the transformation after the store was added to both paths. Note, though, that `light_att2()` may incur a memory access fault on the memory address in `x0` when `(d < 0)` where it would not have in `light_att()`. Similarly, that `light_att2()` may incur a cache miss on the memory address in `x0` when `(d < 0)` where it would not have in `light_att()`. When altering the source code in this manner, understand how `res` is set and used in relation to `d` in the calling functions.

```

void light_att(float &res, float d, float base)
{
    if (d < 0)
        res = base*d;
}

light_att(float&, float, float):
    FCMPL    S0, #0.0
    B.PL     .LBB3_2
    FMUL     S0, S0, S1
    STR      S0, [X0]
.LBB3_2:
    RET

void light_att2(float &res, float d, float base)
{
    res = ((d < 0) ? base*d : res);
}

light_att2(float&, float, float):
    LDR      S2, [X0]
    FMUL     S1, S0, S1
    FCMPL    S0, #0.0
    FCSEL    S0, S1, S2, MI
    STR      S0, [X0]
    RET

```

In this fourth example, `cSET` can be used to set Boolean values based on conditions.

```
bool in_range(int y)
{
    bool result = false;
    if (y < 10 && y > 4)
        result = true;
    return result;
}

in_range(int):
```

```
SUB      W8, W0, #5
CMP      W8, #5
CSET     W0, LO
RET
```

Use the following metrics to identify general and conditional branch misprediction rates.

Table 5.10. General and Conditional Branch Mispredict Metrics

Name and Formula (Event Definitions: Section 8.1, "CPU Counters With Performance Monitoring Events")	Description
Mispredicted Branch Density (of instructions): <Ev BRANCH_MISPRED_NONSPEC> / <Ev INST_ALL>	Proportion retired mispredicted branches of all retired instructions
Mispredicted Branch Density (of branches): <Ev BRANCH_MISPRED_NONSPEC> / <Ev INST_BRANCH>	Proportion retired mispredicted branches of all retired branches
Mispredicted Conditional Branch Density (of instructions): <Ev BRANCH_COND_MISPRED_NONSPEC> / <Ev INST_ALL>	Proportion retired mispredicted conditional branches of all retired instructions

Recommendation: Use conditional instructions to reduce conditional branch mispredictions.

[Magnitude: High | Applicability: Medium] When a conditional branch is difficult to predict and the added cost of executing the instructions on the wrong path is minimal, execute instructions from both paths and use conditional data movement to select the correct outcome.

5.4.6 Indirect Branch and Indirect Call Mispredicts

The processor also predicts indirect branch and indirect call targets. Returns are also indirects but have a dedicated prediction mechanism optimized for their expected pattern (see [Section 5.4.7, "Return Address Stack"](#) for more details). Use the following metrics to identify indirect branch and indirect call misprediction rates:

Table 5.11. Indirect Branch and Indirect Call Mispredict Metrics

Name and Formula (Event Definitions: Section 8.1, "CPU Counters With Performance Monitoring Events")	Description
Mispredicted All Indirect Branch Density (of instructions): <Ev BRANCH_INDIR_MISPRED_NONSPEC> / <Ev INST_ALL>	Proportion retired mispredicted indirect branches including calls and returns of all retired instructions
Mispredicted Indirect Branch Density (of instructions): (<Ev BRANCH_INDIR_MISPRED_NONSPEC> - <Ev BRANCH_RET_INDIR_MISPRED_NONSPEC> - <Ev BRANCH_CALL_INDIR_MISPRED_NONSPEC>) / <Ev INST_ALL>	Proportion retired mispredicted indirect branches (not including calls and returns) of all retired instructions

Table 5.11. Indirect Branch and Indirect Call Mispredict Metrics (cont.)

Name and Formula (Event Definitions: Section 8.1, "CPU Counters With Performance Monitoring Events")	Description
Mispredicted Indirect Call Density (of instructions): $\frac{\text{Ev BRANCH_CALL_INDIR_MISPRED_NONSPEC}}{\text{Ev INST_ALL}}$	Proportion retired mispredicted indirect function calls of all retired instructions

5.4.7 Return Address Stack

When a function call is decoded, the processor pushes the address of the next instruction (the return address) onto the Return Address Stack. This allows a single function to be called from multiple call points while predicting correct return addresses. Like other indirect branch predictions, the return target is verified during execution, and may result in a misprediction (pipeline flush). Most well-formed code should not need modification to optimize for Return Address Stack behavior. However, follow these guidelines:

- **Use recognized call and return instructions.** Use the expected instructions so that the processor properly manages the Return Address Stack. Avoid custom code sequences, such as regular branches, to implement returns.
 - For calls: `BL` or `BLR`. Branch with Link branches to a PC-relative offset or to an address in a register, setting the register `x30` to `PC+4`. It provides a hint that this is a subroutine call.
 - For returns: `RET`, which is a specific flavor branch through register. It provides a hint that this is a subroutine return.
- **Use properly paired calls and returns.** Avoid using a single return instruction to simultaneously return through multiple stack levels. This restriction mostly affects unusual code such as the C language `longjmp` standard library routine, which can corrupt the RAS if it jumps back to the location of the last `setjmp` call without simulating any returns that would have otherwise occurred. Counterintuitively, a "fast" single `RET` that skips over several intervening returns may end up being slower overall than a "slow" return sequence that performs all of the `RET` instructions. By skipping returns, the Return Address Stack may become misaligned with actual execution, resulting in many mispredictions afterwards.
- **Restructure algorithms to avoid deep call stacks.** Inline short functions where possible to reduce call stack depth, especially those that primarily just call another function. Use recursion sparingly, if at all. Inlining also integrates the function body into the caller, eliminating any computational redundancy and data movement overhead, often resulting in further performance improvement.

Use the following metrics to measure return mispredict rates. Occasional return mispredictions occur even if all of the above guidelines are followed. For instance, the processor and operating system do not save and restore the contents of the RAS during process switches. Therefore returns executed immediately after the process is switched back into the processor may encounter an empty RAS.

Table 5.12. Return Mispredict Metrics

Name and Formula (Event Definitions: Section 8.1, "CPU Counters With Performance Monitoring Events")	Description
Mispredicted Return (of instructions): <Ev BRANCH_RET_INDIR_MISPRED_NONSPEC> / <Ev INST_ALL>	Proportion retired mispredicted function returns of all retired instructions
Mispredicted Return (of returns): <Ev BRANCH_RET_INDIR_MISPRED_NONSPEC> / <Ev INST_BRANCH_RET>	Proportion retired mispredicted function returns of all retired returns

Recommendation: Leverage the return address stack by using well formed call-return sequences and limiting call depth.

[Magnitude: Medium | Applicability: Low] When calling and returning from functions, use properly paired recognized call and return instructions. Examine the call stacks for any regions of execution that commonly exhibit high return mispredicts. Correct any improperly paired or non-standard calls and returns. If significant mispredicts remain, limit call stack depth through algorithmic changes and function inlining.

5.4.8 Multi- μ op Instructions

Some complex instructions require multiple μ ops to implement the instruction's functionality. These include instructions that read more than three source registers, write multiple destination registers, and use functionality from multiple pipelines. However, use of these instructions improves computational density by packing a lot of functionality into a single 32b instruction.

Cracked instructions require sequences of 2 or 3 μ ops. The processor inserts the μ op sequence seamlessly between μ ops from the prior and subsequent instructions.

Microcoded instructions require sequences of 4 or more μ ops. In addition, the processor may not be able to seamlessly integrate these μ ops into the μ op stream and thus incur further overhead.

As a general guideline, when using the full capability of these instructions, the added microoperations are more than worth their cost, compared with implementing the same functionality with a collection of available individual instructions. However, if the full functionality is not needed, consider alternatives.

Multi- μ op instructions include:

- **Load and store operations with address writeback:** These common cracked instructions require an extra μ op to perform the address update. Avoid chains of these instructions because of the added μ op bandwidth consumed as well as the added dependencies. Use one adjustment per block of loads or stores. (See [Section 2.8.1, "Avoid Chains of Pre- and Post-Indexed Operations"](#)). Formats:

```
LDx/STx Rt, [Xn, #imm]!           // Pre-index addressing mode
LDx/STx Rt, [Xn], #imm            // Post-index addressing mode
```

- **Paired load operations:** These common cracked instructions have two destination registers and are cracked before renaming. However, unless the operands are Q-size, the processor re-fuses them back into a single μ op before sending them to the Load and Store Execution Units. Use these instructions wherever possible. Example instructions:

```
LDP{SW} Rt1, Rt2, ...
LD{N}P Dt1, Dt2, ...
```

- **Paired Advanced SIMD and FP store operations:** These common cracked instructions require an additional μ op. Use them wherever possible to improve code density. Example instructions:

```
ST{N}P {SDQ}t1, {SDQ}t2, ...
```

- **Advanced SIMD and FP load and store operations that use an `lsl #4` or `<extend> #4`:** These less common cracked instructions require an additional μ op to compute the address. Use them wherever possible to improve code density. Example instructions:

```
STR Qt, [Xn, Xm, lsl #4]
```

- **Instructions that move data between integer and Advanced SIMD and FP and include another operation:** These less common cracked instructions require two operations such as conversion along with data movement. Use them when necessary to improve code density. Example instructions:

```
FCVTAS Rd, Dm
SCVTF Dd, Rm
DUP Vd.2D, Rm
INS Vd.2D[x], Rm
```

- **Integer execution unit multiply-accumulate operations:** Beginning on the M4 and A18 families of chips, the P cores crack these instructions into a MUL-class operation and an ALU-class operation. See [Section 3.5.4, "Fused Op with Add/Accumulate Instructions \(Integer and ASIMD and FP Types\)"](#) and [Section A.1, "Core Integer Execution Units"](#).
- **Atomics (Load-and-operation (LDop), Store-and-operation (STop), Compare-and-Swap (CAS), Swap):** These instructions perform atomic operations and replace more complex sequences of instructions with barriers. If the full instruction behavior matches the required functionality, in particular for synchronization operations, use these instructions. These instructions are cracked with the exception of CAS. Example instructions:

```
LDSMAX Rs, Rt, [Xn]
STCLRA Rs, [Xn]
CAS Rs, Rt, [Xn]
```

- **Table vector lookup:** These instructions may be single μ op, cracked, or microcoded depending on the size of the table. Use the smallest sized table that meets the needs of the algorithm. Example instructions:

```
TBL Vd, {Vn, Vn+1, Vn+2}, Vm
TBX Vd, {Vn, Vn+1, Vn+2}, Vm
```

- **Vector element loads and stores:** These instructions offer powerful element interleaving and de-interleaving that is not easily achievable with combinations of other instructions. Most of these instructions are microcoded. Use them when necessary, but consider reorganizing the data structure when possible to eliminate frequent interleaving and de-interleaving. Example instructions:

```
LD3 { Vn.<T>, Vn+1.<T>, Vn+2.<T> }, [Xn]
```

- **Brain floating point (bf16) mathematical operations:** These instructions (BFDOT, BFMMMLA) require a sequence of 3-8 general vector μ ops to complete.

Recommendation: Leverage cracked and microcoded instructions when the full instruction functionality is needed.

[Magnitude: Low | Applicability: Medium] Although cracked and microcoded instructions require multiple μ ops to implement, they typically offer complex operations and improve code density. Selectively use microcoded instructions due to their higher overhead, using simpler versions or recoding the algorithm when possible.

5.5 Instruction Processing Map and Execution Optimization

The following sections describe optimization opportunities for identifying and improving Map and Execution issues in Instruction Processing.

[Bottleneck Analysis](#) in the Instruments tool offers empirical measurements about the relative impact of Instruction Processing and its components, as well as preconfigured sets of performance monitoring events for deep analysis.

Sustained μ op bandwidth (assuming a broad mix of μ op types) is limited by Map bandwidth, in other words, the number of μ ops that can be inserted into the instruction window per cycle. Each Map slot is capable of handling any type of μ op.

Burst μ op bandwidth is achieved when all of the functional units execute a μ op at the same time. This is possible due to buffering of instructions in the instruction window. Note that store μ op address and data parts are counted separately (see [Section A.3.4, “Load and Store Execution Unit Bandwidth”](#) for more details). However, when averaged over time, execution cannot exceed the sustained μ op bandwidth dictated by Map bandwidth.

Table 5.13. μ op Execution Bandwidth: M-Series Chips

Chip	μ ops Per Cycle									
	Performance Cores					Efficiency Cores				
	Sus-tained	Burst				Sus-tained	Burst			
		Total	Int	AS&FP	Mem		Total	Int	AS&FP	Mem
M1 Family	8	17	6	4	3LD, 2STA, 2STD	4	9	3	2	2LD/STA, 2STD

Table 5.13. μ op Execution Bandwidth: M-Series Chips (cont.)

Chip	μ ops Per Cycle									
	Performance Cores					Efficiency Cores				
	Sus-tained	Burst				Sus-tained	Burst			
		Total	Int	AS&FP	Mem		Total	Int	AS&FP	Mem
M2 Family	8	17	6	4	3LD, 2STA, 2STD	5	10	4	2	2LD/STA, 2STD
M3/M3 Pro	9	19	8	4	3LD, 2STA, 2STD	5	10	4	2	2LD/STA, 2STD
M3 Max	9	19	8	4	3LD, 2STA, 2STD	5	11	4	3	2LD/STA, 2STD
M4 Family	10	19	8	4	3LD, 2STA, 2STD	5	11	4	3	2LD/STA, 2STD

Table 5.14. μ op Execution Bandwidth: A-Series Chips

Chip	μ ops Per Cycle									
	Performance Cores					Efficiency Cores				
	Sus-tained	Burst				Sus-tained	Burst			
		Total	Int	AS&FP	Mem		Total	Int	AS&FP	Mem
A14 Bionic	8	17	6	4	3LD, 2STA, 2STD	4	9	3	2	2LD/STA, 2STD
A15 Bionic	8	17	6	4	3LD, 2STA, 2STD	5	10	4	2	2LD/STA, 2STD
A16 Bionic	8	17	6	4	3LD, 2STA, 2STD	5	10	4	2	2LD/STA, 2STD
A17 Pro	9	19	8	4	3LD, 2STA, 2STD	5	11	4	3	2LD/STA, 2STD
A18 Family	10	19	8	4	3LD, 2STA, 2STD	5	11	4	3	2LD/STA, 2STD

Integer Execution Unit latencies are typically 1 to 3 cycles, excluding divide-related instructions. Advanced SIMD and FP latencies are typically 2-8 cycles, excluding divide-related instructions. Instruction group latency and bandwidth tables are located in [Appendix A, Instruction Latency and Bandwidth](#).

The instruction throughput metric, measured as instructions retired per clock (IPC), provides a first-order indication of processor performance. High throughput workloads have an IPC near the sustained μ op bandwidth of the processor. In these cases, the processor is encountering no bottlenecks and finding sufficient independent instructions when executing the given code and is effectively using its maximum sustainable resources.

Table 5.15. Instruction Throughput Metric

Name and Formula (Event Definitions: Section 8.1, "CPU Counters With Performance Monitoring Events")	Description
Instructions Per Clock (IPC): <Ev INST_ALL> / <Ev CORE_ACTIVE_CYCLE>	Rate of Instruction Processing

However, even with a high IPC, it may well be possible to improve performance further. High IPC only means that the processor is effectively executing the instructions as specified. Depending on the algorithm, there may be other instructions available that perform more actual work per instruction. For example, a vectorized workload that runs 25% lower IPC than a scalar implementation of that same workload likely out performs that scalar implementation. In other cases, one instruction may perform the work of several and require fewer clocks cycles. The net result may be a lower IPC but higher overall performance. For example, `UBLAL` performs an unsigned absolute difference, up converts that result, and then accumulates it with previous results.

Recommendation: High IPC does not necessarily mean high performance. Improve the fundamental algorithm or implementation of the algorithm.

[Magnitude: High | Applicability: High] Especially when the IPC is high, explore algorithmic improvements such as more efficient data structures that require fewer instructions to access and manipulate. Explore vectorization to improve throughput per instruction, and leverage the rich set of instructions available in the Arm ISA to efficiently implement the algorithm (see [Chapter 2, ISA Optimization: Overview and Integer Unit](#)).

5.5.1

Movement of Data from General Purpose Registers to Vector Registers

The Arm ISA offers several instructions that transfer data from the general purpose registers to the vector registers, including `DUP(general)`, `FMOV(general)`, `INS(general)`, `MOV(from general)`, `SCVTF(scalar)`, `UCVTF(scalar)`. When these instructions execute, they consume a load issue slot. The movement portion of these instructions (MOVE2VEC) has a latency of 4 or 5 cycles, depending on microarchitectural condition. The remainder of this section assumes 4 for illustrative purposes.

When the data is written directly to the H/S/D subset of the Vector register (for example, versions of the `FMOV` instruction), no additional latency is incurred. However, if the data is destined for other portions of the vector register or needs to be duplicated or type converted, an additional logic operation is required. For instructions that leave some elements of the destination vector register unwritten, the latency through that vector register source is 2 cycles.

In general, minimize data movement from the general purpose registers to the vector registers since the operations incur additional latency and may block loads from issuing. For example, consider the following code snippet. According to the program specification, the algorithm may load an integer value and convert it into a floating-point value:

```
LDR    W0, [X1] // Loads into a general purpose register
SCVTF D1, W0   // 4+3 cycle latency, 4 for the movement and 3 for the conversion
```

The above sequence incurs 11 cycles of latency: 4 for the load instruction and 7 for the convert instruction. Instead, do the following:

```
LDR    S0, [X1] // Loads directly into a vector register
SCVTF D1, S0    // 3 cycle latency for the conversion, with no movement delay
```

Although s0 is nominally a *floating point* register, it can still contain integer values. By avoiding a superfluous trip between functional units, this version is significantly faster, 7 cycles instead of 11.

It can sometimes be more efficient to perform simple mathematical operations involving integers in the Advanced SIMD and FP Execution Unit, if it avoids excess movement between the Integer and Advanced SIMD and FP Execution Units. In the previous example, if a multiplication or addition is necessary to scale or shift the loaded integer value before conversion with `SCVTF`, it is often faster to insert an integer-type SIMD instruction into the middle of the second sequence than an integer unit instruction into the first. This integer-type SIMD instruction operates over all elements, and effectively wastes all but element 0. However, because SIMD instruction latencies are typically longer than similar operations in the Integer Execution Unit, the latency benefits can be lost if too many integer instructions are executed in the SIMD unit.

Recommendation: Minimize data movement between general purpose registers and vector registers.

[Magnitude: Low | Applicability: Medium] In general, minimize movement back and forth between the general purpose registers and the vector registers to avoid latency associated with the movement. For short sequences of scalar integer code where data is destined for the Advanced SIMD and FP Execution Units, consider loading it directly into the vector register file (or leaving it in the vector register file), and operating on it using SIMD integer-type instructions (wasting all but element 0) to avoid excess data movement. Data movement between registers, though, is still typically significantly faster than executing a vector store instruction followed by an integer load instruction.

5.5.2

Movement of Data from Vector Registers to General Purpose Registers

When moving results from the vector registers to the general purpose registers without any type conversion, the ARM ISA offers `UMOV`, `SMOV`, `FMOV`, or `MOV(to general)`. `FMOV` just moves the FP register contents directly over to the integer register file without any type of conversion, while `UMOV` and `SMOV` perform a move along with a sign or zero extend of the integer value being moved, requiring extra latency. This may be common especially for byte- and halfword-size integers. However, for cases when sign or zero extension are not needed, use `FMOV`, even if the value being moved is an integer. These data movement instructions execute in 3 or 4 cycles of latency.

Recommendation: Use `FMOV` instruction when moving data from vector registers to GPRs when no conversion is needed.

[Magnitude: Low | Applicability: Medium] The `FMOV` instruction does not perform any conversion when moving data from the vector registers to the general purpose registers, whereas `UMOV` and `SMOV` do. Use the lower bandwidth and lower latency `FMOV` wherever possible.

The Arm ISA offers a number of related instructions to convert and move floating point values in vector registers H/S/D to integer (or fixed point) values in the general purpose registers: `FCVT(AMNPZ)(SU)(scalar)`. These require an operation to perform the conversion as well as an operation to move the data, resulting in a latency of 6 or 7 cycles.

5.5.3 Preferred Instructions for Common Operations

Use the following preferred instructions for common operations.

5.5.3.1 Register Data Copy

Use the following preferred instructions for creating copies of values in registers.

Table 5.16. Register Data Copy Instructions

Register Type	Register Data Copy Instructions
Integer	<code>MOV Xd, Xn</code> <code>ORR Xd, XZR, Xn</code> // OR a 0 with Xn into Xn <code>ADD Xd, Xn, #0</code> // Add a 0 to Xn into Xd
ASIMD and FP	<code>FMOV Dd, Dn</code> <code>MOV.2D Qd, Qn</code> <code>ORR.16B Vd, Vn, Vm</code> when <code>(Vm==Vn)</code> // Or Vm with itself into Vd

When creating multiple copies of a value in registers, avoid chaining, and instead create multiple copies from the original:

```
MOV X6, X5
MOV X7, X6    // Avoid chaining copies

MOV X6, X5
MOV X7, X5    // Prefer creating multiple copies from the original
```

Recommendation: Use preferred instructions for creating copies of a value in registers.

[Magnitude: Medium | Applicability: Medium] Use the preferred instructions and avoid chaining.

5.5.3.2 Register Zeroing

Use the following preferred instructions for zeroing registers and breaking dependence chains.

Table 5.17. Register Data Zeroing Instructions

Register Type	Register Data Zeroing Instructions
Integer	MOVZ Xd Wd, #0 // Equivalent to MOV Xd Wd, #0
ASIMD and FP	MOVI.2D Qd, #0

Because the Arm ISA is a fixed-length instruction set, moving an immediate of 0 to a register is the same number of instructions bytes as any other instruction. Other variable-length instruction sets recommend using other operations to zero registers that do not involve an immediate to reduce code size.

Note that while Exclusive-OR of a register with itself produces a 0 value on the Arm ISA, it does not break dependency chains like it may on other ISAs. That is, the Exclusive-OR cannot execute until the source register has been written by the producer instruction, which delays the consumer of the Exclusive-OR accordingly. Specifically, in the following example, the `EOR` enforces a specific ordering of the two loads:

```
LDR X2, [X1]
EOR X3, X2, X2    // Dependencies through X2 and X3 are enforced
LDR X4, [X1, X3]
```

See the description of "Address dependency" in Section E2.3.2 of the Arm Architecture Reference Manual for more information.

Recommendation: Use preferred instructions for zeroing registers.

[Magnitude: Low | Applicability: Low] To zero a register, use specific move immediate instructions. Unlike other architectures, move immediates are the same code size as other instructions that could zero a register, and thus other combinations are not needed. Also unlike other ISAs, Exclusive-OR operations where both inputs are the same register must obey producer dependencies.

5.5.3.3 Register Constants

Use the following preferred instructions for loading constants into registers:

Table 5.18. Register Data Constant Instructions

Register Type	Register Data Constant Instructions
Integer	MOVZ X Wd, #imm MOVN X Wd, #imm ORR X Wd, X Wzr, #imm EOR X Wd, X Wzr, #imm ADR Xd, <label> ADRP Xd, <label> BL <label>

Recommendation: Use preferred instructions for loading constants into registers.

[Magnitude: Low | Applicability: Medium] Use the prefer instructions for loading constants. However, prioritize the use preferred instruction pairs where applicable.

5.5.3.4 Limited FPCR and FPSR Bandwidth

Reads and writes to the floating point Status Register (FPSR) and floating point Control Register (FPCR) incur high access cost. Because of the high cost, do not read and write these registers frequently, but rather limit access to no more than every few thousand instructions.

Recommendation: Limit read and write frequency to floating point control and status registers (FPCR and FPSR).

[Magnitude: Low | Applicability: Low] To avoid the high cost of these operations, limit access to no more than every few thousand instructions.

5.5.4 Unroll and Software Pipeline Long Sequential Loop Bodies

When executing loop bodies with long regions of sequential code (for example, several dozen instructions or more), the processor must fetch and map the entire loop body into the instruction window before proceeding to the next iteration. This may delay execution of independent work available in subsequent iterations while fetching and mapping remaining dependent work from the first iteration. And, because the instruction window is of finite size, few iterations may be resident in the instruction window prior to it filling and stalling the Mapper.

Unrolling and interleaving instructions from multiple iterations may allow independent work from multiple iterations to move into the instruction window faster. Unrolling comes with a cost, however, in that it often increases the code size and can cause instruction cache and TLB capacity issues. Software pipelining techniques also aim to overlap loop iterations by staggering the start of subsequent iterations in a continuous manner before previous iterations have been completely inserted into the instruction window. Software pipeline transformations are more complex but can make independent work available to the processor with, in some cases, less code size increase.

Compilers often unroll automatically according to various heuristics when higher levels of optimization are specified.

For more direct control, see `#pragma unroll loop` in the [Clang Language Extensions](#) guide.

Recommendation: Perform limited unrolling with interleaving or software pipelining of long loop bodies for improved parallelism.

[Magnitude: Medium | Applicability: Medium] For code sequences that consist of long regions of sequential code, consider limited loop unrolling and interleaving of the instructions of the unrolled copies. Consider this technique when parallelism across iterations is high but actual IPC is not. After performing this optimization, check instruction cache and instruction TLB miss rates to identify any new bottlenecks that might have been created.

5.6 Instruction Processing Memory Optimization

The following sections describe optimization opportunities to identify and improve memory issues.

Data movement between memory and the processor cores frequently limits overall application performance. This section describes the memory-system hierarchy and highlights important principles to effectively use resources. Note that inefficient use of cache capacities, organization of data, and patterns of access can have a major impact on application performance.

[Bottleneck Analysis](#) in the Instruments tool offers empirical measurements about the relative impact of memory bottlenecks within the Instruction Processing bottleneck analysis, the relative impacts of incorrect store-to-load forwarding prediction, and additional preconfigured sets of performance monitoring events for deep analysis.

5.6.1 Address Generation

Once the Scheduler issues a memory μ op, the Load and Store Execution Units calculate the virtual address. The address-generation logic contains the necessary adders and shifters to handle inline most of the addressing formats defined in the Arm ISA (see [Section 2.8, “Addressing Forms, Instruction immediates, and Operand Shifts”](#)). The only exception is for Advanced SIMD and FP loads and stores when using `ls1 #imm` or `<extend> #imm` forms when the immediate is 4 or greater. As noted in [Section 5.4.8, “Multi- \$\mu\$ op Instructions”](#), these require an extra μ op to calculate the address.

Other addressing forms may require additional μ ops to update base registers or update multiple elements.

5.6.2 L1 Data (L1D) TLB and Shared L2 TLB

After the processor computes a virtual address, the Load and Store Execution Unit converts the virtual address to a physical address in order to access the memory system. The translation process occurs in the Memory Management Unit which caches the result of the translations in the Translation Lookaside Buffers. Application data pages are 16KiB.

Table 5.19. L1D 16KiB-Page TLB Organization: M-Series Chips

Chip	Entries (Coverage)	
	Performance Cores	Efficiency Cores
M1 Family	160 entries (2.5MiB)	128 entries (2MiB)
M2 Family	256 entries (4MiB)	192 entries (3MiB)
M3 Family	160 [†] entries (2.5MiB)	192 entries (3MiB)
M4 Family	160 entries (2.5MiB)	192 entries (3MiB)

[†]Reduced capacity over prior generation but includes microarchitectural improvements

Table 5.20. L1D 16KiB-Page TLB Organization: A-Series Chips

Chip	Entries (Coverage)	
	Performance Cores	Efficiency Cores
A14 Bionic	256 entries (4MiB)	128 entries (2MiB)
A15 Bionic	256 entries (4MiB)	192 entries (3MiB)
A16 Bionic	160 [†] entries (2.5MiB)	192 entries (3MiB)

Table 5.20. L1D 16KiB-Page TLB Organization: A-Series Chips (cont.)

Chip	Entries (Coverage)	
	Performance Cores	Efficiency Cores
A17 Pro	160 entries (2.5MiB)	192 entries (3MiB)
A18 Family	160 entries (2.5MiB)	192 entries (3MiB)

[†]Reduced capacity over prior generation but includes microarchitectural improvements

If the processor cannot find a translation in the L1D TLB, it looks up the virtual address in the much larger Shared L2 TLB. This L2 TLB also backs the L1I TLB.

Table 5.21. Shared L2 16KiB-Page TLB Organization: M-Series Chips

Chip	Entries (Coverage)	
	Performance Cores	Efficiency Cores
M1 Family	3072 entries (48MiB)	1024 entries (16MiB)
M2 Family	3072 entries (48MiB)	2048 entries (32MiB)
M3 Family	3072 entries (48MiB)	2048 entries (32MiB)
M4 Family	3072 entries (48MiB)	2048 entries (32MiB)

Table 5.22. Shared L2 16KiB-Page TLB Organization: A-Series Chips

Chip	Entries (Coverage)	
	Performance Cores	Efficiency Cores
A14 Bionic	3072 entries (48MiB)	1024 entries (16MiB)
A15 Bionic	3072 entries (48MiB)	2048 entries (32MiB)
A16 Bionic	3072 entries (48MiB)	2048 entries (32MiB)
A17 Pro	3072 entries (48MiB)	2048 entries (32MiB)
A18 Family	3072 entries (48MiB)	2048 entries (32MiB)

If the processor cannot find a translation in any of the TLBs, then it automatically initiates a page table walk for the virtual address via Memory Management Unit. A full page table walk is a multi-step walk of a tree structure starting at the base node and ending at a leaf node that contains the page's physical address. Steps require a memory operation to access the appropriate node in the page table structure. The MMU issues the memory request to the Shared L2 Cache, which may have the data cached. Otherwise, the L2 Cache sends the request over the fabric to the memory system.

By leveraging multilevel TLBs and the Shared L2 Cache, the overhead due to virtual memory translation gradually worsens as the working set size increases. Nonetheless, Shared L2 TLB and MMU bandwidth is limited.

Use the following metrics to evaluate TLB performance.

Table 5.23. Common L1D TLB, Shared L2 TLB, and Address Translation Metrics

Name and Formula (Event Definitions: Section 8.1, "CPU Counters With Performance Monitoring Events")	Description
L1D TLB Miss Retired Density (of loads and stores): <Ev L1D_TLB_MISS_NONSPEC> / <Ev INST_LDST>	Proportion retired load and store instructions that missed L1D TLB of all retired load and store instructions
L1D TLB Miss Density (of L1D TLB accesses): <Ev L1D_TLB_MISS> / <Ev L1D_TLB_ACCESS>	Proportion speculative L1D TLB misses of L1D TLB accesses including loads, stores, prefetches, etc.
L1D TLB Fill Density (of L1D TLB accesses): <Ev L1D_TLB_FILL> / <Ev L1D_TLB_ACCESS>	Proportion speculative L1D TLB fills for any reason of L1D TLB accesses including loads, stores, prefetches, etc.
Instruction L2 TLB Miss Density (of L1 TLB misses): <Ev L2_TLB_MISS_INSTRUCTION> / <Ev L1_TLB_FILL>	Proportion speculative L2 TLB instruction misses of unique L1 TLB misses (counted by fills into the L1 TLB)
Data L2 TLB Miss Density (of L1D TLB misses): <Ev L2_TLB_MISS_DATA> / <Ev L1D_TLB_FILL>	Proportion speculative L2 TLB instruction misses of unique L1D TLB Misses (counted by fills into the L1D TLB)
Instruction Table Walk Request Density (of data L2 TLB misses): <Ev MMU_TABLE_WALK_INSTRUCTION> / <Ev L2_TLB_MISS_INSTRUCTION>	Mean speculative instruction table walk read requests per instruction L2 TLB miss
Data Table Walk Request Density (of data L2 TLB misses): <Ev MMU_TABLE_WALK_DATA> / <Ev L2_TLB_MISS_DATA>	Mean speculative data table walk requests per data L2 TLB miss

Recommendation: Compact data or improve temporal access locality to reduce virtual address translation overhead.

[Magnitude: Medium | Applicability: Medium] Avoid frequent long-latency page table walks due to sporadically accessing sparse data. Utilize data structures that keep the working page set within the TLB capacity limits. Or, organize accesses to data to improve page temporal locality.

5.6.3 L1 Data (L1D) Cache

Each core uses its own private L1 Data Cache. It usually operates as a write-back write-allocate cache. The write allocation policy keeps the recently written data in the L1 cache, where it is conveniently located for re-reading. Load μops that hit in the L1D Cache execute with a latency of 4 cycles. See [Section A.3.1, "Load Latency"](#) for latency ranges of more complex load instructions.

Although the system operates on 128B cache lines, the L1D Cache operates on 64B cache lines. See [Section 5.6.6, "Improving Cache Hierarchy Performance"](#) for recommendations on improving data cache hierarchy performance.

Table 5.24. L1D Cache Organization: M-Series Chips

Chip	Capacity, Associativity, and Line Size	
	Performance Cores	Efficiency Cores
M1 Family	128KiB, 8-way, 64B lines	64KiB, 8-way, 64B lines
M2 Family	128KiB, 8-way, 64B lines	64KiB, 8-way, 64B lines
M3 Family	128KiB, 8-way, 64B lines	64KiB, 8-way, 64B lines
M4 Family	128KiB, 8-way, 64B lines	64KiB, 8-way, 64B lines

Table 5.25. L1D Cache Organization: A-Series Chips

Chip	Capacity, Associativity, and Line Size	
	Performance Cores	Efficiency Cores
A14 Bionic	128KiB, 8-way, 64B lines	64KiB, 8-way, 64B lines
A15 Bionic	128KiB, 8-way, 64B lines	64KiB, 8-way, 64B lines
A16 Bionic	128KiB, 8-way, 64B lines	64KiB, 8-way, 64B lines
A17 Pro	128KiB, 8-way, 64B lines	64KiB, 8-way, 64B lines
A18 Family	128KiB, 8-way, 64B lines	64KiB, 8-way, 64B lines

Note: Capacity and associativity specifications are provided for reference. Software should check the appropriate `sysctl` parameter to dynamically adjust to CPU and chip configurations. See [Appendix B, Dynamic Determination of Chip-Specific Capabilities](#) for more information.

Use the following metrics to evaluate L1D Cache performance.

Table 5.26. Common Data Cache Metrics

Name and Formula (Event Definitions: Section 8.1, "CPU Counters With Performance Monitoring Events")	Description
Load L1D Cache Miss Retired Density (of instructions): <code><Ev L1D_CACHE_MISS_LD_NONSPEC> / <Ev INST_ALL></code>	Proportion retired L1D Cache loads that miss the cache of all instructions.
Store L1D Cache Miss Retired Density (of instructions): <code><Ev L1D_CACHE_MISS_ST_NONSPEC> / <Ev INST_ALL></code>	Proportion retired L1D Cache stores that miss the cache of all instructions.
Load L1D Cache Miss Retired Density (of retired loads): <code><Ev L1D_CACHE_MISS_LD_NONSPEC> / (<Ev INST_INT_LD> + <Ev INST_SIMD_LD>)</code>	Proportion retired L1D Cache loads that miss the cache.
Store L1D Cache Miss Retired Density (of retired stores): <code><Ev L1D_CACHE_MISS_ST_NONSPEC> / (<Ev INST_INT_ST> + <Ev INST_SIMD_ST>)</code>	Proportion retired L1D Cache stores that miss the cache.
Load L1D Cache Miss Density (of load pipeline accesses):	Proportion speculative load pipeline accesses that miss the L1D Cache. Accesses may count more than

Table 5.26. Common Data Cache Metrics (cont.)

