Apple Silicon CPU Optimization Guide: 3.0

2024-03-21

Author
Apple Inc.

Contents

List of Figures

List of Tables

Document Release Notes

Table 1. Change History

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

Chapter 1. Introduction

Apple silicon M Series and recent A Series chip designs include two flavors of general purpose cores, 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 (CPUs) and related components. Within a generation, such as M1 Generation chips (see Section 1.1, "Generation and Series Naming Conventions"), the general purpose compute core designs are the same. However, the core counts, chip topology, bandwidths, and other systems components and characteristics may differ. This guide covers Apple silicon M Series CPUs beginning with the M1 generation and Apple silicon A Series CPUs beginning with the 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 generations, 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 generations 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 Generation and Series Naming Conventions

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

- **M*x* Generation:** All M*x*, M*x* Pro, M*x* Max, and M*x* Ultra chips, such as M1, M1 Pro, M1 Max, and M1 Ultra chips

  - Note that for any particular Generation, only a subset of the CPU classes may be available
- **M Series:** All M*x* Generations, starting with M1 Generation
- **A Series:** All A*x* Bionic chips, starting with A14 Bionic chips
- **Apple silicon:** M Series and A Series

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.

While 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 complier optimizations that will optimize the code for the platform. For example, apply `-O3`. "Debug" configurations typically map to lower levels of optimization which results in less efficient code. Use "Release" configurations to enable more optimizations. Enable the compiler to use specific optimizations, transformations, or advanced instructions, many of which are tuned for both the ISA and the microarchitecture. For example, apply `-ffast-math` to allow the compiler to reorder floating point operations and use multiply-add

fused instructions. (See Section 3.5.4, "Fused Op with Add/Accumulate Instructions (Integer and ASIMD&FP Types)".) 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 5.4, "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 6, *Performance Monitoring*)
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, leveraging the full Instruction Set Architecture (ISA), especially if the compiler will not vectorize:** Insert sequences of custom of 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")
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 4.6.3, "L1 Data (L1D) Cache" and Section 4.6.12, "Prefetching")
7. **Optimized for the Microarchitecture, tuning for maximum throughput:** Use performance monitoring events in Instruments to identify inefficiencies. Carefully craft code sequences to achieve maximum execution throughput based on microarchitectural characteristics. (See topics across Chapter 4, *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 effort on the opportunities that matter.

1.3 High Magnitude and/or High Applicability Recommendations

While there are many potential optimization opportunities, consider these High Magnitude and/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: Table 4.10
- 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 (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 4.4.4, "Taken Branch Reduction"
- Use Conditional Instructions to Reduce Branch Mispredictions: Section 4.4.5, "Conditional Branch Mispredicts and Conditional Instructions"
- Avoid False Sharing by Allocating Independent Shared Variables to Different 128B Cachelines: Section 4.6.6, "Improving Cache Hierarchy Performance"
- Pack Hot Variables into the Smallest Set of Cachelines for Improved Cache Hierarchy Performance: Section 4.6.6, "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 v8.5 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 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 will make an "educated guess" as to the condition and/or target address of the branch. The processor will begin fetching instructions after fetching a branch according to the prediction. The processor will check 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 will start 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.

- **Biased Branch:** A branch that has a highly consistent outcome. For a conditional branch, execution is predominantly taken or is predominantly fall through. For an indirect branch, the target is predominantly the same target address.
- **Highly Predictable Branch:** A branch where the processor's prediction algorithms are highly likely to pick the correct outcome, and hence the processor is highly likely to fetch instructions along the correct path.

1.5 Performance Monitoring Events

Performance Monitoring Unit (PMU) Events provide insight into processor behavior. These events are denoted <Ev *event_name*> such as <Ev L1D_TLB_MISS_NONSPEC>. These events are available via the Instruments profiling application. See Chapter 6, *Performance Monitoring* for information about the Instruments application and for a list of available events and their definitions.

1.6 Decimal and Binary Data Quantities

This document uses the conventions specified in Section 2.5 of the January 1972 IEEE Recommended Practice: Rules for the Use of Units of the International System of Units standards document for decimal data quantities representation and Section 3 of the Mar 2008 IEEE Standard for Prefixes for Binary Multiples standards document for binary data quantities representation.

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

Note that in general, bandwidths are calculated in decimal quantities (e.g., MB/s), whereas memory sizes and footprints are calculated in binary quantities (e.g., MiB).

Chapter 2. ISA Optimization: Overview & Integer Unit

The following chapter describes aspects of the processor's Instruction Set Architecture. 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 LZFSE, 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 5, *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 generations. 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.

All Apple silicon CPUs documented in this guide (see Section 1.1, "Generation and Series Naming Conventions")x support floating point, floating point half precision converts, Advanced SIMD, and the CRC32 instruction.

Table 2.1 lists the base Arm ISA version and supported EL0 (unprivileged, colloquially "user code") features, base and optional, for each CPU.

Software can dynamically check for the existence of the features via the `sysctl` interface. See Appendix B, *Dynamic Determination of Chip-Specific Capabilities* for details.

Table 2.1. Supported Arm Base ISA and EL0 Features

Feature	Description [Clang ISA Extension Name]	CPU Support		
		M1 Gen. A14 Bionic	**M2 Gen. A15 Bionic**	**M3 Gen. A16 Bionic**
	Base ARM ISA	v8.5 (excluding FEAT_BTI)	v8.6	v8.6
FEAT_AES	Advanced SIMD AES instructions	Yes	Yes	Yes
FEAT_AFP	Alternate floating-point behavior	No	No	Yes
FEAT_BF16	Storage and arithmetic instructions of the Brain Floating Point (`BFloat16`) data type [`bf16`]	No	Yes	Yes
FEAT_BTI	Instructions to guard against the execution of instructions that aren't the intended target of a branch	No	Yes	Yes
FEAT_DPB	Clean data cache by address to Point of Persistence DC CVAP instruction	Yes	Yes	Yes
FEAT_DPB2	Clean data cache by address to Point of Deep Persistence DC CVADP instruction	Yes	Yes	Yes
FEAT_DotProd	Advanced SIMD Int8 dot product instructions	Yes	Yes	Yes
FEAT_ECV	Support for Enhanced Counter Virtualization, which enhances the Generic Timer architecture	No	Yes	Yes
FEAT_FCMA	Floating-point complex number instructions	Yes	Yes	Yes
FEAT_FHM	Floating-point half-precision multiplication instructions	Yes	Yes	Yes
FEAT_FlagM	Condition flag manipulation instructions	Yes	Yes	Yes
FEAT_FlagM2	Enhancements to condition flag manipulation instructions	Yes	Yes	Yes
FEAT_FP16	General half-precision floating-point data processing instructions	Yes	Yes	Yes

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

Feature	Description [Clang ISA Extension Name]	CPU Support		
		M1 Gen. A14 Bionic	M2 Gen. A15 Bionic	M3 Gen. A16 Bionic
FEAT_FRINTTS	Floating-point to integral valued floating-point number rounding instructions	Yes	Yes	Yes
FEAT_I8MM	Advanced SIMD Int8 matrix multiplication instructions [`i8mm`]	No	Yes	Yes
FEAT_JSCVT	JavaScript conversion instruction	Yes	Yes	Yes
FEAT_LRCPC	Load-acquire Release Consistency processor consistent (`RCpc`) instructions	Yes	Yes	Yes
FEAT_LRCPC2	Load-acquire Release Consistency processor consistent (`RCpc`) instructions version 2	Yes	Yes	Yes
FEAT_LSE	Atomic instructions to support large systems	Yes	Yes	Yes
FEAT_LSE2	Changes to single-copy atomicity and alignment requirements for loads and stores for large systems	Yes	Yes	Yes
FEAT_PMULL	Advanced SIMD PMULL instructions	Yes	Yes	Yes
FEAT_RDM	Advanced SIMD rounding double multiply accumulate instructions	Yes	Yes	Yes
FEAT_RPRES	Increased precision of Reciprocal Estimate and Reciprocal Square Root Estimate	No	No	Yes
FEAT_SB	Barrier instruction to control speculation	Yes	Yes	Yes
FEAT_SHA1	Advanced SIMD SHA1 instructions	Yes	Yes	Yes
FEAT_SHA256	Advanced SIMD SHA256 instructions	Yes	Yes	Yes
FEAT_SHA3	Advanced SIMD SHA-3 instructions	Yes	Yes	Yes
FEAT_SHA512	Advanced SIMD SHA512 instructions	Yes	Yes	Yes
FEAT_SSBS	Instructions to control speculation of loads and stores	Yes	Yes	Yes

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, though, 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 and dispatch the preferred path based on the dynamic checking strategy mentioned above.

2.3 Arm Reference Documents

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

- **A-Profile:** https://developer.arm.com/architectures/instruction-sets/base-isas/a64
- **Arm Architecture Reference Manual:** https://developer.arm.com/documentation/ddi0487/latest
- **Advanced SIMD and FP ISA (also referred to as NEON):** https://developer.arm.com/architectures/instruction-sets/simd-isas/neon
- **Advanced SIMD and FP ISA high-level language Intrinsics:** https://developer.arm.com/architectures/instruction-sets/intrinsics/
- **Software ABI:** https://developer.arm.com/architectures/system-architectures/software-standards/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:** https://developer.arm.com/documentation/100941/0100/ARMv8-A-Memory-systems
- **Virtual Memory Management:** https://developer.arm.com/architectures/learn-the-architecture/memory-management

2.4 ISA Characteristics

Considering the **Instruction Set Architecture (ISA)**, the foundations of the Arm ISA are similar to other ISAs. Commonalities include:

- **Rich instruction set:** The Arm ISA is a rich instruction set architecture. It features an extensive instruction library of both compiler targetable and special purpose instructions. While 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:** While 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 ASIMD&FP unit vector registers :** Vector registers are 128b in length. These registers support scalar and packed 64b, 32b, and 16b-sized floating point values and packed 64b, 32b, 16b, and 8b-sized integer values. The Arm ISA does not support 80b extended floating point precision.
- **Separate source and destination registers (i.e., non-destructive 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. While 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"

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 will be 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.

ASIMD&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 ASIMD&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 ASIMD&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. Then 64b, the upper 64 bits of the register are ignored on a read and cleared to zero on a write.

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: 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 will 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.

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 ASIMD&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 will also be read as sources according to the instruction

pseudocode. 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&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 Multiply:**
  - `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 Insert**
  - `INS`
- **Second Half of Narrowing Operations:** (see Section 3.3.16, "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 4.4.8, "Multi-µop Instructions" and Section 4.6.1, "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 will still read 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 *elements*)	Single register operations `LDR X1, [X0, #8]`
	9b offset, Sign Extended (-256 to 255 *bytes*)	Single register unscaled operations `LDUR X1, [X0, #-6]`
	7b offset, Sign Extended, implied scale by transfer size (-64 to 63 *elements*)	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 *bytes*)	Single register operations `LDR X1, [X0, #8]!`
	7b offset, Sign Extended, implied scale by transfer size (-64 to 63 *elements*).	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 *bytes*)	Single register operations `LDR X1, [X0], #8`
	7b offset, Sign Extended, implied scale by transfer size (-64 to 63 *elements*).	Paired register operations `LDP X1, X2, [X0], #16`
	Offset must equal structure size	ASIMD&FP operations only

Table 2.2. Load and Store Addressing Modes *(cont.)*

Addressing Mode / Form	Immediate	Example
		`LD1 {Q1.16B,Q2.16B}, [X0], #32`
Post-indexed (register) Address = base Base updated with base + reg offset `[base], X`	None	ASIMD&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 *bytes*)	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.

- Zero extended 32b: `Wm`, or `Wm UXTW`, or `Wm UXTW #0`, or `Xm UXTW`, or `Xm UXTW #0`
- Zero extended 32b with scale by transfer size: `Wm UXTW #2/#3`, or `Xm UXTW #2/#3`
- Sign extended 32b: `Wm SXTW`, or `Wm SXTW #0`, or `Xm SXTW`, or `Xm SXTW #0`
- Sign extended 32b with scale by transfer size: `Wm SXTW #2/#3`, or `Xm SXTW #2/#3`
- 64b: `Xm`, or `Xm LSL #0`, or `Xm SXTX`, or `Xm SXTX #0`
- 64b with scale by transfer size: `Xm LSL #2/#3`, or `Xm SXTX #2/#3`

> **Recommendation: Use the `LSL` Option in Memory Addressing to Scale the Second Register Operand by Transfer Size:**
>
> **[Magnitude: Low | Applicability: High]** Specify `LSL #<transfer_size>` to scale the second register operation by the transferl size. This is often used to convert an array index to an array byte offset.

General-purpose arithmetic (ALU) instructions can calculate the result of most addressing modes and write the address to a general-purpose register. The Base Only, Base Plus Offset (immediate), Base Plus Offset (register), and Literal (PC-relative) are available.

With regard to extended register mode, the ALU instructions support 1B (B) and 2B (H) base sizes in addition to the 4B (W) and 8B (X) base sizes supported by the memory instructions: `(S|U)XT(W|H|B|X)`. Further, the shift amount is a 3b value that can be set between 0 and 4, rather than just to the transfer size.

Last, a more flexible shifted register form is available with a shift amount from 0-63 for the following shift types: `LSL` (Logical Shift Left), `LSR` (Logical Shift Right), and `ASR` (Arithmetic Shift Right).

For example:

- **ADD (immediate):** `ADD Xd|SP, Xn|SP, #imm{, shift}`
- **ADD (shifted register):** `ADD Xd, Xn, Xm{, shift #amount}`

- **ADD (extended register):** `ADD Xd|SP, Xn|SP, Rm{, extend {#amount}}`

Recommendation: Use Immediate Values as Shift Amounts Applied to the Second Register Operand to Replace Multi-Instruction Sequences:

[Magnitude: Low | Applicability: Medium] Where possible, use the implicit shift functionality available in various instructions to avoid a separate shift instruction.