Name and Formula (Event Definitions: Section 8.1, "CPU Counters With Performance Monitoring Events")	Description
$\text{<Ev L1D_CACHE_MISS_LD> / <Ev LD_UNIT_UOP>}$	once if the load is replayed within the Load and Store Execution Unit or is split across a cache line, and includes prefetches and other operations that may use the pipeline.
Store L1D Cache Miss Density (of store pipeline accesses): $\text{<Ev L1D_CACHE_MISS_ST> / <Ev ST_UNIT_UOP>}$	Proportion speculative store pipeline accesses that miss the L1D Cache. Accesses may count more than once if the store is replayed within the Load and Store Execution Unit or is split across a cache line, and includes prefetches and other operations that may use the pipeline.
L1D Dirty Writeback Density (of load and store pipeline accesses): $\text{<Ev L1D_CACHE_WRITEBACK> / (<Ev LD_UNIT_UOP> + <Ev ST_UNIT_UOP>)}$	Proportion speculative load and store pipeline accesses that result in writeback of dirty data out of the L1D Cache toward the L2 Cache.

5.6.4 Shared L2 Cache

Each cluster of cores uses a Shared L2 Cache which operates on 128B cache lines. It is shared between instructions and data, and between all cores in the cluster. Load μops that hit in the Shared L2 Cache execute with an average latency of 15 cycles under unloaded conditions. Some loads may be a few cycles faster and some slower, depending on microarchitectural conditions. The Shared L2 Cache is address banked to allow multiple accesses in parallel for increased throughput. In rare cases, intense strided accesses can cause imbalance in activity across the banks, resulting in a bottleneck on one bank and reduced overall throughput. The mapping of addresses to banks is a hash of select address bits designed to avoid imbalance, and that hash pattern may change across chips and families.

Table 5.27. Shared L2 Cache Organization: M-Series Chips

Chip	Capacity, Associativity, and Line Size	
	Performance Cluster	Efficiency Cluster
M1 Family	12MiB, 12-way, 128B lines	4MiB, 16-way, 128B lines
M2 Family	16MiB, 16-way, 128B lines	4MiB, 16-way, 128B lines
M3 Family	16MiB, 16-way, 128B lines	4MiB, 16-way, 128B lines
M4 Family	16MiB, 16-way, 128B lines	4MiB, 16-way, 128B lines

Table 5.28. Shared L2 Cache Organization: A-Series Chips

Chip	Capacity, Associativity, and Line Size	
	Performance Cluster	Efficiency Cluster
A14 Bionic	8MiB, 16-way, 128B lines	4MiB, 16-way, 128B lines
A15 Bionic	12MiB, 12-way, 128B lines	4MiB, 16-way, 128B lines
A16 Bionic	16MiB, 16-way, 128B lines	4MiB, 16-way, 128B lines

Table 5.28. Shared L2 Cache Organization: A-Series Chips (cont.)

Chip	Capacity, Associativity, and Line Size	
	Performance Cluster	Efficiency Cluster
A17 Pro	16MiB, 16-way, 128B lines	4MiB, 16-way, 128B lines
A18	8MiB, 16-way, 128B lines	4MiB, 16-way, 128B lines
A18 Pro	16MiB, 16-way, 128B lines	4MiB, 16-way, 128B lines

Note: Capacity and associativity specifications are provided for reference. Software should check the appropriate `sysctl` parameter to dynamically adjust to CPU and chip configurations. See [Appendix B, Dynamic Determination of Chip-Specific Capabilities](#) for more information.

Shared L2 Cache misses may be serviced by either the memory system, including the Memory Cache (See [Section 5.6.5, “Memory Cache”](#)), or another cluster's caching hierarchy depending on data ownership. Load μ ops that hit in another cluster's caching hierarchy execute with a latency in the range of 50ns, which is closer to that of the Memory Cache than the local cluster's L2. Heavy system traffic may increase these latencies.

5.6.5 Memory Cache

Attached to the DRAM interface is a Memory Cache (M Cache) operating on 128B cache lines. It is shared by all agents on the chip that can access DRAM directly, including the GPU and other co-processors and fixed function hardware. Load μ ops that hit in the M Cache execute with a latency in the range of 35ns, while load μ ops that access DRAM execute with latency in the range of 95ns. These latencies may increase if the system is experiencing heavy traffic.

Table 5.29. Memory Cache Organization: M-Series Chips

Chip	Capacity, Associativity, and Line Size
M1	8MiB, 16-way, 128B lines
M1 Pro	24MiB, 12-way, 128B lines
M1 Max	48MiB, 12-way, 128B lines
M1 Ultra	96MiB, 12-way, 128B lines
M2	8MiB, 16-way, 128B lines
M2 Pro	24MiB, 12-way, 128B lines
M2 Max	48MiB, 12-way, 128B lines
M2 Ultra	96MiB, 12-way, 128B lines
M3	8MiB, 16-way, 128B lines
M3 Pro	12MiB [†] , 16-way, 128B lines
M3 Max	48MiB, 12-way, 128B lines
M3 Ultra	96MiB, 12-way, 128B lines
M4	8MiB, 16-way, 128B lines
M4 Pro	24MiB, 12-way, 128B lines

Table 5.29. Memory Cache Organization: M-Series Chips (cont.)

Chip	Capacity, Associativity, and Line Size
M4 Max	48MiB, 12-way, 128B lines

[†]Reduced capacity over prior generation but includes microarchitectural improvements

Table 5.30. Memory Cache Organization: A-Series Chips

Chip	Capacity, Associativity, and Line Size
A14 Bionic	16MiB, 16-way, 128B lines
A15 Bionic	32MiB, 16-way, 128B lines
A16 Bionic	24MiB [†] , 12-way, 128B lines
A17 Pro	24MiB, 12-way, 128B lines
A18	12MiB, 12-way, 128B lines
A18 Pro	24MiB, 12-way, 128B lines

[†]Reduced capacity over prior generation but includes microarchitectural improvements

Recommendation: Perform cache blocking optimization based on L1D and L2 Shared Cache sizes.

[Magnitude: Medium | Applicability: Medium] Some algorithms benefit from dividing up the data set into blocks that fit into the cache, then operating on the blocks one-by-one. Do not block algorithms based on the M Cache capacity. The M Cache is shared amongst several agents and large portions of the cache may be consumed by those agents.

5.6.6 Improving Cache Hierarchy Performance

Consider these data layout optimizations to improve cache hierarchy performance:

- Eliminate false sharing:** A performance bottleneck can occur when two or more independent variables are allocated to the same 128B cache line and are modified by threads running on different core clusters. When a core in one cluster writes to one of the independent variables, it caches the line in its own Shared L2 Cache and L1D Cache. This forces the line to be evicted from all other caches in the system. Meanwhile, a core in the other cluster can attempt to write to the second variable, request the cache line, and the caching subsystem moves the entire line containing both variables over to its Shared L2 Cache and L1D Cache. If the first core continues to modify its variable, the line then moves back to the first core, and so on as the cycle repeats.

This unnecessary cache line *thrashing*, or rapid and repeated movement of a cache line back and forth between the clusters, can result in poor multithreaded performance. To avoid this problem, simply add padding as needed between any independent variables that may be used simultaneously by more than one core, thereby ensuring that they are always allocated to different 128B cache lines. In properly tuned code, any remaining cache line thrashing should only occur when "true" sharing occurs, when *individual* variables are actively being shared by multiple cores.
- Use cache line sharing constructively:** Having multiple variables packed together in the same cache line can be beneficial. If two or more variables are commonly used

together by one core, but seldom used by different cores at the same time, pack them into a single cache line to improve performance. Pack variables or data structure elements that are accessed together into one 64B cache line to match the L1D Cache line size. When one datum is requested, the related data naturally arrive with it in the same cache line, avoiding further cache misses. Data packed into the same 128B cache line also benefit from constructive sharing when this larger line is cached in the Shared L2 Cache, although two L1D misses may be incurred to bring both constituent 64B lines into the L1D Cache from the Shared L2 Cache.

More broadly, identify the most commonly accessed (*hot*) variables in structures and place them together so they fall in the same cache line(s). Afterwards place infrequently used (*cold*) variables where space allows. Splitting hot from cold in this manner may reduce the working set size, and thus improve the effectiveness of the cache, because only the hot portion of the structures occupy cache space in the common case. For example, group all of the pointers interlinking data nodes together at the beginning of nodes for data structures such as linked lists or trees, so that only one cache line per node loads during a typical list or tree traversal.

- **Order data fields to minimize padding:** Some high-level languages do not allow the compiler to reorder individual data elements from that specified in the source, such as fields in a structure in C language code. Further, these fields are also often aligned according to their size (1B elements to 1B boundaries, 4B elements to 4B boundaries, etc.). To achieve this alignment, padding bytes may be present in between fields. Order data elements such that few gaps are present from the end of one element to the start of the next, according to their natural alignment. In this example, placing the one byte `bool` elements next to each other reduces padding bytes and overall structure size:

```
typedef struct _mystruct {           // sizeof() = 12
    bool valid;                      // byte 0
    int data;                        // [3 byte gap]; bytes 4-7
    bool ready;                      // byte 7
} mystruct;

typedef struct _mystruct2 {          // sizeof() = 8
    bool valid;                      // byte 0
    bool ready;                      // byte 1
    int data;                        // [2 byte gap]; bytes 4-7
} mystruct2;
```

False sharing often causes dramatic performance loss, and fixing such issues can lead to significant performance improvement. Typically, identify and fix these issues first. Then identify hot variables, placing them together in a compact order according to their natural alignment. Pack cold variables into gaps to eliminate padding. Finally, pack the remaining cold variables.

For information on the sizes of various data types, see [Table 3.12: “Advanced SIMD Intrinsic Data Types”](#) and [Writing Arm64 Core for Apple Platforms](#).

Recommendation:

Avoid false sharing by allocating independent shared variables to different 128B cache lines.

[Magnitude: High | Applicability: Low] Avoid false sharing bottlenecks by allocating each independent shared variable to a different 128B cache line. This avoids situations where a cached shared variable, possibly related a software semaphore, is not inadvertently invalidated from the cache due another core using a different shared variable. Thrashing in this manner may significantly slow multi-threaded performance.

Recommendation:

Pack hot variables into the smallest set of cache lines for improved cache hierarchy performance, concentrating data commonly used together into the same 64B cache lines.

[Magnitude: High | Applicability: Medium] Identify and place the commonly used variables into cache lines in a compact order to reduce padding according to natural alignments. Place cold variables into gaps, and pack remaining cold variables adjacent to the hot variables.

5.6.7 Fast Load-to-Load Latency (Pointer Chasing)

The processor features a fast pointer chasing path with a 3-cycle latency. This works with all forms of addressing when the base operand arrives directly from the load execution unit as the data result of a prior load. The specific register number does not matter. For example:

```

LDR x2, [xn, #imm]!           // Latency is 3 cycles when feeding into
                                // the base operand of a subsequent load
LDR Xt, [x2, Xm, lsl #imm]
```

An example of a pointer chasing loop:

```

label:
    LDR R1, [R1, #8]
    ... // Other Instructions: No writes to R1
    BNE <label>
```

To further optimize this loop, pack the critical fields used by the "Other Instructions" along with the next pointer into a single cache line. See discussion of "hot" fields in [Section 5.6.2, "L1 Data \(L1D\) TLB and Shared L2 TLB"](#).

Recommendation:

Leverage fast load-to-load latency when pointer chasing.

[Magnitude: Medium | Applicability: Medium] When quickly walking from one data structure node to the next, ensure fast load latency by directly using the result register from the first load as the *xn* (base) operand in the second load. Do not put the result register in the *xm* (offset) operand of the second load nor pass the result value through any intervening instructions. Instructions unrelated to the result register in between the loads do not break this optimization.

5.6.8 Non-Temporal (NT) Accesses

Caches naturally store data close to the processor under the assumption that the data in the cache line is likely to be used again in the near term. Data that benefits from caching is said to have temporal locality. However, some algorithms stream (reading or writing) through large quantities of data with little temporal reuse. In these cases, storing the data in the cache is counterproductive because it floods (or "pollutes") the cache with unneeded data, forcing out potentially useful data.

The Load Pair with Non-Temporal Hint (LDNP) and Store Pair with Non-temporal Hint (STNP) instructions inform the memory system that the application is not likely to reuse the data again soon. Based on these hints, the processor optimizes cache allocation policies for particular cache lines to minimize pollution.

Use the Non-Temporal instructions in the following circumstances:

- **Non-Temporal Loads:** Use these versions of load instructions when the data is likely to be used only briefly, such as when streaming through large blocks of memory. The processor retains the cache line for a short time to allow accesses to additional data elements in the same cache line. Do not use non-temporal loads if the algorithm is likely to frequently use the data, nor if the algorithm subsequently modifies the data. And do not mix non-temporal accesses with regular accesses during the window of use. The processor ensures correct memory ordering in these cases, but the algorithm may suffer from poor performance.
- **Non-Temporal Stores:** Use these versions of store instructions when the data is being streamed out to memory. The processor is most efficient when all 128 bytes of a particular cache line are written by non-temporal stores within a brief window of time. Within that window, do not mix regular store and non-temporal store accesses to the same line, and do not use non-temporal stores if the algorithm is likely to read any of the written data in the near future. The processor ensures correct memory ordering in these cases, but the algorithm may suffer from poor performance.

The Clang compiler provides a builtin that allows the compiler to generated non-temporal stores.

```
void __builtin_nontemporal_store(T value, T *addr);
```

The types `T` currently supported are integer, floating point, and vector types. Note that the compiler does not guarantee insertion of non-temporal stores.

Recommendation: Utilize non-temporal stores for large data sets with low temporal locality.

[Magnitude: Medium | Applicability: Low] When streaming through large data sets where the application is unlikely to access individual data elements again in the near future, use non-temporal stores to avoid polluting the caches and evicting more useful data.

Some library functions that implement memory movement, in particular `memcpy()`, are optimized and use NT accesses where beneficial.

Recommendation: Utilize tuned library functions for data movement, such as `memcpy()`, rather than writing your own.

[Magnitude: Medium | Applicability: High] These functions are optimized for various scenarios, and include non-temporal accesses to improve performance.

Measure the non-temporal load and store rates with the following metrics:

Table 5.31. Common Non-Temporal Access Metrics

Name and Formula (Event Definitions: Section 8.1, "CPU Counters With Performance Monitoring Events")	Description
Non-Temporal Load Density: <Ev LD_NT_UOP> / <Ev LD_UNIT_UOP>	Proportion speculative issued non-temporal loads of all issued load operations
Non-Temporal Store Density: <Ev ST_NT_UOP> / <Ev ST_UNIT_UOP>	Proportion speculative issued non-temporal stores of all issued store operations

5.6.9 Unaligned Accesses

The cores support unaligned accesses to normal, cacheable memory for all available data sizes. When executed sporadically, unaligned accesses generally complete without any observable performance impact to overall execution. Some accesses suffer delays, however, and the impact is often greater for stores than loads.

- Loads:** The L1 Data Cache is designed to be able to read data at any alignment. An unaligned load that does not cross a 64B boundary has no associated performance penalty and therefore executes with the same latency as an aligned load. If the unaligned load spans a 64B boundary and one portion experiences a cache miss, however, then additional delay may occur. This is especially the case if both portions experience cache misses, because the two miss requests are handled serially. Thus, in the worst case, an unaligned load that crosses a 64B boundary can take at least twice as long to resolve as a normal cache miss. An unaligned load that spans a virtual page boundary experiences a greater delay. It waits until it is the oldest active load before executing, in order to safely perform both page translations, as needed. Avoid this case if possible.
- Stores:** Unaligned stores that cross 16-byte boundaries use two write ports into the L1 Data Cache, one port for the portion of the store on either side of the boundary. As a result, if a stream of unaligned stores occurs, performance might be up to 2x slower than that of the aligned case, which can occur if every store crosses a 16-byte boundary. In addition, just as with unaligned loads that span page boundaries, a store that spans a virtual page boundary must wait until it is the oldest active store to complete the page translations.

For example, optimized streaming memory operations like `memcpy()` and `memset()` typically use 16B load and store operations. If software does not use 16B-aligned source and destination buffers, then every store crosses a 16B boundary and every fourth load crosses a 64B boundary. Both can also suffer from page-spanning penalties. Such operations see a significant drop in performance without alignment.

Finally, if either loads or stores must be unaligned in a particular algorithm, but not both, then it is usually preferable to have the loads be unaligned. Although both suffer page boundary spanning penalties, stores require more resources when spanning 16B boundaries, while loads don't and further may not even suffer penalties for spanning 64B boundaries.

As a guideline, occasional unaligned accesses do not result in significant performance loss, because the hardware efficiently manages accesses that span boundaries. However, avoid accessing large memory buffers using continuous streams of unaligned memory accesses, especially in performance-critical inner loops. Any misalignment penalty is likely to be triggered repeatedly for a large number of sequential accesses, which eventually make unaligned access penalties become apparent. Accesses to large buffers inevitably lead to a fair number of loads and stores that span cache line or page boundaries, which may result in reduced performance.

Table 5.32. Common Access Alignment Metrics

Name and Formula (Event Definitions: Section 8.1, "CPU Counters With Performance Monitoring Events")	Description
L1D 64B Cache Line Crossing Access Density (of load and store pipeline accesses): $\langle \text{Ev LDST_X64_UOP} \rangle / (\langle \text{Ev LD_UNIT_UOP} \rangle + \langle \text{Ev ST_UNIT_UOP} \rangle)$	Proportion of speculative load and store pipeline accesses that cross a 64B cache line boundary.
L1D Page Crossing Access Density (of load and store pipeline accesses): $\langle \text{Ev LDST_XPG_UOP} \rangle / (\langle \text{Ev LD_UNIT_UOP} \rangle + \langle \text{Ev ST_UNIT_UOP} \rangle)$	Proportion of speculative load and store pipeline accesses that cross a 16KiB page boundary

Recommendation: Minimize unaligned data access for large arrays.

[Magnitude: Medium | Applicability: Low] When accessing large arrays, align data according to the access size. Ex., for data accesses of 4B words, align data elements to 4B boundaries (least significant 2 bits are 0; #bxx..xx00). This prevents accesses from spanning boundaries.

Recommendation: If unaligned accesses are unavoidable, prefer unaligned loads over stores.

[Magnitude: Medium | Applicability: Low] If unaligned accesses are unavoidable due to algorithmic constraints, generally prefer unaligned loads to unaligned stores. Stores suffer delays for more boundary cases than loads. However, if loads significantly outnumber stores in the algorithm, eliminating load penalties by aligning loads may improve performance if the store penalties are hidden behind load execution.

5.6.10 Memory Dependence Prediction

Load and store μops may execute and send their requests to the rest of the memory subsystem out of order and speculatively. Most of the time, this allows the memory units to process multiple memory accesses in parallel, which translates directly into higher performance execution. As with any other μops , however, the processor enforces in-order

execution when there is a true dependence between the μ ops. With load and store μ ops, it is possible to have dependencies through memory if a store is closely followed by a load of an overlapping data element. Such dependencies are not caught by the usual register-tracking mechanisms within each core. When a store and dependent load execute out of order, a memory ordering violation occurs and the offending load must be re-executed. If these violations occur often, performance can suffer significantly.

To minimize the number of violations while still allowing out-of-order execution, each core includes mechanisms to predict whether a load may consume data written by an older store that has not yet executed. This is especially challenging when the address of an older store is not known when the load is ready to execute.

The matrix of potential outcomes is depicted in [Table 5.33: "Store-to-Load Dependence Prediction Outcome"](#).

- **Worst Performance:** When a younger load is predicted to not depend on an older store and is allowed to speculatively execute. As noted above, when the store's address is computed and the dependence violation is detected, the processor flushes the pipeline and restarts the load and subsequent instructions. This memory order violation leads to wasted execution resources both to perform the flush, as well as to re-execute of all μ ops that had speculatively executed.
- **Poor Performance:** When a younger load is predicted to depend on an older store, but actually does not. These "phantom" dependences unnecessarily delay load execution.

In some cases, the predictor must decide whether or not to insert a dependency before data addresses can be computed for both the load and store in a particular store-to-load pair. As a result, the processor must make an educated guess whether or not a dependence may occur based on the past behavior for that pair of instructions. Unfortunately, it may predict too conservatively when a store-to-load pair is only occasionally dependent. Pipeline flushes are quite expensive, so it may favor the modest expense of enforcing dependences. This might occur when address pointers for the load and store are briefly sweeping past each other or when buffers only partially overlap, but usually the instructions are not dependent. Thrashing may occur if the dependent and not-dependent cases both regularly occur for the same store-load pairs. This may result in patterns where dependences are enforced when none are needed and never enforced when they are needed, leading to significant slowdown.

Table 5.33. Store-to-Load Dependence Prediction Outcome

Actual Behavior	Predicted Outcome	
	Not Dependent	Dependent
Load Not dependent on Store (Non-Overlapping Memory Range)	High Performance: Loads and stores may execute at any time without conflict	Low Performance: Although a load was dependent on a store at some point in the past, this overly conservative prediction unnecessarily forces further instances of that load to stall and wait for store completion
Load Dependent on Store (At Least Partially Overlapping Memory Range)	Lowest Performance: Memory Order Violation: If the older store's address is unknown	Moderate Performance: Loads are delayed until just after stores complete, properly honoring

Table 5.33. Store-to-Load Dependence Prediction Outcome (cont.)

Actual Behavior	Predicted Outcome	
	Not Dependent	Dependent
	when the younger load is ready to execute, the younger load may prematurely execute and require a pipeline flush with restart. OR Moderate Performance: If the older store's address is known, the processor may stall the load until the store is known.[†]	dependencies but not stalling excessively. [†]

[†]Note that data forwarding from store to load is often slower than through registers, resulting in moderate performance compared with versions that communicate through registers.

In real code, forwarding from a store to a load occurs surprisingly often. In particular, any code that uses numerous read-modify-write accesses to its data structures is likely to trigger these sorts of problems. Due to the large out-of-order window, the P cores can easily attempt to process multiple read-modify-write sequences at once, so the read of one sequence can occur before the write of the previous sequence is complete. If repeated accesses to particular elements in the data structure sometimes (but not always) occur "close" to each other, temporally, then memory ordering violations and subsequent pipeline flushes can occur as a result of these accesses. Note that "close" in this context means within the core's out-of-order instruction window: repeated read-modify-writes of the same element may trigger ordering violations if they occur within a few loop iterations of each other, but not if they occur hundreds or thousands of loop iterations apart.

Here are several types of data structures from real algorithms where store-to-load stalling or predictor thrashing are likely to occur:

- **Histograms:** These data structures tally input data according to some characteristic, and typically involve a read of the existing tally, followed by an increment and store of the updated tally. When histogram updates occur at relatively high rates, the processor may not have completed updating a first tally prior to beginning a second. In such cases, the processor predicts whether the tally loaded for the second update comes from memory or from the first tally's store. If from memory, the tally updates can be performed in parallel, otherwise the processor must serialize the tally updates.
- **Growing Data Structures:** Suppose there is a loop that accesses a rapidly growing data structure using read-modify-write accesses. At the beginning of execution when the data structure is small, it is likely that older stores feed younger loads because of the small data set. The predictor is trained accordingly to enforce those dependencies. If the data structure stays approximately the same size, then this is the ideal behavior. However, if the data structure grows, there are more data elements, and the likelihood of accessing the same data node within the out-of-order window is reduced. But, the predictor will already have been trained and will tend to enforce phantom dependencies during execution that are not necessary.

- **In-Place, Cache-Optimized FFT:** A variant of the previous situation can occur when the data structure stays the same size, but the access patterns to it create frequent read-modify-write operations to individual elements at first, but fewer later on.

A real-world case of this can be found in an in-place FFT that has been optimized to improve cache locality. One way to do this is to have loops that execute FFT butterflies with the following data elements: (0 1)(2 3)(0 1 2 3)(4 5)(6 7)(4 5 6 7)(0 1 2 3 4 5 6 7) . . . The algorithm begins with very small, radix-2 loops, then proceeds into the radix-4 loops that combine the results from the previous loops together. From there, it proceeds to radix-8 loops that combine those results together, and so on. This improves cache locality by performing as many FFT passes as possible on each block of data while it fits within the cache, no matter what the cache size might be.

Unfortunately it also tends to create store-to-load dependencies as the radix-4 loops consume the outputs of the radix-2 loops very quickly. As with the previous case, these legitimate dependencies tend to train the predictor conservatively, so that store-to-load dependence enforcement still occurs during later FFT passes. But during these later passes, the working set is larger, and so the reuse of the same data elements are spaced out much farther apart in time and seldom cause true dependencies.

- **Small, in-place, two-dimensional buffers:** In operations involving two-dimensional buffers, nearest neighbor elements are often read as input to help update the value of element (m, n). If the elements are modified in-place, then element (m, n-1) will be read back in to help compute (m, n) shortly after (m, n-1) was written. Depending on the operation and data set size, the instructions to compute an entire row may fit into a core's active instruction window. When this happens, store-to-load forwarding scenarios can occur as the loads from (m, n) start executing before the stores from (m, n-1) have completed. Similar scenarios can be constructed for in-place buffers with larger numbers of dimensions.
- **Passing Pointers:** If a buffer is being accessed and updated by multiple pointers that are incremented by different amounts during each loop iteration, they may cross at some point during the loop. A one-time flush occurs after the pointer collision, which then is followed by a number of iterations of overly conservative dependence enforcement.
- **List of Pointers to a Small Object Pool:** Perhaps the most pernicious access pattern can be triggered when loop iterations must perform read-modify-write updates to a small pool of data structures based on a long list of pointers. This pattern is quite common in graphics or physics algorithms that maintain a pool of objects and then a larger list of "links" between objects that are close to each other in space and need to be considered during each simulation step. When the same object just happens to be used in calculations during adjacent loop iterations, then dependence collisions and flushes are quite likely. This sort of hard-to-predict dependence behavior can easily cause thrashing between overly-conservative serialization and too little serialization.
- **Frequently modified top of stack:** In code that regularly pushes data to a stack data structure and then quickly pops it back off again, load-store dependence violations are likely. Moreover, if these locations on the stack are then used to hold different data variables for other functions soon thereafter, then the stale store-to-load predictions trained into the predictor for the earlier stack contents may trigger excessive serialization of loads and stores for these later uses of the stack. On the other hand,

address calculations for stack accesses are often quite simple, so store and load addresses can often be calculated quickly enough to avoid relying on the predictor.

Although these scenarios are not particularly common, speculative store-to-load prediction thrashing can seriously degrade performance. Production code examples have been seen that reduce performance by 50%.

Consider these strategies to avoid dependence prediction problems:

- **Avoid Read-Modify-Write Data Accesses:** The simplest way to avoid speculative store-to-load thrashing is to avoid looping algorithms that require read-modify-write access to data structures that are too large to be kept in registers. Algorithms where the input and output buffers for each loop are separate naturally avoid memory dependence violations. In contrast, violations may occur when one data structure is both read and written simultaneously within a single loop, so loads may read what was just written by a recent store. Although avoiding read-modify-write behavior is often possible, it may require a radically different algorithm and data structures that take up more space in memory than one might use with a read-modify-write-based algorithm. For example, an algorithm that uses double-buffered data structures instead of modify-in-place structures may be required. Another example is described in [Section 5.6.10.1, "Memory Dependence Prediction Example"](#).
- **Explicit Test for Identical Elements:** It is possible to use an `if` statement to check explicitly for the case when pointers may happen to be pointing at the same element, and treat these cases with special code. This may occasionally help, but is not recommended because it often just replaces store-to-load dependence problems with branch mispredictions, which may well be worse. Also, the test code must execute on every loop iteration, slowing down the common case of no collision.
- **Different Loops for Different Data Sizes:** A better way to select special code for colliding cases is to do the testing outside of the loop, which minimizes the number of tests and branch mispredictions. For situations like the "growing data structure" or "in-place FFT" examples, where it is known a priori that the early, short loops tend to trigger store-to-load dependencies, it can be helpful to use a second copy of the loop body included specifically for these "small loop" cases. Any store-to-load speculation problems that occur during these "early" loops do *not* affect speculative predictions for loads and stores in the "long version" loop body, which is only used when there is little or no potential for store-hit-younger-load problems. The small drop in instruction cache performance associated with having two essentially identical copies of the loop body can be easily amortized by the resulting performance savings.
- **Rearrange Read-Modify-Write Data Accesses:** For a case like the FFT example, it is possible to shuffle the loop iterations to avoid dependence violations. A simple reordering of the loop iterations keep read-modify-writes of the same iterations far apart, while still performing an FFT. For instance, change the order to (0 1)(2 3)(4 5)(6 7)(0 1 2 3)(4 5 6 7) (0 1 2 3 4 5 6 7) . . . , performing all butterflies of the same radix together as a complete pass before proceeding to the next radix level. Cache performance is not as optimized with this ordering, but performing iterations ordered this way up to about 1,024 data points or so is all that is necessary to avoid memory ordering violations. One can then switch to the original cache-optimized algorithm for later passes. This completely avoids store-to-load forwarding, which is an even better solution than the previous "Different Loops for Different Data Sizes" option above, and

can be cache-optimized for any realistic L1 cache size. Any algorithm where the order of accessing read-modify-write data elements can be safely changed to move accesses to the same data element farther apart without modifying the task that the code performs is amenable to this kind of alteration.

- **Resorted Pointer List:** Situations similar to the "List of Pointers" example, where the store-to-load dependencies occur more randomly, may be more difficult to optimize. One way that can be effective, however, is to sort the list of pointers to move read-modify-write operations to the same objects away from each other, so that at least they do not occur in adjacent loop iterations. This sorting can be done implicitly as the list is initially assembled, by adding pointers to the list in a way that inherently keeps re-uses of any particular object far apart. Or, it can be done between the creation of the list and its use by executing a sorting pass that finds cases where adjacent loop iterations access the same objects and shuffling them farther apart in the operation sequence.
- **Unroll Loops:** In some algorithms, a load from every n th iteration of the loop consumes data from a prior store. Unrolling the loop n times creates n copies of the loads and stores and stabilizes the dependence pattern. 1 copy of the load always obtains data from a prior store, while $n-1$ copies always obtain data from memory. Unrolling may generally help even when such a pattern is not obvious. To an extent, more copies of the loads and stores create more opportunities for the predictor to find stable dependence patterns between them.

These improvements or others like them are not always possible, either because the original algorithm is not amenable to change or because the changes cause performance degradation for other reasons, such as lower cache hit rates. However, if they are possible, then performance of the algorithm often increases in *any* out-of-order core. Application of these techniques can result in performance increases of 50-150% on programs that were originally limited by speculative store-to-load prediction thrashing.

Recommendation: Reduce hard to predict store-to-load forwarding.

[Magnitude: High | Applicability: Medium] Avoid algorithms where loads occasionally read recently written data. Consider restructuring data accesses to increase the number of dynamic instructions between the write and the subsequent read, so the read finds the data already written into the cache.

Alternatively, consider restructuring algorithms to avoid sporadic reads after writes, potentially leveraging registers to communicate the data.

When multiple in-flight stores simultaneously forward bytes to a single load, the predictor may be unable to properly enforce dependence ordering between loads and individual stores. For instance, a string buffer may be written using `STRB` instructions and subsequently loaded as a 16-byte aligned vector, instead. In these cases, the predictor may force the load to wait until all older stores have issued instead of being dependent on a specific store. Nevertheless, performance is still usually better than if the load receive incorrect data, and the processor is forced to flush the pipeline and re-execute the load. Because of the large instruction window, stores several hundred instructions older than the load may not yet have written the cache when the load is read to execute.

Recommendation: Reduce scenarios where multiple smaller stores simultaneously forward to a single larger load.

[Magnitude: Medium | Applicability: Medium] Minimize cases where memory is accessed as two differently-size data types, especially if those accesses may occur within a several hundred instructions of each other.

Table 5.34. Common Memory Dependence Metrics

Name and Formula (Event Definitions: Section 8.1, "CPU Counters With Performance Monitoring Events")	Description
Memory Order Violation Density: <Ev ST_MEMORY_ORDER_VIOLATION_NONSPEC> / <Ev ST_UNIT_UOP>	Proportion retired stores that caused a pipeline flush and replay of loads and subsequent instructions of all stores.

5.6.10.1 Memory Dependence Prediction Example

Consider the following code example. The "Add AND assignment" operation (+=) on `autoc[coeff]` performs a load of the existing value, adds to it, and stores out the new value. As long as the value of `d` remains `>= 0`, the load obtains the value from the previous store. But, in some iterations when `d < 0`, `coeff` index is incremented such that the load obtains the value from memory instead.

```
void filter(float *autoc, float *data, uint32_t data_len, uint32_t lag)
{
    float d;
    uint32_t sample, coeff = 0;
    const uint32_t limit = data_len - lag;

    for(coeff = 0; coeff < lag; coeff++)
        autoc[coeff] = 0.0;

    for(sample = 0; sample <= limit; sample++) {
        d = data[sample];
        if (d < 0) {
            coeff++;
            if (coeff == lag)
                break;
        }
        autoc[coeff] += d * data[sample+coeff];
    }
}
```

Knowledge of the data and algorithm may be useful when rewriting code to eliminate the memory dependences. In this example, all updates to a particular `autoc[coeff]` happen sequentially. The algorithm can be altered to load the initial value of `autoc[coeff]` into a local variable `sum`, accumulate all updates into that local variable, then write out that local variable into `autoc[coeff]` when the algorithm moves onto the next `coeff`. The compiler is likely to register allocate the local variable `sum` eliminating the per-iteration memory loads and stores altogether.

```
void filter_opt(float *autoc, float *data, uint32_t data_len, uint32_t lag)
{
    float d;
    uint32_t sample, coeff = 0;
    float sum = 0.0f;
    const uint32_t limit = data_len - lag;

    for(coeff = 0; coeff < lag; coeff++)
        autoc[coeff] = 0.0;

    for(sample = 0; sample <= limit; sample++) {
        d = data[sample];
        if (d < 0) {
            autoc[coeff] = sum;
            coeff++;
            sum = 0.0f;
            if (coeff == lag)
                break;
        }
        sum += d * data[sample+coeff];
    }

    if (coeff < lag)
    {
        autoc[coeff] = sum;
    }
}
```

5.6.11 Store-to-Load Forwarding

When the processor predicts that a load reads data written in all or in part by an older store, the load waits to execute until the data portion of the store is available. In such cases, the load may read its data from internal buffers prior to the store actually writing the cache. This is commonly referred to as *store-to-load forwarding*. The P cores feature an aggressive algorithm that allows loads to read data from multiple sources generally without additional delay, including reading some bytes from various younger stores (that have not yet written to the cache) and bytes from the cache itself.

Although data alignment for store-to-load-forwarding isn't often a limiter, cache line and page spanning loads may experience delays. (See [Section 5.6.9, "Unaligned Accesses"](#)). Similarly, when multiple stores simultaneously feed a single load, the dependence predictor may enforce conservative dependencies, requiring the store's addresses to be known prior to forwarding. (See [Section 5.6.10, "Memory Dependence Prediction"](#)).

In some instances, the processor may be able to further optimize store-to-load forwarding when resources are available:

- Single element 4B or 8B integer loads and stores only (without sign extension).
- Simple base + offset addressing only

Recommendation: Leverage flexible store-to-load forwarding when register communication is complex.

[Magnitude: Low | Applicability: Low] For algorithms with complex memory access patterns, forwarding data via registers might require complex data selection and shift/merge sequences. Rely on the aggressive store-to-load forwarding network to complete such data movement. Further, rely on optimized store-to-load forwarding of whole integer registers with simple addressing in situations such as register spills/fills.

While the use of `STP/LDP` instructions is generally encouraged, sequences of nonstack (non-`SP` base register) paired instructions that store-to-load forward may not allow certain hardware optimizations to engage, which may limit performance especially in tight loops. Use single-register store and loads in these cases.