2.8.1 Avoid Chains of Pre- and Post-Indexed Operations

Pre- and post-indexed instructions compress two operations into one instruction and are useful for improving code density. However, executing a chain of pre-indexed or post-indexed memory operations is also inefficient because each instruction includes a separate update of the address register, making subsequent instructions dependent on prior instructions. For example, the load into `X2` cannot begin until `X0` is updated by the load into `X1`.

```
LDR X1, [X0], #8
LDR X2, [X0], #8
LDR X3, [X0], #8
LDR X4, [X0], #8
```

Instead, make a single update to the address register either through a dedicated `ADD` or `SUB` instruction or though a single pre-/post-indexed memory operation, along with additional memory operations using base plus immediate addressing.

```
LDR X2, [X0, #8]
LDR X3, [X0, #16]
LDR X4, [X0, #24]
LDR X1, [X0], #32
```

Note that base plus immediate offset for single registers offers only an unsigned positive offset range. Therefore when accessing a sequence of elements, access the elements from the original pointer first, before advancing the pointer.

Stack push and pop operations can be naively translated into pre-indexed and post-indexed addressing modes. The Arm equivalent of a push operation to a stack that grows down toward 0 is `STR Xn, [XSP, #-8]!` which first decrements the stack pointer by the operand size, then stores the value to the computed address, and finally updates the stack pointer with the computed address. The Arm equivalent of a pop operation is `LDR Xn, [XSP], #8` which first loads the value from stack pointer address, then adds the operand size to the stack pointer, and then updates the stack pointer. However, rather than using a sequence of pre-indexed and post-indexed memory operations, use a single pre-indexed store to allocate all of required the stack space and write the first value, followed by base plus offset stores. Similarly, read all but the last of the values off of the stack with base plus offset loads, followed by a final post-indexed load that also deallocates the stack space.

Further, use load and store pair instructions (`LDP` and `STP`), which read from two sequential locations in memory into two registers and write to two sequential locations in memory from two registers. Use of these instructions increases code density.

```
# start of function
STP X28, X27, [XSP, #-96]!
STP X26, X25, [XSP, #16]
STP X24, X23, [XSP, #32]
STP X22, X21, [XSP, #48]
STP X20, X19, [XSP, #64]
STP X29, X30, [XSP, #80]
# function body
LDP X29, X30, [XSP, #80]
LDP X20, X19, [XSP, #64]
LDP X22, X21, [XSP, #48]
LDP X24, X23, [XSP, #32]
LDP X26, X25, [XSP, #16]
LDP X28, X27, [XSP], #96
RET X30
```

The Arm Application Binary Interface (ABI) states that stacks grow downward toward 0. Further, a process may only store data in the closed interval of the entire stack delimited by [SP, stack base - 1]. Therefore, when applying the optimization above, allocate space on the stack first with an arithmetic operation (or along with the first store), and deallocate 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-sized 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 then 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 will typically result 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 will be 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 4.5.3.3, "Register Constants".

Recommendation: Use Short Sequences of ALU Operations to Construct Constants:

[Magnitude: Low | Applicability: Medium] While creating a slight increase in code size, use short sequences of moves, adds, and shifts to create constants. This will 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 will include a name instead of the immediate values, which will be converted to immediate values by the linker.

```
000000000000132c  ADRP X0, #0x84          ; 0x84000 + 0x001000 = 0x85000
0000000000001330  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
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

†Unordered means at least one NaN operand.

††The NV condition code does not appear in the Arm Architecture Reference Manual, however some assemblers accept it. While 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.

2.10 Conditional Instructions

The Arm instruction set offers a number of instructions that modify register values depending on a condition. See Section 4.4.5, "Conditional Branch Mispredicts and Conditional Instructions" for a list of these instructions and recommendations on when to employ them.

2.11 Fused Op with Add/Accumulate Operations (Integer Unit)

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.

The benefits are similar to the Advanced SIMD Fused Op with Add/Accumulate Instructions and both are detailed jointly in Section 3.5.4, "Fused Op with Add/Accumulate Instructions (Integer and ASIMD&FP Types)".

2.12 Memory Model, Barriers, and Synchronization

Almost all EL0 instruction and data bytes reside in Arm "normal" type memory. Normal memory employs a "weakly ordered" memory model. While a thread's own loads and stores will 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.

Apple silicon offers additional memory types for specific uses with different memory ordering properties. Given their narrow uses, they are beyond the scope of this document.

For more information on Arm memory types, weakly ordered memory, and barriers, see ARMv8-A Memory Systems".

Arm offers further examples of barrier behavior in Appendix K11 "Barrier Litmus Tests" in the Arm Architecture Reference Manual.

Last, see topics in Chapter 5, *Asymmetric Multiprocessor (AMP) Optimization* that discuss APIs for inter-thread communication and thread checking.

Recommendation: Use Barrier Instructions to Enforce Memory Ordering Due to Weakly Ordered Memory Model:

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.

[Magnitude: High | Applicability: Low]

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:

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 will evict other useful cached instructions. Avoid frequent code deployment, and hence frequent instruction cache invalidations, as much as possible.

[Magnitude: Low | Applicability: Low]

2.14 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 6.2, "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 Generation and following, and A15 Bionic 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 4.6: "Common Branch Instruction Mix and Direction Metrics".

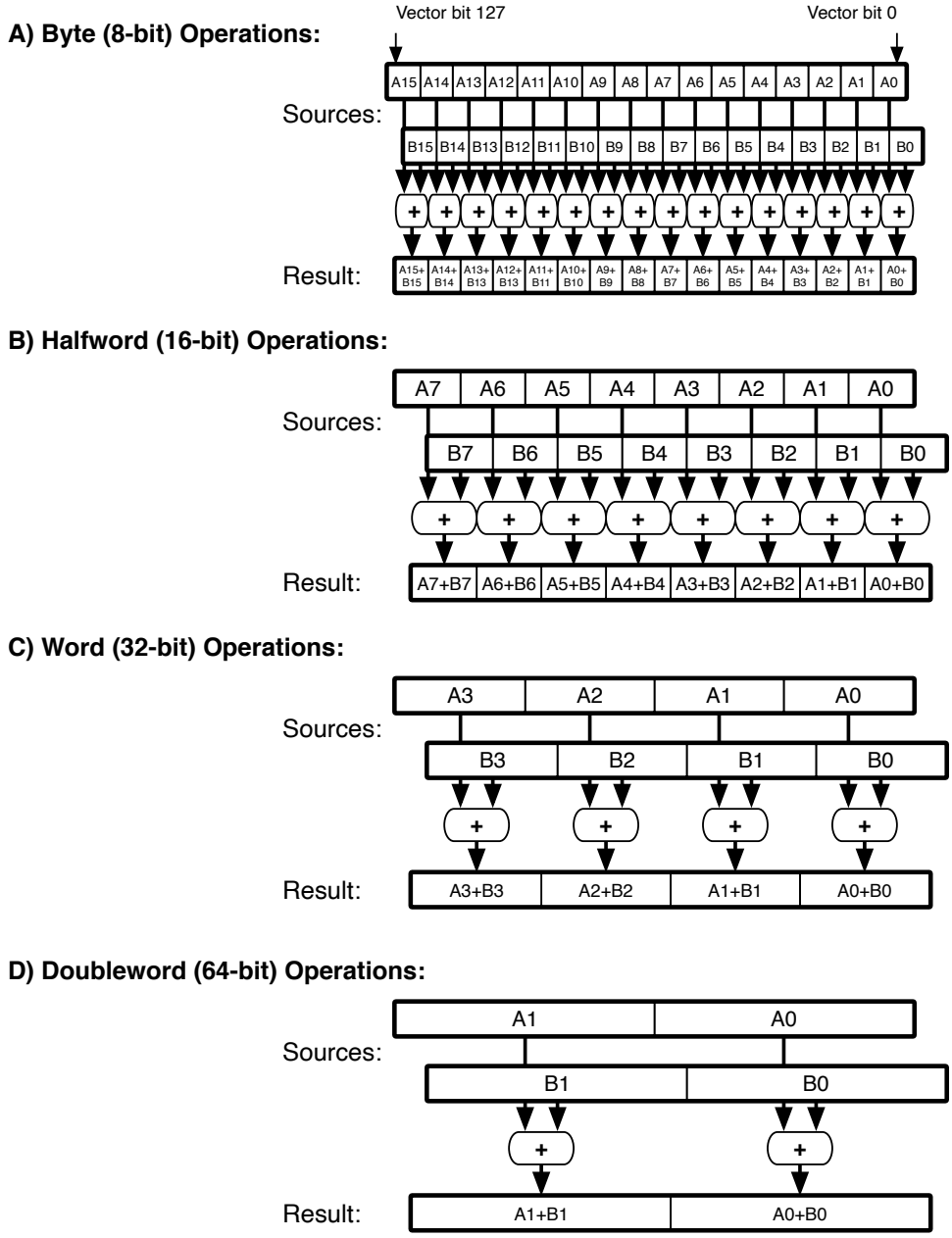
Chapter 3. ISA Optimization: Advanced SIMD and FP Unit

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.

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)

"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 precision 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.

3.1.1 FP Rounding, NaN, and Denormal Controls

Various rounding, NaN handling, and denormal handling controls are available in the FPCR (Floating Point Control Register). FPCR can be accessed directly through the MSR/MRS instructions or preferably through `fenv` functions declared in `fenv.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 Available in the Floating Point Control Register (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.
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 https://developer.arm.com/documentation/ddi0595/2021-03/AArch64-Registers/FPCR--Floating-point-Control-Register for more details.

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 3.4: “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 3.5: “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 8/16/32/64 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/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 3.4. 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 3.5. 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 Neon 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. While 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 3.5: "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-sized number of prefixes, suffixes, and about 60 core 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 3.6: "Advanced SIMD Instruction Mnemonic Prefixes" and the suffixes are summarized in Table 3.7: "Advanced SIMD Instruction Mnemonic Suffixes". The prefixes, core mnemonics, and suffixes are further described in subsequent sections.

Table 3.6. 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 / 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
Negate	`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

Table 3.7. Advanced SIMD Instruction Mnemonic Suffixes

Field Name	Coding	Types	Description
Cryptographic	Various	N/A	Various AES / SHA / etc. cryptographic instruction specifications
Load/Store Interleaving	`1` / `2` / `3` / `4`	Load/Store	Selects how elements are interleaved as they are loaded from memory
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 ±∞ 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)

Table 3.7. Advanced SIMD Instruction Mnemonic Suffixes *(cont.)*

Field Name	Coding	Types	Description
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 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 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 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 will 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 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 will result in a result 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 will be multiplied by a 0.5x factor 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 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 Negated: N

The floating point multiplication operations with the `N` prefix negate their results before returning the value. They can save many explicit `FNEG` instructions when multiplying-and-adding long sequences of terms.

3.3.7 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 Core Instruction Mnemonic

In the middle of each mnemonic is the abbreviated name of the actual operation being performed by the instruction. Most are three letters long. Table 3.8: "Advanced SIMD

Core Instructions Grouped By General Function" groups the instructions by type. 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 Core Instructions 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`

3.3.9 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.

3.3.10 Load/Store by Element Interleaving Factor: 1 / 2 / 3 / 4

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 3.3.18, "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.

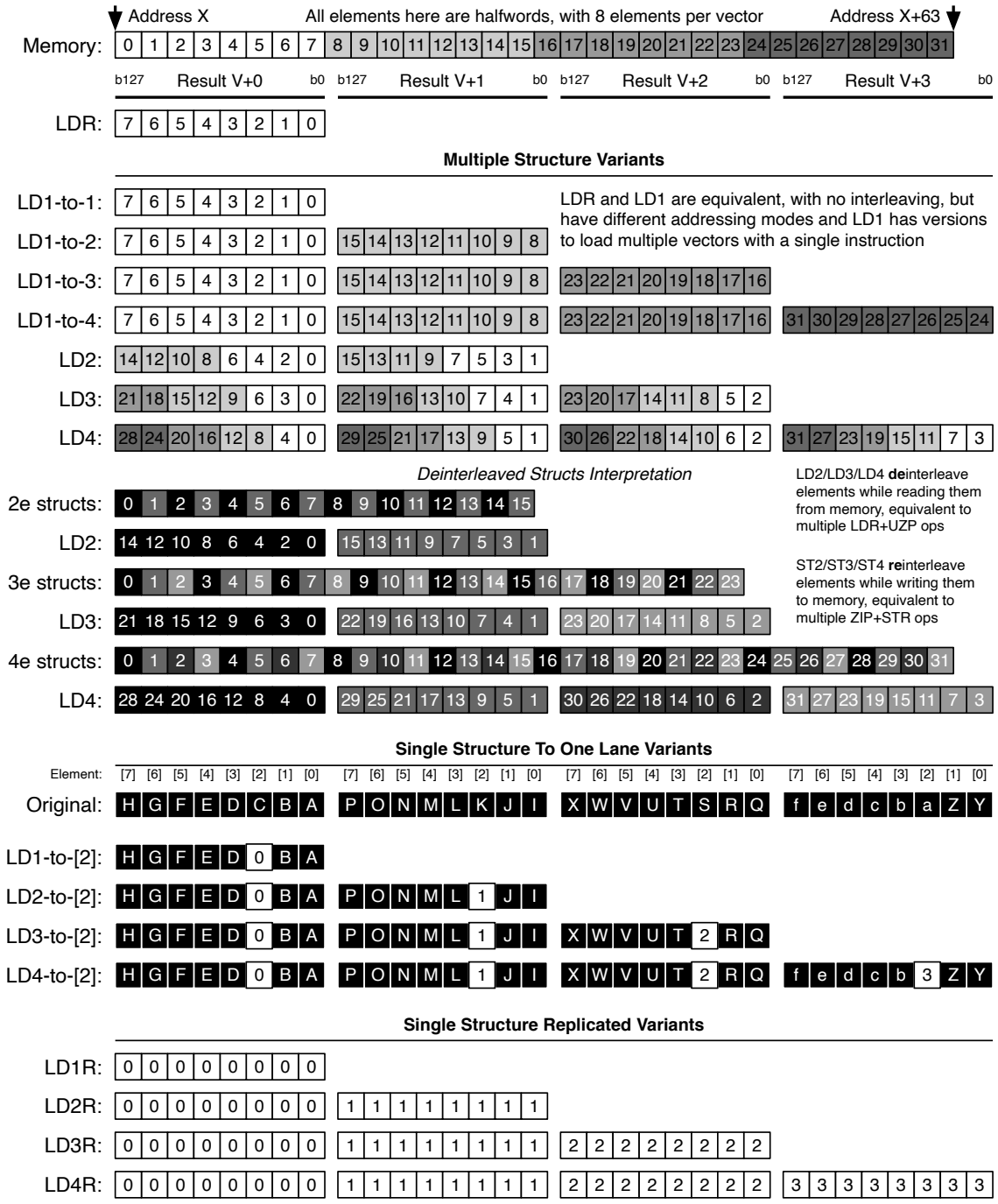
For example, when loading an array of pixel values consisting of interleaved RGB values, a `LD3` instruction can be used to de-interleave these values and bring them in as three separate all-R, all-G, and all-B vectors. A similarly packed array of RGBa pixels (RGB

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 3.2: "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 4.4.8, "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 3.2. Advanced SIMD Load/Store Element Interleaving and De-Interleaving



Recommendation: Leverage `LD<n>` and `ST<n>` Instructions for Interleaved Memory Structures:

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.

[Magnitude: Medium | Applicability: Low]

3.3.11 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 3.9. 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 `MOVI` of `#0` instead. See "Modified immediate constants in A64 instructions" in the Arm Architecture Reference Manual for additional details.

3.3.12 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 3.10: "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 will include a condition code that is evaluated against the flags.

Table 3.10. Advanced SIMD Comparison Conditionals

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

3.3.13 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

will be 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.

3.3.14 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 will need to be rounded off. Variations include round towards: nearest, zero, +∞, –∞, 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 3.3.15, "Different Output Type: (none) / F / S / U" just after the N suffix.

3.3.15 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, "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.

3.3.16 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 and/or precision 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.

Narrowing versions in particular often have saturating and/or rounding options available to define the behavior regarding bits that are cut off at the high or low ends as the elements are packed down to a smaller size. They also sometimes come in high-half versions (see Section 3.3.17, "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 3.3.18, "Lower-Half / Upper-Half: (none) / 1 / 2 " and Section 3.3.19, "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 ASIMD&FP register to the corresponding vector element in the second source ASIMD&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 ASIMD&FP register, and RADDHN2 writes the upper half of the destination ASIMD&FP register.

Recommendation: Use Mathematical Instructions that Include Type Conversions:

Instead of using explicit extension instructions, such SXTL, on inputs prior to a calculation that results in a larger-typed size, use instructions that include the conversion in with the calculation, such as SADDL and SADDL2.

[Magnitude: Medium | Applicability: Medium]

3.3.17 Return High Half : H

Eligible core 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 core 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 will be in one vector, while the low-order half will be in a second. While this is feasible for some scenarios, usually pairs of L suffix lengthening operations (see Section 3.3.16, "Narrow, Long,

Wide: N / L / W") will be 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).

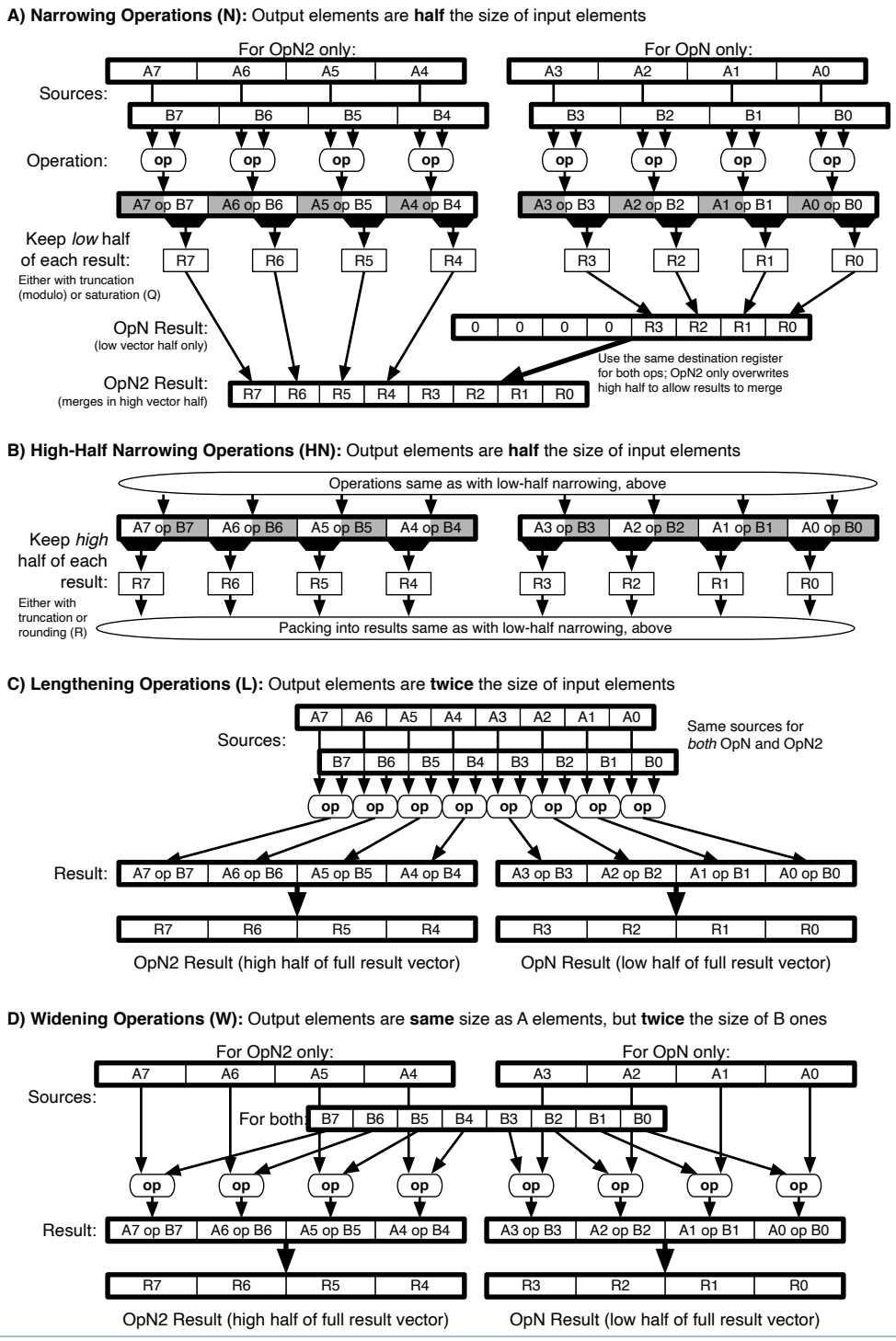
3.3.18 Lower-Half / Upper-Half: (none) / 1 / 2

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 will be 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 3.3: "Advanced SIMD Narrow, Lengthen, and Widen".

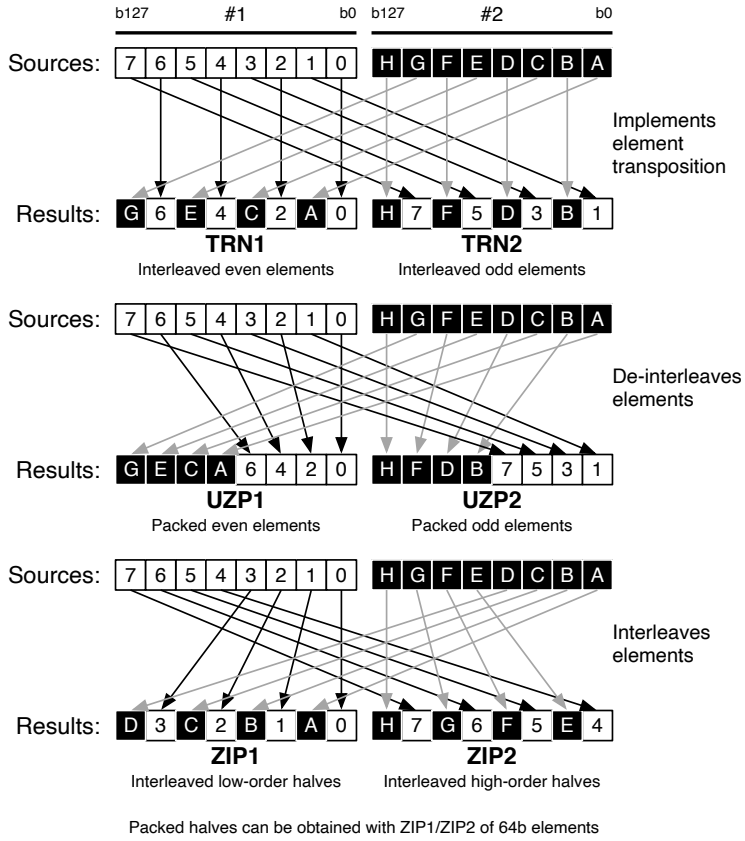
Figure 3.3. Advanced SIMD Narrow, Lengthen, and Widen



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 3.4: “Advanced SIMD Byte Transformations”. An explicit `1` notation is required for these instructions; assemblers will not assume that for you. Shuffle and Permute Instructions: “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 3.4. Advanced SIMD Byte Transformations



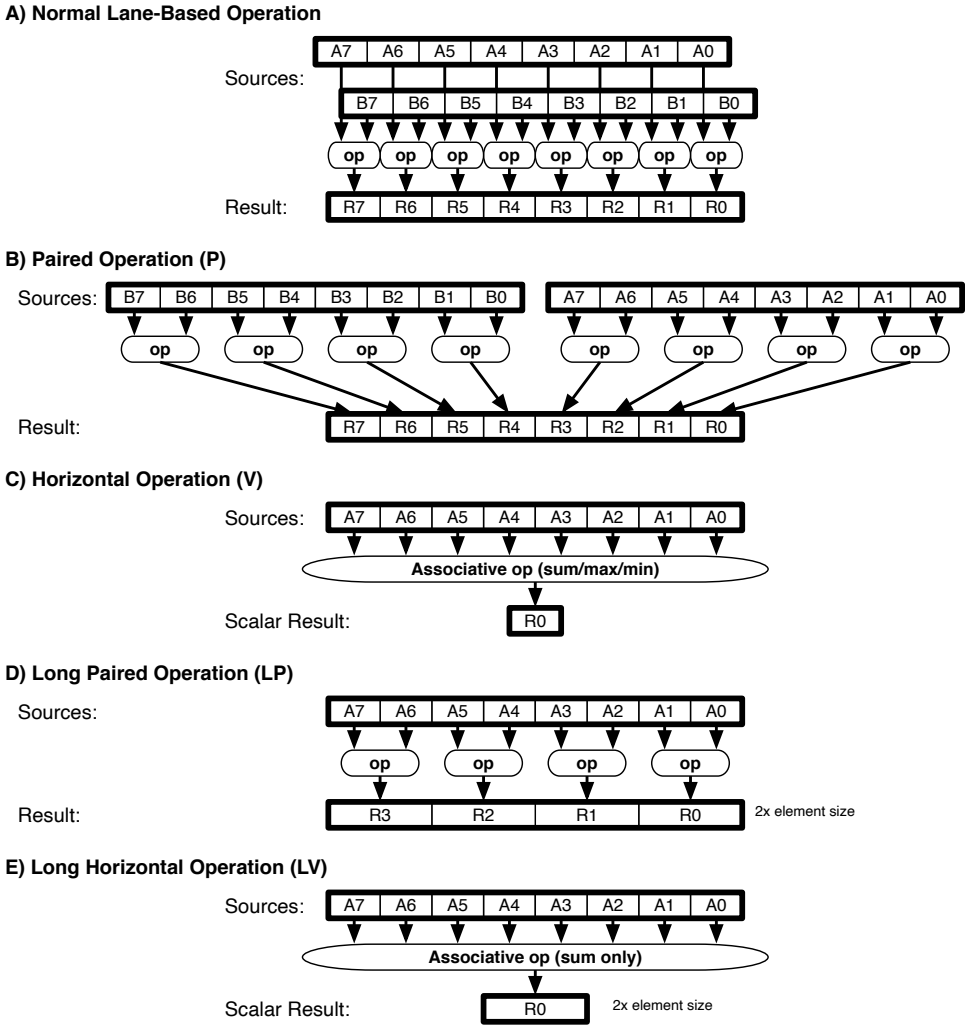
3.3.19 Cross-Lane Paired or Horizontal: P / V

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 3.5: "Advanced SIMD Cross Lane Paired and Horizontal Instructions" (B) illustrates a paired instruction. Trees of repeated *paired* operations will 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 3.5: "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 3.5: "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 `L` (long) suffix from Section 3.3.16, "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 3.5: "Advanced SIMD Cross Lane Paired and Horizontal Instructions" (D-E). These can prevent overflow from occurring, if it might be a concern.

Figure 3.5. Advanced SIMD Cross Lane Paired and Horizontal Instructions

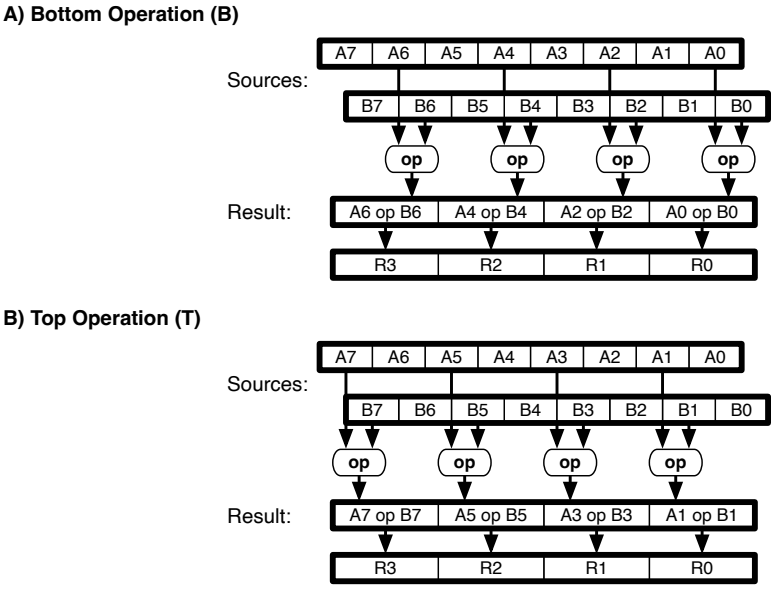


3.3.20 Bottom or Top Half of Element Pairs: B / T

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 3.3.18, "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 will 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 depictexd in Figure 3.6. Only a few brain floating point (`bfloat16`) operations utilize this interleaved approach.

Figure 3.6. 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.

Before writing custom code using assembly or intrinsics, explore the auto-vectorization capabilities of the compiler. Use the `-ffast-math` compiler flag, when applicable, to allow the compiler to aggressively optimize floating point code. For a full list of the transformations, see the Clang User Manual.

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:

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 will typically implement these tasks efficiently without any developer intervention.

[Magnitude: High | Applicability: High]

3.4.1 C Vector Types

Before using the intrinsics themselves, the developer must convert variables that will be used with the intrinsics into the appropriate vector types. The available types are summarized in Table 3.11: "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 3.11. 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`
Unsigned Integer or Fixed Point	8b	`uint8_t`	`uint8x8_t`	`uint8x16_t`
	16b	`uint16_t`	`uint16x4_t`	`uint16x8_t`

Table 3.11. Advanced SIMD Intrinsic Data Types *(cont.)*

Type	Element Size	Scalar	64b Vector	128b Vector
	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 `float32x4`**`x4`**`_t`. The various `float32x4_t` sub-vectors inside are defined as `val[<M>]`.

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

While 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. `float32x4`**`x1`**`_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, "Overall instruction data type: (none) / BF / F / S / U") and half-select suffix (`1`/`2` from Section 3.3.18, "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 3.3.18, "Lower-Half / Upper-Half: (none) / 1 / 2 ").

• **`<input_type>`:** The source data type. These indicators are taken from the list in Table 3.5: "Vector Register Interpretation", and specify both the type and size of the *sources*, which need to be specified with the types from Table 3.11: "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.

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

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").

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

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&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 will allocate and manage 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 will occur 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>
```