5.6.12 Prefetching

For memory-intensive applications, prefetching data into the caches from main memory prior to use can result in dramatic performance increases. To accommodate this, the cores include mechanisms for both hardware-driven and software-driven prefetching. This section gives a brief description of how to avoid pitfalls that might cause significant performance problems.

5.6.12.1 Hardware Prefetcher

The hardware prefetcher is a unit attached to each core that analyzes the cache misses generated by that core in an effort to find predictable streams of misses. When its heuristic identifies a predictable stream, it attempts to launch main memory accesses in advance in order to avoid future cache misses to the same stream of references. In general, any large data structures accessed in a linear fashion should be successfully prefetched, and even somewhat irregular patterns are also often quite prefetchable. In general, it is successful on all but the most irregular and hard-to-predict scenarios. Nevertheless, the hardware prefetcher does have a few limitations, such as the maximum stride that can be detected successfully.

Benefits to relying on the hardware prefetcher:

- **No developer intervention:** This mechanism is automatic and does not require any code modifications.
- **Automatically tuned for each core:** The hardware prefetching aggressiveness is tuned for each core's performance characteristic and each chip's latencies, so that data arrives in the core just before it is used. In contrast, the count of instructions between software prefetches and the subsequent demand accesses must be manually tuned for each core to ensure that they are early enough, but not too early. Because the P cores usually require prefetching 4 to 30 lines ahead to allow the data to arrive in time (depending on the specific software algorithm), this can be quite difficult to tune manually. In contrast, the E cores usually require less aggressive prefetching. When the same code is executed on the E cores, the prefetcher's aggressiveness is retuned for those characteristics. Such adjustment is not possible utilizing software prefetches.
- **Reduced resource contention with loads and stores:** Software prefetch instructions occupy load units in the pipeline just like any other load. Reducing the number of

instructions that must go through those units can help avoid turning them into a critical bottleneck. In contrast, the hardware prefetcher usually only inserts prefetches into the load or store pipelines on spare idle cycles. Hardware prefetches may sometimes still compete for memory system resources with demand accesses, but the hardware prefetcher algorithm adjusts for that too. Software prefetches are *always* mandatory and must execute even in the face of heavy memory system contention. In contrast, hardware prefetches are always optional and can be dropped if necessary.

5.6.12.2 Software Prefetch Instructions

The cores support the `PRFM` and `PRFUM` software prefetch instructions, which differ only in their available selection of addressing modes. The former have the same addressing modes as `LDR` or `STR`, while the latter handles modes available to `LDUR` or `STUR`. These prefetch instructions include prefetch options that include operation type {prefetch load, prefetch store}, data destination {cache levels 1 or 2}, and temporal behavior {temporal, non-temporal}. When using software prefetch instructions, issue only 1 instruction per cache line, striding the address based on the particular cache's line size. The address specified within a particular cache line does not matter.

Table 5.35. Recommended Software Prefetch Instructions

Instruction	Operation	Destination	Temporal Behavior
<code>PRF(U)M PLDL1KEEP</code>	Load / Read	L1D Cache	Temporal
<code>PRF(U)M PLDL1STRM</code>			Non-Temporal
<code>PRF(U)M PLDL2KEEP</code>		Shared L2 Cache	Temporal
<code>PRF(U)M PLDL2STRM</code>			Non-Temporal
<code>PRF(U)M PSTL1KEEP</code>	Store / Write	L1D Cache	Temporal
<code>PRF(U)M PSTL1STRM</code>			Non-Temporal
<code>PRF(U)M PSTL2KEEP</code>		Shared L2 Cache	Temporal
<code>PRF(U)M PSTL2STRM</code>			Non-Temporal

Unlike normal memory accesses, these instructions cannot trigger exceptions (e.g., unmapped virtual to physical addresses, illegal page access request types, etc.). As a result, software prefetch instructions may safely fetch past the ends of arrays and access `NULL` pointers, without slowing down the system with spurious faults. Because prefetch instructions often access data a few iterations ahead of the actual demand accesses in the loop, preventing "bad" accesses would require extra range checks and branches that would complicate and slow the loop. Unfortunately, this feature also prevents prefetch instructions from triggering true page faults early; only actual demand misses can force the core to handle page faults.

When the hardware prefetcher can successfully predict memory access patterns, inclusion of software prefetch instructions may cause a decrease in performance because the software prefetch instructions may inhibit the abilities of the hardware prefetcher. The prefetch instructions don't trigger the hardware prefetcher themselves, so their use can mask any natural miss patterns in the code that would normally trigger the detection of hardware prefetchable patterns. In addition, the prefetch instructions tie up resources within the cores, which may increase competition for critical pipeline resources. In

particular, they occupy decode slots, take up space in the reorder buffer, and can compete with actual loads and stores for issue slots in the Load and Store Units.

Furthermore, you may find it difficult to properly place software prefetches into your code. To be fully effective, software prefetches must launch memory requests a hundred or more cycles ahead of the natural use in the code. Thus developers need to insert software prefetch instructions potentially hundreds of dynamic instructions ahead of the use. This distance is likely to increase on future faster processors.

Recommendation: Sparingly use software prefetch instructions.

[Magnitude: Medium | Applicability: Low] Consider adding software prefetch instructions to improve load and store latency, but only after analyzing data miss patterns and exploring algorithmic improvements to better facilitate hardware prefetching. Select the appropriate software prefetch instruction based on expected demand operation type and temporal locality.

Before inserting software prefetches, evaluate cache performance using the metrics listed in [Table 5.26: “Common Data Cache Metrics”](#). Identify loads and stores that commonly miss and analyze the access patterns. The follow scenarios may warrant use of software prefetching. After attempting insertion of software prefetches, it is best to analyze performance both with and without the software prefetches to see whether or not they are properly placed and verify that they provide an actual performance improvement.

- **Any access pattern:** The biggest advantage for software prefetches is that they can prefetch data using any memory access pattern, while the hardware prefetcher is limited to a finite number of repetitive patterns. Hence, prefetching in complex data structures such as trees is really only possible using software prefetching. That being said, when accessing these sorts of data structures it is often too difficult or sometimes even impossible to calculate the necessary addresses sufficiently early. For example, pointer-chasing delays usually cannot be avoided by prefetching, because the program doesn't know the addresses to prefetch in advance.
- **Multiple streams too close together:** The hardware prefetcher is designed to detect streams while accounting for out-of-order execution of accesses. When multiple separate streams are located close together, typically on the same page, and are accessed simultaneously in an interleaved manner, the prefetcher may sometimes identify them as a single irregular stream. The prefetcher may then initiate either erratic prefetching or, in the worst case, no prefetching at all. For example, significant performance degradation can occur when a single loop reads values from adjacent rows in 2D data buffers (matrices, photos, and video frames, for example). The full buffers in these scenarios are usually large enough to easily exceed the sizes of the caches, so prefetching is important for good performance. However, rows of these large buffers can still be short enough so that streams of accesses to two adjacent rows of the buffer at once can be misinterpreted as a single irregular stream. There are several ways to address this problem, so it is covered in more detail later in this section.
- **Large strides:** Software prefetching can also help if large strides are needed, typically those nearing the size of memory page or larger. Any stride larger than about a memory page may potentially be missed and tracked as separate streams, instead. For example, a loop accessing a large (i.e. thousands x thousands of elements) row-major array in a column-wise manner often reads the array using strides that greatly exceed the maximum stride that the hardware prefetcher can detect.

5.6.12.3 Prefetch Performance Tips

Although rare, the most difficult-to-debug problems with the hardware prefetcher tend to occur when software simultaneously accesses multiple streams that are too close together in memory. To avoid these problems, don't mix two or more simultaneous streams of cache misses within the same memory page. Consider these transformations to improve the ability of the hardware prefetcher to lock on to streams:

- **Avoid adjacent rows:** If possible, change the algorithm so that it does not need to access adjacent rows of a 2D buffer simultaneously. For example, consider an algorithm that operates row-by-row over a large 2D buffer. You might be tempted to unroll the outer loop such that the algorithm operates on two rows (m and $m+1$) simultaneously, exposing parallelism, like the following:

```
// Fused outer loop, loops over even/odd pairs of rows together
FOR (m = 0; m < M; m += 2) {
    // Inner loop over columns of the buffer
    // -- Cache misses on both rows m & m+1, interleaved
    FOR (n = 0; n < N; n++) {
        Compute row pair output elements (m,n) and (m+1,n)
        // -- Can interleave instructions from both rows together now
    }
}
```

Unfortunately, this loop accesses two rows of the 2D buffer simultaneously. If both rows are located in the same memory page, the hardware prefetcher may be unable to clearly identify the streams.

However, because the algorithm processes each row independently, pick two non-adjacent rows to process simultaneously. For example, instead of adjacent rows, interleave a row from the top half of the buffer (m) and a row from the bottom half ($m+(M/2)$):

```
// Fused outer loop, loops over pairs of rows together
FOR (m = 0; m < M/2; m++) {
    // Inner loop over columns of the buffer
    // -- Cache misses on both rows m & m+M/2, interleaved
    FOR (n = 0; n < N; n++) {
        Compute row pair output elements (m,n) and (m+(M/2),n)
        // -- Can interleave instructions from both rows together now
    }
}
```

Simple changes like this can ensure that any two (or more) rows being read simultaneously are always far apart instead of just a single row apart.

- **Rearrange access patterns:** The next possible solution is to rearrange the accesses to memory so the cache misses all occur in a single stream that the hardware prefetcher can easily understand. Instead of interleaving two streams in confusing patterns like $x, X+M, x+1, X+M+1, x+2, X+M+2, \dots$, for rows of length M , one can instead rearrange the references into streams like $x, x+1, x+2, \dots, x+M-1, X+M, X+M+1, X+M+2, \dots$. The simplest way to do this is to add a small "read ahead" loop that triggers the cache misses ahead of time for all rows or all but the last row. Here is a simple example for

when two rows are read together that triggers the misses to the first of the two rows early:

```
// Original outer loop, loops over even/odd pairs of rows
FOR (m = 0; m < M; m += 2) {
    // Added "read ahead" loop
    FOR (n = 0; n < N; n += L1_CACHE_LINE_SIZE) {
        Load at (m,n) to trigger cache misses from row m in correct order
    }
    // Original inner loop, loops over columns of the buffer
    // -- Originally missed on both rows m & m+1, now just misses on m+1
    FOR (n = 0; n < N; n++) {
        Compute row pair output elements (m,n) and (m+1,n)
    }
}
```

This small addition reorders all of the cache misses so they occur in the "correct" order and thereby generally present the hardware prefetcher with a single, simple stream of misses. Depending upon the nature of the inner loop, it may sometimes be better to load ahead for both rows m and $m+1$ in a read ahead loop like the following:

```
// Added "read ahead" loop
FOR (n = 0; n < 2*N; n += L1_CACHE_LINE_SIZE) {
    Load at (m,n) to trigger cache misses from row m and then row m+1
}
```

After the processor executes the "read ahead" loop, the computation loop won't generate any cache misses at all, because all of its data will have been fetched into the cache by the "read ahead" loop. Experiment with different solutions to see which performs better in a given situation.

- **Partial software prefetch:** Typically, read ahead loops use normal "demand" accesses which leverage all of the hardware prefetcher's advantages, such as having the prefetch aggressiveness automatically scaled properly for the underlying core. However, read ahead loops using demand accesses may stall the processor waiting for load data that will be immediately discarded. In select cases, use software prefetch instructions in the "read ahead" loop to avoid these stalls. In this case, the read ahead loop looks slightly different:

```
// Added "read ahead" loop
FOR (n = 0; n < N; n += L1_CACHE_LINE_SIZE) {
    PRFM at (m+2,n) to prefetch from row m+2
}
```

When coded with demand accesses, the read ahead loop accesses data used in the subsequent loop iteration. The hardware prefetcher identifies the pattern and prefetches out ahead, prefetching data for future iterations. However, software prefetches do not train the hardware prefetcher. A read ahead loop consisting of software prefetches must manually prefetch out ahead for future iterations. This can be seen in the above code example where the `PRFM` instruction accesses $(m+2, n)$.

This is because of a fundamental difference between hardware and software prefetching. The hardware prefetcher spots patterns in a stream of accesses and then automatically tries to run far enough ahead to avoid pipeline stalls. However, software

prefetch instructions only initiate prefetches at the time that they are executed, and therefore must appear in the code stream sufficiently ahead of the demand loads to cover the cache miss latency of the access, which may be hundreds of instructions. In this case, prefetch for the computation loop of the next outer loop iteration instead of the one that will execute immediately after this "read ahead" loop in order to provide enough lookahead.

Although using software prefetch to access the "even" (m) rows, this hybrid scheme continues to access the "odd" ($m+1$) rows using standard demand misses during the computation loop. As a result, the code is actually still reliant on the hardware prefetcher, but it now only has to deal with prefetching a single stream of references (for row $m+1$) during each inner loop and hence should work much better.

- **All software prefetch:** The final step is to just software prefetch everything and remove the hardware prefetcher from the equation altogether. This can be accomplished with a straightforward variation on the last read ahead loop:

```
// Added "read ahead" loop
FOR (n = 0; n < N; n += L1_CACHE_LINE_SIZE) {
    PRFM at (m+2,n) to prefetch from row m+2
    PRFM at (m+3,n) to prefetch from row m+3
}
```

Much like the second read ahead loop variation from the "rearrange access patterns" case, this runs ahead and fetches two rows at once. Unlike that scenario, however, one can fetch the two rows using an interleaved access pattern. Because the code now bypasses the hardware prefetcher entirely, there is no reason to keep the accesses laid out in a strictly linear fashion. (It may still be a good idea to use the linear access pattern from the "rearrange access patterns" case, because it may improve DRAM performance. However, it is not necessary to guide prefetching, so this alternate code works too.) In fact, it may even be desirable to fuse the "read ahead" loop and the computation loop together so that the software prefetches are fed into the Load and Store Units gradually over the course of the inner loop instead of in a big chunk at between inner loops, more like the following:

```
// Original outer loop, loops over even/odd pairs of rows
FOR (m = 0; m < M; m += 2) {
    // Fused read ahead+inner loops, loops over columns of the buffer // --
    Compute rows m & m+1 + prefetch rows m+2 & m+3
    FOR (ns = 0; ns < N; ns += L1_CACHE_LINE_SIZE) {
        // Read ahead section
        PRFM at (m+2,ns) to prefetch from row m+2
        PRFM at (m+3,ns) to prefetch from row m+3
        // Compute section
        FOR (n = ns; n < ns + L1_CACHE_LINE_SIZE; n++) {
            Compute row pair output elements (m,n) and (m+1,n)
        }
    }
}
```

This kind of pattern can be continued to prefetch even farther ahead, but be careful not to prefetch ahead so far that the data starts falling out of the L1D Cache before it can be used by the compute loop. If L1D Cache capacity is a problem, then it would probably

be better to go back to one of the solutions that uses the hardware prefetcher. Finally, note that it is not possible to fuse the read ahead loop in this manner unless the read ahead loop is only using software prefetches, because otherwise the code will still have multiple load streams and lead to prefetcher confusion.

Lastly, neither software nor hardware prefetching is likely to improve the performance of pointer chasing code. That is, it is hard to predict the need for cache lines far enough in advance for algorithms that predominantly hop from one linked node to the next with minimal additional computation. Many of these data structures feature dynamically allocated nodes, for which the allocation and link ordering form no discernible memory address pattern. Performance improvement may still be possible through intelligent ordering of structure fields to minimize the number of cache lines that are needed for typical access to a node. Specifically, put the key node data items and "next" pointer into a single cache line. See the discussion on "hot/cold" fields and variables in [Section 5.6.3, "L1 Data \(L1D\) Cache"](#).

Recommendation: Recode algorithms to simplify access patterns to aid the hardware prefetcher.

[Magnitude: Medium | Applicability: Medium] Although software prefetch instructions may be able to prefetch complex patterns, it is often difficult to insert them far enough ahead, but not too far ahead, of the demand use. The hardware prefetcher, however, dynamically adjusts according to the circumstances. Before attempting to insert software prefetches, identify regions of high cache miss behavior, and explore pattern simplifications that may allow the hardware prefetcher to identify and prefetch the stream.



Chapter 6. SME Engine Microarchitecture Optimization

[Streaming Scalar Floating Point instructions \(SSFP\)](#), [Streaming Scalable Vector Extension \(SSVE\)](#), and [Scalable Matrix Extension \(SME\) instructions](#) are comingled with other core instructions, presenting a unified programming model. However, these instructions are largely executed in a separate computational block called the *SME engine*. Each cluster contains one SME engine that supports multithreaded execution on behalf of all cores in the cluster, and it maintains a separate copy of the register state associated with each of those cores. When the OS context switches the register state of one thread for another in a core, it also context switches the corresponding SSVE/SME register state in the SME engine.

Like cores, Apple silicon chips feature both performance (P) and efficiency (E) versions of the SME engine. Unless otherwise noted, details presented in this chapter apply to both versions.

The microarchitecture of the SME engine is designed to enable algorithms to achieve a high throughput where latency is a secondary concern. For instance, the microarchitecture does not support speculative execution, thus it only issues instructions when they are guaranteed to be executed in the program flow. The SME engine does support out-of-order execution, but with the goal of keeping the execution units busy (covering memory access latency and some SME engine pipeline conflicts) rather than reducing latency. With this microarchitecture, SME engine instructions *trail* execution of their comingled core instructions by dozens, hundreds, possibly even thousands of cycles.

Per the convention set forth in the [ISA chapter](#), while all capabilities are part of Arm FEAT_SME and subsequent extensions, SME refers instructions that interact with the 2D ZA storage array, and SSVE and SSFP refer to the instructions that interact with the Z vector and P predicate registers.

The SME engine contains several main components that work alongside several enhancements in the core, as shown in [Figure 6.1: “Abstract View of the SME Engine and Connectivity”](#).

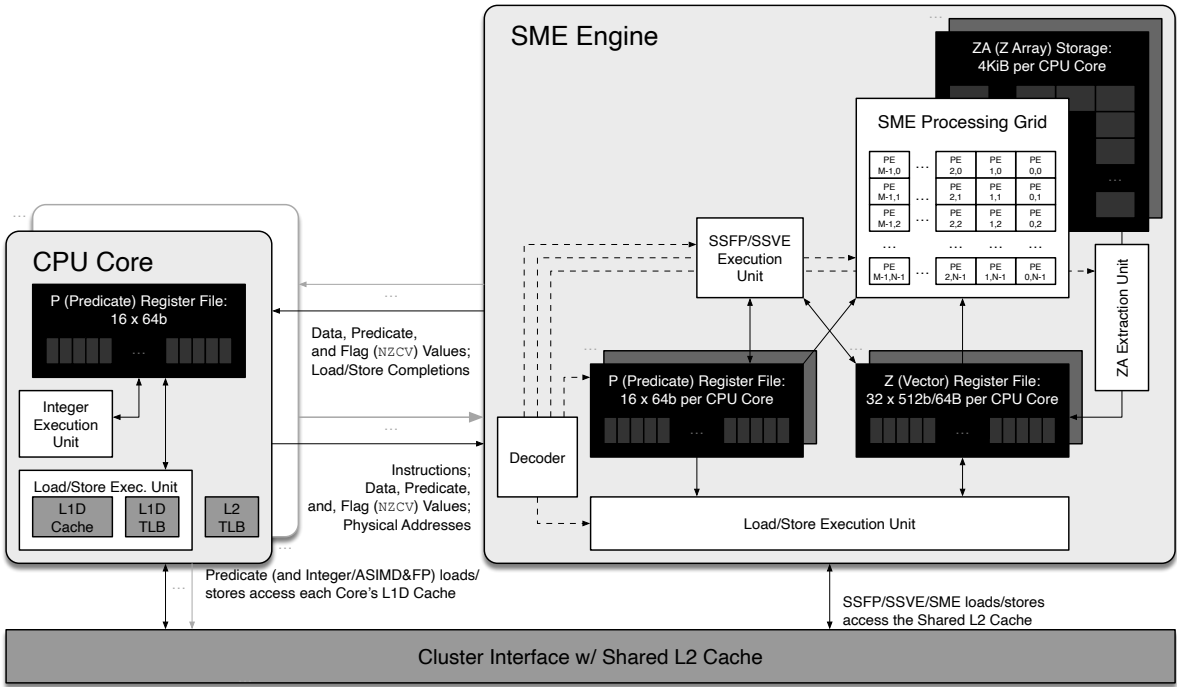
- **Streaming Scalar Floating Point instructions (SSFP)** and **Streaming Scalable Vector Extension (SSVE)** computation instructions are largely executed in the SSVE Execution Unit based on data stored in the Z vector register file, activating elements based on predicates stored in the P register file:
 - **32 x 512b (Z) vector registers** hold SIMD values that are similar to but longer than ASIMD vectors.
 - **16 x 64b (P) predicate registers** hold bits for activating individual elements within a Z vector. Note that the microarchitecture keeps a copy in both the core and the SME engine to reduce local access latency (so the core has fast access for generating and manipulating predicates, and the SME engine has fast access for governing execution).

- **SSVE Vector Execution Unit** performs the desired operation on data associated with the Z vector registers and in some cases the P predicate registers. This pipeline operates similarly to a single, wide Advanced SIMD and FP pipeline. The SSVE instruction set is more sophisticated than Advanced SIMD, working with longer, 64-byte/512-bit vectors, and supporting features like predication of vector lanes that are not needed by a particular instruction. Similar to the core's Instruction Processing component, the unit contains a Map component and schedule component (not shown) for implementing out-of-order execution, although all instructions are nonspeculative. This unit also executes SSFP. Lastly, the unit sends data (element value), predicate, and flag results generated in the SME engine back to the core, when needed.
- **Core Integer Unit enhancements** performs the desired operation on data associated with the P predicate register. The existing core Integer Execution Unit processes most predicate manipulation instructions and provides general purpose register contents for SSFP/SSVE/SME instructions. The unit prepares data and predicate values generated by the core to be sent to the SME engine. Instructions that execute in the core Integer Execution Unit process like all core integer instructions and can execute speculatively and out of order.
- **Scalable Matrix Extension (SME)** computation instructions are largely executed in the SME Processing Grid based on data stored in the Z vector register file and the ZA storage:
 - **4KiB ZA storage** holds data elements that can be accessed in two configurations, first as 2D tiles where the number of tiles depends on the data element size and second as 64 x 512b 1D vectors where the number of elements depends on vector length. Some instructions can concatenate 512b vectors to operate as 1024b or 2048b vectors (referred to as *vector groups*: *vgx2* and *vgx4*), and some instructions can widen input elements to generate 2x (double) or 4x (quad) output vectors.
 - **SME Processing Grid** performs the desired operation on data associated with the ZA storage. This two-dimensional SME processing grid accelerates calculations using many parallel processing elements. It can operate on 2D tiles or 1D vectors. The grid does not support fine-grained renaming, and generally features instructions that touch large portions of the ZA storage. Although the Processing Grid contains a simple schedule component (not shown) for scheduling μ ops that are ready, there is only modest opportunity for out-of-order execution due to the lack of renaming and limited number of tiles and registers. Like for SSFP and SSVE, all instructions are nonspeculative.
 - Note that the microarchitecture co-locates the cells of the ZA storage with the Processing Grid elements that operate on them for high bandwidth and low latency local access.
 - **ZA Extraction Unit** moves data from the ZA storage into Z vector registers. This pipeline extracts rows or columns from the ZA storage in parallel with grid compute instructions. These, too, can execute out of order and are all nonspeculative.
- Memory instructions for SSFP/SSVE/SME execute partly in the core and partly in the SME engine:
 - **Core Load/Store Execution Unit** performs the memory access translation, ordering, and checking operations on behalf of SME engine memory instructions. The

existing core Load/Store Execution Unit performs virtual-to-physical memory address translations on behalf of SSFP/SSVE/SME loads and stores using the existing L1D and L2 Translation Lookaside Buffers (TLBs), and maintains memory ordering consistency between core and SME engine instructions. The core, however, does not access the caches for these instructions, but sends physical addresses to the SME engine for SSFP/SSVE/SME load/store instructions. This unit is responsible for loading and storing predicate values, and for preparing and shipping each instruction destined for the SME engine. All of these operations process like all other core load and store instructions, executing speculatively and out of order inside the unit, with stores and engine instructions sent out obeying ordering rules.

- **SME Engine Load/Store Execution Unit** performs the reads and writes directly from and to the Shared L2 Cache within this SME engine's cluster. Load latencies are longer than in the core, but direct Shared L2 Cache access allows the unit to work with much larger data sets before needing cache-centric software optimizations such as blocking. The unit optimizes streaming accesses to memory by reorganizing access patterns to use full cache lines where possible. Physical addresses are provided to the unit by the core. Accesses to the Shared L2 Cache can execute out of order but nonspeculatively. Lastly, the unit sends load and store completion indications back to the core for memory ordering enforcement.

Figure 6.1. Abstract View of the SME Engine and Connectivity



The core and SME engine interact in the following ways:

1. The core processes SSFP/SSVE/SME instructions (intermixed with core instructions) as follows:
 - a. Predicate manipulation instructions are executed in the core's Integer Execution Unit. Predicate register updates are processed through the core's Load/Store Execution Unit to be shipped to the SME engine. Similarly, any core integer register

values needed by SME engine instructions are read through the core's Integer Execution Unit and processed through the core's Load/Store Execution Unit to be shipped to the SME engine.

- b. SME engine load/store instructions process through the core's Load/Store Execution Unit to perform address translations, permission checks, and memory hazard tracking between core and SME engine accesses.
 - c. All instructions, data values, and addresses destined for the SME engine are each processed through the core's Load/Store Execution Unit to be shipped to the SME engine. When instructions are determined to be nonspeculative, they are sent *in program order* to the SME engine. These instructions are not stored to memory by this unit, but use some of the same datapaths as stores to reach the SME engine, and must be properly ordered with stores. Transit to the SME engine takes on the order of 20 cycles.
2. The SME engine processes SSFP/SSVE/SME instructions as follows:
- a. SME engine instructions are decoded into one or more SME engine μ ops and written into μ op schedulers associated with each SME engine unit (not shown). Incoming predicate values are updated in the SME engine's copy of the predicate register file, and incoming data values and addresses are coupled with their instructions.
 - b. The μ op stream typically loads a large volume of data from the Shared L2 cache and computes/accumulates intermediate results into the ZA storage. As SME engine memory instructions complete, the core is notified in order to release any core memory instructions blocked due to memory hazard tracking. Transit to the core takes on the order of 20 cycles, but see the important note below on the effects of nonspeculative execution.
 - c. When the compute loop is complete and the final results are ready in the ZA storage, the data is extracted from the ZA storage and typically stored into the Shared L2 Cache. Data, predicate, and nzcv flag values are sent back to the core, according to the instruction stream. Transit to the core takes on the order of 20 cycles, but see the important note below on the effects of nonspeculative execution.

Recommendation: Avoid execution sequences where dependant data traverses back and forth between the core and engine.

[Magnitude: High | Applicability: High] Because of nonspeculative execution, communication latencies, and in some cases long memory and computation latencies, SME engine instructions *trail* execution in the core by dozens to thousands of cycles. Any core compute instructions that consume data produced by the SME engine may have to wait an indeterminate (but long) amount of time for the data to arrive.

Lastly, an entire cluster runs at the same frequency, thus core, Shared L2 Cache, and SME engine clock frequencies are the same across a cluster. When any core in a cluster is in Streaming SVE or ZA mode, the entire cluster may run at slightly lower clock frequency.

Note: The details covered in this chapter and [Appendix A, *Instruction Latency and Bandwidth*](#) represent the most important information to consider when optimizing software for the SME engine microarchitecture. The CPUs employ additional techniques to improve performance beyond what is described here. These may change from family to family, and it may be difficult or counterproductive to try to transform software to match the heuristics. The guidelines in this chapter represent the optimization opportunities with the most significant impact across various SME engine configurations, and across chip series and families.

6.1 Overall Optimization Guidance

When optimizing code running on the SME engine, optimize the code with the following in mind:

- [Design algorithms to use SME instructions](#) targeting ZA storage and the SME Processing Grid. Use SSVE instructions targeting Z registers and the SSVE Execution Unit in a supporting role. The SME Processing Grid offers considerably higher operation bandwidth. Algorithms that can't utilize the SME engine are typically best executed on the core in the Advanced SIMD and FP execution units.
- [Avoid memory hazards between the core and the SME engine](#) by isolating SME engine data structures to separate 4KiB sections of memory. Avoid accessing the stack from the SME engine's Load/Store Execution Unit, that is, loading from or storing to the stack using ZA, Z, B, H, S, D, Q registers when in Streaming SVE mode (SM=1). Wherever possible, dynamically allocate memory for data structures used by the SME engine, and do not access it from the core beyond initialization and consumption of final results.
- [Avoid data critical paths](#) through registers or memory that traverse back and forth between the core and the SME engine.
- [Use consecutive memory accesses](#) as much as possible to use all of the banks in the Shared L2 Cache in order to maximize parallelism and hardware prefetching. For cases where strided accesses cannot be avoided, such as for accesses down a column of a matrix, consider padding the length of the matrix's rows to distribute the column across the banks. Pad for aligned stores over aligned loads, when necessary. Some padding experimentation may be required.
- [Develop the algorithm kernel to cover operation accumulator latencies](#) to expose sufficient parallelism thereby maximizing SME Processing Grid throughput. [Distribute data across as much of the ZA storage](#) as possible (vectors or tiles) to leverage [PE row parallelism](#). Nominal accumulation latencies in the SME Processing Grid are 4 cycles, while reading out vectors from the ZA storage are 15 cycles.
- [Maximize use of the Z vector registers for staging data](#) in and out of the Processing Grid, reducing pressure on the out-of-order schedulers.
- [Organize code to maximize instruction operation fusion](#) to reduce operation latency and bandwidth.
- [Pack SME engine instructions next to each other](#) where possible and avoid fine-grained interleaving of SME engine instructions with core memory instructions to optimize delivery of instructions and data to the SME engine.

6.2 SSVE Vector Execution Unit Optimization

The SSVE Vector Execution Unit is similar to the vector pipeline for Advanced SIMD and FP. A maximum of one vector instruction can be issued to the SSVE Vector Execution Unit per cycle in the SME engine. The SSVE Vector Execution Unit offers traditional register renaming to avoid various data dependency hazards and has a scheduler that can issue ready μ ops.

Broadly, this unit is designed to *support* long vector and matrix operations performed on ZA storage *in the SME Processing Grid*.

Although the Z vectors (64B) are longer than ASIMD V vectors (16B), the overall operation bandwidth is similar to ASIMD in the core due to the single SME engine issue port compared with the multiple issue ports of ASIMD (e.g., four on M4 P core). Some SSVE instructions can operate on multi-vectors of up to four Z vectors, however many of these require an issue cycle per individual Z vector. Lastly, instructions that perform multiplication and those that perform floating point addition require multiple issue cycles even when processing a single 64B vector, such that their latency is longer than ASIMD.

SSFP executes in this unit as well. Although the latencies are similar to ASIMD, the overall bandwidth is much smaller (e.g., $\frac{1}{4}$ bandwidth of floating point on M4 P core). When any significant scalar floating point is required, transition out of Streaming SVE mode and execute the operations in the core.

See also [Section 6.8, "SSVE/SME Instruction Operation Fusion"](#).

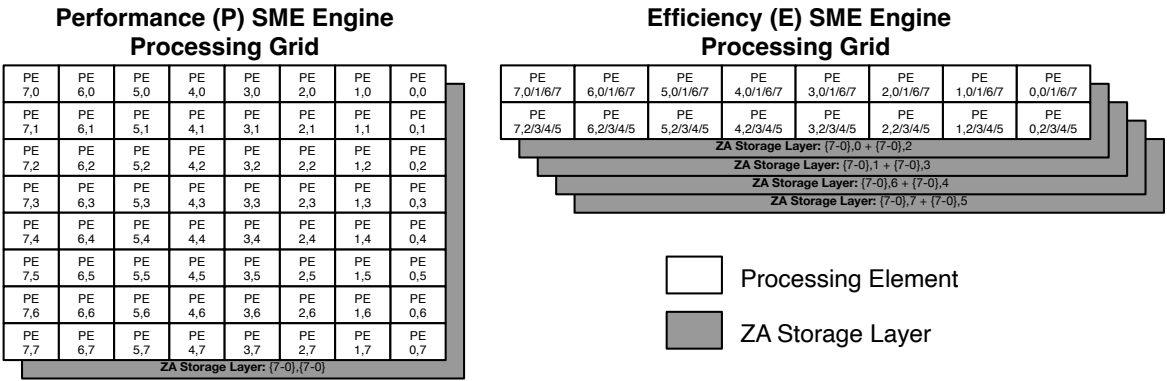
Recommendation: Use SSVE in a supporting role to enable high throughput SME grid computation.

[Magnitude: High | Applicability: High] SSVE offers wide 64B vectors. While the ISA includes instructions that can operate on multi-vectors, the throughput is often only one 64B vector per cycle. Use SSVE to enable SME, which offers higher parallelism.

6.3 SME Processing Grid Optimization

For parallelism purposes, the ZA storage (64 rows of 64 bytes) is divided into cells of 8 rows of 8 bytes, as described in [Section 4.6.11.1, "SME Outer Product \(MOP\) Tile Instructions"](#). Compute instructions generally confine operations to elements within the cell, and the compute block that operates on that cell is called a Processing Element (PE). To achieve high throughput, each performance (P) SME engine implements 1 PE per cell for a total of 64 PEs, as shown in [Figure 6.2: "Performance and Efficiency SME Engine Processing Element Grids"](#). To improve efficiency, each efficiency (E) SME engine *folds* the ZA storage such that each PE operates on four cells for a total of 16 PEs. The ISA instructions perform the exact same function on P and E SME engines, and thus the same code can execute on P or E SME engines. However, the instruction latencies may be longer on the E SME engine, as a PE may need to process through four cell's worth of data instead of one, therefore also reducing instruction throughput.

Figure 6.2. Performance and Efficiency SME Engine Processing Element Grids



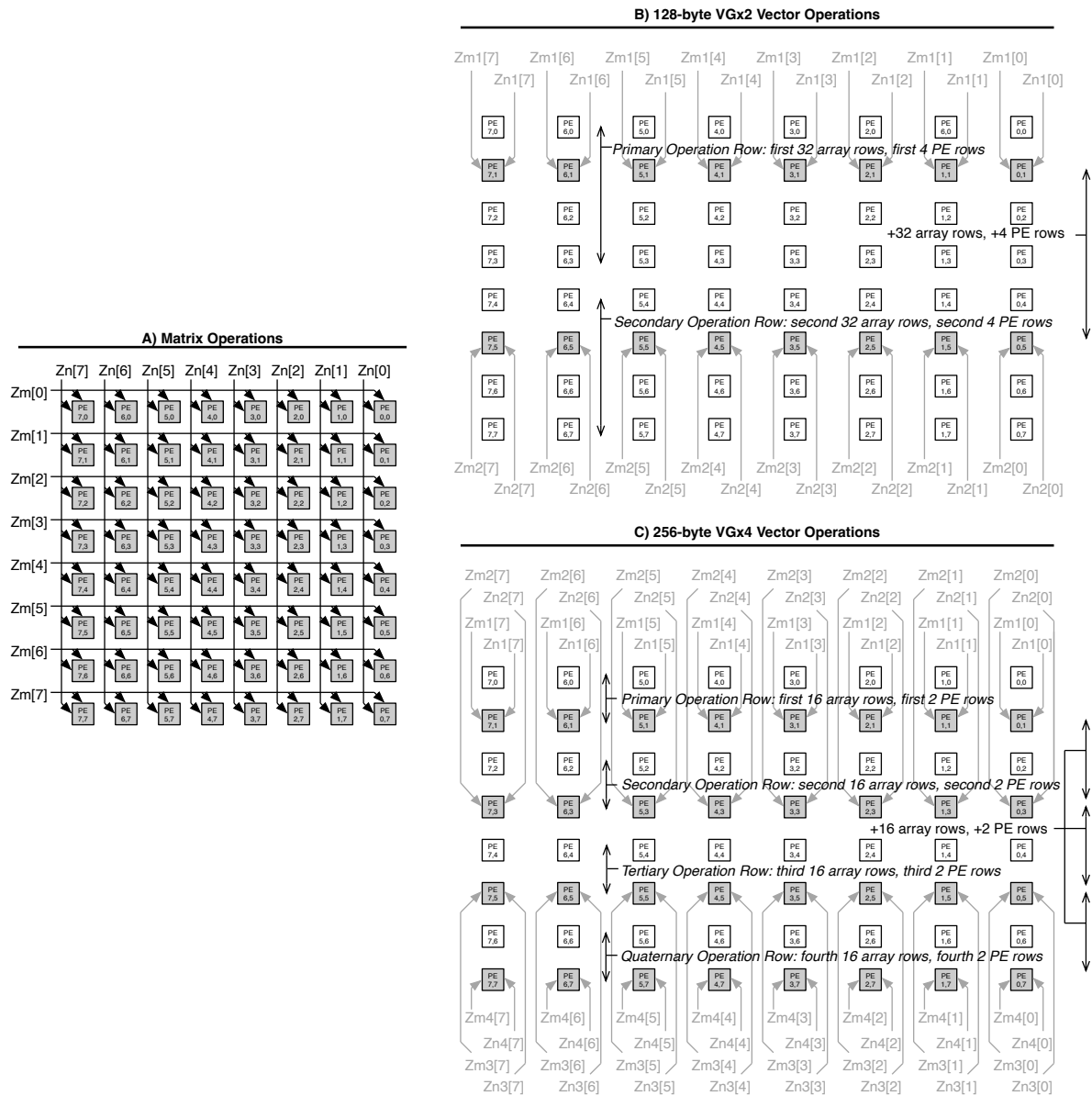
Recommendation: Optimize code to achieve high throughput on the P SME engine.

[Magnitude: High | Applicability: High] Code that executes with high throughput on the P SME engine also runs well on the E SME engine, albeit at a lower rate.

A maximum of one μop can be issued to the Processing Grid per cycle. The Processing Grid offers coarse-grained "register" renaming support of the ZA storage but only on a *whole ZA storage* basis. A `ZERO { ZA }` instruction can rename the entire array to a different portion of internal storage. This allows the Processing Grid scheduler to overlap execution of the tail of the algorithm prior to the `ZERO` and the start of the algorithm after the `ZERO`. The scheduler can track dependencies between μops on a row-by-row basis. μops operating on different rows can be reordered with respect to one another, but all operations on a particular row occur in program order, including writes after reads.

Tile operations such as `MOB` essentially feed one vector on the x-axis of the Grid and the other vector on the y-axis, as shown in (A) of [Figure 6.3: "Full-Tile and Single-Vector Group SME Operations in the P SME Engine Processing Grid"](#), and generally use the full grid of PEs. Note that the indices shown in the figure are in terms of 8-row x 8B cell size, and do not depend on element size. (B) and (C) show two and four single-vector group (VGx2 and VGx4) μop execution such as `MLA`. As described in [Section 4.1.3, "SME ZA Storage"](#), when using ZA vectors, two or four rows are concatenated to form long vectors. ZA storage is divided into two or four sections, with the vector using two or four rows at the same offset in each half or quarter, thus spreading out the operation to independent PEs. ZA storage in the E SME engine is folded to maximize throughput of two and four single-vector group instructions.

Figure 6.3. Full-Tile and Single-Vector Group SME Operations in the P SME Engine Processing Grid



Although the PEs are pipelined, some μ ops that execute in the Processing Grid require the PEs for more than one cycle. These come in two forms:

- **Scheduler multi-issue:** These μ ops are issued across multiple cycles from the scheduler. Although issuing, no other μ ops are able to issue into the Processing Grid. This is common in the E SME engine to handle the folded ZA storage.
- **PE multi-issue:** These μ ops require one cycle of issue from the scheduler, but require additional cycles of issue internal to the row of PEs. During those additional cycles, other μ ops can issue from the scheduler, but only to other rows of PEs in the Processing Grid. For instance, some μ ops require one issue from the scheduler but two internal issues to the row of PEs. In that second cycle, the row(s) of PEs are still busy and cannot accept a new μ op, but the scheduler can issue a μ op to different row(s) of PEs. Note that the

latency of the μop may be longer, e.g., eight cycles. Because the PEs are pipelined, other μops can be issued to the same PEs after those first two cycles. It is difficult to code a software loop to follow a specific schedule due to out-of-order execution. Rather, ensure there is enough PE row parallelism by spreading the ZA vectors across ZA storage, allowing the out-of-order execution engine to fill the issue slots.

These requirements are annotated in the latency and issue slot tables in [Section A.6, “SME Processing Grid”](#).

To achieve maximum throughput in the Processing Grid, a new μop must be ready to issue each cycle (or, rather, each cycle that the scheduler is able to issue). Individual μops often have an accumulator latency of four cycles (some eight cycles), and the higher throughput P SME engine can generally issue one μop every cycle. For instructions that only require one μop , the code stream should generally be unrolled to expose four independent chains of execution. The target unroll amount varies by accumulator latency, by the required number of issue slots mentioned above, and by the number of μops required to implement the instruction. In general, use as much of the ZA storage (tiles or vectors) through unrolling as reasonable in order to expose sufficient parallelism.

See also [Section 6.8, “SSVE/SME Instruction Operation Fusion”](#).

Recommendation: Target high throughput by exposing sufficient parallelism through use of the entire ZA storage.

[Magnitude: High | Applicability: High] To ensure the Processing Grid remains busy, unroll loops creating sufficient parallelism to cover the (typical) four-cycle latency of computation instructions. Use as many tiles or vectors in ZA storage as possible to expose the parallelism.

6.4 ZA Extraction Unit Optimization

The ZA Extraction unit pulls data from the ZA storage in parallel with independent instructions executing in the Processing Grid. This pipeline is key to maintaining high computation throughput so the Processing Grid does not have to stall compute when reading out of vectors from the ZA storage. This unit is used both when moving vectors from ZA storage to the Z vector register file, as well as for the first stage μop for instructions that store ZA storage contents to memory. In these latter cases, the ZA Extraction Unit places the ZA storage data into temporary storage alongside the Z vector register file and relies on Z vector store μops to do the write to memory. Extractions are performed at 64B per cycle. Although extractions can be reordered with respect to operations on other rows, the extraction must be issued before any subsequent write to the same row can issue. Both 128B or 256B `mov` instructions do not reduce latency compared to a set of 64B `mov` instructions, but the wider instructions reduce instruction and μop counts that have benefits elsewhere in the microarchitecture.

Not all instructions that move data from the ZA storage to Z vectors can execute on the ZA Extraction Pipeline. Eligible instructions include `movas` that do not have a governing predicate and do not require a conversion or other computation. Those that cannot use the Extraction Unit must execute on Processing Grid itself, which includes most single-vector extractions because these instructions typically have a governing predicate.

Only a few instructions store data directly from the grid into memory. The first instruction below (and similar instructions with other element sizes) cannot use the ZA Extraction Unit

due to the governing predicate, while the second can. In general, however, move ZA data through Z vector registers using multi-vector `MOVAS` and then store it from the Z vector registers to benefit from this unit. See also [Section 6.8, “SSVE/SME Instruction Operation Fusion”](#).

```
ST1D { <ZAt><HV>.D[<Ws>, <offs>] }, <Pg>, [<Xn|SP>{, <Xm>, LSL #3}]
STR ZA[<Wv>, <offs>], [<Xn|SP>{, #<offs>, MUL VL}]
// Grid extraction followed by a store
```

Recommendation: Use instructions that execute in the ZA Extraction Unit to extract data from ZA storage in parallel with other Processing Grid instructions.

[Magnitude: High | Applicability: High] The ZA Extraction Unit provides an additional pipeline for accessing ZA storage. Use this where possible to increase parallelism. However, prefer operation fusion opportunities, even if they execute in the Processing Grid.

6.5 SME Engine Load/Store Execution Unit Optimization

The SME engine Load/Store Execution Unit accesses memory data for both Z vectors as well as for ZA tiles and ZA vectors. The unit is connected directly to the Shared L2 Cache to benefit from its capacity and bandwidth. Virtual to physical address translations are performed in the core and use the core's TLBs. All memory-related exceptions are detected and signalled in the core before SSFP/SSVE/SME instructions are sent to the SME engine. And similarly, the core tracks and enforces all memory dependence conflicts between the core and the SME engine. (See [Section 6.7, “Core Load/Store Execution Unit Optimization”](#).) Like all other SSFP/SSVE/SME instructions, load operations executed in this unit are not performed speculatively (and store operations in general are never performed speculatively). See [Section 6.9, “Memory Access Optimizations”](#) for strategies for managing the data working set size.

The SME engine cracks each load instruction into one μop per accessed 128B cache line, creating a 64B μop if the access is contained within either the low or high 64B half of that cache line, or a 128B μop if the access requires bytes from both halves. For example, consider a 128B load that starts at offset byte 48 of a 128B cache line A. This instruction requires two load μops : a 128B access to cache line A (both low and high portions) to retrieve the first 80 bytes, followed by a 64B access to the low portion of cache line A+1 for the remaining 48 bytes. Then, the SME engine attempts to *merge* $\mu\text{ops} across instructions to the same cache line to both reduce μop count and accesses to the Shared L2 Cache. [Section 6.5.1, “Access Merging”](#) describes this process in more detail. Similarly, the SME engine cracks each store instruction into one μop per 64B cache half-line and attempts similar μop merging.$

The SME engine contains a scheduler that can issue up to two ready (merged) μops out of order to the SME engine Load/Store Execution Unit. For each accessed cache line, the scheduler enforces program order, so that younger loads always see the effect of older stores. The Shared L2 Cache can send up to 256B of data from the cache to the Z vector register file on the P SME engine and 128B of data on the E SME engine each cycle. The Shared L2 Cache can receive up to 64B of store data per cycle. Store (merged) μops that write only portions of a 64B cache half-line require two operations to perform

a read-modify-write of the cache line in order to preserve bytes that are not written to by the store, and may additionally suffer from limited internal read-modify-write tracking resources. For algorithms that heavily use predicated stores, when possible, merge data in registers to reduce the number of stores that require read-modify-write.

Strided accesses are more likely to cause Shared L2 Cache bank imbalance, as described in [Section 5.6.4, "Shared L2 Cache"](#) because there is no L1D cache to buffer against these patterns. Although strided accesses cannot always be avoided, reducing them is important for performance compatibility because the address banking hash differs between chips and families. For matrices that require reading down a column, the rows can be padded such that their length is not a multiple of 256B, which should help avoid bank conflicts in most chip families, at the expense of increased memory footprint.

Padding rows to $(\text{multiples of } 256B) + 128B$ in length may alleviate both banking and unaligned access concerns but with greater impact on memory footprint. However, if possible, restructure the matrix to enable sequential accesses. Some experimentation with alignments may be required.

Recommendation: Experiment with alignment and padding to ensure maximum SME engine Load/Store Execution Unit throughput.

[Magnitude: Medium | Applicability: High] The engine offers high bandwidth to the Shared L2 Cache, and can often perform unaligned but contiguous accesses without additional overhead. However, avoid unaligned stores to eliminate read-modify-write sequences, and use selective padding to avoid Shared L2 Cache bank imbalance.

Most load and store instructions interact with the Z vector registers. There are only a few load instructions that directly write ZA storage. For example:

```
LD1Q { <ZAt><HV>.Q[<Ws>, <offs>] }, <Pg>/Z, [<Xn|SP>{, <Xm>, LSL #4}]
LDR ZA[<Wv>, <offs>], [<Xn|SP>{, #<offs>, MUL VL}]
// Load followed by grid insertion
```

Load instructions that move data into the ZA storage require a two- μ op sequence, a first Z vector load μ op that writes the data into temporary storage near the Z vector register file, and a second grid-issued insertion μ op that writes from that temporary storage into the ZA storage. Note that the grid-issued μ op prevents other Processing Grid operations from being issued (started) in the same cycle. However, in ZA vector code algorithms, this limitation can be largely overcome in the P SME engine by arranging the instructions so that ZA storage insertion operations access unused Processing Element rows of the grid where computation is not occurring (see discussion on "PE multi-issue" μ ops in [Section 6.3, "SME Processing Grid Optimization"](#)).

Similarly, there are only a few store instructions that directly read ZA storage.

```
ST1D { <ZAt><HV>.D[<Ws>, <offs>] }, <Pg>, [<Xn|SP>{, <Xm>, LSL #3}]
STR ZA[<Wv>, <offs>], [<Xn|SP>{, #<offs>, MUL VL}]
// Grid Extraction followed by a Store
```

These instructions require a Processing Grid μ op (for ST1D) or Extraction Unit μ op (for STR) (as noted in [Section 6.4, "ZA Extraction Unit Optimization"](#)) that write into temporary

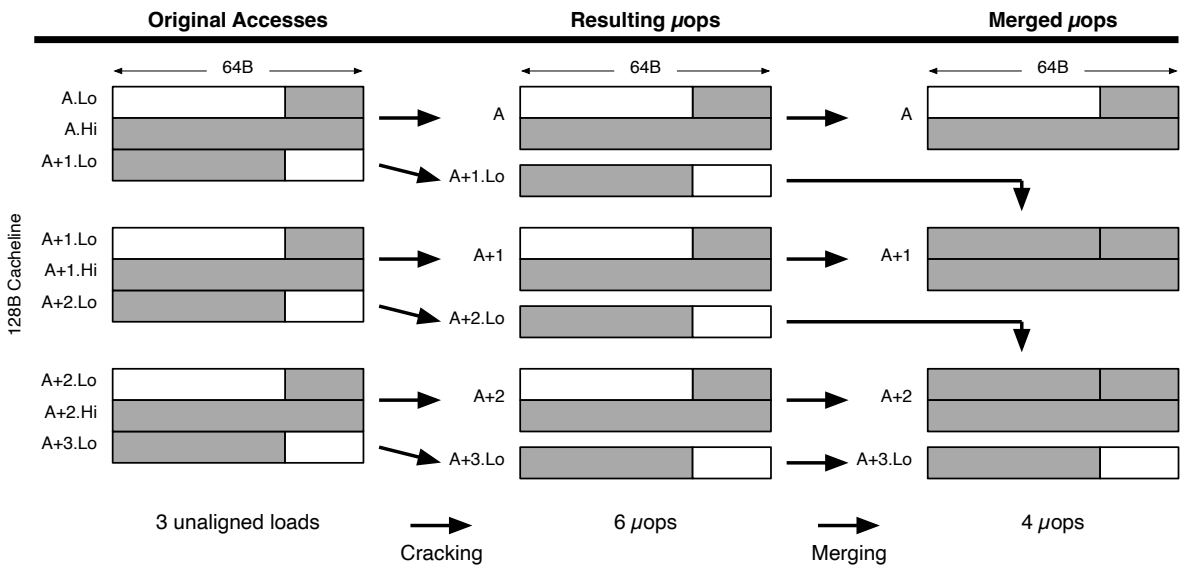
storage near the Z vector register file, as well as a store μop that reads from the temporary store and writes to memory.

Lastly, note that instructions that load and store predicate values are performed in the core.

6.5.1 Access Merging

Unaligned streaming accesses across 128B cache line boundaries are inevitable in some algorithms. Consider the example in [Figure 6.4: "Access Merging"](#) which depicts three sequential 128B load instructions that start at offset byte 48 of a 128B cache line. The first instruction requires two load μops: a 128B access to cache line A (both low and high portions) to retrieve the first 80 bytes, followed by a 64B access to the low portion of cache line A+1 for the remaining 48 bytes. The second instruction also requires two μops, a 128B access to cache line A+1 (both low and high portions) to retrieve the first 80 bytes followed by a 64B access to the low portion of cache line A+2 for the remaining 48 bytes. Note that all the bytes from cache line A+1 were consumed, but over two μops, wasting cache access bandwidth. The third instruction continues this pattern. The SME engine attempts to *merge* μops, such as the access to A+1.Lo with the subsequent access to A+1, into a single 128B μop that fulfills both requests and only require one issue into the SME engine Load/Store Execution Unit.

Figure 6.4. Access Merging



Overall, the mechanism attempts to merge up to three load μops to make complete 128B cache line requests, and up to two store μops to make complete 64B cache half-line requests. The two (or three) μops do not necessarily need to be from back-to-back instructions in the SME engine instruction stream, but they must be available in the SME engine when the first is ready to issue, ideally no more than two or three load/store instructions apart. For loads, this mechanism specifically operates on two or three vector μops that each access a 64B cache half-line or more of memory that together completely access the 128 bytes of the cache line. Similarly for stores, it operates on two vector μops that each access a 64B cache half-line that together completely access the 64

bytes of the cache half-line. Instructions such as SSFP loads and stores, broadcasting or replicating SSVE loads, SSVE loads and stores where the memory element size is smaller than the register element size, etc., are not eligible for merging. Store merging is particularly important for creating whole 64B cache half-line accesses that avoid read-modify-write penalties.

As mentioned, the SME engine can only merge if the second μop arrives before the first is issued. In particular, group unaligned stores together as much as possible in the code sequence to provide the best opportunity for merging. Construct data structures and algorithms to avoid strided stores (or groups of stores) that result in partial cache line writes. If strided stores are unavoidable, especially for temporary matrices where storage size is less of a concern, consider padding matrix rows to align the accesses and thus avoid the added costs of read-modify-write.

6.5.2 Optimizing for Long Memory μ op Latencies

Each of the SME engine execution units features a μ op scheduler capable of buffering μ ops and issuing them when they are ready. Even with Z vector register renaming, the long load latency (20 or more cycles for a Shared L2 Cache hit, depending on level of traffic) leads to significant capacity pressure on those schedulers.

Consider a data dependent sequence that loads data into Z registers, computes results in the grid into ZA storage, extracts data from ZA storage into Z registers, and stores that data to memory. Even if the load issues immediately, the grid compute operation waits in the SME Processing Grid scheduler for about 20 cycles. The extraction operation waits in its scheduler for the latency of the load and the compute, while the store operation waits in its scheduler for all of them to complete. (Note that the latencies documented in [Section A.6, “SME Processing Grid”](#) for grid computation operations focus on the latency through the accumulator while the latencies through the source operands are considerably longer.) When such a sequence comprises the body of a loop, any of the schedulers can fill up (become clogged) with waiting μ ops, which can cause a back-pressure stall that prevents new iterations from being accepted into the SME engine. With no new iterations being accepted into the SME engine, subsequent loads for new iterations cannot issue, which may lead to pipeline bubbles and wasted bandwidth across all units in the SME engine.

Unrolled and software pipelined performance-critical loops reduce some of the burden on the schedulers by using many architectural Z vector registers. Rather than position a consumer compute operation immediately after its load(s) in the code stream, continue with loads for subsequent iterations using different Z vector registers, delaying the arrival of the first consumer compute operation to a time when it is more likely to have its sources ready. The same applies to extracts and stores. Position them in the code sequence after several iterations of compute instructions (and after several iterations of extracts for store instructions) to delay their insertion into their respective schedulers, using the Z vector registers as a buffer. However, because many compute kernels have high load-to-store ratios, extracts and stores are less likely to significantly clog their respective schedulers. In general, prioritize use of the Z vector registers to buffer loads from their consumer operations.

6.6 Core Integer Unit Optimization

The core Integer Execution Unit adjusts pointers by element counts, manipulates predicate values, and resolves branches. Further, it manages integer and flag register data consumed or produced by the SME engine.

The Integer Execution Unit in the core orchestrates execution in the SME engine. Loop control generally stems from memory pointers and data structure dimensions stored in general purpose registers. By resolving loop branches, only correct path instructions destined for the SME engine retire and actually get sent to the SME engine. The core may generate predicate values for managing edges of data structures where the dimension is not a multiple of the vector length ([Section 4.4.3, “Data Structure Edge Control”](#)). And, it may generate `wv` register operands that serve as row indices into ZA storage for some SME instructions. When used in this manner, these control code sequences execute entirely in the core based on information contained entirely within the core. These sequences are not dependent on data values computed in the SME engine. This enables the core to *run ahead* of SME engine execution. The nonspeculative SME engine instructions, predicate updates, and pointers (physical addresses) are merely pushed to the SME engine where it queues them up and executes them in time, *trailing* the core.

Predicate manipulations on the result of data element value comparisons in the SME engine have a much longer latency and are not recommended. Because execution in the SME engine occurs nonspeculatively, and generally trails execution in the core (sometimes significantly), the core could wait from dozens to thousands of cycles for the predicate to be returned, depending on the algorithm and observed memory latencies in the SME engine. Transit of predicate values requires an additional ~20 cycles to return to the core after they are computed. Then, after manipulation in the core, the resulting predicate values take another ~20 cycles to transit back to the SME engine. These updated predicate values must be sent to the SME engine in sequential order with other SME engine instructions, and thus all subsequent SME engine instructions are delayed from even being sent.

The same also applies to scalar integer and flag values sent from the SME engine back to the core. This delay may be unavoidable when an algorithm consumes a large amount of data and returns a single (or a few) data values at the end. In these scenarios, group the data return instructions together, preferably with no further subsequent SME engine instructions.

Some instructions are cracked into several `μops` or require microcoded sequences, which may vary by core type and family. Predicate manipulation instructions can themselves have a governing predicate, where merge-style predication (`Pg/M`) requires a read of the destination register in order to preserve predicated-off bits. All of these `μops` are generally confined to a single Integer Execution Unit pipeline and generally take three cycles to complete.

6.7 Core Load/Store Execution Unit Optimization

Loads and stores of predicate values are performed in the core Load/Store Execution Unit, alongside predicate manipulations, which are also performed in the core.

The core Load/Store Execution Unit also performs all address-related aspects of SSVE/SME memory operations.

And, the core issues hardware prefetches targeting the Shared L2 Cache that based on SME engine instruction memory access patterns.

6.7.1 Address Translations and Permission Checks

Address translation from virtual to physical address occurs in the core's Load/Store Execution Unit, regardless if the memory access instruction ultimately executes in the core or the SME engine. Thus the coverage of the L1D TLB and L2 Shared TLB ([Section 5.6.2, "L1 Data \(L1D\) TLB and Shared L2 TLB"](#)) includes data accessed by SME engine instructions. Like in the core, TLB misses can cause delays in execution while the translation is obtained. As noted in the ISA chapter, the translations are not performed, and thus TLB misses are not issued for inactive elements due to governing predicates.

Manage the TLB working set of SME engine-based computations like for those that execute in the core. See [Section 6.9, "Memory Access Optimizations"](#) for strategies for managing the data working set size.

6.7.2 Memory Ordering Between Core and SME Engine

The core Load/Store Execution Unit enforces memory ordering between the core and the SME engine. From within a thread of execution, data values for a particular memory location must appear to execute in program order, regardless of where the access occurs, core or SME engine. Tracking and enforcement occurs on 4KiB granularity (quarters of 16KiB pages), and not on cache line granularity. Enforcement takes the form of barriers that ensure that all accesses to a 4KiB page in the SME engine complete before subsequent accesses in the core begin (and vice versa). Combined with the trailing nature of SME engine execution, the core may have to wait dozens to thousands of cycles for the SME engine to conclude its accesses to a particular 4KiB page, causing significant core delays if the core needs access to anything in the same 4KiB.

As a result, avoid algorithms that require frequent reading and writing of the same data structures by both the core and the SME engine. And further, avoid placing unrelated core and SME engine data structures in the same 4KiB page. This most commonly occurs when SME engine data is placed on the program stack; use heap memory for SME engine data. Returning values via memory to the core at the end of a large computation is often not a performance concern, but the core likely ends up waiting for the result. Group such communication together where possible.

Standard cache coherence protocols govern the movement and states of cache lines between the core's L1D Cache and the Shared L2 Cache.

Recommendation: Avoid memory hazards between the core and the SME engine by isolating data accessed by the engine to separate 4KiB sections of memory.

[Magnitude: High | Applicability: Medium] From a single thread perspective, the ISA requires that all accesses to memory appear in program order. The core enforces this ordering through coarse-grained tracking of 4KiB regions. Wherever possible, ensure separation of data between the core and the SME engine, including the stack.

6.7.3 SME Engine Instruction Delivery and Packing Fusion

Each instruction destined for the SME engine processes through the store portion of the core Load/Store Execution Unit for packaging and shipment. The unit performs writes for any older core stores before it ships younger instructions to the SME engine, and won't perform writes for yet younger core stores until all prior SME engine instructions are sent. Intermixed core stores that hit the L1D Cache create a several cycle bubble in the instruction stream being delivered to the SME engine. However, core stores that miss can cause much longer delays. Avoid fine-grained interleaving of stores and SME engine instructions in the unified instruction stream, preferring blocks of each.

Predicate updates made by the core must be reflected in the SME engine's copy of the predicate register file. When a predicate is updated in the core, a predicate update instruction with the new data value is sent to the SME engine in order, consuming the same resources and following the same rules as any other instruction being sent to the SME engine.

Core store bandwidth is described in [Section A.3.4, "Load and Store Execution Unit Bandwidth"](#). Each SME engine instruction and predicate update requires the same unit resources as one store, in particular the equivalent of one write into the cache. Note that the maximum number of writes into the cache (hence the maximum instruction bandwidth from the core to the SME engine) is less than the maximum issue rate inside the SME engine itself. However, many instructions require multiple μ ops or multiple issue slots in the SME engine, and thus the actual achieved SME engine instruction bandwidth is lower than the instruction transmittal bandwidth.

Still, some algorithms benefit from instruction packing fusion that combines two computation instructions into one, reducing the pressure on the core store pipelines. Most adjacent SSFP/SSVE/SME computation instructions may fuse in this manner. SSFP/SSVE/SME load/store or control instructions are not eligible, nor are SSSFP/SVE/SME computation instructions that pass data back and forth between the core and SME engine. Specifically, instructions that require NZCV flags or general purpose registers (except for ws/wv used as row indices into ZA storage) as sources are ineligible. Similarly, SSFP/SSVE/SME instructions that produce new NZCV flags, general purpose register contents, or predicate results are also ineligible. There are other occasional microarchitectural constraints that may prevent fusion, however these are difficult to predict from the instruction stream itself. In general, pair up SME engine computation instructions to benefit from packing fusion and [instruction operation fusion](#), and interleave blocks of compute and blocks of memory instructions to avoid over-filling any out-of-order execution tracking structures.

Recommendation: Pack computation instructions back-to-back in the instruction stream to maximize instruction delivery to the SME engine.

[Magnitude: Medium | Applicability: High] The core performs packing fusion of eligible computation instructions to increase the bandwidth of instruction delivery to the SME engine. Follow the specified guidelines where possible.

6.8 SSVE/SME Instruction Operation Fusion

Some pairs of instructions can be fused by the microarchitecture inside the SME engine to reduce latency, reduce accesses to the register file, and in some cases, reduce

instruction issue. [Table 6.1: “SSVE/SME Instruction Operation Fusion Pairings”](#) lists the supported instruction pairs. These combinations combine an SSVE instruction with an SME instruction. (Note that instruction operation fusion is distinct from [packing fusion](#).)

Table 6.1. SSVE/SME Instruction Operation Fusion Pairings

Row	First Instruction		Second Instruction		Operand Rule
1	Table lookup using ZT0 Write 2 or 4 vectors	LUTI2, LUTI4	ZA vector compute Read 2 or 4 vectors	Most ZA vector computation instructions	2 or 4 vector set must match exactly with either/both Zn/Zm source registers of second instruction
2	Table lookup using ZT0 Write 1 vector	LUTI2, LUTI4	ZA tile compute Read 1 vector	Most ZA tile computation instructions	1 vector set must match exactly with either/both Zn/Zm source register of second instruction
3	Move (extract) horizontal from ZA byte tile Write 2 or 4 vectors	MOVA (ZA tile slice)	Downcast/shift/saturate/zip Read 2 or 4 vectors	BFCVT, BFCVTN, FCVT, FCVTN, SQCVT, SQCVTN, SQCVTU, SQCVTUN, SQRSHR, SQRSHRN, SQRSHRU, SQRSHRUN, SQSHRN, SQSHRUN, UQCVT, UQCVTN, UQRSHR, UQRSHRN, UZP, ZIP	2 or 4 vector set must match exactly
4	Move (extract) horizontal from ZA byte tile Write 2 vectors	MOVA (ZA tile slice)	Move (extract) horizontal from ZA, byte tile Write 2 vectors	MOVA (ZA tile slice)	None.
5	Move (extract) horizontal from ZA storage as part of a store	ST1* (ZA tile slice) STR (ZA row)	Move (extract) horizontal from ZA storage as part of a store	ST1* (ZA tile slice) STR (ZA row)	None

Table lookup fusion ([Table 6.1](#) rows 1-2) is particularly useful for SME algorithms that work with data that is stored in memory in compressed form. [Vector lookup table \(LUTI2 or LUTI4\) operations](#) are often used to decompress data that has been compressed down to 2-bit or 4-bit memory formats and often feed [ZA vector multiply-accumulate operations](#). When used per the patterns described below, the fusion reduces the latency of the lookup table instruction to a single cycle on top of the consumer vector latency, and eliminates the one to four μ ops needed to execute the lookup table instruction.

Fusion occurs between a first instruction and *each of a set of second instructions in a contiguous block* that meet the requirements for a fused second instruction. This block must occur immediately after the first instruction. In the example below, the LUTI2 is fused with each of the FMLAs because each FMLA consumes the exact same registers produced

by the `LUTI2`. This can reduce the latency of the data flow through the `LUTI2` and `FMLAS`. Note that if another SSFP/SSVE/SME instruction that writes a Z register appears between the `FMLAS`, the second `FMLA` is not able to fuse. Note that in general, instructions that execute only in the core can be scattered in among these instructions and not affect fusion. However, fusion is dynamically performed, and if the instructions arrive too far apart in time, fusion does not occur.

```

LUTI2 {z1,z2}, zT0, z5[0]      // Table lookup: first instruction
FMLA  ZA[W8], {z1,z2}, z6[0]   // Table lookup: second instruction
FMLA  ZA[W8], {z1,z2}, z6[1]   // Table lookup: additional second instruction

```

In some cases, the first instruction must still execute singly (on its own). In this following sequence extended from above, the `FSUB` consumes only `z1`, and thus the `LUTI2` needs to execute on its own to feed the `FSUB`. Similarly, if the `FSUB` dpes not use `z1` or `z2`, other instructions later in the code may, and thus the `LUTI2` also needs to execute singly. And if no other consumer actually needs `z1` or `z2`, then the single execution of the `LUTI2` is only wasted SSVE bandwidth and energy.

```

LUTI2 {z1,z2}, zT0, z5[0]      // Table lookup: first instruction
FMLA  ZA[W8], {z1,z2}, z6[0]   // Table lookup: second instruction
FMLA  ZA[W8], {z1,z2}, z6[1]   // Table lookup: additional second instruction
FSUB  z3, z1, z7               // Non-fusion use of z1 requires independent
                               //      LUTI2 execution
FMLA  ZA[W8], {z1,z2}, z6[2]   // Non-fusion because the contiguous block of
                               //      second instructions was broken by the FSUB

```

In the following example, `z1` and `z2` are immediately and completely overwritten by the second `LUTI2` instruction that breaks the contiguous block of second instructions (here, just one `FMLA`). In this case, there are no other possible consumers of `z1` and `z2` as produced by the first `LUTI2`, and thus it does not execute singly. Note that the second `LUTI2` is itself a first instruction and can then match a set of subsequent second instructions.

```

LUTI2 {z1,z2}, zT0, z5[0]      // Table lookup: first instruction
FMLA  ZA[8], {z1,z2}, {z6}     // Table lookup: second instruction
LUTI2 {z1,z2}, zT0, z7[0]     // Overwrite of z1 and z2; No single execution of
                               //      first LUTI2 needed

```

In summary, when using `LUTI` instructions, create a contiguous set of computation instructions that consume the results of the same `LUTI`. End the set with an instruction that completely overwrites the involved vector registers to achieve the largest gains and avoid wasted single execution of the `LUTI`. Do not break the fusion sequence with any other SSVE/SME instructions that write Z registers.

Extraction with downcast/shift/saturate/zip fusion (Table 6.1 row 3) is useful for preparing data for storage in memory, where smaller element sizes are often preferred. See Section 3.1.2, “Integer Data Types”, in particular Figure 3.2: “ASIMD Integer Processing Example”, for more on integer data type size reductions. This type of fusion can merge the `MOVA` extract and down-conversion operations together in order to reduce the number of issued `μops` and reduce latency. For calculations with floating point values,

this fusion is only relevant when converting `float32` accumulator values into either `float16` or `bfloat16` formats. For integer calculations, down-conversions are common due to the large accumulator element sizes, and could include truncation, right shifting, rounding, and saturation as the accumulation registers are halved or quartered in width. Finally, `MOVA` extract instructions can fuse with `ZIP` and `UZP` that perform interleaving and deinterleaving. Note that some down-conversion instructions, such as `FCVTN` (`float32` to `float16`), already include an interleaving operation, which thus enables an extraction/down-conversion/interleave all-in-one fused instruction.

The following code shows an example of this type of fusion that combines extraction, conversion, and interleaving:

```

MOVA { Z0.B, Z1.B }, ZA0H.B[W12, 0:1]
BFCVTN Z4.H, { Z0.S, Z1.S }           // Fuses with prev MOVA
ADD W12, W12, #16
MOVA { Z0.B, Z1.B }, ZA0H.B[W12, 0:1] // Second MOVA overwrite of first MOVA
                                         //   dests; single exec of first MOVA
                                         //   not needed
BFCVTN Z5.H, { Z0.S, Z1.S }           // Fuses with previous MOVA
ADD W12, W12, #16
MOVA { Z0.B, Z1.B }, ZA0H.B[W12, 0:1]
BFCVTN Z6.H, { Z0.S, Z1.S }           // Fuses with previous MOVA
ADD W12, W12, #16
MOVA { Z0.B, Z1.B }, ZA0H.B[W12, 0:1] // If this block of code repeats, the MOVA
                                         //   at the top of the block serves to
                                         //   overwrite, and single exec not needed;
                                         //   otherwise requires single execution
BFCVTN Z7.H, { Z0.S, Z1.S }           // Fuses with previous MOVA
ST1H { Z4.H, Z5.H, Z6.H, Z7.H }, PN8, [X10] // Does not break a fusion sequence
INCB X10, ALL, MUL #4

```

For the downcast/shift/saturate forms, failing to overwrite the destination registers of the `MOVA` thus requiring single execution of the first instruction (in addition to the fused executions) is quite expensive. Fusion enables the conversion to be performed in the Processing Grid, where the data is reduced to just a single vector that needs to be extracted from `ZA` storage. If single execution is required of the `MOVA` as well, then four additional vectors must be extracted from `ZA` storage. Extraction is a common bottleneck out of the `ZA` storage, and requiring the four extractions when unneeded is quite wasteful, and notably worse bandwidth than not fusing at all.