Since 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`-sized) 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. Since 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 3.3.14, "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 3.3.18, "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, "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>
```

While 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(vcvt_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 will insert 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#>);
```

While 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 3.11: "Advanced SIMD Intrinsic Data Types".

3.4.4.6 Byte Rotations Operations

The "extract" operation allows a vector to be extracted from any bytewise 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 will be 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-typed 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 3.3.18, "Lower-Half / Upper-Half: (none) / 1 / 2 " and Section 3.5.7, "Shuffle and Permute Instructions"). The intrinsics or these instrucitons 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 will be necessary if you want to use `TBL` or `TBX` with non-byte type vectors. 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 ARMv7 operations without this `q`, which used 64-bit vectors for their tables. While still defined, intrinsics for these old operations should generally *not* be used in ARMv8 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 will automatically insert 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 3.3.10, "Load/Store by Element Interleaving Factor: 1 / 2 / 3 / 4"):

```
<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, a typical use of `LD3/ST3` would be to load 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. While complex numbers are typically loaded in interleaved form in order to leverage the complex number instructions (see Section 3.3.7, "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. While 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 will inevitably arrive 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 hardware will provide 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 4.3, "Processor Pipeline Fundamentals".
- **Extensive loop unrolling and interleaving:** In order to provide enough additional inter-instruction parallelism, software will often contain 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. While 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 zeroes, filter weights, byte masks, permutation tables (for `TBL` and `TBX` instructions), and the like. While 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 will usually be 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:

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 will spill variables to memory and fill them back when needed, resulting 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.

[Magnitude: Medium | Applicability: Medium]

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 4.5.1, "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 3.3.19, "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 zeroes 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-sized. 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 (`bfloat16`) 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:

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.

[Magnitude: Medium | Applicability: Medium]

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

The Arm ISA offers a limited set of "fused" instructions that combine an operation with an add or accumulate — for example, `FMLA`, is a floating point fused multiply-add to accumulator. The processor decodes, issues, executes, and retires these instructions as single µops for increased computation efficiency.

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 ASIMD&FP registers, and accumulates the results with the vector elements of the destination ASIMD&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 decode and issue throughput:** All op-add instructions effectively combine two operations. As long as both operations are useful, combining them in the same instruction effectively doubles throughput of basic operations throughout the processor.
- **Lower overall latency:** In many cases, the op-add instruction executes with the same latency as an instruction that only executes the base op. However, the latency

through the add input may be longer that 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.) While 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 alleviated by unrolling and restructuring the algorithm to have 4 independent chains of `FMLA`s that are combined at the end. See Appendix A, *Instruction Latency and Bandwidth*.

- **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 about the "`-ffast-math`" compiler flag below.
- **Reduced latency through the add/accumulator source (Integer Execution Unit only):** In the integer unit, an 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.

Fused two operation instructions:

- Integer Execution Instructions:
  - Multiply-add/subtract (separate destination): `MADD`, `MSUB`
  - Multiply-add/subtract long (separate destination): `SMADDL`, `UMADDL`, `SMSUBL`, `UMSUBL`
- SIMD 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 (`bfloat16`) dot product (accumulator): `BFDOT`
- Brain floating point (`bfloat16`) widening multiply-add (accumulator): `BFMLALB`, `BFMLALT`
- Brain floating point (`bfloat16`) 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.

Recommendation: Use Op-Add (e.g., Multiply-Accumulate) to Increase Code Density and Reduce Latency:

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.

[Magnitude: High | Applicability: Medium]

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 3.12. Example Complex Number Operations

Operation	Instruction Sequence
A += B x C	`FCMLA.<T> Va, Vb, Vc, #0` `FCMLA.<T> Va, Vb, Vc, #90`
A += conj(B) x C	`FCMLA.<T> Va, Vb, Vc, #0` `FCMLA.<T> Va, Vb, Vc, #270`
A –= B x C	`FCMLA.<T> Va, Vb, Vc, #180` `FCMLA.<T> Va, Vb, Vc, #270`
A –= conj(B) x 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 generations 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:

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

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:

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

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

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 2x8 matrix of 8-bit integers by another containing an 8x2 matrix of 8-bit integers, with the individual integers either signed, unsigned, or both (in the latter case, the 2x8 is unsigned, the 8x2 signed). Much like with the dot product instructions, these results are then accumulated into a vector containing four 32-bit accumulators, arranged as a 2x2 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 will need to be used regularly in quick succession, several registers may have to be dedicated to holding these mapping control vectors.

As a result, it is a better to use one of the "dedicated" byte-rearranging instructions (`DUP`, `EXT`, `INS`, `REV`, `TRN`, `UZP`, and `ZIP`) if possible. Table 3.13: "Advanced SIMD Shuffling Operations" lists these various instructions, along with brief descriptions and a summary of their primary intended uses.

Table 3.13. 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 3.14. Input-Output Patterns for the Advanced SIMD Shuffling Operations: Byte-Sized 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 3.15. Input-Output Patterns for the Advanced SIMD Shuffling Operations: Half-Sized 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
	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 3.16. Input-Output Patterns for the Advanced SIMD Shuffling Operations: Word-Sized 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

Table 3.16. Input-Output Patterns for the Advanced SIMD Shuffling Operations: Word-Sized Elements

Format	Op	Vector Byte Lanes			
		15,14,13,12	11,10,9,8	7,6,5,4	3,2,1,0
	ZIP2	D	3	C	2

Table 3.17. Input-Output Patterns for the Advanced SIMD Shuffling Operations: Double-Sized 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
	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. While 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 1–4 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:

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.

[Magnitude: Medium | Applicability: Medium]

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 2x2 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) {

        // -- Read block
        a0 = in[k*BLOCK_SIZE+0][l*BLOCK_SIZE];
        a1 = in[k*BLOCK_SIZE+1][l*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[l*BLOCK_SIZE+0][k*BLOCK_SIZE] = b0;
        in_t[l*BLOCK_SIZE+1][k*BLOCK_SIZE] = b1;
    }
}
```

Each block only requires 2 loads, 2 stores, and 2 `TRN` instructions, instead of 4 scalar loads and 4 scalar stores. While 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-sized `TRN` instructions alone will not completely transpose a vector containing 4 word-sized values. Instead, 32-bit `TRN` instructions will need to be mixed with 64-bit `TRN` instructions in a 2-pass sequence on a 4x4 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][l*BLOCK_SIZE];
        a1 = in[k*BLOCK_SIZE+1][l*BLOCK_SIZE];
        a2 = in[k*BLOCK_SIZE+2][l*BLOCK_SIZE];
        a3 = in[k*BLOCK_SIZE+3][l*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[l*BLOCK_SIZE+0][k*BLOCK_SIZE] = c0;
            in_t[l*BLOCK_SIZE+1][k*BLOCK_SIZE] = c1;
            in_t[l*BLOCK_SIZE+2][k*BLOCK_SIZE] = c2;
            in_t[l*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 4 loads, 4 stores, and 8 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-sized elements using 8x8 blocks, or even matrices of byte-sized elements using 16x16 blocks. For smaller element sizes, additional passes are required, moving from transposing entire halves of each vector down to transposing individual elements: 8x8 uses 3 passes of TRN instructions per block (64-bit / 32-bit / 16-bit), while 16x16 uses 4 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. While 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:** BFMMLA
- **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 3.3.20, “Bottom or Top Half of Element Pairs: B / T”). BFDOT and BFMMLA 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&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 will 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`:

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

Use these dedicated instructions instead of an `SHRN` because they will 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 MxP X PxN = MxN 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. While 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 1xP X PxN = 1xN (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 MxP X Px1 = Mx1 (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 will encounter 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 and/or cache configurations.
- **Blocking:** The matrix-matrix example code will read 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 matrix will tend to stream through and experience constant cache misses for larger matrix 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 4.6.12.2, "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 will likely be 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 4. This will allow 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 will need 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

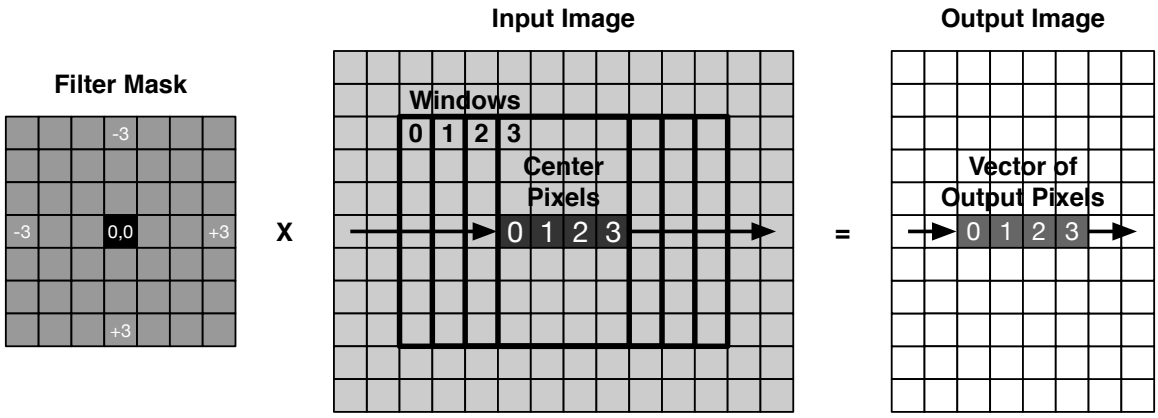
Filtering techniques are used regularly in the field of image processing. Many of these filters are two-dimensional versions of FIR filters, modifying values of pixels based on their current value and those of their nearest neighbors. This section illustrates a few ways that such filters can be constructed, 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 3.7: “Advanced SIMD Pixel Filter”.

Figure 3.7. 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)    (*((uint8x16_t *)(&(uint8Ref))))
#define ST_U8V(uint8Ref)    (*((uint8x16_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. While 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, $\texttt{MASKSIZE}^2$ 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 `LD`s and `ST`s 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 will need to insert 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(vqrshrn_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 7 of 8 parallel lanes will be in use in each row, wasting 12.5% of each vector. While reasonable with this configuration, smaller values of MASKSIZE will 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 will become less signficant for very large `MASKSIZE` values.
- Since 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, then there will be more pressure on the load/store unit because the filter weights need to be re-loaded 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 will 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 will only need to run on more recent chips.

Chapter 4. Core Microarchitecture Optimization

The details covered in Chapter 4, *Core Microarchitecture Optimization* 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 from generation to generation, 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 CPU series and generations.