For code that does not perform conversion and only follows the `MOVA` with a `ZIP` or `UZP`, overwriting the result of the `MOVA` with the result of the `ZIP` or `UZP` can conveniently satisfy the overwrite requirement.

```

// First μop: Extract 4 vectors from ZA storage
MOVA {Z0.D, Z1.D, Z2.D, Z3.D}, ZA.D[W8, #0, VGx4]
// Second μop: Interleave vectors; overwrite all registers written by MOVA
//   MOVA does not have to execute singly
ZIP {Z0.S, Z1.S, Z2.S, Z3.S}, {Z0.S, Z1.S, Z2.S, Z3.S}

```

In summary, pair extraction instructions with conversion instructions, where possible, to improve efficiency. If conversion is not needed, pair extraction instructions, where possible, with interleave/deinterleave instructions. Do not break the fusion sequence with

any other SSFP/SSVE/SME instructions, and immediately follow the sequence with a complete overwrite of the involved Z registers to gain the most benefit.

Aggregated extraction fusion (Table 6.1 rows 4-5) is useful for combining two two-vector `MOVA` extractions together into a single four-vector extraction, as well as combining two single-vector extractions as part of `STR` and `ST1*` instructions. In the `MOVA` case, this pairing can be useful if the extractions do not happen to align in a way that could normally be accomplished with a single four-vector extraction instruction.

Recommendation: Create code sequences that leverage operation fusion to reduce latency and execution.

[Magnitude: High | Applicability: High] The SME engine offers several forms of operation fusion that can reduce latency and reduce the utilization of the execution units. Extracting with downcast/shift/saturate/zip fusion is especially applicable to most algorithms. Generally, prefer the other fusion types over aggregated extraction when available. If used, do not break the fusion sequence with any other SSFP/SSVE/SME instructions. Follow the guidelines carefully to maximize the benefits.

6.9 Memory Access Optimizations

As previously described, SME engine memory accesses use the core's data TLB hierarchy for translations and the Shared L2 Cache for data. In addition, the core issues hardware prefetch operations that target the Shared L2 Cache based on SME engine access patterns. For workloads with large data footprints, or non-streaming memory accesses, consider the following software optimizations.

6.9.1 Cache Blocking

Cache blocking is a critical optimization for core-based algorithms because of the limited size of the L1D Cache. This common technique is used to stage computation such that only small sections of the input data set that fit into the L1D Cache are needed at once. The input data is consumed to compute relevant portions of all dependent outputs, and then those partial results are added to partial results from earlier stages. In this way, L1D Cache misses on the input data can be minimized across the duration of the larger computation. This adds software complexity in the form of additional nested loops that select blocks of input data, but can be effective in improving the utilization of the compute hardware.

Blocking can also be beneficial when performing large SSVE/SME matrix operations. Because the SME engine is connected to the Shared L2 Cache, the size of the data blocks can be larger than for core ASIMD versions of the same algorithm (but note that all the cores in the cluster also share the Shared L2 Cache). For instance, a matrix of 384 x 384 single-precision floating point elements can fit into just over ½ MiB, which can easily fit into P and E Cluster Shared L2 Caches, with P Cluster Shared L2 Caches generally being considerably larger than their E Cluster counterparts. Note that the cache sizes along with number of cores present in each cluster varies between chips, and using somewhat smaller blocks can help ensure performance compatibility. Blocking can similarly help contain TLB footprint. See [Section 5.6.4, "Shared L2 Cache"](#) and [Section 5.6.2, "L1 Data \(L1D\) TLB and Shared L2 TLB"](#) for sizes and coverage.

6.9.2 Software Prefetching

SVSE/SME loads and stores trigger the same hardware prefetching mechanisms used by the core, but in a mode that only prefetches from main memory into the Shared L2 Cache. Thus the same types of data access patterns that are successfully prefetched into the core's L1D Cache are also successfully prefetched into the Shared L2 Cache. For instance, access patterns that stream down rows for matrices stored in a row-major organization (or, equivalently, down columns for matrices stored in a column-major organization) should stream in from memory automatically. Hardware prefetchers are adaptive and attempt to autonomously bring in data when it are needed without prefetching too far ahead. Streaming across rows in row-major organization or across columns in a column-major organization, which is sometimes unavoidable, can be more difficult for the hardware prefetchers because of the unusual strides (+ row length or + column length).

For cases of unusual strides, software prefetching can be advantageous. Like in the core, software prefetches must be inserted sufficiently ahead of loads or stores to the same cache line in order to see benefit. Software prefetches are often issued early, often before SME engine instructions in the same routine become nonspeculative and are sent to the SME engine by the core, which naturally allows them to execute ahead of the associated loads and stores. Use the `L2` destination encoding to load data into the Shared L2 cache while not polluting the L1D Cache. Software prefetches are most important when the stride is more than 8KiB. In general, however, blocking or otherwise rearranging the data that permits more consistent and shorter strides is likely to perform better. See [Section 5.6.12, “Prefetching”](#) for more information on general prefetching recommendations strategies.

6.9.3 Non-Temporal Memory Access

SSVE offers non-temporal versions of some of the fundamental vector loads and stores, `LDNT1[B,H,W,D]/STNT1[B,H,W,D]`, similar to those in the core described in [Section 5.6.8, “Non-Temporal \(NT\) Accesses”](#). Non-temporal accesses can improve overall memory performance when applied to large blocks of data that will not be reused in the near term, thus avoiding cache pollution, such as using SSVE for very large vector/matrix manipulation and for memory copy operations. Choosing between temporal and non-temporal accesses can be made per data structure or per function, such that large tables with high reuse can be cached while others can be streamed in and out, avoiding cache pollution.

6.10 Preferred Instructions for Common Operations

Use the following preferred instructions for common operations.

6.10.1 Register Data Copy

Use the following preferred instructions for creating copies of values in registers.

Table 6.2. Register Data Copy Instructions

Register Type	Register Data Copy Instructions
Z Vector	<code>ORR Zd.D, Zn.D, Zm.D</code> // (When <code>n == m</code>) // Usually aliased to <code>MOV Zd.D, Zn.D</code>

Register Type	Register Data Copy Instructions
	When copying to avoid a destructive destination, use MOVPRFX .
ZA Storage	There are no instructions that copy a ZA tile or vector to another.

Use the following preferred instructions for zeroing registers and breaking dependence chains.

Register Type	Register Data Zeroing Instructions
Z Vector	DUP Zd.D, #0, LSL #0 // Usually aliased to MOV Zd.D, #0, LSL #0
ZT0 LUT Vector	ZERO { ZT0 }
ZA Storage	ZERO { <list> } // List of tiles up to full array. // { ZA } offers coarse-grained renaming // benefits.

The SME engine is a shared processing resource that executes issued instructions from all cores in the cluster in a fairly fine-grained manner, similar to that of simultaneous multithreading. It is possible for a small number of cores (even one or two) to fully consume the SME engine's processing capacity in that cluster. SME engine throughput may actually decrease somewhat when oversubscribed due to sharing of internal resources and general contention. As a general guideline, do not create more than two threads per cluster that heavily use the SME engine. The system scheduler attempts to spread out (load balance) these threads across the clusters. However, some experimentation may be required to find the optimal thread count for a given application considering the intensity and burstiness of the SSVE/SME portion of the code.

In order to use SSVE/SME instructions, Streaming SVE mode and (in most cases) ZA mode must be enabled, as are described in [Section 4.2, “SVE Streaming and ZA Modes”](#). In brief, Streaming SVE mode enables SSFP/SSVE Z and P registers and associated instructions while disabling Advanced SIMD and FP in the core. ZA mode enables the ZA storage. Both modes together enable the SME instructions that target ZA storage.

Applications should only enter and remain in Streaming SVE mode or ZA mode (or both) when the functionality is required. The SME engine allocates internal resources when either of the modes are enabled, and those resources are then unavailable to other cores that might benefit. For example, when any core is in Streaming SVE or ZA mode in the cluster, the SME engine in that cluster is powered up and ready to process incoming instructions. When powered up, the system allots power to the SME engine that could cause the entire cluster to operate at a lower performance state. When enabled, the new large Z, P, and ZA storage states increase OS context switch latency. And lastly, the SME engine executes scalar floating point (SSFP) at a lower rate than when executed in the core. Note, however, that switching modes does incur some execution overhead, so fine-grained switching is not advised, and that Advanced SIMD and FP and Z/P registers are cleared on Streaming SVE mode switches.

Most software enables and disables both modes together. However, the ISA does allow Streaming SVE and ZA modes to be controlled independently.

- **Only Streaming SVE mode:** Software can use the longer SSVE registers along with predication without enabling the ZA storage state and related SME instructions. However, the narrower single-issue SSVE Execution Unit and the slower load and store access latency, along with the slow data transit rates to and from the core, reduce the benefits of this mode over Advanced SIMD and FP. In general, favor Advanced SIMD and FP in the core over SSVE-only execution.
- **Only ZA mode:** The CPU holds ZA storage state while software uses Advanced SIMD and FP instructions in the core. This mode is most likely entered by exiting Streaming SVE mode (`SMSTOP SM`) after both are engaged. Primarily, this is used for making brief excursions to core-only code (especially for scalar floating point) from within an SSVE/SME routine because it does not require a full spill and fill of ZA storage around the excursion. It should not be used for long sequences of non-SSVE/SME code because of the power and context switch penalties mentioned above. Note that the scalar floating point registers are not preserved across Streaming SVE mode switches. They are subsets of the V registers in non-Streaming SVE mode and subsets of the Z registers in Streaming SVE mode.



Chapter 7. Asymmetric Multiprocessor (AMP) Optimization

As mentioned in the introduction, Apple silicon chips feature two types of general-purpose CPU cores, performance cores (P cores) and efficiency cores (E cores). Both types of cores use the same instruction set architecture (ISA) and coherently access the same memory, so threads may freely move back and forth between the two types of cores. The E cores are physically smaller cores with shorter wires and smaller transistors, but are based on a similar microarchitecture as the P cores, thus workloads optimized for one are expected to perform well on the other. Having both types allows the system to optimize for performance when performance is a priority, and optimize for efficiency (for example, improving battery life) when it isn't. Because E cores offer compelling high performance on their own, certain tasks may perform sufficiently well on the E cores which can free up P cores for other more demanding tasks. Lastly, E cores can be used in multithreaded workloads along with the P cores to add significant additional performance.

Optimizing multithreaded applications requires careful consideration. In addition to the discussion below, see [Tuning Your Code's Performance for Apple Silicon](#) and [Addressing Architectural Differences in Your macOS Code](#) on the Apple developer website.

To programmatically determine the CPU and chip configurations, see [Appendix B, Dynamic Determination of Chip-Specific Capabilities](#).

7.1 Prioritizing Work

Developers can prioritize work by providing hints to the system as to which work items are the most performance-critical, and hence should be favored for execution on the P cores, using the quality of service (QoS) flags. Likewise, developers can indicate which work items are less performance-critical ("background") and thus should be favored for execution on the E cores. Nevertheless, the system may execute performance-critical threads on E cores when there are more such threads than available P cores. Especially when the desired performance level is unspecified, the system monitors applications dynamically and automatically prioritizes CPU-intensive threads for the P cores for maximum performance. Similarly, to save power, the system may assign less CPU-intensive threads to the E cores. Documentation on the use of the QoS classes can be found on the [Prioritize Work at the Task Level](#) page of the [Energy Efficiency Guide for Mac Apps](#) guide.

Recommendation: Use quality of service (QoS) APIs to prioritize work.

[Magnitude: Medium | Applicability: Low] Software cannot directly control the type of core on which any task executes. However, use Quality of Service classes to provide hints to the system.

7.2 Partitioning Work

When developing applications with large numbers of worker threads that all need maximum performance, consider the implications of asymmetric multiprocessing. At a

high level, there are two general approaches used by developers to divide up work on symmetric MP systems, but only one works well on an asymmetric system:

- **Static Partitioning:** The partitioning algorithm divides up work into pre-determined, equal-size chunks. Software assigns each worker thread a particular set of chunks to complete. Synchronization techniques such as barriers occasionally synchronize all threads, potentially after a phase of execution. Although this works well in a symmetric multi-processor system it often doesn't work well in an asymmetric system because the underlying assumption doesn't apply: in a symmetric system, software can divide a task into in equal size chunks because the underlying processing elements are all equally capable. But this is not the case in an asymmetric system.
- **Dynamic Partitioning:** The partitioning algorithm divides up work into tasks, potentially of various sizes, stashing them in a task queue. Each worker thread retrieves a new task from the queue when it finishes the previous task. Although this strategy may incur slightly higher task-management overhead, the benefits of dynamically assigning chunks of the work to available threads typically allows the work to be completed faster.

[Section 7.2.1, "Avoid Static Partitioning"](#) describes some of the pitfalls associated with static partitioning, while [Section 7.2.2, "Use Dynamic Partitioning"](#) describes the benefits of dynamic partitioning.

7.2.1 Avoid Static Partitioning

Static partitioning can lead to longer latency execution on asymmetric MP systems. Although these equal-size chunks may all complete in about the same amount of time on a symmetric MP system, asymmetric cores execute them at different speeds and thereby introduce dynamic load imbalance. Some threads will end up running faster (on P cores) and some slower (on E cores), sometimes switching cores in the middle of execution, usually in a manner that the program itself cannot control.

Because of the asymmetry, threads running primarily on P cores make more progress than those running primarily on E cores. Although the operating system (OS) moves those primarily E core threads to the P cores when P cores become available (through rebalancing), those primarily E core threads may already be behind. Note that OS also often rotates threads amongst the P and E cores to load balance when running for multiple OS time quanta.

Similarly, static partitioning strategies tend to assume that little or no other time-consuming tasks are running on the system, and that the static partitioning algorithm can statically and evenly partition work across all of the compute resources. When other unrelated tasks are running on a core, a static slice of the MP application may be delayed, which delays the overall application in the same manner as static slice running on a slower core. Therefore, even on symmetric multiprocessors, dynamic partitioning is often a better strategy.

If static thread scheduling is the only viable option, use `sysctl` queries to determine the core count. See [Appendix B, Dynamic Determination of Chip-Specific Capabilities](#).

7.2.2 Use Dynamic Partitioning

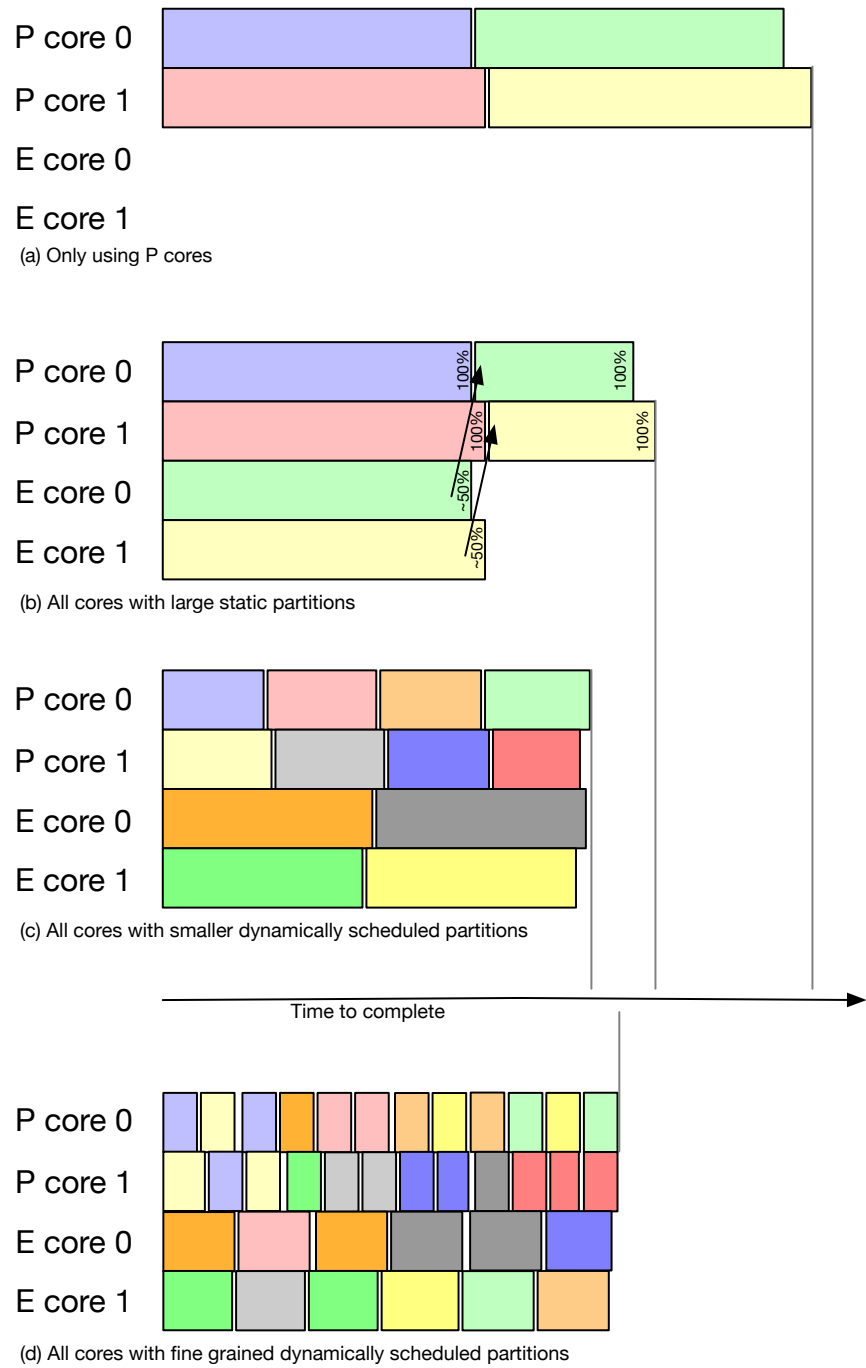
Dynamic partitioning is a better approach than static partitioning in most cases. Threads that happen to be assigned to P cores naturally complete work more quickly, taking a

larger number of parallel tasks from the task queue, while threads assigned to E cores churn through a smaller number of tasks more slowly. As threads move around the cores, they naturally adapt and start picking up more tasks when they are running on the faster P cores or fewer when they happen to be assigned to the slower E cores.

To use all the available computing power of all the cores in the system, applications typically need to spawn a number of worker threads equal to the number of available cores (e.g., 8 for the M1's 4 P cores + 4 E cores). Use `sysctl` for determining the number of cores in the system (see [Appendix B, Dynamic Determination of Chip-Specific Capabilities](#)). As a broad guideline, divide work into as many discrete tasks as possible, at least 3x the number of cores and preferably more. Of course, when work items are too small, work-queue management overhead can begin to dominate the actual work, in terms of both instruction count and queue memory contention. Work-queue management overhead also increases with additional cores due to communication latency and contention, so consider the range of chips your code may run on, from an A14 Bionic chip with 2 P cores and 4 E cores to an M2 Ultra chip with 16 P cores and 8 E cores. Lastly, single-threaded CPU performance tends to increase faster than multi-threaded work-queue management performance from family to family, so work items that are the proper size in one family may be too small in a future family. Because work-queue management itself has a direct impact on scaling performance, keep this code as short and efficient as possible. Finding the proper work item size requires experimentation and tuning.

Consider the illustrative example [Figure 7.1: "Partitioning Strategies: An Illustrative Example"](#) for a 2 P core + 2 E core chip where the P cores are roughly double the performance on the particular workload. A static partitioning into 4 threads and using all 4 cores (b) is faster than using just 2 threads prioritized to run on the 2 P cores (a). Dividing the work into smaller chunks and allowing threads to dynamically select tasks from a common work pool (c) can achieve better utilization of all the cores and the shortest runtime. However, further reducing chunk size (d) results in worse performance because of increased management overhead (depicted as more gaps between work items) and contention on the work queue (depicted as larger gaps). The core count, relative performance, and workload management overheads in this example illustrate the concepts and don't prescribe a particular workload partition size. The aforementioned 3x partitions compared with available cores is a good starting point, but experimentation and tuning is required. If execution completion is delayed by a small number of straggler threads (threads that continue running long after the others have completed), finer-grained work partitioning may help.

Figure 7.1. Partitioning Strategies: An Illustrative Example



As illustrated in [Figure 7.1: “Partitioning Strategies: An Illustrative Example”](#), a dynamic work partitioning strategy most often leads to the best performance because of its flexible work assignment. Threads synchronized using static techniques like barriers do not work well when some cores can execute code significantly faster than others. Management of multiprocessing at a low level can be accomplished through C++ threads or [pthreads](#). These allow for direct creation and control of multiple POSIX-compatible threads managed by the application, usually with at least one thread per available core. Distribute work

amongst these threads by building dynamic task queues on top of these APIs using basic primitives such as locks and condition variables. However, because this code is complex and often requires extensive tuning, consider a pre-built task management API like [Grand Central Dispatch](#). This framework provides mechanisms to manage and synchronize large numbers of concurrent parallel tasks and automatically handles low-level thread management.

Recommendation: Use Grand Central Dispatch for dynamic work and thread management.

[Magnitude: Medium | Applicability: Low] Avoid static work partitioning techniques that assign equal portions of the work to each thread, with one thread per core. Instead, divide the work into more (smaller) pieces, 3x the number of cores as a starting point. Use dynamic techniques that allow faster threads to pull more work from a shared work queue. Consider using Grand Central Dispatch which is tuned for each platform to offer the best performance and fairness. C++ threads and pthreads are also available as cross-platform alternatives.

7.3 Avoid Spin-Wait

Regardless of work partitioning strategy, avoid keeping threads active but effectively idle in spin-wait loops. In the rare circumstance the spin-wait is known to be short lived, where the wait time is on the same order as the thread scheduling overhead, spin-wait may be a workable choice. However, most wait times are unknown and often longer than the scheduling overhead. By occupying cores, especially P cores, with spin-wait loops, work that might have fallen behind on E cores may be forced by the system to continue on the E cores. Likewise, the system often has other unrelated work to perform. Spin-waiting prevents the system from using that effectively idle time for other system work, and may force a context switch of application work at less optimal times. See "Don't Keep Threads Active and Idle" at [Tuning Your Code's Performance for Apple Silicon](#).

Recommendation: Block threads when idle and avoid spin-wait loops.

[Magnitude: Medium | Applicability: Low] When the wait is expected to be very short, on par with the cost of a thread switch, use spin-wait loops to cause a thread to wait for next stage. In all other cases, block the thread such that the operating system can schedule other work on the core.

7.4 APIs for Synchronization and Thread Communication

Experienced MP developers may be tempted to try to achieve maximum performance by building their own primitives (e.g., spin locks) from scratch using Arm ISA Load/Store exclusive or atomic instructions. However, these primitives require complex algorithms (e.g., queued locks) to ensure fairness between all threads, avoiding bias (or even monopolization) to cores within a cluster. Also, topologies, protocols, and latencies are likely to change between families and even between chips in a particular family. Custom synchronization primitives tuned for one product may not function well in another.

Before attempting to write your own primitives, try APIs that are specific for Apple languages or cross-platform APIs to ensure reliably good performance. These APIs are tuned for each chip to account for topology, protocol, and latency changes.

The section "Synchronize Access to Shared Data in Memory" in [Addressing Architectural Differences in Your macOS Code](#) offers various API options summarized here:

- [Grand Central Dispatch](#) (GCD) provides serial queues and other ways to synchronize tasks.
- The `@synchronized` directive creates a mutex lock for Objective-C code.
- The [Foundation](#) framework defines standard mutexes, conditions, and other types of locks.
- The `os` framework provides "unfair" locks for synchronization.
- The `pthread`s library defines standard mutexes and condition variables.
- The C11/C++11 primitives in `stdatomic.h` support custom memory ordering in atomic operations.

Recommendation: Use tuned APIs for synchronization primitives.

[Magnitude: Medium | Applicability: Low] Use APIs specific to Apple languages or cross-platform APIs for high performance thread synchronization. These libraries are tuned for the specific platform to offer the best performance and fairness.

Use the following metrics to measure atomic and exclusive instruction rates, as well as success and fail percentages. For exclusive instructions, such as `LDREX/STREX`, the events count depending on whether the processor was able to maintain exclusive access of the cache line between the load and the store. For compare-and-swap instructions, such as `CAS`, the events count depending on whether the compare condition was met.

Table 7.1. Common Atomic and Exclusive Metrics

Name and Formula (Event Definitions: Section 8.1, "CPU Counters With Performance Monitoring Events")	Description
Atomic/Exclusive Success Density: <Ev ATOMIC_OR_EXCLUSIVE_SUCC> / <Ev INST_ALL>	Proportion atomic and exclusive instructions that succeed of all retired instructions See note regarding the ATOMIC_OR_EXCLUSIVE events in Table 8.1 .
Atomic/Exclusive Fail Density: <Ev ATOMIC_OR_EXCLUSIVE_FAIL> / <Ev INST_ALL>	Proportion atomic and exclusive instructions that fail of all retired instructions See note regarding the ATOMIC_OR_EXCLUSIVE events in Table 8.1 .

7.5 Thread Communication Tradeoffs

Communication latencies are asymmetric between various cores in the system. Data written by one core and read by another in the same cluster is fast via the Shared L2 Cache. If a multithreaded application has a thread count equal to or smaller than the number of cores in a cluster, the system tries to keep the threads together on the same cluster. Thus, the threads tend to share data using the faster paths through the Shared L2 Cache.

Sharing of data between clusters must pass through the slower inter-cluster communication network (the fabric). This significantly increases the latency of

the communication by roughly an order-of-magnitude compared with intra-cluster communications. Applications with more threads than the number of cores available in a single cluster almost inevitably encounter these longer communication latencies. The system keeps as many of the threads active as possible, likely employing cores in multiple clusters at least part of the time. If each of the threads in an application deals with a more or less independent pool of data, these longer latencies are typically not an issue. But any applications where worker threads regularly communicate and share large amounts data can require careful tuning to minimize the effect of varying communication latencies.

See [Section A.3.5, “Memory Hierarchy Access Latencies”](#) for typical best case latencies.

When managing shared data such as a work queue, semaphore, or atomic counter, a core snoops away the cache line from other cores to perform the read-modify-write operation. In some code sequences, the thread may first read the status of the queue or semaphore non-atomically prior to the read-modify-write operation causing additional traffic. Other custom sequences may cause additional traffic that is not optimal for Apple silicon. Where possible, use the platform’s tuned APIs (see [Section 7.4, “APIs for Synchronization and Thread Communication”](#)) to efficiently access shared structures. And, increase work element size to reduce the frequency of multiple threads accessing the shared structures simultaneously. Tuning may be required to limit contention of shared structures by employing sufficiently large work items, while avoiding imbalance due to excessively large work items.

Even with efficient accesses, contention may arise between threads particularly when work items are necessarily small. A symptom of this issue may be that multi-threaded performance does not scale with additional threads up to the available core count. Consider *sharding* shared structures, that is, partitioning work into several work queues, semaphore-protected data sets, or atomic counters to reduce contention. Sharding atomic counters is generally straightforward and only requires accumulating the separate counts at the end. Sharding other structures comes with the risk of imbalance that may lead to overall worse performance, such as when one queue dedicated to a subset of the threads takes much longer to complete than another queue. Job stealing or rebalancing algorithms may be needed, adding to the overall complexity. If needed, create shards associated with individual or groups of threads. In very rare occasions, such as when the thread count must be much larger than the core count, it may be appropriate to create shards associated with individual or groups of cores to limit the amount of memory consumed. Use `int pthread_cpu_number_np(size_t *cpu_number_out)` to determine the current core ID and access the appropriate shard. Note that the OS may move a particular thread to a different core at any time, and therefore this is a mostly-correct heuristic for using the appropriate shard. Significant tuning and advanced algorithms are required to ensure shards improve overall performance while limiting worst-case slowdowns. Use sharding, in particular core-based sharding, cautiously and sparingly.

Lastly, avoid false sharing, where two or more independent variables occupy the same 128B cache line. This organization may lead to thrashing of the cache line between cores. See [Section 5.6.6, “Improving Cache Hierarchy Performance”](#).

Recommendation: Minimize data sharing when increasing thread counts.

[Magnitude: Medium | Applicability: Low] Sharing modified data between cores in a cluster is relatively fast due to the Shared L2 Cache. When expanding a workload beyond the number of threads found in a single cluster, consider how much data is written by one core and consumed by another. Excessive sharing may lead to lower-than-expected performance especially with increased thread counts. Reduce the amount of data shared where possible, either algorithmically or via increased data packing within cache lines without introducing false sharing.

7.6 Xcode: Sanitizer Tools

Xcode offers a number of tools to aid in multithreaded code development. The [Thread Sanitizer](#) tool inserts diagnostics into the code to record each memory read or write operation. These diagnostics generate a timestamp for each operation, as well as its location in memory. The tool then reports any operations that occur at the same location at approximately the same time. The tool also detects other thread-related bugs, such as uninitialized mutexes and thread leaks.

Recommendation: Use Thread Sanitizer and other Xcode tools to aid multithreaded code development.

[Magnitude: Medium | Applicability: Low] Correct multithreaded applications require the developer to pay careful attention to synchronization, data ordering, and leaks. Use available tools to verify correctness.



Chapter 8. Xcode: Performance Monitoring and Analysis

Xcode features an integrated development environment tool called **Instruments**. This tool can help you profile your apps in order to better understand and optimize their behavior and performance. The tool offers many different individual instruments (monitors) for tracking CPU performance, GPU performance, network traffic, memory management, audio performance, and more. To look deeper into CPU performance, the Instruments tool offers several primary instruments:

- **Time Profiler:** Periodic (based on wall-clock time) and simultaneous sampler across all CPUs for collecting synchronized call stacks. It provides a broad view of work done from a whole-chip perspective. The sample rate is not adjusted for CPU frequency nor CPU residency of any process or thread, and is not recommended for deep ISA or microarchitectural analysis.
- **CPU Profiler:** Cycle-based profiler that operates independently across the CPUs for analyzing workloads. The profiler uses hardware performance monitoring interrupts (PMIs) to provide more stable and accurate measurements to identify the frequently executed code.
- **CPU Counters:** Counter and sampler of CPU ISA and microarchitectural activity based on performance monitoring events during program execution.
 - Beginning in Instruments 17.0, CPU Counters offers a *guided mode* called CPU Bottlenecks with preset configurations that present recommendations for deeper exploration.
- **Processor Trace:** Hardware-supported, low-overhead CPU execution trace collector to accurately reconstruct program execution.

At launch, Instruments presents a list of templates that each open a preconfigured collection of individual instruments designed to help answer a performance question. Once selected, the tool opens with a blank document ready with the collection of instruments. The collection can be edited to add or remove specific Instruments.

For more information on how to use Xcode and Instruments, see:

- [Getting Started with Instruments \(WWDC19 Video\)](#)
- [Improving your app's performance](#)
- [Profiling apps using Instruments](#)
- [Performance and Metrics](#)
- [Instruments Help](#)

8.1 CPU Counters With Performance Monitoring Events

The CPU Counters instrument uses hardware-based performance counters to record a time-series of various events during execution of a program. Beginning in Instruments 17.0,

CPU Counters offers a *guided mode* called [CPU Bottlenecks](#) with preset configurations that present recommendations for deeper exploration. It also offers the ability for the user to [program individual events](#) manually.

8.1.1 CPU Bottleneck Analysis

The CPU Bottlenecks menu offers a guided experience to help identify bottlenecks in CPU programs. At the top levels, the CPU employs empirical measurements to reflect the relative impact of each category. While the root analysis is available on all CPUs referenced in this guide, some categories and deeper analyses are offered only on newer chips. Additional deeper analysis modes may be available. See the Instruments tool for additional options.

CPU bottleneck analysis categorizes the [sustainable instruction bandwidth](#) of the CPU into four categories: one that represents useful work and three that represent various high-level sources of inefficiency. The [CPU pipeline](#) is divided into two primary phases: Instruction Delivery (obtaining instructions from memory along a predicted execution path) and Instruction Processing (performing the action of the instructions). Both phases are potential sources of inefficiency, as well as the CPU-wide impact of mispredicting the execution path. More specifically, bandwidth is analyzed at a micro-operation (μop) granularity, where instructions are typically translated into one μop , but some instructions require more than one μop . Lastly, because the categories are presented as fractions of sustainable instruction (μop) bandwidth, and sustainable bandwidth tends to increase from family to family, it is difficult to directly compare the fractions across families. For some software, the improvement in instructions (μops) per cycle is less than the increase in sustainable instruction (μop) bandwidth, resulting in higher fractions of inefficiency even though performance actually improves. Use the fractions to measure and guide optimization efforts on your CPU.

- **Useful:** The fraction of sustainable instruction bandwidth that retired, meaning it was on the correct path through the code, and therefore contributed to forward progress of the application. More specifically, this fraction directly relates to μops per cycle (μPC) and largely to instructions per cycle (IPC). Rather than presenting the measured μPC directly, this category shows the measured μPC divided by the maximum μPC for the particular core in order to provide a fraction of achievable μPC . A high fraction indicates that the code is executing in the CPU at a high rate. To improve performance, improve the efficiency of the algorithm or the instruction sequences you use to implement the algorithm.
- **Discarded bottleneck:** The fraction of sustainable instruction bandwidth wasted due to the effects of a branch misprediction or a pipeline restart. More specifically, this fraction represents μop bandwidth wasted while processing down the wrong predicted path, as well as lost bandwidth required to recover from the misprediction. A high fraction indicates that the control flow or memory dependencies or both are difficult to predict and the CPU is throwing away a significant amount of predicted work. To improve performance, focus on removing branches through conditional moves or altering data structures and decision trees for a more stable path through the code. Because the [instruction window](#) and overall speculative execution capability of the CPU tends to increase from family to family, the amount of discarded work due to similar misprediction rates may also increase.

- Further note that starting on the M4 and A18 families chips, the underlying CPU performance monitoring events more accurately account for bottlenecks in this category. The discarded bottleneck will likely appear larger than on previous families when running similar code.
- **Instruction Delivery bottleneck:** The fraction of instruction bandwidth lost because the Instruction Delivery component did not deliver instructions at an adequate rate for the Instruction Processing component. More specifically, the Instruction Delivery component reads bytes from memory, decodes them into instructions, and delivers them as μ ops into the Instruction Processing component's instruction window. This fraction represents lost bandwidth when the processing component has room in the instruction window, but new μ ops are not available from the predicted instruction stream. A high fraction indicates a higher Instruction Processing bandwidth might be possible if the Instruction Delivery component delivers new instructions faster. To improve performance, reduce instruction memory delays by improving the locality of hot functions, and inline, unroll, and straighten common code sequences to create longer streams of sequentially fetched instructions. For less common code, optimize for smaller code size to improve cache performance using `-Os` with C-based languages or `-Osize`.
- **Instruction Processing bottleneck:** The fraction of instruction bandwidth lost because the Instruction Processing component is executing instructions at a rate lower than the sustainable bandwidth. More specifically, this component maintains a forward-looking set of μ ops, called the instruction window, that the CPU collects from along a predicted instruction path and executes those μ ops as they become ready. This fraction represents μ op bandwidth lost when the instruction window becomes full due to slow processing. A high fraction indicates insufficient instruction-level parallelism in the code, and that the Instruction Delivery component is inserting instructions into the instruction window faster than they are completed. To improve performance, reduce memory delays and improve available instruction-level parallelism. For example, make data structure locality improvements, prefetch data from other cache lines, and unroll loops to create additional parallel computational sequences for Swift.

Bottleneck analysis is offered as a preconfigured analysis mode in the [Instruments](#) Xcode tool starting with Instruments 17.

Additional preconfigured analysis modes and sampling modes are available for deeper insight into each of these categories. See the Instruments tool for the full selection. The following two subsections describe the next-level preconfigured analysis mode for Instruction Processing bottleneck and Instruction Delivery bottleneck.

8.1.1.1 Instruction Delivery Bottleneck Analysis

The Instruction Delivery bottlenecks analysis examines the efficiency of the Instruction Delivery component of the CPU, which reads instruction bytes from memory, translates them into instructions, and delivers them as μ ops into the Instruction Processing component. This analysis further categorizes the sustainable bandwidth lost due to delivery while the Instruction Processing component is able to accept additional μ ops, with some categories related to latency issues and others related to reduced bandwidth issues. Unlike the Instruction Processing bottleneck analysis, which categorizes the *overall activity of the Instruction Processing component each cycle*, this analysis effectively *zooms in* on the Instruction Delivery bottleneck to reveal further categorization.

- **Instruction Delivery latency:** The fraction of sustainable bandwidth lost because the Instruction Delivery component experienced an event that delayed the delivery of the next instructions. To improve performance, inline common code sequences and consolidate frequently executed functions together in memory. For less common code, optimize for smaller code size to improve cache performance using `-Os` with C-based languages or `-Osize` for Swift.
- **Instruction Delivery latency from instruction cache:** The fraction of sustainable bandwidth lost because the Instruction Delivery component experienced an L1 Instruction Cache miss while reading instruction bytes from memory. This is a subset of the Instruction Delivery latency category. A high fraction is often associated with fetching small blocks of infrequently used instructions scattered around memory. To improve performance, inline common code sequences and consolidate frequently executed functions together in memory. For less common code, optimize for smaller code size to improve cache performance using `-Os` with C-based languages or `-Osize` for Swift.
 - Note that this category is available starting on the M3 family and A16 Bionic chips.
- **Instruction Delivery latency from instruction TLB:** The fraction of sustainable bandwidth lost because the Instruction Delivery component experienced an L1 Instruction Translation Lookaside Buffer (TLB) miss while reading instruction bytes from memory. This is a subset of the Instruction Delivery latency category. A high fraction is often associated with fetching instructions scattered across the memory space of the application. To improve performance, consolidate frequently executed functions together in memory and inline common code sequences to create integrated streams of instructions. For less common code, optimize for smaller code size to improve cache performance using `-Os` with C-based languages or `-Osize` for Swift.
 - Note that this category is available starting on the M3 family and A16 Bionic chips.
- **Instruction Delivery latency from other reason:** The fraction of sustainable bandwidth lost because the Instruction Delivery component experienced an event that delayed the delivery of the next instructions due to an undetermined cause. This is a subset of the Instruction Delivery latency category. In some cases, these delays might be artifacts of the other Instruction Delivery issues, and focusing on solutions for those issues can help with this category.
 - Note that this category is available starting on the M3 family and A16 Bionic chips.
- **Instruction Delivery bandwidth:** The fraction of the sustainable bandwidth lost because the Instruction Delivery component was unable to deliver μ ops at full bandwidth in a cycle. A high fraction is often associated with an instruction stream with too many taken branches, including calls and returns, leading to sequences that are shorter than the sustainable bandwidth per cycle. To improve performance, inline, unroll, and straighten common code sequences to create longer streams of sequentially fetched instructions. For example, replace simple branches that skip assignments with conditional select or set instructions.

8.1.1.2 Instruction Processing Bottleneck Analysis

Available on the M3 and A16 families of chips and later, the Instruction Processing bottlenecks analysis examines the efficiency of the Instruction Processing component

of the CPU. This component maintains a forward-looking set of instructions (called the instruction window) that it collects along a predicted instruction path, and executes those instructions as they become ready.

Note: This analysis categorizes the *overall activity of the Instruction Processing component each cycle*, when there is at least one μ op in the instruction window, into a number of categories. Unlike the Instruction Delivery bottleneck analysis which *zooms in* to reveal the cause of each lost delivery, this mode examines *all cycles where there is work to do* in the instruction window.

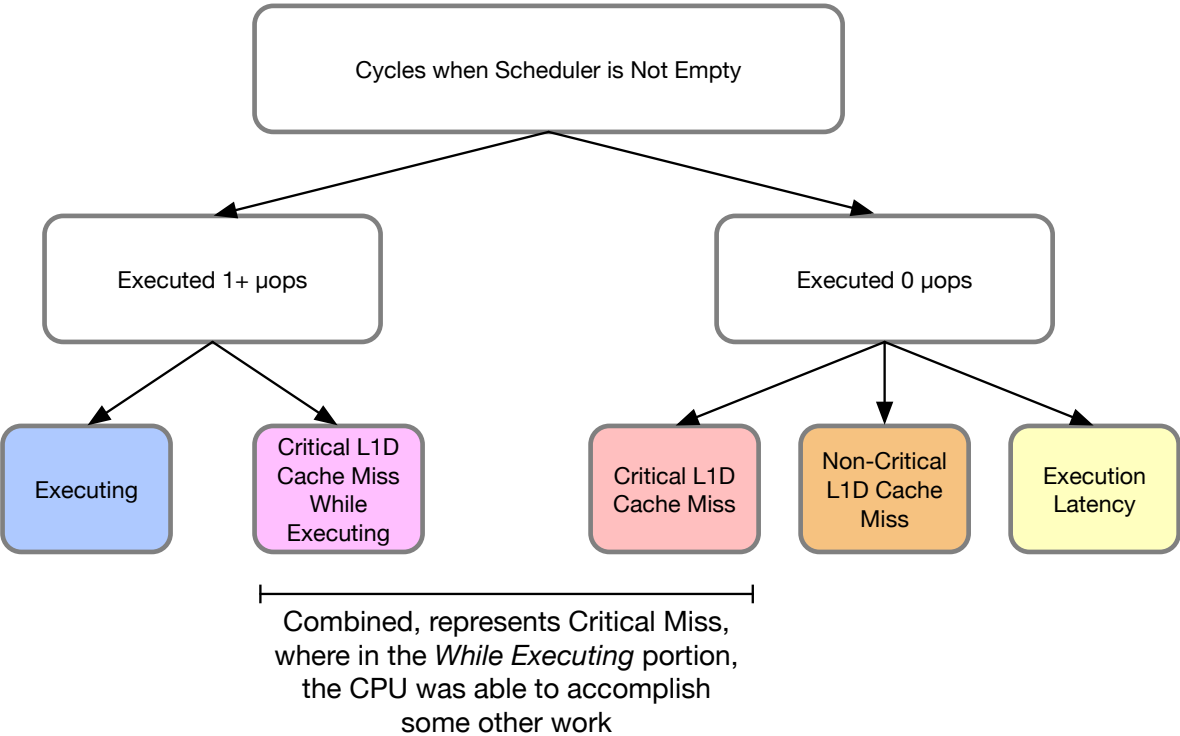
Specifically, this mode helps to analyze whether the workload is more compute-bound (bottlenecked on compute instruction bandwidth and latency) or memory-bound (bottlenecked on obtaining data from the memory hierarchy). Because of the large instruction instruction window, this is a complex question because the Instruction Processing component can often find *some* speculative work to do even while waiting for data to be supplied from a distant source, and can have requests outstanding for many (speculative) memory accesses at any time.

- **Executing:** The fraction of cycles where the instruction window contains μ ops, executed at least one of them, and the oldest load or store operations are not waiting on an L1D Cache miss. A high fraction indicates that the Instruction Processing component is making progress executing μ ops in the instruction window, but combined with a high Processing Bottleneck from the CPU Bottleneck Counting Mode, indicates there is insufficient parallelism to keep it very busy. To improve performance, reduce critical paths through code sequences and parallelize code to eliminate dependencies.
- **Critical L1D Cache Miss While Executing:** The fraction of cycles where the Instruction Processing component is executing at least 1 μ op from the instruction window, while the oldest tracked load or store operation is experiencing an L1D Cache miss. Combined with the Critical L1D Cache Miss category (below), the combined fraction represents the overall impact of a critical memory miss, and generally reflects the degree to which the workload is memory bound. A high fraction is often associated with scattered accesses to data memory such that there is poor cache-line reuse with access patterns that are difficult for the CPU to predict. However, the Instruction Processing component is still able to find μ ops to execute. To improve performance, reduce the working set of data and access your data in regular, strided patterns. For multi-threaded applications, ensure that independent variables that might be actively read by different threads are in separate 128B cache lines to avoid false sharing.
- **Critical L1D Cache Miss:** The fraction of cycles where the instruction window contains μ ops, did not execute any of them, and the oldest tracked load or store operation is experiencing an L1D Cache miss. Combined with the Critical L1D Cache Miss While Executing category (above), the combined fraction represents the overall impact of a critical memory miss, and generally reflects the degree to which the workload is memory bound. See above for recommendations on improving memory performance.
- **Non-Critical L1D Cache Miss:** The fraction of cycles where the instruction window contains μ ops, did not execute any of them, and a younger tracked load operation is experiencing an L1 Data Cache miss. The CPU is able to issue younger loads speculatively while waiting for other older work to complete.
- **Execution Latency:** The fraction of cycles where the instruction window contains μ ops, did not execute any of them, and the critical path through the code is limited

by instruction latencies (possibly including load instructions that hit in the L1D Cache). To improve performance, reduce critical paths through code sequences and parallelize code to eliminate dependencies.

The mode also contains an additional track that indicates whether the SME engine (available starting on the M4 and A18 families of chips) is used by the workload. If so, additional analysis modes may be available to explore interactions between the core and the engine.

Figure 8.1. Instruction Processing Bottleneck Categories



8.1.2 Performance Monitoring Events

Event names used throughout this document are denoted as <EV *event_name*> where the *event_name* is defined in [Table 8.1: “Performance Monitoring Events”](#) and follow this convention:

- **INST_*:** ISA instruction type profiling event. These events count when a matching instruction retires. By definition, these are nonspeculative.
- ***_NONSPEC:** Nonspeculative microarchitectural event. These events count when an instruction that caused or experienced the microarchitectural event retires.
- **(other):** Microarchitectural events. These events count when the event occurs. The observed PC at the time of the associated counter increment is not necessarily indicative of an instruction that caused or experienced the event. Often referred to as "speculative" since they count before it is known whether the related instructions retire or are cleared due to a misprediction.

To simplify the table, the CPU Support columns are named as follows:

Apple Silicon CPU Optimization Guide: 4.0

Xcode: Performance Monitoring and Analysis

Performance Monitoring Events

- **M1:** M1 family, A14 family
- **M2:** M2 family, A15 Bionic
- **M3:** M3 family, A16 Bionic, A17 Pro
- **M4:** M4 family, A18 family

Table 8.1. Performance Monitoring Events

Event Name	Brief Description	CPU Support			
		M1	M2	M3	M4
ARM_BR_MIS_PRED	Mispredicted or not predicted branch Speculatively executed				Y
ARM_BR_PRED	Predictable branch Speculatively executed				Y
ARM_L1D_CACHE	Level 1 data cache access				Y
ARM_L1D_CACHE_LMISS_RD	Level 1 data cache long-latency read miss				Y
ARM_L1D_CACHE_RD	Attributable Level 1 data cache access, read				Y
ARM_L1D_CACHE_REFILL	Level 1 data cache refill				Y
ARM_STALL	No operation sent for execution				Y
ARM_STALL_FRONTEND	No operation issued due to the frontend				Y
ARM_STALL_BACKEND	No operation issued due to the backend				Y
ARM_STALL_SLOT	No operation sent for execution on a slot				Y
ARM_STALL_SLOT_BACKEND	No operation sent for execution on a Slot due to the backend				Y
ARM_STALL_SLOT_FRONTEND	No operation sent for execution on a Slot due to the frontend				Y
ATOMIC_OR_EXCLUSIVE_FAIL	Atomic or exclusive instruction failed due to contention *For exclusives, incorrectly undercounts for exclusives when the cache line is initially found in shared state, however counts correctly for atomics	Y*	Y*	Y*	Y*
ATOMIC_OR_EXCLUSIVE_SUCC	Atomic or exclusive instruction successfully completed *For exclusives, incorrectly undercounts for exclusives when the cache line is initially found in shared state, however counts correctly for atomics	Y*	Y*	Y*	Y*
BRANCH_CALL_INDIR_MISPRED_NONSPEC	Retired indirect call instructions mispredicted	Y	Y	Y	Y
BRANCH_COND_MISPRED_NONSPEC	Retired conditional branch instructions that mispredicted	Y	Y	Y	Y
BRANCH_INDIR_MISPRED_NONSPEC	Retired indirect branch instructions including calls and returns that mispredicted	Y	Y	Y	Y
BRANCH_MISPRED_NONSPEC (maps to ARM_BR_MIS_PRED_RETIRED beginning on M4)	Retired branch instructions including calls and returns that mispredicted	Y	Y	Y	Y
BRANCH_RET_INDIR_MISPRED_NONSPEC	Retired return instructions that mispredicted	Y	Y	Y	Y
CORE_ACTIVE_CYCLE (maps to ARM_CPU_CYCLES beginning on M4)	Cycles while the core was active	Y	Y	Y	Y
FETCH_RESTART	Fetch Unit internal restarts for any reason. Does not include branch mispredicts	Y	Y	Y	Y

Table 8.1. Performance Monitoring Events (cont.)

Event Name	Brief Description	CPU Support			
		M1	M2	M3	M4
FLUSH_RESTART_OTHER_NONSPEC	Pipeline flush and restarts that were not due to branch mispredictions or memory order violations	Y	Y	Y	Y
INST_ALL (maps to ARM_INST_RETIRED beginning on M4)	All retired instructions	Y	Y	Y	Y
INST_BARRIER	Retired barrier instructions	Y	Y	Y	Y
INST_BRANCH (maps to ARM_BR_RETIRED beginning on M4)	Retired branch instructions including calls and returns	Y	Y	Y	Y
INST_BRANCH_CALL	Retired subroutine call instructions	Y	Y	Y	Y
INST_BRANCH_COND	Retired conditional branch instructions *On M3 and prior, only counts only B • cond instructions. On M4 and following, adds CBZ/CBNZ/TBZ/TBNZ instructions to form the complete set of conditional branch instructions	Y*	Y*	Y*	Y
INST_BRANCH_INDIR	Retired indirect branch instructions including indirect calls	Y	Y	Y	Y
INST_BRANCH_RET	Retired subroutine return instructions	Y	Y	Y	Y
INST_BRANCH_TAKEN	Retired taken branch instructions	Y	Y	Y	Y
INST_INT_ALU	Retired non-branch and non-load/store Integer Unit instructions	Y	Y	Y	Y
INST_INT_LD	Retired load Integer Unit instructions	Y	Y	Y	Y
INST_INT_ST	Retired store Integer Unit instructions; does not count DC ZVA (Data Cache Zero by VA).	Y	Y	Y	Y
INST_LDST	Retired load and store instructions; does not count DC ZVA (Data Cache Zero by VA).	Y	Y	Y	Y
INST_SIMD_ALU	Retired non-load/store Advanced SIMD and FP Unit instructions	Y	Y	Y	Y
INST_SIMD_ALU_VEC	Retired non-load/store Advanced SIMD instructions (including integer and floating point data types)		Y	Y	Y
INST_SIMD_LD	Retired load Advanced SIMD and FP Unit instructions	Y	Y	Y	Y
INST_SIMD_ST	Retired store Advanced SIMD and FP Unit instructions	Y	Y	Y	Y
INST_SME_ENGINE_ALU	Retired non-load/store SME engine instructions				Y
INST_SME_ENGINE_LD	Retired load SME engine instructions				Y
INST_SME_ENGINE_PACKING_FUSED	Retired non-load/store SME engine instructions that were packed with another to reduce instruction bandwidth to the SME engine				Y
INST_SME_ENGINE_SCALARFP	Retired scalar floating-point SME engine instructions				Y
INST_SME_ENGINE_ST	Retired store SME engine instructions				Y
INTERRUPT_PENDING	Cycles while an interrupt was pending because it was masked	Y	Y	Y	Y
L1D_CACHE_MISS_LD	Loads that missed the L1 Data Cache	Y	Y	Y	Y
L1D_CACHE_MISS_LD_NONSPEC	Retired loads that missed in the L1 Data Cache	Y	Y	Y	Y
L1D_CACHE_MISS_ST	Stores that missed the L1 Data Cache	Y	Y	Y	Y
L1D_CACHE_MISS_ST_NONSPEC	Retired stores that missed in the L1 Data Cache	Y	Y	Y	Y

Table 8.1. Performance Monitoring Events (cont.)

Event Name	Brief Description	CPU Support			
		M1	M2	M3	M4
L1D_CACHE_WRITEBACK	Dirty cache lines written back from the L1D Cache toward the Shared L2 Cache	Y	Y	Y	Y
L1D_TLB_ACCESS	Load and store accesses to the L1 Data TLB	Y	Y	Y	Y
L1D_TLB_FILL	Translations filled into the L1 Data TLB	Y	Y	Y	Y
L1D_TLB_MISS	Load and store accesses that missed the L1 Data TLB	Y	Y	Y	Y
L1D_TLB_MISS_NONSPEC	Retired loads and stores that missed in the L1 Data TLB	Y	Y	Y	Y
L1I_CACHE_MISS_DEMAND (maps to ARM_L1I_CACHE_LMISS beginning on M4)	Demand instruction fetches that missed in the L1 Instruction Cache	Y	Y	Y	Y
L1I_TLB_FILL	Translations filled into the L1 Instruction TLB	Y	Y	Y	Y
L1I_TLB_MISS_DEMAND	Demand instruction fetches that missed in the L1 Instruction TLB	Y	Y	Y	Y
L2_TLB_MISS_DATA	Loads and stores that missed in the L2 TLB	Y	Y	Y	Y
L2_TLB_MISS_INSTRUCTION	Instruction fetches that missed in the L2 TLB	Y	Y	Y	Y
LD_BLOCKED_BY_SME_LDST	Core load μ ops blocked by SME accesses to same 4KiB page				Y
LD_NT_UOP	Load μ ops that executed with non-temporal hint; excludes SSVE/SME loads because they utilize the Store Unit	Y	Y	Y	Y
LD_SME_MEM_ORDER_VIOL_LD_NONSPEC	Retired SME loads that triggered memory order violations with core loads				Y
LD_SME_NORMAL_UOP	SME engine load μ ops with Normal memory type				Y
LD_SME_NT_UOP	SME engine load μ ops that executed with non-temporal hint				Y
LD_UNIT_UOP	μ ops that flowed through the Load Unit	Y	Y	Y	Y
LD_UNIT_WAITING YOUNG_L1D_CACHE_MISS	Cycles while a younger load μ op is waiting for data after an L1 Data Cache miss, and no μ op was issued by the scheduler with no critical miss, prioritized				Y
LDST_SME_PRED_INACTIVE	SME engine load and store μ ops where all lanes are inactive due to the governing predicate; for a page-crossing load or store, the event may incorrectly count when all of the elements of the low page are predicated off, even if some of the elements on the high page are active. In Apple silicon cores, where predication is recommend primarily for data structure edge control (discarding elements 'past the end of the data structure'), this scenario should not be common.				Y
LDST_SME_XPG_UOP	SME engine load and store accesses that crossed a 16KiB page boundary; an access is considered cross-page if any bytes are accessed in the high portion (second page), regardless if any bytes are accessed in the low portion (first page), after predication is applied. An SME operation that only touches the low portion (first page) after predication is applied is not considered cross-page.				Y
LDST_UNIT_OLD_L1D_CACHE_MISS	Cycles while an old load or store μ op is waiting for data after an L1 Data Cache miss			Y	Y

Table 8.1. Performance Monitoring Events (cont.)

Event Name	Brief Description	CPU Support			
		M1	M2	M3	M4
LDST_UNIT_WAITING_OLD_L1D_CACHE_MISS	Cycles while an old load or store μ op is waiting for data after an L1 Data Cache miss, and no μ op was issued by the scheduler, prioritized			Y	Y
LDST_UNIT_WAITING_SME_ENGINE_INST_QUEUE_FULL	Cycles while the instruction queue to the SME engine is full, and no μ op was issued by the scheduler, prioritized				Y
LDST_WAITING_SME_ENGINE_MEM_DATA	Cycles while the core is waiting for the SME engine to produce memory data, and no μ op was issued by the scheduler, prioritized				Y
LDST_X64_UOP	Load and store μ ops that crossed a 64B boundary	Y	Y	Y	Y
LDST_XPG_UOP	Load and store μ ops that crossed a 16KiB page boundary; an SME access is considered cross-page if any bytes are accessed in the high portion (second page), regardless if any bytes are accessed in the low portion (first page), after predication is applied. An SME operation that only touches the low portion (first page) after predication is applied is not considered cross-page.	Y	Y	Y	Y
MAP_DISPATCH_BUBBLE	Cycles while the Map Unit was not stalled and Decode Unit did not send any μ ops	Y	Y	Y	Y
MAP_DISPATCH_BUBBLE_IC	Cycles while the Map Unit had no μ ops to process due to L1 Instruction Cache and was not stalled			Y	Y
MAP_DISPATCH_BUBBLE_ITLB	Cycles while the Map Unit had no μ ops to process due to L1 Instruction TLB and was not stalled			Y	Y
MAP_DISPATCH_BUBBLE_SLOT	Slots where the Map Unit had no μ ops to process and was not stalled				Y
MAP_INT_SME_UOP	Mapped core Integer Unit μ ops for SME engine instructions				Y
MAP_INT_UOP	Mapped Integer Unit μ ops	Y	Y	Y	Y
MAP_LDST_UOP	Mapped Load and Store Unit μ ops, including GPR to vector register converts; includes all instructions sent to the SME engine because they are processed through the Store Unit	Y	Y	Y	Y
MAP_RECOVERY	Cycles while the Map Unit was stalled while recovering from a flush and restart				Y
MAP_REWIND	Cycles while rewinding the Map Unit due to flush and restart	Y	Y	Y	Y
MAP_SIMD_UOP	Mapped Advanced SIMD and FP Unit μ ops	Y	Y	Y	Y
MAP_STALL	Cycles while the Map Unit was stalled for any reason	Y	Y	Y	Y
MAP_STALL_DISPATCH	Cycles while the Map Unit was stalled because of Dispatch back pressure	Y	Y	Y	Y
MAP_STALL_NONRECOVERY	Cycles while the Map Unit was stalled for any reason other than recovery				Y
MAP_UOP (maps to ARM_OP_SPEC beginning on M4)	Mapped μ ops			Y	Y
MMU_TABLE_WALK_DATA	Table walk memory requests on behalf of data accesses	Y	Y	Y	Y
MMU_TABLE_WALK_INSTRUCTION	Table walk memory requests on behalf of instruction fetches	Y	Y	Y	Y
RETIRE_UOP	All retired μ ops	Y	Y	Y	Y

Table 8.1. Performance Monitoring Events (cont.)

Event Name	Brief Description	CPU Support			
		M1	M2	M3	M4
(maps to ARM_OP_RETIRED beginning on M4)					
SCHEDULE_EMPTY	Cycles while the μ op scheduler is empty	Y	Y	Y	Y
SCHEDULE_UOP	μ ops issued by the scheduler to any execution unit	Y	Y	Y	Y
SCHEDULE_UOP_ANY	Cycles while the μ op scheduler issued at least 1 μ op to any execution unit			Y	Y
SCHEDULE_WAITING_SME_ENGINE_REG_DATA	Cycles while the core is waiting for register, predicate, or flag data from the SME engine, and no μ op was issued by the scheduler, prioritized				Y
SME_ENGINE_SM_ENABLE	Transitions into SME engine Streaming Mode (PSTATE.SM: 0 to 1)				Y
SME_ENGINE_SM_ZA_ENABLE	Simultaneous transitions into SME engine Streaming Mode and ZA Mode (PSTATE.SM: 0 to 1 and PSTATE.ZA: 0 to 1)				Y
SME_ENGINE_ZA_ENABLED_SM_DISABLED	Cycles while SME engine ZA Mode is enabled but Streaming Mode is not (PSTATE.ZA=1 and PSTATE.SM=0)				Y
ST_BARRIER_BLOCKED_BY_SME_LDST	Core store μ ops blocked by SME accesses to same 4KiB page, and any barriers or store-release μ ops blocked by SME accesses				Y
ST_MEMORY_ORDER_VIOLATION_NONSPEC	Retired core store μ ops that triggered memory order violations with core load μ ops	Y	Y	Y	Y
ST_NT_UOP	Store μ ops that executed with non-temporal hint; includes SSVE/SME loads because they utilize the Store Unit	Y	Y	Y	Y
ST_SME_MEM_ORDER_VIOL_LD_NONSPEC	Retired SME stores that triggered memory order violations with core loads				Y
ST_SME_NORMAL_UOP	SME engine store μ ops with Normal memory type				Y
ST_SME_NT_UOP	SME engine store μ ops that executed with non-temporal hint				Y
ST_UNIT_UOP	μ ops that flowed through the Store Unit	Y	Y	Y	Y

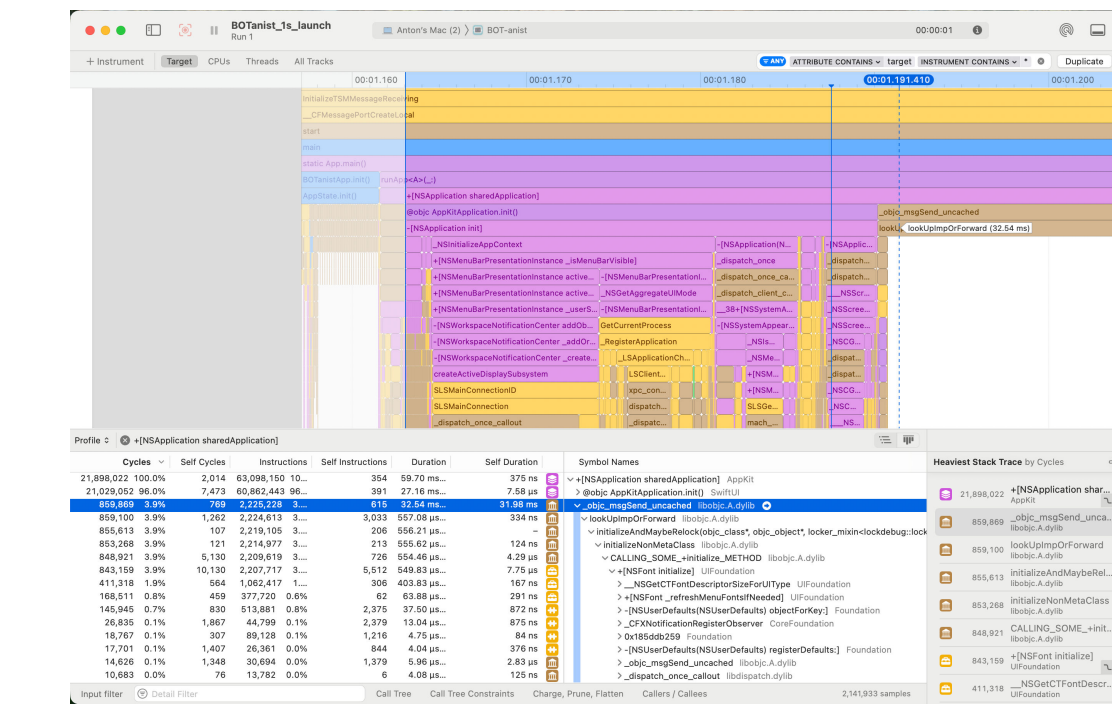
8.2 Processor Trace

Instruments 16.3 includes a new Processor Trace Instrument that uses hardware-supported, low-overhead CPU execution tracing to accurately reconstruct execution of the ELO (unprivileged, user code) portions of a program. This tool provides metrics like duration, number of cycles, and instructions retired for every function executed on the CPU. Timeline in Instruments presents an execution flame graph (see [Figure 8.2: “Example of a Processor Trace Flame Graph”](#)) and detail views provide aggregate-level data like Call Tree or metrics (min, max, count, sum), divided by function. Unlike sampling profilers that can miss critical code paths between sampling intervals, this tool reveals every function call, including compiler-generated code like ARC memory management in Swift and C++ constructors.

Use the Processor Trace instrument to gather information about the functions your code runs while you perform particular activities in your app. Analyze slow or processor-intensive sequences to identify whether your app might be able to process its data in more efficient ways, such as by using different algorithms or caching the results of calculations to use again.

For more details on using Processor Trace, see [Analyzing CPU Usage with Processor Trace Instrument](#).

Figure 8.2. Example of a Processor Trace Flame Graph





Appendix A. Instruction Latency and Bandwidth

The details covered in [Chapter 5, Core Microarchitecture Optimization](#), [Chapter 6, SME Engine Microarchitecture Optimization](#), and [Appendix A, Instruction Latency and Bandwidth](#) represent the most important information to consider when optimizing software. The CPUs employ additional techniques beyond what is described here that may reduce the latency of an instruction sequence. Similarly the CPU may have other internal resource or datapath limitations that may increase latency under certain less common conditions. These may change across chip families, and it may be difficult or counter-productive to try to transform software to match the conditions. This appendix describes the foundational latency for the operations from which code sequences should be developed.

Operations are divided into three categories for the core (integer, Advanced SIMD and FP, and load and store) and four categories for the SME engine (SSVE vector including SSFP, SME grid, SME extraction, and load and store). Instructions often map directly to single μ ops, thus instruction mnemonics mentioned serve as proxies for the relevant μ ops.

For each category, the Operation Class table presents descriptions with latencies and instruction mnemonics. The list of example mnemonics is not necessarily exhaustive. Use the descriptions and examples to determine the Operation Class for instructions of interest. Operation Class table contents apply to both P cores and E cores, except where noted. The Operation Latency column provides the primary latency of the operation, and may be differentiated between P cores and E cores. Latencies inside [x] provide an alternate latency typically through a specific operand or a special condition, and are described in the Operation Types and Descriptions column.

Note that operation latencies describe the number of cycles required to complete execution of the operation. In the best case, consumer operations begin execution immediately after producer operations complete. However, there are numerous reasons consumer operations may be delayed, including but not limited to:

- Consumer operation entering the instruction window
- Other input operands
- Unavailable internal microarchitectural resources
- Limited datapath bypassing/forwarding paths

Operation Class Availability tables provide the mapping of operation classes to execution units for the P and E cores to describe operation bandwidth and concurrence.

A.1 Core Integer Execution Units

Integer operations are divided into classes, as described in [Table A.1: "Integer Execution Classes"](#).

Note that the Load and Store Execution Unit handles movement of data from general purpose registers to Advanced SIMD and FP (vector) registers. See [Section 5.5.1, “Movement of Data from General Purpose Registers to Vector Registers”](#).

Table A.1. Integer Execution Classes

Class	Latency	Operation Types and Descriptions (Example Instruction Mnemonics)
ALU	1[2]	General purpose register and immediate value to general purpose register moves (MOVK, MOVN, MOVZ, ORR (with XZR)) Arithmetic and logic operations that do not read or write flags (ADD, ADR, ADRP, AND, BIC, EON, EOR, ORR, ORN, SUB). Note: Multiplies and divides are in separate classes Shift and bitfield operations (ASRV, CLS, CLZ, EXTR, LSLV, LSRV, RBIT, REV, REV16, REV32, RORV, SBFM, UBFM) Note: BFM is not included in this class P core: Beginning with the M4 and A18 families of chips, the accumulate portion of multiply-accumulate (i.e., multiply-add/sub) operations (MADD, MSUB, SMADDL, SMSUBL, UMADDL, UMSUBL) Extract register from a pair of registers (EXTR). Note in cases where the two input registers are different, the instruction requires an additional single-cycle MISC-class μop Writeback portion of pre-index and post-index memory addressing modes Operation portion of LD-op and ST-op instructions, including all flavors of acquire and release. See Table A.11 [2]: Non-moves that operate on a shifted or extended register operand occupy the unit for 2 cycles (depipelined)
ALUf (ALU/f)	1[2]	Arithmetic and logic operations that consume or produce flags (ADC, ADCS, ADDS, ANDS, AXFLAG, BICS, CCMN, CCMP, CFINV, CMN, CMP, CSEL, CSINC, CSINV, CSNEG, RMIF, SBC, SBCL, SETF8, SETF16, SUBS, TST, XAFLAG) [2]: Operations that operate on a shifted or extended register operand occupy the unit for 2 cycles (depipelined) ALU/f: Notation for combined ALU + ALUf unit
BRud	-	Unconditional direct branches (B, BL) Note: These instructions do not require an execution unit and thus do not appear in the availability tables
BRc	1	Conditional branches (B.cond, CBZ, CBNZ, TBZ, TBNZ)
BRi (BRc/i)	1	Indirect branches, including returns and indirect calls (BLR, BR, RET) BRc/i: Notation for combined BRc + BRi unit
MAC	3[1]	Multiply-accumulate (aka Multiply-add/sub) operations (MADD, MSUB, SMADDL, SMSUBL, UMADDL, UMSUBL) [1]: The accumulator input of these instructions has a latency of 1 cycle if it comes directly from the output of a previous multiply-accumulate instruction P core: This execution class has been removed starting on the M4 and A18 families of chips. Instead, these instructions are cracked into a MUL-class operation and an ALU-class operation
MUL	3	Multiplies without additional add/sub operation (MUL, MNEG, SMNEGL, SMULL, SMULH, UMNEGL, UMULH, UMULL)

Table A.1. Integer Execution Classes (cont.)

Class	Latency	Operation Types and Descriptions (Example Instruction Mnemonics)
		P core: Beginning with the M4 and A18 families of chips, the multiply portion of multiply-accumulate (aka Multiply-add/sub) operations (MADD, MSUB, SMADDL, SMSUBL, UMADDL, UMSUBL)
DIV	Various	Divides (SDIV, UDIV) P core: 32b: 7-8 cycles, 64b: 7-9 cycles. A new divide can be issued every other cycle E core: 32b: 7-13 cycles, 64b: 7-21 cycles. A new divide can be issued only once the previous divide is complete
MISC	1	Bitfield insert with merge (BFM)
	3	CRC calculation (CRC32*)
PRED/f	3[6][7]	Predicate logical operations, including those that write flags (AND, ANDS, BIC, BICS, EOR, EORS, MOV, MOVs, NAND, NANDS, NOT, NOTS, ORN, ORNS, ORR, ORRS, SEL) P: Instructions using the form Pd, Pg/Z, Pn, Pm require 2 μops with latency 6 through Pn and Pm, and 3 through Pg P: Instructions using the form Pd, Pg, Pn, Pm require 3 μops with latency 7 through Pn and Pg, and 6 through Pm Arithmetic by element counts (DECB, DECD, DECH, DECP, DECW, INCB, INCD, INCH, INCP, INCW, SQDECB, SQDECD, SQDECH, SQDECP, SQDECW, SQINCB, SQINCD, SQINCH, SQINCP, SQINCW, UQDECB, UQDECD, UQDECH, UQDECP, UQDECW, UQINCB, UQINCD, UQINCH, UQINCP, UQINCW) Loop control (BRKA, BRKAS, BRKB, BRKBS, BRKN, BRKNS, BRKPA, BRKPAS, BRKPB, BRKPBS, CTERMEO, CTERMNE, WHILEGE, WHILEGT, WHILEHI, WHILEHS, WHILELE, WHILELO, WHILELS, WHILELT, WHILERW, WHILEWR) WHILE* instructions that write {Pd1, Pd2}: Requires 2 independent μops each with latency 3 P: BRK{AB}{S}: Instructions using the form Pd, Pg/M, Pn require 3 μops with latency 7 through Pn and Pg, and 6 through Pm P: BRKN{S}: Instructions using the form Pdm, Pg/Z, Pn, Pdm require 2 μops with latency 6 through Pn and Pdm, and 3 through Pg Miscellaneous (ADDPL, ADDVL, ADDSPL, ADDSVL, CNTB, CNTD, CNTH, CNTP, CNTW, PEXT, PFIRST, PNEXT, PSEL, PTEST, PUNPKHI, PUNPKLO, RDVL, RDSVL, REV, TRN1, TRN2, UZP1, UZP2, ZIP1, ZIP2) PEXT that write {Pd1, Pd2}: Requires 2 independent μops each with latency 3 P: PSEL: Instructions using the form Pd, Pn, Pm[Wv, imm] require 2 μops with latency 6 through Pm and Wv, and 3 through Pn

The following instructions serialize the instruction delivery pipeline according to their conditions: ISB, SB.