4.1 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 performance 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 will 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 5.1, "Prioritizing Work".

Several cores of the same type are grouped together and share a second level cache and are collectively known as a cluster.

The overall parametrized topology is shown in Figure 4.1: "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 4.1. In Ultra chips, the two dies are connected via low-latency high-bandwidth Ultra Fusion interconnect.

Figure 4.1. General Purpose Compute Complex and Related Components

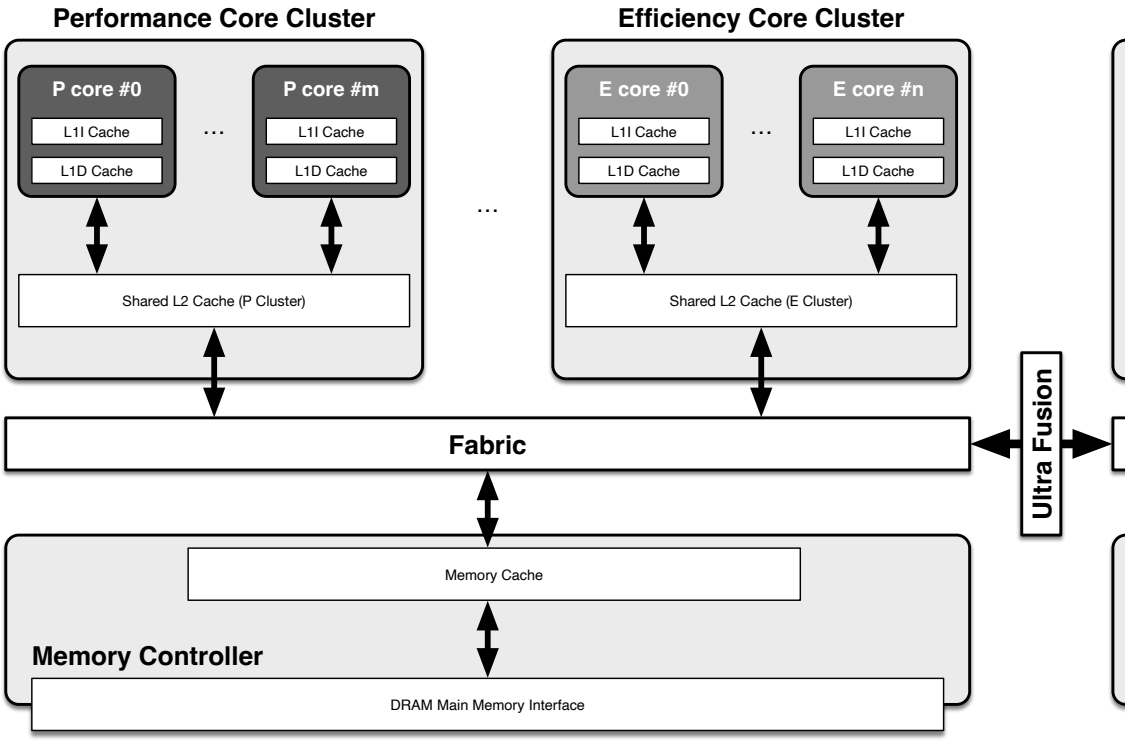


Table 4.1. Apple silicon Topology Configurations

Chip	Cores per P Cluster	Number of P Clusters	Cores per E Cluster	Number of E Clusters	Shorthand NumDie x (Cluster + ... + Cluster)
M1	4	1	4	1	1 x (4P + 4E)
M1 Pro	3 or 4	2	2	1	1 x (3P + 3P + 2E) 1 x (4P + 4P + 2E)
M1 Max	4	2	2	1	1 x (4P + 4P + 2E)
M1 Ultra	4	4	2	2	2 x (4P + 4P + 2E)
M2	4	1	4	1	1 x (4P + 4E)
M2 Pro	3 or 4	2	4	1	1 x (3P + 3P + 4E) 1 x (4P + 4P + 4E)
M2 Max	4	2	4	1	1 x (4P + 4P + 4E)
M2 Ultra	4	4	4	2	2 x (4P + 4P + 4E)
M3	4	1	4	1	1 x (4P + 4E)
M3 Pro	5 or 6	1	6	1	1 x (5P + 6E) 1 x (6P + 6E)
M3 Max	5 or 6	2	4	1	1 x (5P + 5P + 4E)

Table 4.1. Apple silicon Topology Configurations *(cont.)*

Chip	Cores per P Cluster	Number of P Clusters	Cores per E Cluster	Number of E Clusters	Shorthand NumDie x (Cluster + ... + Cluster)
					1 x (6P + 6P + 4E)
A14 Bionic	2	1	4	1	1 x (2P + 4E)
A15 Bionic	2	1	4	1	1 x (2P + 4E)
A16 Bionic	2	1	4	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.

4.2 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 efficiency 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 will 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 will be prefetched without issue, however applications may need to be tuned differently in the presence of more unusual patterns.

4.3 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-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 will look for independent work. The P cores are capable of looking ahead up to several hundred 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 will have completed execution and will have had their results reflected in register and memory state, while no younger results will 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 will 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.

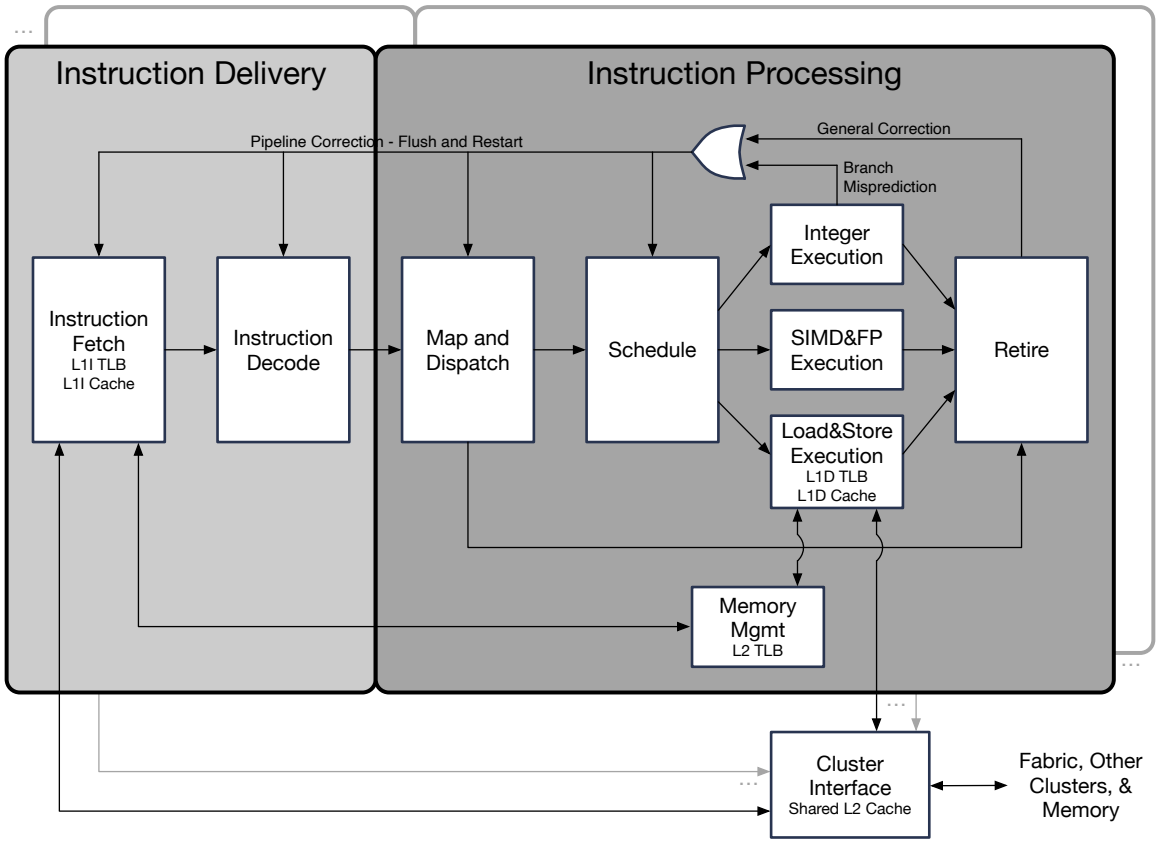
Similarly, the processor may speculate past memory dependences. That is, the processor may guess which, if any, older stores in the instruction window may write values to memory that need to be read by younger loads in the window. In cases where it guesses wrong and misses a dependence, the load will have 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 4.2. Abstract View of the 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 (µops). These µops 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" will be used when specifying a particular instruction at the source code level and "µop" when referring to specific hardware.

Instruction Delivery consists of

- **Instruction Fetch Unit (Fetch):** Reads instruction bytes from the memory system based on various prediction mechanisms. Contains the Instruction Translation Lookaside Buffer (L1I 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

- **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 (Int 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&FP Execution):** Performs the desired operation on data associated with the Advanced SIMD and FP (Vector) Registers.
- **Load and Store Execution Units (Load&Store Execution):** Reads 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

- **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 cachelines to and from the fabric. The fabric transports cachelines between other clusters, memory, and other devices and chip resources.

This guide uses the following additional microarchitectural definitions:

- **Dispatch:** Send µops from the Map and Dispatch Unit to the Schedule Unit.
- **Issue:** Send µops from the Schedule Unit to an Execution Unit.
- **Slot:** A parallel lane in which a µop travels, usually in in-order portions of the processor.
- **Demand:** A required access to a cache structure (including TLB) according to the predicted flow of instructions, as opposed to a prefetch access that is an educated guess of future need.
- **Unique Miss:** The initial miss in a caching structure (including TLB) that initiates a fill. Subsequent misses while the 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:** 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.

4.4 Instruction Delivery Optimization

The following sections describe optimization opportunities for identifying and improving Fetch and Decode Unit bottlenecks in Instruction Delivery.

4.4.1 L1 Instruction (L1I) TLB

The Instruction Translation Lookaside Buffer 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 4.6.2, "L1 Data (L1D) TLB and Shared L2 TLB".

Table 4.2. L1I 16KiB-Page TLB Organization

Chip	Entries (Coverage)	
	Performance Cores	**Efficiency Cores**
M1 Generation and A14 Bionic	192 entries (3MiB)	128 entries (2MiB)
M2 Generation and A15 Bionic		192 entries (3MiB)
M3 Generation and A16 Bionic		

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 µops being delivered to Map until the translation is available.

Table 4.3. Common L1 Instruction TLB Metrics

Name and Formula (Event Definitions: Section 6.2, "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 4.17: "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.

4.4.2 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 4.6.4, "Shared L2 Cache".

Table 4.4. L1I Cache Organization

Chip	Capacity, Associativity, and Line Size	
	Performance Cores	**Efficiency Cores**
M1 Generation and A14 Bionic	192KiB, 6-way, 64B lines	128KiB, 8-way, 64B lines
M2 Generation and A15 Bionic		128KiB, 4-way, 64B lines (reduced associativity)
M3 Generation and A16 Bionic		

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 will 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 cachelines that occupy space in the cache for common execution patterns. Collect uncommonly executed portions and place them together, typically at the end.

Table 4.5. Common L1 Instruction Cache Metrics

Name and Formula (Event Definitions: Section 6.2, "Performance Monitoring Events")	**Description**
L1I Cache Miss Density: <Ev L1I_CACHE_MISS_DEMAND> / <Ev INST_ALL>	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 Cachelines:

Separate portions of functions into commonly used (hot) and uncommonly used (cold). Place the commonly used portions of functions into a small set of cachelines 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 cachelines will only occupy space in the instruction cache in rare circumstances.

[Magnitude: Medium | Applicability: Medium]

Recommendation: Avoid L1I Cache Set Conflicts:

[Magnitude: Low | Applicability: Low] While 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.

4.4.3 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 cacheline. Thus, when the processor fetches a cacheline at the target of a branch, the Instruction Fetch Unit will provide 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 cacheline, 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 cacheline 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.

4.4.4 Taken Branch Reduction

Taken branches cause an interruption in the fetch sequence. As described in Section 4.4.3, “Branch Target Alignment”, the Instruction Fetch Unit reads groups of instructions even when they span a cacheline 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 4.4.5, “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 https://en.cppreference.com/w/cpp/language/attributes/likely for more information.
  - In C and older versions of C++, use `__builtin_expect()` or `__builtin_expect_with_probability()` to empower the compiler. See https://llvm.org/docs/BranchWeightMetadata.html 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 4.6. Common Branch Instruction Mix and Direction Metrics

Name and Formula (Event Definitions: Section 6.2, "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

Table 4.6. Common Branch Instruction Mix and Direction Metrics *(cont.)*

Name and Formula (Event Definitions: Section 6.2, "Performance Monitoring Events")	**Description**
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.

4.4.5 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:

1. **Conditional moves:** For example, `CSEL` that writes the destination with one of two input values based on the condition.
2. **Conditional sets:** For example, `CSET` that sets the destination to one of two fixed values based on the condition.
3. **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 will be 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 will be low, and the code will be executed 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 4.4.4, “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):
        FCMP    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
        FCMP    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 4.7. General and Conditional Branch Mispredict Metrics

Name and Formula (Event Definitions: Section 6.2, "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.

4.4.6 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 4.4.7, "Return Address Stack" for more details). Use the following metrics to identify indirect branch and indirect call misprediction rates:

Table 4.8. Indirect Branch and Indirect Call Mispredict Metrics

Name and Formula (Event Definitions: Section 6.2, "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
Mispredicted Indirect Call Density (of instructions): <Ev BRANCH_CALL_INDIR_MISPRED_NONSPEC> / <Ev INST_ALL>	Proportion retired mispredicted indirect function calls of all retired instructions

4.4.7 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 will 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 4.9. Return Mispredict Metrics

Name and Formula (Event Definitions: Section 6.2, "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.

4.4.8 Multi-µop Instructions

Some complex instructions require multiple µops 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 µops. The processor inserts the µop sequence seamlessly between µops from the prior and subsequent instructions.

Microcoded instructions require sequences of 4 or more µops. In addition, the processor may not be able to seamlessly integrate these µops into the µop stream and thus will 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-µop instructions include:

- **Load and store operations with address writeback:** These common cracked instructions require an extra µop to perform the address update. Avoid chains of these instructions because of the added µop bandwidth consumed as well as the added dependencies. Use one adjustment per block of loads and/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-sized, the processor will re-fuse them back into a single µop 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 ASIMD&FP store operations:** These common cracked instructions require an additional µop. Use them wherever possible to improve code density. Example instructions:

```
ST{N}P {SDQ}t1, {SDQ}t2, ...
```

- **ASIMD&FP load and store operations that use an `lsl #4` or `<extend> #4`:** These less common cracked instructions require an additional µop 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 ASIMD&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
```

- **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 µop, 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 (`bfloat16`) mathematical operations:** These instructions (*`BFDOT`*, *`BFMMLA`*) require a sequence of 3-8 general vector µops to complete.

Recommendation: Leverage Cracked and Microcoded Instructions When the Full Instruction Functionality is Needed:

[Magnitude: Low | Applicability: Medium] While cracked and microcoded instructions require multiple µops 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.

4.5 Instruction Processing Map and Execution Optimization

The following sections describe optimization opportunities for identifying and improving Map and Execution issues in Instruction Processing.

Sustained µop bandwidth (assuming a broad mix of µop types) is limited by Map bandwidth, in other words, the number of µops that can be inserted into the Instruction Window per cycle. Each Map slot is capable of handling any type of µop, and on average, execution will not exceed more than the sustained µop bandwidth.

Burst µop bandwidth is achieved when all of the functional units execute a µop at the same time. This is possible due to buffering of instructions in the instruction window. Note that store µop address and data parts are counted separately (See Section A.3.4, "Load and Store Execution Unit Bandwidth" for more details).

Table 4.10. µop Execution Bandwidth

Chip	µops Per Cycle			
	Performance Cores		Efficiency Cores	
	Sustained	Burst	Sustained	Burst
M1 Generation A14 Bionic	8	17: 6 Integer 4 ASIMD&FP 3 LD + 2 ST(A) + 2 ST(D)	4	10: 3 Integer 2 ASIMD&FP 2 LD/ST(A) + 2 ST(D)
M2 Generation A15 Bionic and A16 Bionic			5	14: 4 Integer 2 ASIMD&FP 2 LD/ST(A) + 2 ST(D)
M3 Generation	9	19: 8 Integer 4 ASIMD&FP 3 LD + 2 ST(A) + 2 ST(D)		

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

will have an IPC near the sustained µop 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 4.11. Instruction Throughput Metric

Name and Formula (Event Definitions: Section 6.2, "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 & Integer Unit*).

4.5.1 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
```

While `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 ASIMD&FP Execution Unit, if it avoids excess movement between the Integer and ASIMD&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 will often be faster to insert an integer-typed SIMD instruction into the middle of the second sequence than an integer unit instruction into the first. This integer-typed 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 ASIMD&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-typed 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.

4.5.2 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-sized 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 or 6 or 7 cycles.

4.5.3 Preferred Instructions for Common Operations

Use the following preferred instructions for common operations.

4.5.3.1 Register Data Copy

Use the following preferred instructions for creating copies of values in registers:

Table 4.12. Register Data Copy Instructions

Register Type	Register Data Copy Instructions

Integer	`MOV Xd, Xn`
	`ORR Xd, XZR, Xn  // OR a 0 with Xn into Xn`
	`ADD Xd, Xn, #0  // Add a 0 to Xn into Xd`

ASIMD&FP	`FMOV  Dd, Dn`
	`MOV.2D  Qd, Qn`
	`ORR.16B Vd, Vn, Vm when (Vm==Vn)  // 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.

4.5.3.2 Register Zeros

Use the following preferred instructions for zeroing registers and breaking dependence chains:

Table 4.13. Register Data Zero Instructions

Register Type	Register Data Zero Instructions
Integer	`MOVZ Xd|Wd, #0 ; Equivalent to MOV Xd|Wd, #0`
ASIMD&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 will not execute until the source register has been written by the producer instruction, and that will in turn delay the consume of the Exclusive-OR. 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.

4.5.3.3 Register Constants

Use the following preferred instructions for loading constants into registers:

Table 4.14. 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.

4.5.3.4 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.

4.5.4 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 will 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.

4.6 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.

4.6.1 Address Generation

Once the Scheduler issues a memory µop, 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 ASIMD&FP loads and stores when using `lsl #imm` or `<extend> #imm` forms when the immediate is 4 or greater. As noted in Section 4.4.8, "Multi-µop Instructions", these require an extra µop to calculate the address.

Other addressing forms may require additional µops to update base registers or update multiple elements.

4.6.2 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 4.15. L1D 16KiB-Page TLB Organization

Chip	Entries (Coverage)	
	Performance Cores	**Efficiency Cores**
M1 Generation	160 entries (2.5MiB)	128 entries (2MiB)
A14 Bionic	256 entries (4MiB)	
M2 Generation and A15 Bionic	256 entries (4MiB)	192 entries (3MiB)
M3 Generation	256 entries (4MiB)	

Table 4.15. L1D 16KiB-Page TLB Organization *(cont.)*

Chip	Entries (Coverage)	
	Performance Cores	**Efficiency Cores**
A16 Bionic	160 entries (2.5MiB) (reduction)	

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 4.16. Shared L2 16KiB-Page TLB Organization

Chip	Entries (Coverage)	
	Performance Cores	**Efficiency Cores**
M1 Generation and A14 Bionic	3072 entries (48MiB)	1024 entries (16MiB)
M2 Generation and A15 Bionic		2048 entries (32MiB)
M3 Generation and A16 Bionic		

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 will send 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 4.17. Common L1D TLB, Shared L2 TLB, and Address Translation Metrics

Name and Formula (Event Definitions: Section 6.2, "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 L1I TLB misses): <Ev L2_TLB_MISS_INSTRUCTION> / <Ev L1I_TLB_FILL>	Proportion speculative L2 TLB instruction misses of unique L1I TLB misses (counted by fills into the L1I TLB)

Table 4.17. Common L1D TLB, Shared L2 TLB, and Address Translation Metrics *(cont.)*

Name and Formula (Event Definitions: Section 6.2, "Performance Monitoring Events")	Description
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.

4.6.3 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.

While the system operates on 128B cachelines, the L1D Cache operates on 64B cachelines. See Section 4.6.6, "Improving Cache Hierarchy Performance" for recommendations on improving data cache hierarchy performance.

Table 4.18. L1D Cache Organization

Chip	Capacity, Associativity, and Line Size	
	Performance Cores	**Efficiency Cores**
M1 Generation and A14 Bionic	128KiB, 8-way, 64B lines	64KiB, 8-way, 64B lines
M2 Generation and A15 Bionic		
M3 Generation and A16 Bionic		

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 4.19. Common Data Cache Metrics

Name and Formula (Event Definitions: Section 6.2, "Performance Monitoring Events")	**Description**
Load L1D Cache Miss Retired Density (of instructions): <Ev L1D_CACHE_MISS_LD_NONSPEC> / <Ev INST_ALL>	Proportion retired L1D Cache loads that miss the cache of all instructions.
Store L1D Cache Miss Retired Density (of instructions): <Ev L1D_CACHE_MISS_ST_NONSPEC> / <Ev INST_ALL>	Proportion retired L1D Cache stores that miss the cache of all instructions.
Load L1D Cache Miss Retired Density (of retired loads): <Ev L1D_CACHE_MISS_LD_NONSPEC> / (<Ev INST_INT_LD> + <Ev INST_SIMD_LD>)	Proportion retired L1D Cache loads that miss the cache.
Store L1D Cache Miss Retired Density (of retired stores): <Ev L1D_CACHE_MISS_ST_NONSPEC> / (<Ev INST_INT_ST> + <Ev INST_SIMD_ST>)	Proportion retired L1D Cache stores that miss the cache.
Load L1D Cache Miss Density (of load pipeline accesses): <Ev L1D_CACHE_MISS_LD> / <Ev LD_UNIT_UOP>	Proportion speculative load pipeline accesses that miss the L1D Cache. Accesses may count more than once if the load is replayed within the Load and Store Execution Unit or is split across a cacheline, and includes prefetches and other operations that may use the pipeline.
Store L1D Cache Miss Density (of store pipeline accesses): <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 cacheline, and includes prefetches and other operations that may use the pipeline.
L1D Dirty Writeback Density (of load and store pipeline accesses): <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.

4.6.4 Shared L2 Cache

Each cluster of cores uses a Shared L2 Cache which operates on 128B cachelines. 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.

Table 4.20. Shared L2 Cache Organization

Chip	Capacity, Associativity, and Line Size	
	Performance Cluster	**Efficiency Cluster**
M1 Generation	12MiB, 12-way, 128B lines	4MiB, 16-way, 128B lines
M2 Generation	16MiB, 16-way, 128B lines	
M3 Generation		
A14 Bionic	8MiB, 16-way, 128B lines	
A15 Bionic	12MiB, 12-way, 128B lines	
A16 Bionic	16MiB, 16-way, 128B lines	

Note: Capacity and associativity 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.

Shared L2 Misses may be serviced by either the memory system, including the Memory Cache (See Section 4.6.5, "Memory Cache"), or another cluster's caching hierarchy depending on data ownership. Load µops 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.

4.6.5 Memory Cache

Attached to the DRAM interface is a Memory Cache (M Cache) operating on 128B cachelines. 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 µops that hit in the M Cache execute with a latency in the range of 35ns, while load µops that access DRAM will execute with latency in the range of 95ns. These latencies may increase if the system is experiencing heavy traffic.

Table 4.21. Memory Cache Organization

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 (reduced)
M3 Max	48MiB, 12-way, 128B lines
A14 Bionic	16MiB, 16-way, 128B lines
A15 Bionic	32MiB, 16-way, 128B lines

Table 4.21. Memory Cache Organization *(cont.)*

Chip	Capacity, Associativity, and Line Size
A16 Bionic	24MiB, 12-way, 128B lines (reduced)

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 will 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.

4.6.6 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 cacheline and are modified by threads running on different core clusters. When a core in one cluster writes to one of the independent variables, it will cache 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 vararіable, request the cache line, and the caching subsystem will move 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 will then move 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 cachelines. 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 cacheline 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 cacheline to match the L1D Cache line size. When one datum is requested, the related data will naturally arrive with it in the same cacheline, avoiding further cache misses. Data packed into the same 128B cacheline will 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 will 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. 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 1 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 3.11: "Advanced SIMD Intrinsic Data Types" and Writing Arm64 Core for Apple Platforms.

Recommendation: Avoid False Sharing by Allocating Independent Shared Variables to Different 128B Cachelines:

[Magnitude: High | Applicability: Low] Avoid false sharing bottlenecks by allocating each independent shared variable to a different 128B cacheline. This avoids situations where a cached shared variable, possibly related a software semaphore, is not inadvertently invalided 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 Cachelines for Improved Cache Hierarchy Performance, Concentrating Data Commonly Used Together into the same 64B Cachelines.:

[Magnitude: High | Applicability: Medium] Identify and place the commonly used variables into cachelines 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.

4.6.7 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 4.6.2, “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 will not break this optimization.

4.6.8 Non-Temporal (NT) Accesses

Caches naturally store data close to the processor under the assumption that the data in the cache line will be used again in the near term. Data that benefits from caching is said to have temporal locality. However, some algorithms stream (read and/or write) 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 will optimize 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 will retain the cacheline for a short time to allow accesses to additional data elements in the same cacheline. Do not use non-temporal loads if the algorithm

is likely to frequently use the data, nor if the algorithm will subsequently modify the data. And do not mix non-temporal accesses with regular accesses during the window of use. The processor will ensure 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 will ensure correct memory ordering in these cases, but the algorithm may suffer from poor performance.

The Clang compiler provides a builtin that will allow 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 doesn't guarantee that it will insert 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 4.22. Common Non-Temporal Access Metrics

Name and Formula (Event Definitions: Section 6.2, "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

4.6.9 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 will 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 will execute 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 will be 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 will cross a 16B boundary and every fourth load will cross a 64B boundary. Both can also suffer from page-spanning penalties. Such operations will see 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. While 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 will 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 will eventually make unaligned access penalties become apparent. Accesses to large buffers will inevitably lead to a fair number of loads and stores that span cache line and/or page boundaries, which may result in reduced performance.

Table 4.23. Common Access Alignment Metrics

Name and Formula (Event Definitions: Section 6.2, "Performance Monitoring Events")	**Description**
L1D 64B Cacheline Crossing Access Density (of load and store pipeline accesses): <Ev LDST_X64_UOP> / (<Ev LD_UNIT_UOP> + <Ev ST_UNIT_UOP>)	Proportion of speculative load and store pipeline accesses that cross a 64B cacheline boundary.
L1D Page Crossing Access Density (of load and store pipeline accesses): <Ev LDST_XPG_UOP> / (<Ev LD_UNIT_UOP> + <Ev ST_UNIT_UOP>)	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 will prevent 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.