Table A.2. Integer Execution Class Availability: M-Series Chips

Chip	Integer Execution Unit											
	Performance Cores								Efficiency Cores			
	0	1	2	3	4	5	6	7	0	1	2	3
M1 Family	ALU/f BRc/i	ALU/f BRc	ALU/f	ALU MUL MAC MISC	ALU MUL DIV	ALU			ALU/f MUL MAC MISC	ALU/f BRi DIV	ALU/f BRc	
M2 Family	ALU/f BRc/i	ALU/f BRc	ALU/f	ALU MUL MAC MISC	ALU MUL DIV	ALU			ALU/f MUL MAC MISC	ALU/f BRi DIV	ALU/f BRc	ALU/f
M3 Family	ALU/f BRc/i	ALU/f BRc	ALU/f	ALU/f	ALU MUL MAC MISC	ALU MUL DIV	ALU	ALU	ALU/f MUL MAC MISC	ALU/f BRi DIV	ALU/f BRc	ALU/f
M4 Family	ALU/f BRc/i	ALU/f BRc	ALU/f	ALU/f PRED/f	ALU MUL MISC	ALU DIV	ALU MUL	ALU MUL	ALU/f MUL MAC MISC PRED/f	ALU/f BRi DIV	ALU/f BRc	ALU/f

Table A.3. Integer Execution Class Availability: A-Series Chips

Chip	Integer Execution Unit											
	Performance Cores								Efficiency Cores			
	0	1	2	3	4	5	6	7	0	1	2	3
A14 Bionic	ALU/f BRc/i	ALU/f BRc	ALU/f	ALU MUL MAC MISC	ALU MUL DIV	ALU			ALU/f MUL MAC MISC	ALU/f BRi DIV	ALU/f BRc	
A15 Bionic	ALU/f BRc/i	ALU/f BRc	ALU/f	ALU MUL MAC MISC	ALU MUL DIV	ALU			ALU/f MUL MAC MISC	ALU/f BRi DIV	ALU/f BRc	ALU/f
A16 Bionic	ALU/f BRc/i	ALU/f BRc	ALU/f	ALU MUL MAC MISC	ALU MUL DIV	ALU			ALU/f MUL MAC MISC	ALU/f BRi DIV	ALU/f BRc	ALU/f
A17 Pro	ALU/f BRc/i	ALU/f BRc	ALU/f	ALU/f	ALU MUL MISC	ALU MUL DIV	ALU	ALU	ALU/f MUL MAC MISC	ALU/f BRi DIV	ALU/f BRc	ALU/f
A18 Family	ALU/f BRc/i	ALU/f BRc	ALU/f	ALU/f PRED/f	ALU MUL MISC	ALU DIV	ALU MUL	ALU MUL	ALU/f MUL MAC MISC PRED/f	ALU/f BRi DIV	ALU/f BRc	ALU/f

A.2 Core Advanced SIMD and FP Execution Units

Advanced SIMD and FP operations are divided into classes, as described in [Table A.4: “Advanced SIMD and FP Execution Classes”](#). The Advanced SIMD and FP execution units feature floating-point scalar and SIMD types as well as integer SIMD types. Integer type instructions in this section are not the same as integer registers and execution units described in the previous section.

As noted in [Section 2.6, “Registers”](#), the Advanced SIMD and FP registers and related instructions are often simply referred to as "Vector" registers and instructions. In the context of this document, the term vector doesn't imply only SIMD operations, but does include scalar floating-point operations which use the same registers and datapath.

Note that the Load and Store Execution Unit handles movement of data from general purpose registers to Advanced SIMD and FP (vector) registers. See [Section 5.5.1, “Movement of Data from General Purpose Registers to Vector Registers”](#).

Table A.4. Advanced SIMD and FP Execution Classes

Class	Latency	Operation Types and Descriptions (Example Instruction Mnemonics)
GENERAL	2-8	Wide range of arithmetic, logic, shift, vector-vector moves, and multiplies or both integer and floating point types. See Table A.5: "GENERAL Advanced SIMD and FP Class Details" for further breakdown
MOVE2GPR	P: 3-4 E: 3	Vector to general purpose registers moves (FMOV, SMOV, UMOV) Move portion of vector to general purpose register converts (FCVT (AMNPZ) (SU)) P core: S, D, and D [1] subsets of FMOV and FCVT* have a latency of 3 cycles. Others are 4 cycles
FCMPf	P: 5[9] E: 4[8]	Flag producing FP comparison operations (FCCMP, FCCMPE, FCMP, FCMPE) [9/8]: Latency through the input condition flags is +4 cycles for FCCMP*
FCSElf	2[6]	Flag consuming FP select operations (FCSEL) [6]: Latency through the input condition flags is 6 cycles for FCSEL
FDIV	Various	Divides (FDIV): P core: 1 per cycle, 7 cycles for H sizes, 8 cycles for S sizes, 10 cycles for D sizes E core: Same as P core, but for 128b vectors: 1 per 2 cycles with +1 cycle latency Square roots (FSQRT): P core: 1 per 2 cycles, 8 cycles for H sizes, 10 cycles for S sizes, 13 cycles for D sizes E core: Same as P core, but for 128b vectors: 1 per 4 cycles with +2 cycle latency Reciprocal / reciprocal square root estimates (FRECPE, FRECPX, FRSQRTE, URECPE, URSQRTE): P core: 3 cycles E core: 4 cycles Note that reciprocal / reciprocal square root steps (FRECPS, FRSQRTS) are multiply operations covered in the GENERAL class
MUL	3	Integer multiplies, multiply+accumulates, dot products, polynomial multiplies (MLA, MLS, MUL, PMUL, PMULL, PMULL2, SDOT, SUDOT, SMLAL, SMLAL2, SMLSL, SMLSL2, SMMLA, SMULL, SMULL2, SQDMLAL, SQDMLAL2, SQDMLSL, SQDMLSL2, SQDMULL, SQDMULL2, SQDMULH, SQRDMULH, SQRDMULAH, SQRDMULSH, UDOT, UMLAL, UMLAL2, UMLSL, UMLSL2, UMMLA, UMULL, UMULL2, USDOT, USMMLA)
	4	FP/BF16 multiplies and multiply+adds/accumulates (BFMLALB, BFMLALT, FCMLA, FMADD, FMLA, FMLAL, FMLAL2, FMLSL, FMLSL2, FMLS, FMSUB, FMUL, FMULX, FNMADD, FNMSUB, FNMUL, FRECPX, FRSQRTS)
	P: 10 E: 12	Brain floating point (bf16) dot product (BFDOT) Note this is a cracked/microcoded flow. See Section 5.4.8, "Multi-µop Instructions"
	P: 13 E: 16	Brain floating point (bf16) matrix multiply-accumulate (BFMMLA) Note this is a cracked/microcoded flow. See Section 5.4.8, "Multi-µop Instructions"
SHA	2-5	SHA1, SHA2-256, SHA2-512 cryptography 2 cycle: SHA1H, SHA1SU0, SHA1SU1, SHA256SU0, SHA512SU0, SHA512SU1 3 cycle: SHA256SU1, SHA512H, SHA512H2 5 cycle: SHA1C, SHA1M, SHA1P, SHA256H, SHA256H2

Table A.4. Advanced SIMD and FP Execution Classes (cont.)

Class	Latency	Operation Types and Descriptions (Example Instruction Mnemonics)
		Note that SHA3-related logic instructions (BCAX, EOR3, RAX1, XAR) are included in the GENERAL class

Table A.5. GENERAL Advanced SIMD and FP Class Details

Type	Latency	Operation Types and Descriptions (Example Instruction Mnemonics)
MOVE	2	Vector register/immediate to vector register moves (without conversion), including transpose and interleave (DUP, FMOV, INS, MOVI, MVN, MVNI, TRN1, TRN2, UZP1, UZP2, ZIP1, ZIP2)
	2-8	Table lookup with and without insert (TBL, TBX) Note these are cracked/microcoded flow with varying latency depending on source operand. See Section 5.4.8, “Multi-μop Instructions”
INT	2	Simple arithmetic and logic with a single operation per destination element including widen and lengthen (ADD, ADDP, AND, BIC, CMTST, EOR, MVN(reg), NEG, NOT, ORN ORR, SADDL, SADDL2, SADDLP, SADDW, SADDW2, SHADD, SHSUB, SRHADD, SSUBL, SSUBL2, SSUBW, SSUBW2, SUB, UADDL, UADDL2, UADDLP, UADDW, UADDW2, UHADD, UHSUB, URHADD, USUBL, USUBL2, USUBW, USUBW2) Simple shifts, shifts and inserts, bit selection, bit count, reversals, simple extract and narrow (BIF, BIT, BSL, CLS, CLZ, CNT, EXT, RBIT, REV16, REV32, REV64, SHL, SHLL, SHLL2, SLI, SQSHL, SQSHLU, SRI, SSHL, SSHLL, SSHLL2, SSHR, REV64, UQSHL, USHL, USHLL, USHLL2, USHR, XTN, XTN2) SHA3-related logic operations (BCAX, EOR3, RAX1, XAR)
	P: 2 E: 3	Arithmetic comparison with per element write (CMEQ, CMGE, CMGT, CMHI, CMHS, CMLE, CMLT, CMHI, SMAX, SMAXP, SMIN, SMINP, UMAX, UMAXP, UMIN, UMINP)
	3	Complex operations such as absolute value and absolute difference including lengthen (ABS, SABD, SABDL, SABDL2, UABD, UABDL, UABDL2) Horizontals (ADDV, SADDLV, SMAXV, SMINV, UADDLV, UMAXV, UMINV) Simple + saturating (SQADD, SQABS, SQNEG, SQSUB, SUQADD, USQADD, UQADD, UQSUB) Simple + rounding (QQRSHL, SRSHL, SRSHR, UQRSHL, URSHL, URSHR) Simple + accumulate (SABA, SABAL, SABAL2, SADALP, SRSRA, SSRA, UABA, UABAL, UABAL2, UADALP, URSRA, USRA)
	P: 3 E: 4	Simple + narrowing (ADDHN, ADDHN2, RADDHN, RADDHN2, RSHRN, RSHRN2, RSUBHN, RSUBHN2, SHRN, SHRN2, QQRSHRN, QQRSHRN2, QQRSHRUN, QQRSHRUN2, SQSHRN, SQSHRN2, SQSHRUN, SQSHRUN2, SQXTN, SQXTN2, SQXTUN, SQXTUN2, SUBHN, SUBHN2, UQRSHRN, UQRSHRN2, UQSHRN, UQSHRN2, UQXTN, UQXTN2)
FP/BF16	2	Comparisons, 2-operand comparison-based operations, negation (FABS, FACGE, FACGT, FCMEQ, FCMGE, FCMGT, FCMLE, FCMLT, FMAX, FMAXP, FMAXNM, FMAXNMP, FMIN, FMINP, FMINNM, FMINNMP, FNEG) Note: comparisons and comparison-based operations that involve the flags are in a separate class
	3	Full vector horizontal min-max (FMAXV, FMAXNMV, FMINV, FMINNMV) Full arithmetic (FABD, FADD, FADDP, FCADD, FSUB)

Table A.5. GENERAL Advanced SIMD and FP Class Details (cont.)

Type	Latency	Operation Types and Descriptions (Example Instruction Mnemonics)
CVT	3[2] [+4/5] [+4/3]	Conversion operations between integer and floating point types, and floating point rounding operations (BFCVT, BFCVTN, BFCVTN2, FCVT, FCVT (AMNPZ) (SU), FCVTL, FCVTL2, FCVTN, FCVTN2, FCVTXN, FRINT (AIMNPXZ), FRINT32 (XZ), FRINT64 (XZ), SCVTF, UCVTF) [2]: Simple lengthens (FCVT, FCVTL, FCVTL2) on E core are 2 cycles [+4/5]: Four or five additional cycles for a MOVE2VEC operation (if the instruction requires movement from a GPR), depending on microarchitectural condition. See Section 5.5.1, “Movement of Data from General Purpose Registers to Vector Registers” [+4/+3]: Four or three additional cycles for a MOVE2GPR operation (if the instruction requires movement to a GPR). See MOVE2GPR in Table A.4 for details
AESCRYPT	Various	AES Cryptography (AESD, AESE, AESIMC, AESMC): P core: 3 cycles E core: 5 cycles, 1 per 2 cycles

Table A.6. Advanced SIMD and FP Execution Class Availability: M-Series Chips

Chip	Advanced SIMD and FP Execution Unit						
	Performance Cores				Efficiency Cores		
	0	1	2	3	0	1	2
M1 Family	GENERAL MOVE2GPR FCMPf FCSElf FDIV MUL SHA	GENERAL MOVE2GPR FCSElf MUL	GENERAL MUL	GENERAL MUL	GENERAL MOVE2GPR FCMPf FCSElf FDIV MUL SHA	GENERAL FCSElf MUL	
M2 Family	GENERAL MOVE2GPR FCMPf FCSElf FDIV MUL SHA	GENERAL MOVE2GPR FCSElf MUL	GENERAL MUL	GENERAL MUL	GENERAL MOVE2GPR FCMPf FCSElf FDIV MUL SHA	GENERAL FCSElf MUL	
M3/M3 Pro	GENERAL MOVE2GPR FCMPf FCSElf FDIV MUL SHA	GENERAL MOVE2GPR FCMPf FCSElf MUL	GENERAL MUL	GENERAL MUL	GENERAL MOVE2GPR FCMPf FCSElf FDIV MUL SHA	GENERAL FCSElf MUL	
M3 Max	GENERAL MOVE2GPR FCMPf FCSElf FDIV MUL SHA	GENERAL MOVE2GPR FCMPf FCSElf MUL	GENERAL MUL	GENERAL MUL	GENERAL MOVE2GPR FCMPf FCSElf FDIV MUL SHA	GENERAL MOVE2GPR FCMPf FCSElf MUL	GENERAL
M4 Family	GENERAL MOVE2GPR FCMPf FCSElf FDIV MUL SHA	GENERAL MOVE2GPR FCMPf FCSElf MUL	GENERAL MUL	GENERAL MUL	GENERAL MOVE2GPR FCMPf FCSElf FDIV MUL SHA	GENERAL MOVE2GPR FCMPf FCSElf MUL	GENERAL

Table A.7. Advanced SIMD and FP Execution Class Availability: A-Series Chips

Chip	Advanced SIMD and FP Execution Unit						
	Performance Cores				Efficiency Cores		
	0	1	2	3	0	1	2
A14 Bionic	GENERAL MOVE2GPR FCMPf FCSElf FDIV MUL SHA	GENERAL MOVE2GPR FCSElf MUL	GENERAL MUL	GENERAL MUL	GENERAL MOVE2GPR FCMPf FCSElf FDIV MUL SHA	GENERAL FCSElf MUL	
A15 Bionic	GENERAL MOVE2GPR FCMPf FCSElf FDIV MUL SHA	GENERAL MOVE2GPR FCSElf MUL	GENERAL MUL	GENERAL MUL	GENERAL MOVE2GPR FCMPf FCSElf FDIV MUL SHA	GENERAL FCSElf MUL	
A16 Bionic	GENERAL MOVE2GPR FCMPf FCSElf FDIV MUL SHA	GENERAL MOVE2GPR FCMPf FCSElf MUL	GENERAL MUL	GENERAL MUL	GENERAL MOVE2GPR FCMPf FCSElf FDIV MUL SHA	GENERAL FCSElf MUL	
A17 Pro	GENERAL MOVE2GPR FCMPf FCSElf FDIV MUL SHA	GENERAL MOVE2GPR FCMPf FCSElf MUL	GENERAL MUL	GENERAL MUL	GENERAL MOVE2GPR FCMPf FCSElf FDIV MUL SHA	GENERAL MOVE2GPR FCMPf FCSElf MUL	GENERAL
A18 Family	GENERAL MOVE2GPR FCMPf FCSElf FDIV MUL SHA	GENERAL MOVE2GPR FCMPf FCSElf MUL	GENERAL MUL	GENERAL MUL	GENERAL MOVE2GPR FCMPf FCSElf FDIV MUL SHA	GENERAL MOVE2GPR FCMPf FCSElf MUL	GENERAL

A.3 Core Load and Store Execution Units

The cores feature several load and store execution units capable of simultaneous issue. Store μ ops are executed in 2 parts: one that computes the address to be stored and one that obtains the data to be stored. When both parts of a store are complete and ordering rules satisfied, the Load and Store Execution Unit writes the data into the cache.

A.3.1 Load Latency

Load μ ops that hit in the data cache normally execute with a 4-cycle latency.

Some integer-load instructions require an additional arithmetic μ op to complete in order to update addresses for pre- and post-indexing. See [Section 2.8, “Addressing Forms, Instruction Immediates, and Operand Shifts”](#), [Section 5.6.1, “Address Generation”](#), and [Section 5.4.8, “Multi- \$\mu\$ op Instructions”](#) for more details.

Table A.8. Integer Load Latencies

Load Type	Latency of Destination Data Operand(s)	Example Instruction Mnemonics
Load (all sizes)	4	LDAR, LDARB, LDARH, LDAPUR, LDAPURB, LDAPURH, LDAPURSB, LDAPURSH, LDAXR, LDAXRB, LDAXRH, LDAPR, LDAPRB, LDAPRH, LDR, LDRB, LDRH, LDRSB, LDRSH, LDRSW, LDUR, LDURB, LDURH, LDURSB, LDURSH, LDURSW, LDTR, LDTRB, LDTRH, LDTRSB, LDTRSH, LDTRSW, LDXR, LDXRB, LDXRH
Load Pair (all sizes)	4	LDAXP, LDNP, LDP, LDPSW, LDXP

Some vector load instructions require several μ ops to complete, and may include combinations of:

- arithmetic μ ops to compute addresses as well as update address registers for pre- and post-indexing
- load μ ops for multiple elements
- shuffle μ ops to fill specific elements

See [Section 2.8, “Addressing Forms, Instruction Immediates, and Operand Shifts”](#), [Section 5.6.1, “Address Generation”](#), and [Section 5.4.8, “Multi- \$\mu\$ op Instructions”](#) for more details.

Some instructions use a microcode sequence that consists of more loads than available load execution units, or similarly more shuffles than available shuffle execution unit resources. Because, for example, the P core only contains 3 load execution units, a sequence that requires 4 load μ ops has at least 1 μ op execute later than the others. The latencies listed below don't account for any delays after becoming ready due to resource over-subscription or contention with other instructions. However sequences that require more than the 3 loads or 4 shuffles available on the P core are marked with [†]. E core over-subscription (not shown) is more significant due to fewer load and shuffle execution resources.

Table A.9. Vector Load Latencies

Load Type	Latency of Destination Data Operand(s)	Example Instruction Mnemonics with Additional Details
Load and Load Pair (all sizes)	4	LDR, LDUR, LDP, LDNP (see note on lsl/extend #4 in Section 5.4.8, "Multi-μop Instructions")
Load 1 Element Structure	4 [†]	LD1 [†] Resource over-subscription when using 4 register destinations
Load {2, 3, 4} Element Structure	6	LD{2, 3, 4}
Load {1, 2, 3, 4} Element Structure to 1 Lane	6/2 [†]	LD{1, 2, 3, 4} {Vd, . . .} [x] 6 cycles through the load path, 2 cycles for the pass-through data in Vd* operands [†] Resource over-subscription for Load 4 Element Structure to 1 Lane when using 128b destination registers
Load {1, 2, 3, 4} Element Structure Replicated	6	LD{1, 2, 3, 4}R

A.3.2 Store Latency

Store operations do not have a destination register and therefore no destination register latency. Depending on the complexity of the store operation, the instruction may require additional μops for address computation, element shuffling, storing to multiple addresses, and updating pointers, all similar to their load counterparts.

Table A.10. Integer Store Latencies

Store Type	Latency of Destination Data Operand(s)	Example Instruction Mnemonics
Store (all sizes)	N/A	STLR, STLRB, STLRH, STLUR, STLURB, STLURH, STR, STRB, STRH, STUR, STURB, STURH, STTR, STTRB, STTRH
Store Exclusive (all sizes)	4	STXR, STXRB, STXRH, STLXR, STLXRB, STLXRH
Store Pair (all sizes)	N/A	STP, STNP,
Store Pair Exclusive (all sizes)	4	STXP, STLXP

Although the nominal latency for the returned status value of a store exclusive instruction is 4 cycles, execution of the store may be delayed due to ordering requirements.

A.3.3 Atomic Instructions

The Arm ISA contains a number of instructions that execute an entire atomic synchronization operation in a single instruction. Latency can vary significantly depending on the amount of time it takes for the load-and-store exclusive operation pairs that are at the core of these instructions to complete, especially if the memory addresses are being accessed by many processors simultaneously. Despite these instructions being comprised

of several μ ops, the load and store μ ops are guaranteed to execute atomically by the core. That is, once ordering requirements have been met and the core possesses the cache line, both the load and store complete prior to the line being sent to another core.

The compare-and-swap (CAS) operations load the memory contents at the target address and then conditionally store a new value to memory location only if the memory value is equal to the old contents of the destination register.

The swap operations (SWP) load in the current memory contents at the target address and then store a new value to the same memory location from a register, swapping the memory and register contents.

The load-and- $\{$ operation $\}$ instructions, such as LDSMAX, load in the memory contents at the target address to a register, combine this value with the contents of a second register using one of a variety of integer operations, and then store the result back to the same location in memory.

The store-and- $\{$ operation $\}$ instructions are just load-and- $\{$ operation $\}$ instructions that discard the value loaded from memory instead of keeping it around in a register afterwards.

Although these instruction could wait to execute for the input data register(s) to be ready, the instruction latency is typically through the load operation and input address register. The latencies in [Table A.11](#) express this likelihood and reflect the latency after the ordering requirements have been met and assuming a L1D Cache hit.

Table A.11. Synchronization Instruction Latencies

Operation Type (Each have optional acquire and release semantics)	Latency to Destination Register	Example Instruction Mnemonics
Compare-and-Swap	4	CAS, CASA, CASAB, CASAH, CASAL, CASALB, CASALH, CASB, CASH, CASL, CASLB, CASLH
Compare-and-Swap Pair		CASP, CASPA, CASPL, CASPAL
Swap		SWP, SWPB, SWPH, SWPA, SWPAB, SWPAH, SWPL, SWPLB, SWPLH, SWPAL, SWPALB, SWPALH
Load-and- $\{$ Operation $\}$		LDADD, LDADDB, LDADDH, LDCLR, LDCLRB, LDCLRH, LDEOR, LDEORB, LDEORH, LDSET, LDSETB, LDSETH, LDSMAX, LDSMAXB, LDSMAXH, LDSMIN, LDSMINB, LDSMINH, LDUMAX, LDUMAXB, LDUMAXH, LDUMIN, LDUMINB, LDUMINH LDADDA, LDADDAB, LDADDAH, LDCLRA, LDCLRAB, LDCLRAH, LDEORA, LDEORAB, LDEORAH, LDSETA, LDSETAB, LDSETAH, LDSMAXA, LDSMAXAB, LDSMAXAH, LDSMINA, LDSMINAB, LDSMINAH, LDUMAXA, LDUMAXAB, LDUMAXAH, LDUMINA, LDUMINAB, LDUMINAH LDADDL, LDADDLB, LDADDLH, LDCLRL, LDCLRLB, LDCLRLH, LDEORL, LDEORLB, LDEORLH, LDSETL, LDSETLB, LDSETLH, LDSMAXL, LDSMAXLB, LDSMAXLH, LDSMINL, LDSMINLB, LDSMINLH, LDUMAXL, LDUMAXLB, LDUMAXLH, LDUMINL, LDUMINLB, LDUMINLH

Table A.11. Synchronization Instruction Latencies (cont.)

Operation Type (Each have optional acquire and release semantics)	Latency to Destination Register	Example Instruction Mnemonics
		LDADDAL, LDADDALB, LDADDALH, LDCLRAL, LDCLRALB, LDCLRALH, LDEORAL, LDEORALB, LDEORALH, LDSETAL, LDSETALB, LDSETALH, LDSMAXAL, LDSMAXALB, LDSMAXALH, LDSMINAL, LDSMINALB, LDSMINALH, LDUMAXAL, LDUMAXALB, LDUMAXALH, LDUMINAL, LDUMINALB, LDUMINALH
Store-and-{Operation}	N/A	STADD, STADDB, STADDH, STCLR, STCLRB, STCLRH, STEOR, STEORB, STEORH, STSET, STSETB, STSETH, STSMAX, STSMAXB, STSMAXH, STSMIN, STSMINB, STSMINH, STUMAX, STUMAXB, STUMAXH, STUMIN, STUMINB, STUMINH STADDA, STADDAB, STADDAH, STCLRA, STCLRAB, STCLRAH, STEORA, STEORAB, STEORAH, STSETA, STSETAB, STSETAH, STSMAXA, STSMAXAB, STSMAXAH, STSMINA, STSMINAB, STSMINAH, STUMAXA, STUMAXAB, STUMAXAH, STUMINA, STUMINAB, STUMINAH STADDL, STADDLB, STADDLH, STCLRL, STCLRLB, STCLRLH, STEORL, STEORLB, STEORLH, STSETL, STSETLB, STSETLH, STSMAXL, STSMAXLB, STSMAXLH, STSMINL, STSMINLB, STSMINLH, STUMAXL, STUMAXLB, STUMAXLH, STUMINL, STUMINLB, STUMINLH STADDAL, STADDALB, STADDALH, STCLRAL, STCLRALB, STCLRALH, STEORAL, STEORALB, STEORALH, STSETAL, STSETALB, STSETALH, STSMAXAL, STSMAXALB, STSMAXALH, STSMINAL, STSMINALB, STSMINALH, STUMAXAL, STUMAXALB, STUMAXALH, STUMINAL, STUMINALB, STUMINALH

Recommendation: Use single instruction atomics to improve multiprocessor performance.

[Magnitude: Low | Applicability: Low] The CPU executes the load and store operations in an atomic instruction without losing the cache line in between them, improving multithreaded performance over the exclusive instructions.

A.3.4 Load and Store Execution Unit Bandwidth

Loads and stores execute on multiple execution ports. As noted, store μ ops are executed in 2 parts, one that computes the address to be stored and one that obtains the data to be stored.

MOVE2VEC instructions that move data from the GPRs to the Vector registers (DUP(general), FMOV(general), INS(general), MOV(from general), SCVTF(scalar), UCVTF(scalar)) consume load issue slots. The movement portion of these instructions has a latency of 4 cycles. See [Section 5.5.1, “Movement of Data from General Purpose Registers to Vector Registers”](#).

For SSFP/SSVE/SME instructions executed in the SME engine, all instructions (including non-memory instructions) and predicate updates are prepared and sent to the SME engine through the core Load and Store Execution Unit as a store (both address and data parts). The unit additionally performs all required translations and checks for SME engine loads and stores, but does not access the L1D Cache.

Table A.12. Memory Execution Unit Bandwidth: P Core

Chip	Bandwidth			
	Load or MOVE2VEC μ op	Store μ op (address part)	Store μ op (data part)	Combined limits per cycle
M Series	3	2	2	Burst: 3 load μ ops, 2 store μ ops (address part), and 2 store μ ops (data part) Sustained: 4 μ ops Sustained: 2 writes into the cache
A Series				

Table A.13. Memory Execution Unit Bandwidth: E Core

Chip	Bandwidth			
	Load or MOVE2VEC μ op	Store μ op (address part)	Store μ op (data part)	Combined limits per cycle
M Series	2	2	2	Burst: 2 load μ ops, or 2 store μ ops (address part), or 1 of each, along with 2 store μ ops (data part) Sustained: 2 μ ops Sustained: 1 write into the cache
A Series				

The following instructions serialize the Memory Execution Units according to their conditions: DMB, DSB, PSSBB, SSBB.

A.3.5 Memory Hierarchy Access Latencies

Table A.14: “Cache and Memory Access Latencies” lists typical unloaded access latencies for cache lines stored in different locations for M1. There are many factors that can slow an individual access, including but not limited to queuing delays due to other accesses/traffic and clock frequency differences. Generationally, larger caches are typically slower to access, and thus typical access latency may increase by a few cycles accordingly. This is most pertinent when considering the Own Shared L2 Cache latency, where a few additional cycles may represent a non-negligible percentage increase. Similarly, the CPUs in A-series chips may see a slightly faster Own Shared L2 Cache latency compared with the CPUs in M-series chips due to the smaller cache sizes. Use this table to comprehend “ballpark” latencies when analyzing algorithms, especially when evaluating how working sets interact with various caches and when analyzing multithreaded shared variable access patterns.

Table A.14. Cache and Memory Access Latencies

Cache/Memory Level	Typical M1 Unloaded Latency
Own L1 Cache	4 cycles (See Section A.3, “Core Load and Store Execution Units”)
Own Shared L2 Cache	~15 cycles
Other Cluster Shared L2 Cache	~50 ns
Memory Cache	~35 ns
Main Memory	~95 ns

A.4 SSFP/SSVE Execution Unit

The SSFP/SSVE Execution Unit can issue up to one μ op per cycle, where some μ ops may require the unit's issue slot for more than one cycle.

The destination latency reflects the common latency for each of the 1-, 2-, or 4- output vectors accordingly. In some noted cases, four-vector versions are actually implemented as two independent two-vector versions that do not necessarily issue back-to-back. However the table simplifies the latencies assuming they do.

Note that in some specific circumstances, the SSFP/SSVE Execution Unit can issue an unrelated μ op at the same time as the second and following issues of a multi-issue μ op. Because these circumstances are complex, software should be written to keep the unit busy given the latencies and bandwidths in the table, considering any extra issue bandwidth as opportunistic added benefit.

Table A.15. SSVE Execution Classes

Class	Dest Latency	Scheduler Issue Slots	Operation Types and Descriptions (Example Instruction Mnemonics)
INT	3	1	Data movement (CLASTA^{***}, CLASTB^{***}, CPY[*], DUP[*], DUPM, FCPY, FDUP, FMOV^{***}, INDEX[*], INSR[*], LASTA^{***}, LASTB^{***}, MOV[*], unfused MOVPRFX, MOVT^{***}, 1-dest SEL, SMOV^{**}, UMOV^{**}, ZERO (ZT)) Comparison operations that return predicate data to the core (CMEQ^{**}, CMGE^{**}, CMGT^{**}, CMHI^{**}, CMHS^{**}, CMLE^{**}, CMLT^{**}, CMLO^{**}, CMLS^{**}, CMNE^{**}) *Instructions that send data, flag, or predicate values from the core to the SME engine require an additional core μop to read the core register **Instructions that return data, flag, or predicate values from the SME engine to the core require an additional SME engine Load/Store Unit μop and have an undefined latency back to the core due to the trailing-execution nature of the SME engine Arithmetic, logical, and comparison operations (ABS, ADCLB, ADCLT, 1-dest ADD, ADDHNB, ADDHNT, ADDP, AND, BIC, CADD, CNOT, DECD, DECH, DECP, DECW, EON, EOR, EORBT, INCD, INCH, INCP, INCW, NEG, NOT, ORR, ORN, RADDHNB, RADDHNT, RSUBHNB, RSUBHNT, SABA, SABALB, SABALT, SABD, SABDLB, SABDLT, SADALP, SADDLB, SADDLT, SADDLBT, SADDWB, SADDWT, SBCLB, SBCLT, SHADD, SHSUB, SHSUBR, 1-dest SMAX, SMAXP, 1-dest SMIN, SMINP, SQABS, SQADD, SQCADD, SQDECD, SQDECH, SQDECP, SQDECW, SQINCD, SQINCH, SQINCP, SQINCW, SQNEG, SQSUB, SQSUBR,

Table A.15. SSVE Execution Classes

Class	Dest Latency	Scheduler Issue Slots	Operation Types and Descriptions (Example Instruction Mnemonics)
			SRHADD, SSUBLB, SSUBLT, SSUBLBT, SSUBLTB, SSUBWB, SSUBWT, SUB, SUBHNB, SUBHNT, SUBR, SUQADD, UABA, UABALB, UABALT, UABD, UABDLB, UABDLT, UADALP, UADDLB, UADDLT, UADDWB, UADDWT, UHADD, UHSUB, UHSUBR, 1-dest UMAX, UMAXP, 1-dest UMIN, UMINP, UQADD, UQDECD, UQDECH, UQDECP, UQDECW, UQINCD, UQINCH, UQINCP, UQINCW, UQSUB, UQSUBR, URHADD, USQADD, USUBLB, USUBLT, USUBWB, USUBWT) Shift and bitfield operations (ASR, ASRD, ASRR, BSL, BSL1N, BSL2N, EXT, LSL, LSLR, LSR, CLS, CLZ, CNT, LSRR, NBSL, RBIT, REV, REVB, REVD, REVH, REVW, RSHRNB, RSHRNT, SHRNB, SHRNT, SLI, SQRSHL, SQRSHLR, SQRSHRNB, SQRSHRNT, SQRSHRUNB, SQRSHRUNT, SQSHL, SQSHLR, SQSHLU, SQSHRNB, SQSHRNT, SQSHRUNB, SQSHRUNT, SQXTNB, SQXTNT, SQXTUNB, SQXTUNT, SRI, 1-dest SRSHL, SRSHLR, SRSHR, SRSRA, SSHLLB, SSHLLT, SSRA, SXTB, SXTH, SXTW, UQSHL, UQSHLR, UQRSHL, UQRSHLR, UQRSHRNB, UQRSHRNT, UQSHRNB, UQSHRNT, UQXTNB, UQXTNT, 1-dest URSHL, URSHLR, URSHR, URSRA, USHLLB, USHLLT, USRA, UXTB, UXTH, UXTW) SHA3-related logic operations (BCAX, EOR3, XAR) Data transposition operations (SPLICE, SUNPKHI, SUNPKLO, TRN1, TRN2, UUNPKHI, UUNPKLO, UZP1, UZP2, ZIP1, ZIP2) Table lookup operations (unfused LUTI2, unfused LUTI4, TBL, TBX)
	3, 4	2	Data movement (2-dest SEL, 2-dest SMAX, 2-dest SMIN, 2-dest UMAX, 2-dest UMIN) Integer arithmetic operations (2-dest ADD) Shift and bitfield operations (unfused 2-src SQRSHR, unfused 2-src SQRSHRN, unfused 2-src SQRSHRU, unfused 2-src SQRSHRUN, unfused 2-src SQSHRN, unfused 2-src SQSHRUN, 2-dest SRSHL, unfused 2-src UQRSHR, unfused 2-src UQRSHRN, 2-dest URSHL) Data transposition operations (2-dest SUNPK, 2-dest UUNPK, unfused 2-dest UZP, unfused 2-dest ZIP) Table lookup operations (unfused 2-dest LUTI2, unfused 2-dest LUTI4)
	3, 4, 5, 6	4	Data movement (4-dest SEL), implemented as 2 independent μops of the 2-dest version, split between first 2 srcs/dests and second 2 srcs/dests Arithmetic and comparison operations (4-dest ADD, 4-src/dest with 1-src SMAX, 4-src/dest with 1-src SMIN, 4-src/dest with 1-src UMAX, 4-src/dest with 1-src UMIN), implemented as a single μop Arithmetic and comparison operations (4-src/dest with 4-src SMAX, 4-src/dest with 4-src SMIN, 4-src/dest with 4-src UMAX, 4-src/dest with 4-src UMIN), implemented as 2 independent μops of the 2-dest version, split between first 2 srcs/dests and second 2 srcs/dests Shift and bitfield operations (4-src/dest with 1-src SRSHL, 4-src/dest with 1-src URSHL), implemented as a single μop Shift and bitfield operations (unfused 4-src SQRSHR, unfused 4-src SQRSHRN, unfused 4-src SQRSHRU, unfused 4-src SQRSHRUN, unfused

Table A.15. SSVE Execution Classes

Class	Dest Latency	Scheduler Issue Slots	Operation Types and Descriptions (Example Instruction Mnemonics)
			4-src SQSHRN, unfused 4-src SQSHRUN, 4-src/dest with 4-src SRSHL, unfused 4-src UQRSHR, unfused 4-src UQRSHRN, 4-src/dest with 4-src URSHL), implemented as 2 independent μ ops of the 2-dest version, split between first 2 srcs/dests and second 2 srcs/dests Data transposition operations (4-dest SUNPK, 4-dest UUNPK, unfused 4-dest UZP, unfused 4-dest ZIP), implemented as a single μ op Table lookup operations (unfused 4-dest LUTI2, unfused 4-dest LUTI4), implemented as a single μ op
	4	1	Arithmetic and logical operations across vector (ANDV, EORV, ORV, SADDV, SMAXV, SMINV, UADDV, UMAXV, UMINV)
	8	4	Multiplication operations (CDOT, CMLA, MAD, MLA, MLS, MSB, MUL, PMUL, PMULLB, PMULLT, SDOT, SMLALB, SMLALT, SMLSLB, SMLSLT, SMULH, SMULLB, SMULLT, SQDMLALB, SQDMLALBT, SQDMLALT, SQDMLSLB, SQDMLSLBT, SQDMLSLT, 1-dest SQDMULH, SQDMULLB, SQDMULLT, SQRDCMLAH, SQRDMLAH, SQRDMLSH, 1-dest SQRDMULH, SUDOT, UDOT, UMLALB, UMLALT, UMLSLB, UMLSLT, USDOT, UMULH, UMULLB, UMULLT).
	8, 12	8	Multiplication operations (2-dest SQDMULH, 2-dest SQRDMULH)
	8, 12, 16, 20	16	Multiplication operations (4-src/dest with 1-src SQDMULH, 4-src/dest with 1-src SQRDMULH), implemented as a single μ op Multiplication operations (4-src/dest with 4-src SQDMULH, 4-src/dest with 4-src SQRDMULH), implemented as 2 independent μ ops of the 2-dest version, split between first 2 srcs/dests and second 2 srcs/dests
	17	8	Divide operations (SDIV, SDIVR, UDIV, UDIVR)
	4	2	Arithmetic and comparison operations (1-dest SCLAMP, 1-dest UCLAMP)
	4, 6	4	Arithmetic and comparison operations (2-dest SCLAMP, 2-dest UCLAMP)
	4, 6, 8, 10	8	Integer arithmetic and comparison operations (4-dest SCLAMP, 4-dest UCLAMP), implemented as 2 independent μ ops of the 2-dest version, split between first 2 dests and second 2 dests
FP/BFP	3	1	Absolute value, negate operations (FABS, FNEG) Arithmetic and comparison operations (1-dest FMAX, 1-dest FMAXNM, FMAXNMP, FMAXP, 1-dest FMIN, 1-dest FMINNM, FMINNMP, FMINP) Conditional select operations (FCSEL*) Comparison operations that return predicate or flag data to the core (FACGE**, FACGT**, FCCMP**, FCCMPE**, FCMEQ**, FCMGE**, FCMGT**, FCMLE**, FCMLT**, FCMP**, FCMPE**) *Instructions that send data, flag, or predicate values from the core to the SME engine require an additional core μ op to read the core register **Instructions that return data, flag, or predicate values from the SME engine to the core require an additional SME engine Load/Store Unit μ op and have an undefined latency back to the core due to the trailing-execution nature of the SME engine

Table A.15. SSVE Execution Classes (cont.)