4.6.10 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 µops. With load and store µops, 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 4.24: "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 µops 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 4.24. Store-to-Load Dependence Prediction Outcome

		Predicted Outcome	
		Not Dependent	**Dependent**
Actual Outcome	Load Not dependent on Store (Non-Overlapping Memory Range)	High Performance: Loads and stores may execute at any time without conflict	Poor Performance: While 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)	Worst Performance: Memory Order Violation: If the older store's address is unknown when the younger load is ready to execute, the younger load may prematurely execute and require a pipeline flush with restart. OR Modest Performance: If the older store's address is known, the processor may stall the load until the store is known.**	Modest Performance: Loads are delayed until just after stores complete, properly honoring dependencies but not stalling excessively. **

**Note that data forwarding from store to load is often slower than through registers, resulting in modest 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 will come 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 will feed younger loads because of the small data set. The predictor will be 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 will occur after the pointer collision, which will then be 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.

While 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. While 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 4.6.10.1, "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 will often just replace 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 will 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 will *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 will work to 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 will not always be 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 will often increase 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 will find 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-sized data types, especially if those accesses may occur within a several hundred instructions of each other.

Table 4.25. Common Memory Dependence Metrics

Name and Formula (Event Definitions: Section 6.2, "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.

4.6.10.1 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 will need the value from the previous store. But, in some iterations when `d < 0`, `coeff` index will be incremented such that the load will need the value from memory as opposed to the previous store.

```
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;
    }
}
```

4.6.11 Store-to-Load Forwarding

When the processor predicts that a load will read data written all or in part by an older store, the load will wait 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.

While data alignment for store-to-load-forwarding isn't often a limiter, cache line and page spanning loads may experience delays. (See Section 4.6.9, “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 4.6.10, “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.

4.6.12 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.

4.6.12.1 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 performance 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 efficiency cores usually require less aggressive prefetching. When the same code is executed on the efficiency 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.

4.6.12.2 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 cacheline, striding the address based on the particular cache's line size. The address specified within a particular cacheline does not matter.

Table 4.26. Recommended Software Prefetch Instructions

Instruction	Operation	Destination	Temporal Behavior
`PRF(U)M PLDL1KEEP`	Load / Read	L1D Cache	Temporal
`PRF(U)M PLDL1STRM`			Non-Temporal
`PRF(U)M PLDL2KEEP`		Shared L2 Cache	Temporal
`PRF(U)M PLDL2STRM`			Non-Temporal
`PRF(U)M PSTL1KEEP`	Store / Write	L1D Cache	Temporal
`PRF(U)M PSTL1STRM`			Non-Temporal
`PRF(U)M PSTL2KEEP`		Shared L2 Cache	Temporal
`PRF(U)M PSTL2STRM`			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 4.19: "Common Data Cache Metrics". Identify loads and/or 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 will often read the array using strides that greatly exceed the maximum stride that the hardware prefetcher can detect.

4.6.12.3 Prefetch Performance Tips

While 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*, ... , for rows of length `M`, one can instead rearrange the references into streams like `X`, `X+1`, `X+2`, ... , `X+M-1`, *X+M*, *X+M+1*, *X+M+2*, ... 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 will 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 immediately be 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 get far enough ahead of them to avoid pipeline stalls, but software prefetch instructions only initiate prefetches at the time that they are executed. Because they do not attempt to “look ahead” in any way, code that uses software prefetches must always manually insert them far enough in advance so that any needed cache misses can complete before the eventual demand loads occur. In

this case, one must 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.

While 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 like this

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.

Few algorithms access multiple rows of 2D buffers at once in ways that will cause cache misses on multiple rows at once like this, but for those that do these techniques can result in very large performance improvements.

Last, 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 cachelines 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 cachelines that are needed for typical access to a node. Specifically, put the key node data items and "next" pointer into a single cacheline. See the discussion on "hot/cold" fields and variables in Section 4.6.3, “L1 Data (L1D) Cache”.

Recommendation: Recode Algorithms to Simplify Access Patterns to Aid the Hardware Prefetcher:

[Magnitude: Medium | Applicability: Medium] While 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 5. 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 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*.

5.1 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.

5.2 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-sized 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. While 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. While 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 5.2.1, "Avoid Static Partitioning" describes some of the pitfalls associated with static partitioning, while Section 5.2.2, "Use Dynamic Partitioning" describes the benefits of dynamic partitioning.

5.2.1 Avoid Static Partitioning

Static partitioning can lead to longer latency execution on asymmetric MP systems. While these equal-sized chunks may all complete in about the same amount of time on a symmetric MP system, asymmetric cores will 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 will make more progress than those running primarily on E cores. While the system will likely move 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 will often rotate 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 will delay 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*.

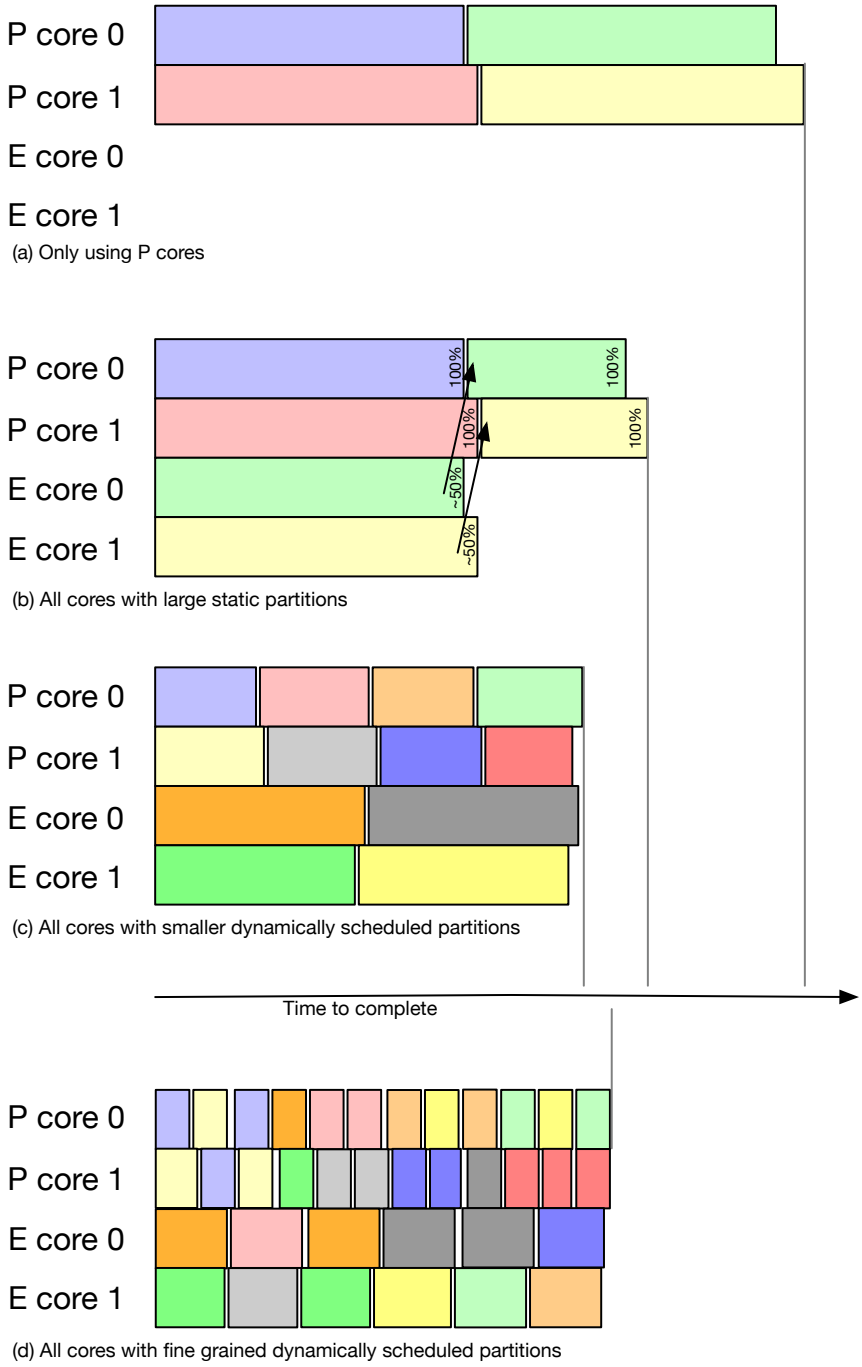
5.2.2 Use Dynamic Partitioning

Dynamic partitioning is a better approach that static partitioning in most cases. Threads that happen to be assigned to P cores will naturally complete work more quickly, taking a larger number of parallel tasks from the task queue, while threads assigned to E cores will churn through a smaller number of tasks more slowly. As threads move around the cores, they will 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. Last, single-threaded CPU performance tends to increase faster than multi-threaded work-queue management performance from generation to generation, so work items that are the proper size in one generation may be too small in a future generation. 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 5.1: “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 5.1. Partitioning Strategies: An Illustrative Example



As illustrated in Figure 5.1: “Partitioning Strategies: An Illustrative Example”, a dynamic work partitioning strategy will most often lead to the best performance because of its flexible work assignment. Threads synchronized using static techniques like barriers will 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.

5.3 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.

5.4 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 generations and even between different products within a particular generation. 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 generation and product 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 pthreads 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 5.1. Common Atomic and Exclusive Metrics

Name and Formula (Event Definitions: Section 6.2, "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
Atomic/Exclusive Fail Density: <Ev ATOMIC_OR_EXCLUSIVE_FAIL> / <Ev INST_ALL>	Proportion atomic and exclusive instructions that fail of all retired instructions

5.5 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 will try to keep the threads together on the same cluster. Thus, the threads will 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 will almost inevitably encounter these longer communication latencies. The system will keep 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 will snoop the cache line away 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 5.4, "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 will be required to ensure shards improve overall performance while limiting worst-case slowdowns. Use sharding, in particular core-based sharding, cautiously and sparingly.

Last, avoid false sharing, where two or more independent variables occupy the same 128B cacheline. This organization may lead to thrashing of the cacheline between cores. See Section 4.6.6, "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 cachelines without introducing false sharing.

5.6 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 6. Performance Monitoring

6.1 Xcode Instruments Tool

Xcode features an Integrated Development Environment (IDE) tool called **Instruments**. This tool can help you profile your apps in order to better understand and optimize their behavior and performance.

For more information on how to use Instruments, see Getting Started with Instruments (WWDC2019 Video).

6.2 Performance Monitoring Events

Event names are denoted <EV *event_name*> where the *event_name* follow this convention:

- **INST_*:** ISA instruction type profiling event. These events count when a matching instruction retires. By definition, these are non-speculative.
- ***_NONSPEC:** Non-speculative 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.

Event Name	Brief Description
ATOMIC_OR_EXCLUSIVE_FAIL	Atomic or exclusive instruction failed (due to contention)
ATOMIC_OR_EXCLUSIVE_SUCC	Atomic or exclusive instruction successfully completed
BRANCH_CALL_INDIR_MISPRED_NONSPEC	Retired indirect call instructions mispredicted
BRANCH_COND_MISPRED_NONSPEC	Retired conditional branch instructions that mispredicted
BRANCH_INDIR_MISPRED_NONSPEC	Retired indirect branch instructions including calls and returns that mispredicted
BRANCH_MISPRED_NONSPEC	Retired branch instructions including calls and returns that mispredicted
BRANCH_RET_INDIR_MISPRED_NONSPEC	Retired return instructions that mispredicted
CORE_ACTIVE_CYCLE	Cycles while the core was active
FETCH_RESTART	Fetch Unit internal restarts for any reason. Does not include branch mispredicts
FLUSH_RESTART_OTHER_NONSPEC	Pipeline flush and restarts that were not due to branch mispredictions or memory order violations
INST_ALL	All retired instructions
INST_BARRIER	Retired barrier instructions
INST_BRANCH	Retired branch instructions including calls and returns

Event Name	Brief Description
INST_BRANCH_CALL	Retired subroutine call instructions
INST_BRANCH_INDIR	Retired indirect branch instructions including indirect calls
INST_BRANCH_RET	Retired subroutine return instructions
INST_BRANCH_TAKEN	Retired taken branch instructions
INST_INT_ALU	Retired non-branch and non-load/store Integer Unit instructions
INST_INT_LD	Retired load Integer Unit instructions
INST_INT_ST	Retired store Integer Unit instructions
INST_LDST	Retired load and store instructions
INST_SIMD_ALU	Retired non-load/store Advanced SIMD and FP Unit instructions
INST_SIMD_ALU_VECTOR	Retired non-load/store Advanced SIMD instructions (including integer and floating point data types) Note: Available on M2 Generation and following, and A15 Bionic and following
INST_SIMD_LD	Retired load Advanced SIMD and FP Unit instructions
INST_SIMD_ST	Retired store Advanced SIMD and FP Unit instructions
INTERRUPT_PENDING	Cycles while an interrupt was pending because it was masked
L1D_CACHE_MISS_LD	Loads that missed the L1 Data Cache
L1D_CACHE_MISS_LD_NONSPEC	Retired loads that missed in the L1 Data Cache
L1D_CACHE_MISS_ST	Stores that missed the L1 Data Cache
L1D_CACHE_MISS_ST_NONSPEC	Retired stores that missed in the L1 Data Cache
L1D_CACHE_WRITEBACK	Dirty cache lines written back from the L1D Cache toward the Shared L2 Cache
L1D_TLB_ACCESS	Load and store accesses to the L1 Data TLB
L1D_TLB_FILL	Translations filled into the L1 Data TLB
L1D_TLB_MISS	Load and store accesses that missed the L1 Data TLB
L1D_TLB_MISS_NONSPEC	Retired loads and stores that missed in the L1 Data TLB
L1I_CACHE_MISS_DEMAND	Demand instruction fetches that missed in the L1 Instruction Cache
L1I_TLB_FILL	Translations filled into the L1 Instruction TLB
L1I_TLB_MISS_DEMAND	Demand instruction fetches that missed in the L1 Instruction TLB
L2_TLB_MISS_DATA	Loads and stores that missed in the L2 TLB
L2_TLB_MISS_INSTRUCTION	Instruction fetches that missed in the L2 TLB
LD_UNIT_UOP	Uops that flowed through the Load Unit
LD_NT_UOP	Load uops that executed with non-temporal hint
LDST_X64_UOP	Load and store uops that crossed a 64B boundary
LDST_XPG_UOP	Load and store uops that crossed a 16KiB page boundary
MAP_DISPATCH_BUBBLE	Cycles while the Map Unit was not stalled and Decode Unit did not send any uops
MAP_INT_UOP	Mapped Integer Unit uops

Event Name	Brief Description
MAP_LDST_UOP	Mapped Load and Store Unit uops, including GPR to vector register converts
MAP_REWIND	Cycles while rewinding the Map Unit due to flush and restart
MAP_SIMD_UOP	Mapped Advanced SIMD and FP Unit uops
MAP_STALL	Cycles while the Map Unit was stalled for any reason
MAP_STALL_DISPATCH	Cycles while the Map Unit was stalled because of Dispatch back pressure
MMU_TABLE_WALK_DATA	Table walk memory requests on behalf of data accesses
MMU_TABLE_WALK_INSTRUCTION	Table walk memory requests on behalf of instruction fetches
RETIRE_UOP	All retired uops
SCHEDULE_UOP	Uops issued by the scheduler to any execution unit
ST_MEMORY_ORDER_VIOLATION_NONSPEC	Retired stores that triggered memory order violations
ST_NT_UOP	Store uops that executed with non-temporal hint
ST_UNIT_UOP	Uops that flowed through the Store Unit

Appendix A. Instruction Latency and Bandwidth

The details covered in Chapter 4, *Core 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 from generation to generation, 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, Integer, ASIMD&FP, and Load&Store. Instructions often map directly to single µops, thus instruction mnemonics mentioned serve as proxies for the relevant µops.

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 will 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 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 ASIMD&FP (vector) registers. See Section 4.5.1, “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.
		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`, `SBCS`, `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.
MUL	3	Multiplies without additional add/sub operation (`MUL`, `MNEG`, `UMNEGL`, `SMNEGL`, `SMULL`, `SMULH`, `UMNEGL`, `UMULH`, `UMULL`)
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*`)

Table A.2. Integer Execution Class Availability: P Core

Chip	Integer Execution Unit							
	0	1	2	3	4	5	6	7
M1 Generation M2 Generation A14 Bionic A15 Bionic A16 Bionic	ALU/f BRc/i	ALU/f BRc	ALU/f	ALU MUL MAC MISC	ALU MUL DIV	ALU		
M3 Generation	ALU/f BRc/i	ALU/f BRc	ALU/f	ALU/f	ALU MUL MAC MISC	ALU MUL DIV	ALU	ALU

Table A.3. Integer Execution Class Availability: E Core

Chip	Integer Execution Unit			
	0	1	2	3
M1 Generation A14 Bionic	ALU/f MUL MAC MISC	ALU/f BRi DIV	ALU/f BRc	
M2 Generation M3 Generation A15 Bionic A16 Bionic	ALU/f MUL MAC MISC	ALU/f BRi DIV	ALU/f BRc	ALU/f

The following instructions serialize the instruction delivery pipeline according to their conditions: `ISB`, `SB`.

A.2 Advanced SIMD and FP Execution Units

ASIMD&FP operations are divided into classes, as described in Table A.4: “ASIMD&FP Execution Classes”. The ASIMD&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 ASIMD&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 ASIMD&FP (vector) registers. See Section 4.5.1, “Movement of Data from General Purpose Registers to Vector Registers”.

Table A.4. ASIMD&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 ASIMD&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
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` Note that SHA3-related logic instructions (`BCAX`, `EOR3`, `RAX1`, `XAR`) are included in the GENERAL class

Table A.5. GENERAL ASIMD&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 4.4.8, "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)`,

Table A.5. GENERAL ASIMD&FP Class Details *(cont.)*

Type	Latency	Operation Types and Descriptions (Example Instruction Mnemonics)
		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 (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)
		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, SQRDMLAH, SQRDMLSH, UDOT, UMLAL, UMLAL2, UMLSL, UMLSL2, UMMLA, UMULL, UMULL2, USDOT, USMMLA)
		Horizontals (ADDV, SADDLV, SMAXV, SMINV, UADDLV, UMAXV, UMINV)
		Simple + saturating (SQADD, SQABS, SQNEG, SQSUB, SUQADD, USQADD, UQADD, UQSUB)
		Simple + rounding (SQRSHL, 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, SQRSHRN, SQRSHRN2, SQRSHRUN, SQRSHRUN2, 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)
	4	Multiplies and multiply+adds/accumulates (BFMLALB, BFMLALT, FCMLA, FMADD, FMLA, FMLAL, FMLAL2, FMLSL, FMLSL2, FMLS, FMSUB, FMUL, FMULX, FNMADD, FNMSUB, FNMUL, FRECPS, FRSQRTS)
	P: 10 E: 12	Brain floating point (bfloat16) dot product (BFDOT) Note this is a cracked/microcoded flow. See Section 4.4.8, "Multi-µop Instructions".
	P: 13 E: 16	Brain floating point (bfloat16) matrix multiply-accumulate (BFMMLA)

Table A.5. GENERAL ASIMD&FP Class Details *(cont.)*

Type	Latency	Operation Types and Descriptions (Example Instruction Mnemonics)
		Note this is a cracked/microcoded flow. See Section 4.4.8, "Multi-µop Instructions".
CVT	3[2] [+4/5] [+4/3]	Base convert operation between integer and floating point types (`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 4.5.1, "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. ASIMD&FP Execution Class Availability: P Core

Chip	ASIMD&FP Execution Unit			
	0	1	2	3
M1 Generation M2 Generation A14 Bionic A15 Bionic A16 Bionic	GENERAL MOVE2GPR FCMPf FCSELf FDIV SHA	GENERAL MOVE2GPR FCSELf	GENERAL	GENERAL
M3 Generation	GENERAL MOVE2GPR FCMPf FCSELf FDIV SHA	GENERAL MOVE2GPR FCMPf FCSELf	GENERAL	GENERAL

Table A.7. ASIMD&FP Execution Class Availability: E Core

Chip	ASIMD&FP Execution Unit		
	0	1	2
M1 Generation M2 Generation M3/M3 Pro A14 Bionic A15 Bionic	GENERAL MOVE2GPR FCMPf FCSELf FDIV	GENERAL FCSELf	

Table A.7. ASIMD&FP Execution Class Availability: E Core

Chip	ASIMD&FP Execution Unit		
	0	1	2
A16 Bionic	SHA		
M3 Max	GENERAL MOVE2GPR FCMPf FCSELf FDIV SHA	GENERAL FCMPf FCSELf	GENERAL

A.3 Load and Store Execution Units

The cores feature several load and store execution units capable of simultaneous issue. Store µops 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 µops that hit in the data cache normally execute with a 4-cycle latency.

Some integer-load instructions require an additional arithmetic µop to complete in order to update addresses for pre- and post-indexing. See Section 2.8, "Addressing Forms, Instruction Immediates, and Operand Shifts", Section 4.6.1, "Address Generation", and Section 4.4.8, "Multi-µop 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 µops to complete, and may include combinations of:

- arithmetic µops to compute addresses and/or update addresses for pre- and post-indexing
- load µops for multiple elements
- shuffle µops to fill specific elements

See Section 2.8, "Addressing Forms, Instruction Immediates, and Operand Shifts", Section 4.6.1, "Address Generation", and Section 4.4.8, "Multi-µop 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 µops will have at least 1 µop 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)	Instruction Mnemonics with Additional Details
Load and Load Pair (all sizes)	4	`LDR`, `LDUR`, `LDP`, `LDNP` (see note on `lsl/extend #4` in Section 4.4.8, "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`

Table A.10. Integer Store Latencies *(cont.)*

Store Type	Latency of Destination Data Operand(s)	Example Instruction Mnemonics
Store Pair (all sizes)	N/A	`STP`, `STNP`,
Store Pair Exclusive (all sizes)	4	`STXP`, `STLXP`

While 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 µops, the load and store µops are guaranteed to execute atomically by the core. That is, once ordering requirements have been met and the core possesses the cacheline, the load and store will 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.

While 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 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,

Table A.11. Synchronization Instruction Latencies *(cont.)*

Operation Type (Each have optional acquire and release semantics)	Latency to Destination Register	Example Instruction Mnemonics
		`STUMAXAL`, `STUMAXALB`, `STUMAXALH`, `STUMINAL`, `STUMINALB`, `STUMINALH`

Recommendation: Use Single Instruction Atomics to Improve Multiprocessor Performance:

[Magnitude: Low | Applicability: Low] Single instruction atomics execute the load and store as an atomic unit, that is, the core will not lose the cacheline in between the load and store.

A.3.4 Load and Store Execution Unit Bandwidth

Loads and stores execute on multiple execution ports. As noted, store µops 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 4.5.1, “Movement of Data from General Purpose Registers to Vector Registers”.

Table A.12. Memory Execution Unit Bandwidth: P Core

Chip	Bandwidth			
	Load or MOVE2VEC µop	Store µop (address part)	Store µop (data part)	Combined limits per cycle
M1 Generation and A14 Bionic	3	2	2	Burst: 3 load µops, 2 store µops (address part), and 2 store µops (data part) Sustained: 4 µops Sustained: 2 writes into the cache
M2 Generation and A15 Bionic				
M3 Generation and A16 Bionic				

Table A.13. Memory Execution Unit Bandwidth: E Core

Chip	Bandwidth			
	Load or MOVE2VEC µop	Store µop (address part)	Store µop (data part)	Combined limits per cycle
M1 Generation and A14 Bionic	2	2	2	Burst: 2 load µops, or 2 store µops (address part), or 1 of each, along with 2 store µops (data part)
M2 Generation and A15 Bionic				
M3 Generation and A16 Bionic				

Table A.13. Memory Execution Unit Bandwidth: E Core *(cont.)*

Chip	Bandwidth			
	Load or MOVE2VEC µop	Store µop (address part)	Store µop (data part)	Combined limits per cycle
				Sustained: 2 µops Sustained: 1 write into the cache

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 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, “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

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 cacheline 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` command 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>`. `sysctlbyname()` will generate 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".

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 instruction set. 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 hw.optional.<parameter>	**Description**
`arm64`	Arm A-Profile 64b AARCH64 v8

Apple silicon chips support several Arm features without official Arm "`FEAT_*`" names. While 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 Unofficially Named EL0 Feature Parameters

Feature	
Parameter name hw.optional.<parameter>	**Description**
`armv8_crc32`	CRC32 instructions
`floatingpoint`	General support for floating-point instructions
`AdvSIMD`	General support for Advanced SIMD instructions
`AdvSIMD_HPFPCvt`	Advanced SIMD half-precision conversion instructions

Support for Arm features with standard names can be checked using `hw.optional.arm.<parameter>`, such as Dot Product: `hw.optional.arm.FEAT_DotProd`. Available features will have a parameter value of 1.

Unavailable features will have a parameter value of 0. Unimplemented features will 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 will 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
`hw.optional.neon` (The term "NEON" is no longer broadly used by Arm. See Chapter 3, *ISA Optimization: Advanced SIMD and FP Unit*.)	`hw.optional.AdvSIMD`
`hw.optional.neon_hpfp`	`hw.optional.AdvSIMD_HPFPCvt`
`hw.optional.neon_fp16`	`hw.optional.arm.FEAT_FP16`
`hw.optional.armv8_1_atomics`	`hw.optional.arm.FEAT_LSE`
`hw.optional.armv8_2_fhm`	`hw.optional.arm.FEAT_FHM`
`hw.optional.armv8_2_sha512`	`hw.optional.arm.FEAT_SHA512`
`hw.optional.armv8_2_sha3`	`hw.optional.arm.FEAT_SHA3`
`hw.optional.armv8_3_compnum`	`hw.optional.arm.FEAT_FCMA`

These `sysctl` feature parameters are also documented at https://developer.apple.com/documentation/kernel/1387446-sysctlbyname/determining_instruction_set_characteristics.

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	
		perflevel0 P core	perflevel1 E core
`hw.nperflevels`	Number of types of general purpose cores in the chip. The	2	

Table B.4. Per Performance Level Sysctl Parameters

New Per Performance Level Parameters	Definition	M1 Ultra Values	
		perflevel0 P core	perflevel1 E core
	lower the perflevel, the higher the performance of the core type		
`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}.l1dcachesize`	For cores of perflevel N, the size in bytes of the L1 Data Cache	131072	65536
`hw.perflevel{N}.l1icachesize`	For cores of perflevel N, the size in bytes of the L1 Instruction Cache	196608	131072
`hw.perflevel{N}.l2cachesize`	For cores of perflevel N, the size in bytes of the L2 Cache	12582912	4194304
`hw.perflevel{N}.l3cachesize`	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 *lowest* performing cores in the system)	M1 Ultra Values
`hw.l1dcachesize`	Size in bytes of the L1 Data Cache	65536
`hw.l1icachesize`	Size in bytes of the L1 Instruction Cache	131072
`hw.l2cachesize`	Size in bytes of the L2 Cache	4194304
`hw.l3cachesize`	Size in bytes of the L3 Cache	N/A

Table B.5. Updated Existing Sysctl Parameters *(cont.)*

Existing Parameters	Updated Definition (where applicable, applies to the *lowest* performing cores in the system)	M1 Ultra Values
`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
`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` cache and topology parameters are also documented at https://developer.apple.com/documentation/kernel/1387446-sysctlbyname/determining_system_capabilities/.