Class	Dest Latency	Scheduler Issue Slots	Operation Types and Descriptions (Example Instruction Mnemonics)
			Reciprocal/reciprocal square root estimate scalar operations (FRECPE, FRECPX, FRSQRTE)
	3, 4	2	Arithmetic and comparison operations (2-dest FMAX, 2-dest FMAXNM, 2-dest FMIN, 2-dest FMINNM)
	3, 4, 5, 6	4	Arithmetic and comparison operations (4-src/dest with 1-src FMAX, 4-src/dest with 1-src FMAXNM, 4-src/dest with 1-src FMIN, 4-src/dest with 1-src FMINNM), implemented as a single μ op Arithmetic and comparison operations (4-src/dest with 4-src FMAX, 4-src/dest with 4-src FMAXNM, 4-src/dest with 4-src FMIN, 4-src/dest with 4-src FMINNM), implemented as 2 independent μ ops of the 2-dest version, split between first 2 srcs/dests and second 2 srcs/dests
	4	1	Arithmetic and comparison operations (FMAXV, FMAXNMV, FMINV, FMINNMV)
	5	1	Arithmetic scalar operations (FADD, FSUB) Multiply scalar operations (FMADD, FMSUB, FMUL, FMULX, FNMADD, FNMSUB, FNMUL) Reciprocal/reciprocal square root step scalar operations (FRECPS, FRSQRTS)
	8	4	Arithmetic vector operations (FABD, FADD, FADDP, FCADD, FSUB, FSUBR) Multiply vector operations (BFMLALB, BFMLALT, BFMLSLB, BFMLSLT, FMAD, FCMLA, FMLA, FMLALB, FMLALT, FMLS, FMLSLB, FMLSLT, FMSB, FMUL, FMULX, FMAD, FNMLA, FNMLS, FNMSB, FNMSUB) Reciprocal/reciprocal square root step vector operations (FRECPS, FRSQRTS) Base 2 Log/Exp operations (FLOGB, FSCALE)
	10	8	Reciprocal/reciprocal square root estimate vector operations (FRECPE, FRECPX, FRSQRTE, URECPE, URSQRTE)
	28	16	Dot product operation (BFDOT), note that this instruction also consumes 4 instruction slots from the core to the SME engine
	4	2	Arithmetic and comparison operations (1-dest FCLAMP)
	4, 6	4	Arithmetic and comparison operations (2-dest FCLAMP)
	4, 6, 8, 10	8	Arithmetic and comparison operations (4-dest FCLAMP), implemented as 2 independent μ ops of the 2-dest version, split between first 2 dests and second 2 dests
	Various	Various	Horizontal arithmetic to scalar operation (FADDV) Type H: 26 cycles (6 issue) Type S: 21 cycles (5 issue) Type D: 16 cycles (4 issue)
	Various	Various	Divide operations (FDIV, FDIVR) Scalar type H: 7 cycles

Table A.15. SSVE Execution Classes (cont.)

Class	Dest Latency	Scheduler Issue Slots	Operation Types and Descriptions (Example Instruction Mnemonics)
			Vector type H: 14 cycles (8 issue) Scalar type S: 8 cycles Vector type S: 15 cycles (8 issue) Scalar type D: 10 cycles Vector type D: 17 cycles (8 issue)
	Various	Various	Square root operations (FSQRT) Scalar type H: 8 cycles (2 issue) Vector type H: 22 cycles (16 issue) Scalar type S: 10 cycles (2 issue) Vector type S: 24 cycles (16 issue) Scalar type D: 13 cycles (2 issue) Vector type D: 27 cycles (16 issue) Note that use or production of subnormals requires extra cycles
CVT	3	1	Conversion operations between integer and floating point types, and floating point rounding operations (1-src BFCVT, BFCVTNT, 1-src FCVT, FCVT (AMNP) (SU) **, 1-dest FCVTZ (SU), FCVTLT, 1-src FCVTN, FCVTNT, FCVTX, FCVTXNT, 1-dest FRINT (AMNP), FRINT (IXZ), FRINT32 (XZ), FRINT64 (XZ), 1-dest SCVTF*, UCVTF*) *Instructions that send data, flag, or predicate values from the core to the SME engine require an additional core μop to read the core register **Instructions that return data, flag, or predicate values from the SME engine to the core require an additional SME engine Load/Store Unit μop and have a undefined latency back to the core due to the trailing-execution nature of the Engine
	3, 4	2	Conversion operations between integer and floating point types, and floating point rounding operations (unfused 2-src BFCVT, unfused 2-src BFCVTN, unfused 2-src FCVT, unfused 2-src FCVTN, 2-dest FCVTZ (SU), 2-dest FRINT (AMNP), 2-dest SCVTF, unfused 2-src SQCVT, unfused 2-src SQCVTN, unfused 2-src SQCVTU, unfused 2-src SQCVTUN, 2-dest UCVTF, unfused 2-src UQCVT, unfused 2-src UQCVTN)
	3, 4, 5, 6	4	Conversion operations between integer and floating point types, and floating point rounding operations (4-dest FCVTZ (SU), 4-dest FRINT (AMNP), unfused 4-src SQCVT, unfused 4-src SQCVTN, unfused 4-src SQCVTU, unfused 4-src SQCVTUN, unfused 4-src UQCVT, unfused 4-src UQCVTN), implemented as a single μop Conversion operations between integer and floating point types, and floating point rounding operations (4-dest SCVTF, 4-dest UCVTF), implemented as 2 independent μops of the 2-dest version, split between first 2 srcs/dests and second 2 srcs/dests

A.5 SME ZA Extraction Unit

The ZA Extraction Unit scheduler can issue one operation per cycle with basic non-renamed out-of-order execution. It can operate in parallel with the SME Processing Grid.

Table A.16. SME ZA Extraction Unit Classes

Class	Latency	Scheduler Issue Slots	Operation Types and Descriptions (Example Instruction Mnemonics)
General	15	1	Data movement operations from ZA storage without governing predicate (1-dest MOV, MOVA)
	15, 16	2	Data movement operations from ZA storage, including the extraction portion of fused stores (2-dest MOV, MOVA, ST1 *, STR)
	15, 16, 17, 18	4	Data movement operations from ZA storage, including MOVA-MOVA fusion (4-dest MOV, MOVA)

A.6 SME Processing Grid

The SME Processing Grid is optimized for throughput, typically multiplying and accumulating values from a large input data set. As such, the critical latency through Processing Grid instructions is through the accumulator. Software should be optimized to load data sufficiently ahead of computation in order maximize accumulator throughput.

Note: The latencies provided for the compute instructions below represent only the latency through the accumulator, whereas the end-to-end instruction latency that includes that of the multiplies is higher.

LUT12 and LUT14 fusion with vector or tile computation instructions adds negligible latency through the multiply sources.

For many instructions, only 1 microoperation (μop) is necessary to execute the instruction. For other instructions, multiple μops are required, as noted in the Descriptions column in the tables below. The Processing Grid scheduler can issue up to 1 μop per cycle, and supports non-renamed out-of-order execution tracked by row of PEs. Each PE is generally pipelined and can start a new operation every cycle while other operations are in various stages of completion. Special cases for latency and issue slots follow these patterns in the following tables:

- **Additional latency annotations, "[+n, +m, +p]" in Latency column:** These μops have been optimized to execute with minimal accumulator latency when consumed by a μop of the same computation and element types, which is common for throughput-oriented algorithms. The added latencies in the brackets represent additional cycles required for [different computation instruction, horizontal extract, vertical extract].
- **Scheduler multi-issue, "n" in Scheduler Issue Slots column:** Indicates that the μop requires n issue slots into the Processing Grid from the scheduler. Consider the common example when latency is 4 and issues slots is also 4. Repeated issuing of μops of this type that depend on one another through the accumulator consume 100% of the Processing Grid. This is most common for E Engine operations across an entire tile due to the smaller Processing Grid.

- **PE multi-issue, "1 (*m* on PE row)" in Schedule Issue Slots column:** Indicates that the μ op requires 1 overall issue slot into the Grid from the scheduler, but consumes *m* issue slots internally on the row of PEs associated with the accessed ZA rows. For the common example of "1 (2 on PE row)", depending on the distribution across PE rows in the entire algorithm, the overall bandwidth can be as little as 1 μ op every other cycle when instructions use the same PE rows, but as high as 1 μ op per cycle when the instructions use the full set of PE rows (ZA vectors distributed across the ZA storage).
- **Multi- μ op sequences, "<specific source pattern> versions require 2 independent μ ops of this class on half of the 4-vector, split between first 2 srcs/dests and second 2 srcs/dests." in Descriptions column:** Indicates that the multi-vector operation is split into 2 μ ops that can issue and execute individually, each working on half of the inputs/outputs. This is most common for ZA vector operations across both P and E engines.
- **Specific element sizes, "<src>-to-<dest>" in Descriptions column:** The latency and slots required for some μ ops depend on the specific source and destination element sizes. This shorthand notation indicates that the instruction reads inputs of one element size while writing results of a different element size, according to those size names defined in [Section 4.1.1, "SSVE Z Vector Registers \(zn\)"](#), which are used across SSVE and SME.

Table A.17. SME Processing Grid Move From Storage Classes

Class	Latency	Scheduler Issue Slots	Operation Types and Descriptions (Example Instruction Mnemonics)
MOV FROM	16	1	Data movement operations from ZA storage with governing predicate (1-dest MOV/MOVB) Fused data movement operations from ZA storage, 2-src (fused MOVB with SQCVTN, SQCVTUN, SQRSHRN, SQRSHRUN, SQSHRN, SQSHRUN, UQCVTN, or UQRSHRN) Fused data movement operations from ZA storage, 2-src (fused MOVB with BFCVTN or FCVTN)
	17	1	Fused data movement operations from ZA storage, 2-src (fused MOVB with SQCVT, SQCVTU, SQRSHR, SQRSHRU, UQCVT, or UQRSHR) Fused data movement operations from ZA storage, 2-src (fused MOVB with BFCVT or FCVT)
	17	2	Fused data movement operations from ZA storage, 4-src (fused MOVB with SQCVTN, SQCVTUN, SQRSHRN, SQRSHRUN, SQSHRN, SQSHRUN, UQCVTN, or UQRSHRN)
	17, 18	2	Fused data movement operations from ZA storage, 4-src (fused MOVB with SQCVT, SQCVTU, SQRSHR, SQRSHRU, UQCVT, or UQRSHR)
	18, 19	2	Fused data movement operations from ZA storage, 2-src (fused MOVB with UZP or ZIP)
	20, 21, 22, 23	4	Fused data movement operations from ZA storage, 4-src (fused MOVB with UZP or ZIP)

Table A.18. SME Processing Grid Computation and Move To Storage Classes

Class	Accumulator Latency	Scheduler Issue Slots	Operation Types and Descriptions (Example Instruction Mnemonics)
MOV TO	4	1	Data movement operations to ZA storage (1-src MOV/MOVA) Data movement operations to ZA storage (D-type 2-src to VGx2 MOV/MOVA) P: Data movement operations to ZA storage (D-type 4-src to VGx4 MOV/MOVA) P: Data movement operations to ZA storage (D-type 2-src to indexed rows MOV/MOVA) P: Data movement operations to ZA storage (D-type 4-src to indexed rows MOV/MOVA)
	4, 5	2	E: Data movement operations to ZA storage (D-type 4-src to VGx4 MOV/MOVA) Data movement operations to ZA storage (B/H/S-type 2-src to indexed rows MOV/MOVA) E: Data movement operations to ZA storage (D-type 2-src to indexed rows MOV/MOVA) Vertical data movement operations to ZA storage (2-src to vertical indexed rows MOV/MOVA) Data movement operations to ZA storage (B/H-type 4-src to indexed rows MOV/MOVA). Requires 2 independent μops of this class on half of the 4-vector. Data movement operations to ZA storage (S-type 4-src to indexed rows MOV/MOVA) E: Data movement operations to ZA storage (D-type 4-src to indexed rows MOV/MOVA)
	4, 5, 6, 7	4	Vertical data movement operations to ZA storage (4-src to vertical indexed rows MOV/MOVA)
INT	P: 4 [+0, +1, +2] ⁺ E: 4 [+3, +4, +5] ⁺	P: 1 E: 4	Tile: Bitwise exclusive NOR population count outer product and accumulate (S-to-S) (BMOPA, BMOPS) Tile: Sum of outer products and double-lengthening accumulate (H-to-D types) (SMOPA, SMOPS, SUMOPA, SUMOPS, UMOA, UMOPS, USMOPA, USMOPS) Tile: Add horizontal/vertical vector across entire tile (S-to-S and D-to-D) (ADDHA, ADDVA)
	P: 4 [+1, +2, +3] ⁺ E: 8 [+3, +4, +5] ⁺	P: 2 E: 8	Tile: Sum of outer products and lengthening accumulate (H-to-S) (SMOPA, SMOPS, UMOA, UMOPS) Tile: Sum of outer products and double-lengthening accumulate (B-to-S) (SMOPA, SMOPS, SUMOPA, SUMOPS, UMOA, UMOPS, USMOPA, USMOPS)
	4 [+0, +1, +2] ⁺	1	Vector: Dot product, including mixed element signed/unsigned (H-to-S, B-to-S, and H-to-D) (SDOT, SUDOT, SUVDOT, SVDOT, UDOT, USDOT, USVDOT, UVDOT)* Vector: Add/subtract (S-to-S and D-to-D) (ADD, SUB)*

Table A.18. SME Processing Grid Computation and Move To Storage Classes (cont.)

Class	Accumulator Latency	Scheduler Issue Slots	Operation Types and Descriptions (Example Instruction Mnemonics)
			*E: "4-vector to array accumulators" versions require 2 independent μops of this class on half of the 4-vector, split between first 2 srcs/dests and second 2 srcs/dests *E: "4-vector to array results with single/indexed vector" versions require 2 independent μops of this class on half of the 4-vector, split between first 2 srcs/dests and second 2 srcs/dests *E/P: "Multiple 4-vectors, both to array accumulators and to array results" versions require 2 independent μops of this class on half of the 4-vector, split between first 2 srcs/dests and second 2 srcs/dests
	P: 4 [+1, +2, +3] [†] E: 4 [+1, +2, +3] [†]	P: 1 (2 on PE row) E: 2	Vector: Multiply-accumulate and lengthen (H-to-S types) (SMLAL, SMLSLL, UMLAL, UMLSLL)* Vector: Multiply-accumulate and double lengthen (B-to-S and H-to-D) (SMLALL, SMLSLL, UMLALL, UMLSLL)* Vector: Multiply-accumulate and double lengthen, signed-unsigned (B-to-S) (SUMLALL, USMLALL)* *E: "4-vector with single/indexed vector" versions require 2 independent μops of this class on half of the 4-vector, split between first 2 srcs/dests and second 2 srcs/dests *E/P: "Multiple 4-vectors" versions require 2 independent μops of this class on half of the 4-vector, split between first 2 srcs/dests and second 2 srcs/dests
FP/BFP	P: 4 [+1, +2, +3] [†] E: 4 [+4, +5, +6] [†]	P: 1 E: 4	Tile: Sum of outer products and accumulate (S-to-S and D-to-D) (FMOPA, FMOPS)
	P: 8 [+1, +2, +3] [†] E: 8 [+4, +5, +6] [†]	P: 2 E: 8	Tile: Sum of outer products and lengthening accumulate (H-to-S) (BFMOPA, BFMOPS, FMOPA, FMOPS)
	P: 8 [+1, +2, +3] [†] E: 8 [+1, +2, +3] [†]	P: 1 (2 on PE row) E: 2	Vector: Dot product (H-to-S) (BFDOT, BFVDOT, FDOT, FVDOT)* *E: "4-vector with single/indexed vector" versions require 2 independent μops of this class on half of the 4-vector, split between first 2 srcs/dests and second 2 srcs/dests *E/P: "Multiple 4-vectors" versions require 2 independent μops of this class on half of the 4-vector, split between first 2 srcs/dests and second 2 srcs/dests
	4 [+1, +2, +3] [†]	1	Vector: Multiply-accumulate (S-to-S and D-to-D) (FMLA, FMLS)* *E: "4-vector with single/indexed vector" versions require 2 independent μops of this class on half of the 4-vector, split between first 2 srcs/dests and second 2 srcs/dests *E/P: "Multiple 4-vectors" versions require 2 independent μops of this class on half of the 4-vector, split between first 2 srcs/dests and second 2 srcs/dests Vector: Add/subtract (S-to-S and D-to-D) (FADD, FSUB)**

Table A.18. SME Processing Grid Computation and Move To Storage Classes (cont.)

Class	Accumulator Latency	Scheduler Issue Slots	Operation Types and Descriptions (Example Instruction Mnemonics)
			**E: "4-vector to array accumulators" versions require 2 independent μ ops of this class on half of the 4-vector, split between first 2 srcs/dests and second 2 srcs/dests
	P: 4 [+2, +3, +4] [†] E: 4 [+2, +3, +4] [†]	P: 1 (2 on PE row) E: 2	Vector: BFloat16/Float16 multiply-accumulate and lengthen (H-to-S) (BFMLAL, BFMLSL, FMLAL, FMLSL)* *E: "4-vector with single/indexed vector" versions require 2 independent μ ops of this class on half of the 4-vector, split between first 2 srcs/dests and second 2 srcs/dests *E/P: "Multiple 4-vectors" versions require 2 independent μ ops of this class on half of the 4-vector, split between first 2 srcs/dests and second 2 srcs/dests

[†]Extra latencies to a consumer that is a [different computation instruction, horizontal extract, vertical extract]

A.7 SME Engine Load and Store Execution Unit Bandwidth

All SSVE loads write the Z vector register file and all SME loads write temporary storage located alongside the Z vector register file before an additional μ op moves the data into the Processing Grid. Stores operate similarly. As described in [Section 6.5, "SME Engine Load/Store Execution Unit Optimization"](#), all loads and stores access the Shared L2 Cache directly, and addresses are directed to a bank of the cache depending on hashed address. All loads nominally execute with a 19-cycle latency, but could be queued up by bank and suffer cache misses, etc. Loads are split into 64B and 128B μ ops, sometimes accessing only a portion of the byte range (due to size or alignment), one per cache line accessed, as described. Stores are split into 64B μ ops. Then two-vector load and store μ ops can be merged across instructions to create full accesses of 64B or 128B for loads and 64B for stores. The P and E SME engines can issue two (merged) load/store μ ops per cycle. The P SME engine can return up to 256 bytes of data per cycle while the E SME engine 128 bytes.

Because of the merging, determining actual load and store issue bandwidth of a code sequence requires analysis of neighboring memory operations. First identify the overall contiguous sizes of each instruction from [Table A.20: "SME Engine Load Instruction Bandwidth"](#) or [Table A.22: "SME Engine Store Instruction Bandwidth"](#), up to 256B, and then determine whether the access is likely to be aligned or unaligned. Then use [Table A.19: "SME Engine Load \$\mu\$ op Issue Bandwidth"](#) and [Table A.21: "SME Engine Store \$\mu\$ op Issue Bandwidth"](#) to determine how many μ ops are created to implement the overall access size. For loads, the specific pattern of 64B or 128B μ ops depends on whether the unaligned load accesses just the upper or lower 64B portion of the line or from both. Finally, examine nearby μ ops based on their relative addresses, identifying any μ ops that can merge to create a single larger or complete 64B or 128B access, and count only one issue for such merged μ ops.

Table A.19. SME Engine Load μ op Issue Bandwidth

Size and Alignment	Number of Load μ ops
64B Aligned	1
64B Unaligned	1 or 2, depending on required bytes
128B Aligned	1
128B Unaligned	2
192B Aligned	2
192B Unaligned	2 or 3, depending on required bytes
256B Aligned	2
256B Unaligned	3

Table A.20. SME Engine Load Instruction Bandwidth

Latency/Bandwidth	Operation Types and Descriptions (Example Instruction Mnemonics)
~19-20: 64B/128B/256B Load μ op	SSFP loads of types B/H/S/D/Q (LDR, LDUR) Note that <extend> addressing modes require an additional core ALU class μ op, as do those with address writeback One-element structure and simple vector loads (LD1, LDNT1, LDR) ZT0 lookup table loads (LDR)
~22-23: 64B Load μ op + 2 SSVE Shuffle μ ops	SSFP load pairs (LDP, LDNP) Note that instructions with address writeback require an additional core ALU class μ op
~23: 64B Load μ op + 1 SSVE Shuffle μ op	Vector load into ZA storage (LDR)
~22: 64B Load μ op + 1 SSVE Shuffle μ op	One-element structure loads, to all lanes (LD1R, LD1RQ)
~23: 128B Load μ op + 2 SSVE Shuffle μ ops	Two-element structure loads (LD2)
~24: 192B Load μ op + 3 SSVE Shuffle μ ops	Three-element structure loads (LD3)
~29: 256B Load μ op + 8 SSVE Shuffle μ ops	Four-element structure loads (LD4)

Table A.21. SME Engine Store μ op Issue Bandwidth

Size and Alignment	Number of Store μ ops
64B Aligned	1
64B Unaligned	2
128B Aligned	2
128B Unaligned	3
192B Aligned	3
192B Unaligned	4
256B Aligned	4
256B Unaligned	5

Table A.22. SME Engine Store Instruction Bandwidth

Bandwidth	Operation Types and Descriptions (Example Instruction Mnemonics)
64B/128B/256B Store μop	SSFP stores of types B/H/S/D/Q (STR, STUR) Note that <extend> addressing modes require an additional core ALU class μ op, as do those with address writeback One-element structure and simple vector stores (ST1, ST1NT, STR) ZT0 lookup table stores (STR)
1 Extract μ op + 64B Store μop	Vector store from ZA storage (STR)
1 SSVE Shuffle μ op + 64B Store μop	SSFP store pairs (STP, STNP) Note that instructions with address writeback require an additional core ALU class μ op
2 SSVE Shuffle μ ops + 128B Store μop	Two-element structure stores (ST2)
3 SSVE Shuffle μ ops + 192B Store μop	Three-element structure stores (ST3)
8 SSVE Shuffle μ ops + 256B Store μop	Four-element structure stores (ST4)



Appendix B. Dynamic Determination of Chip-Specific Capabilities

Some software may depend on knowledge of specific system details or hardware configurations. For example, software may want adapt to the following:

- Execute specific code paths when certain instructions are available
- Spawn different numbers of worker threads depending on the core count
- Apply cache blocking techniques to large data sets based on the cache configuration.

Software that contains hardcoded assumptions (that is, use a predetermined constant that cannot be changed without recompiling the code) about the existence or value of a parameter may not work as expected on systems where the existence or value is different. Instead of hardcoding values related to the underlying system, fetch those values dynamically from system global variables.

System parameters can be obtained both via a commandline tool `sysctl` and software function `sysctlbyname()`.

Some example parameters further described in subsequent sections:

- Virtual memory page size: `hw.pagesize`
- System cache line size: `hw.cachelinesize`
- Number of physical general purpose cores in the chip: `hw.physicalcpu_max`
- Arm-specific ISA feature, such as Dot Product: `hw.optional.arm.FEAT_DotProd`

For a complete list of available parameters on a particular chip, run the command `sysctl -a` in Terminal. Subsets can be selected, such as `hw`, `kern`, `user`, or `machdep`.

`int sysctlbyname(const char *name, void *oldp, size_t *oldlenp, void *newp, size_t newlen)` reads a parameter `name` and stores the parameter's value in `*oldp`. On an error, `sysctlbyname()` returns `-1` and populates `errno` with additional information such as `ENOENT` when the parameter does not exist. Ensure that the memory region pointed to by `oldp` is of sufficient size. Check the parameter's datatype using `sysctl -d <param>`. The function `sysctlbyname()` generates an `errno` of `ENOMEM` when the memory region is too small.

For instance, to programmatically determine when a process is running under [Rosetta translation](#), call the `sysctlbyname()` function with `sysctl.proc_translated` flag, as shown in the following example. The `processIsTranslated()` function returns the value `0` for a native process or `1` for a translated process based on the value stored in `*oldp`. The `processIsTranslated()` function returns `-1` when an error occurs, such as the parameter does not exist, based on the return value of `sysctlbyname()`.

```
int processIsTranslated() {
    int ret = 0;
    size_t size = sizeof(ret);
```

```
if (sysctlbyname("sysctl.proc_translated", &ret, &size, NULL, 0) == -1) {
    return -1;
}
return ret;
}
```

For additional context, see [Addressing Architectural Differences in Your macOS code](#) on the Apple Developer site.

Recommendation: Query `sysctl` parameters for determining ISA feature support and microarchitectural characteristics.

[Magnitude: High | Applicability: Low] When using specific ISA instructions not present on all target processors, use the `sysctl` functionality to determine availability of the feature, branching to alternate code when not available. When optimizing for specific microarchitectural characteristics such as L1 cache size, leverage `sysctl` to determine those characteristics.

B.1 ISA Features

The `sysctl` interface contains parameters that describe the ISA capabilities of the processor.

Apple silicon chips support the Arm A-Profile 64b AArch64 v8 and follow-on AArch64 v9 instruction set, depending on chip generation. Although software is compiled for a specific base instruction set and dynamically checking it is generally unnecessary, software can test for the Arm ISA using the following parameter:

Table B.1. Supported Base Instruction Set Parameter

Feature	
Parameter name <code>hw.optional.<></code>	Description
<code>arm64</code>	Arm A-Profile 64b AArch64 v8/v9

Apple silicon chips support several Arm features without official Arm "FEAT_*" names. Although parameters exist to identify these features, they are present in all Apple silicon chips beginning with M1 and A14 Bionic and do not need to be dynamically checked.

Table B.2. Supported Arm v8/v9 Unofficially Named EL0 Feature Parameters

Feature	
Parameter name <code>hw.optional.<></code>	Description
<code>armv8_crc32</code>	CRC32 instructions
<code>floatingpoint</code>	General support for floating-point instructions
<code>AdvSIMD</code>	General support for Advanced SIMD instructions
<code>AdvSIMD_HPFPcvt</code>	Advanced SIMD half-precision conversion instructions

Support for Arm features with standard names can be checked using `hw.optional.arm.<>`, such as Dot Product: `hw.optional.arm.FEAT_DotProd`. Available

features have a parameter value of 1. Unavailable features have a parameter value of 0. Unimplemented features return an error (ENOENT).

For the full list of implemented ISA feature names, see [Section 2.2, “Arm AArch64 ISA”](#).

Some Arm features also have legacy parameter names created prior to standardization. Legacy parameters continue to exist to support existing software. However, use the new standardized names whenever possible.

Table B.3. Legacy Feature `sysctl` Parameter Names

Legacy Parameter	New Parameter
<code>hw.optional.neon</code> (The term "NEON" is no longer broadly used by Arm. See Chapter 3, ISA Optimization: Advanced SIMD and FP Unit .)	<code>hw.optional.AdvSIMD</code>
<code>hw.optional.neon_hfpfp</code>	<code>hw.optional.AdvSIMD_HPFPFCvt</code>
<code>hw.optional.neon_fvp16</code>	<code>hw.optional.arm.FEAT_FVP16</code>
<code>hw.optional.armv8_1_atomics</code>	<code>hw.optional.arm.FEAT_LSE</code>
<code>hw.optional.armv8_2_fhm</code>	<code>hw.optional.arm.FEAT_FHM</code>
<code>hw.optional.armv8_2_sha512</code>	<code>hw.optional.arm.FEAT_SHA512</code>
<code>hw.optional.armv8_2_sha3</code>	<code>hw.optional.arm.FEAT_SHA3</code>
<code>hw.optional.armv8_3_compnum</code>	<code>hw.optional.arm.FEAT_FCMA</code>

These `sysctl` [hw feature parameters](#) are also documented on the Apple Developer site.

B.2 Cache and Topology Characteristics

The `sysctl` interface contains parameters that describe cache organization and topology. Software can query the interface to determine the configuration and adapt the workload accordingly, such as to block a data set for a particular cache size. The cache topology parameters do not include the M Cache, and as noted, software should not optimize for its capacity.

[Table B.4: “Per Performance Level Sysctl Parameters”](#) lists parameters that clearly define the system topology and include an example of M1 Ultra. The parameters include a hierarchy level called `perflevel{N}` that specify parameters by core type. Higher performing cores have lower perflevels. On M1 Ultra, the P core parameters are located under `perflevel0` and the E core parameters are located under `perflevel1`. These parameters are available in macOS 12, iOS 15, iPadOS 15, and later.

Table B.4. Per Performance Level Sysctl Parameters

New Per Performance Level Parameters	Definition	M1 Ultra Values	
		<code>perflevel0</code> P core	<code>perflevel1</code> E core
<code>hw.nperflevels</code>	Number of types of general purpose cores in the chip. The lower the perflevel, the higher the performance of the core type	2	

Table B.4. Per Performance Level Sysctl Parameters (cont.)

New Per Performance Level Parameters	Definition	M1 Ultra Values	
		perflevel0 P core	perflevel1 E core
hw.perflevel{N}.cpusperl2	For cores of perflevel N, the number of cores that share a L2 Cache	4	2
hw.perflevel{N}.cpusperl3	For cores of perflevel N, the number of cores that share a L3 Cache	N/A	
hw.perflevel{N}.l1d cachesize	For cores of perflevel N, the size in bytes of the L1 Data Cache	131072	65536
hw.perflevel{N}.l1i cachesize	For cores of perflevel N, the size in bytes of the L1 Instruction Cache	196608	131072
hw.perflevel{N}.l2 cachesize	For cores of perflevel N, the size in bytes of the L2 Cache	12582912	4194304
hw.perflevel{N}.l3 cachesize	For cores of perflevel N, the size in bytes of the L3 Cache	N/A	
hw.perflevel{N}.logicalcpu	For cores of perflevel N, the number of enabled logical cores in the chip	16	4
hw.perflevel{N}.logicalcpu_max	For cores of perflevel N, the number of logical cores in the chip	16	4
hw.perflevel{N}.physicalcpu	For cores of perflevel N, the number of enabled physical cores in the chip	16	4
hw.perflevel{N}.physicalcpu_max	For cores of perflevel N, the number of physical cores in the chip	16	4

Table B.5: “Updated Existing Sysctl Parameters” lists the legacy cache and topology parameter names and includes an example of M1 Ultra. The definitions of the cache parameters have been updated to *reflect the lowest performing cores* in the system beginning with macOS 12, iOS 15, and iPadOS 15. Prior versions of the operating systems reported values based on the boot core, which may have been either a P core or an E core, depending on the particular device.

Table B.5. Updated Existing Sysctl Parameters

Existing Parameters	Updated Definition (where applicable, applies to the <i>lowest performing cores in the system</i>)	M1 Ultra Values
hw.l1d cachesize	Size in bytes of the L1 Data Cache	65536
hw.l1i cachesize	Size in bytes of the L1 Instruction Cache	131072
hw.l2 cachesize	Size in bytes of the L2 Cache	4194304
hw.l3 cachesize	Size in bytes of the L3 Cache	N/A
hw.cachelinesize	Size in bytes of the system cache line	128
hw.memsize	Size in bytes of DRAM	<Various>
hw.pagesize	Size in bytes of a virtual memory page	16384

Table B.5. Updated Existing Sysctl Parameters (cont.)

Existing Parameters	Updated Definition (where applicable, applies to the <i>lowest performing cores in the system</i>)	M1 Ultra Values
hw.activecpu	The number of enabled logical processor cores in the chip (alias of hw.logicalcpu)	20
hw.ncpu	The number of logical processor cores in the chip (alias of hw.logicalcpu_max)	20
hw.logicalcpu	The number of enabled logical processor cores in the chip (alias of hw.activecpu)	20
hw.logicalcpu_max	The number of logical processor cores in the chip (alias of hw.ncpu)	20
hw.physicalcpu	The number of enabled physical processor cores in the chip	20
hw.physicalcpu_max	The number of physical processor cores in the chip	20

These `sysctl hw cache and topology parameters` are also documented on the Apple Developer site.