Intel® 64 and IA-32 Architectures Optimization Reference Manual

Documentation Changes

Document Number: 355308-003US

January 2024

Contents

Revision History

Date	Revision	Description
August 2023	048	Initial release of document.
January 2024	049	Q1 Release
April 2024	050	Q2 Release

Preface

This document is an update to the optimization recommendations contained in the **Intel® 64 and IA-32 Architectures Optimization Reference Manual,** also known as:

- Software Optimization Manual (SOM)
- Optimization Manual
- Optimization Reference Manual (ORM)

This document is a compilation of device and documentation errata, specification clarifications and changes. It is intended for hardware system manufacturers and software developers of applications, operating systems, or tools.

Nomenclature

Documentation Changes include gross content changes or omissions from the current published specifications . These changes are those beyond typos, capitalization, or basic edits. These will be incorporated in any new release of the specification upon approval.

Summary Tables of Changes

The following table indicates documentation changes which apply to the Intel® 64 and IA-32 Architecture software optimization topics covered by this reference manual.

No.	DOCUMENTATION CHANGES
1	Updates to Chapter 1
2	Updates to Chapter 2
3	Updates to Chapter 4
4	Updates to Chapter 17
5	Chapter 16 is now Chapter 8 in Volume 2.

Documentation Changes

Changes to the **Intel® 64 and IA-32 Architectures Optimization Reference Manual** volumes follow, and are listed by chapter. Only chapters with changes are included in this document.

1. Updates to Chapter 1

Change bars and **violet** text show changes to **Chapter 1** of the *Intel® 64 and IA-32 Architectures Optimization Reference Manual*: **Introduction**.

Changes to this chapter:

- Section 1.3: Updated Table 1-1 with Redwood Cove and Crestmont microarchitectures.
- Section 1.5: Added Definitions
- Section 1.6: Added links.

CHAPTER 1 INTRODUCTION

1.1 ABOUT THIS MANUAL

The *Intel® 64 and IA-32 Architectures Optimization Reference Manual* describes how to optimize software to take advantage of the performance characteristics of IA-32 and Intel 64 architecture processors.

The target audience for this manual includes software programmers and compiler writers. This manual assumes that the reader is familiar with the basics of the IA-32 architecture and has access to the *Intel® 64 and IA-32 Architectures Software Developer's Manual*. A detailed understanding of Intel 64 and IA-32 processors is often required. In many cases, knowledge of the underlying microarchitectures is required.

The design guidelines discussed in this manual for developing high-performance software generally apply to current and future IA-32 and Intel 64 processors. In most cases, coding rules apply to software running in 64-bit mode of Intel 64 architecture, compatibility mode of Intel 64 architecture, and IA-32 modes (IA-32 modes are supported in IA-32 and Intel 64 architectures). Coding rules specific to 64-bit modes are noted separately.

NOTE

A public repository is available with open source code samples from select chapters of this manual. These code samples are released under a 0-Clause BSD license. Intel provides additional code samples and updates to the repository as the samples are created and verified.

Public repository: https://github.com/intel/optimization-manual.

Link to license: https://github.com/intel/optimization-manual/blob/master/COPYING.

1.2 TUNING YOUR APPLICATION

Tuning an application for high performance on any Intel 64 or IA-32 processor requires understanding and basic skills in:

- Intel 64 and IA-32 architecture.
- C and Assembly language.
- Hot-spot regions in the application that impact performance.
- Optimization capabilities of the compiler.
- Techniques used to evaluate application performance.

The Intel® VTune™ Profiler can help you analyze and locate hot-spot regions in your applications.

On many Intel processors, this tool can monitor an application through a selection of performance monitoring events and analyze the performance event data that is gathered during code execution.

This manual also describes data that can be gathered using the performance counters through the processor's performance monitoring events.

1.3 INTEL PROCESSORS SUPPORTING THE INTEL® 64 ARCHITECTURE

The following is a list of Intel processors, series, and product families that support the Intel 64 Architecture[1]. The list is organized by microarchitecture.

Table 1-1. Intel Processors Organized by Microarchitecture

Microarchitecture	Processor(s), Series, Product(s)
Nehalem microarchitecture (45nm)	• The Intel® Core™ i7 processor • Intel® Xeon® processor 3400, 5500, 7500 series
Westmere microarchitecture (32nm)	• Intel® Xeon® processor 5600 series • Intel® Xeon® processor E7 • Various Intel® Core™ i7, i5, i3 processors
Sandy Bridge microarchitecture	• Intel® Xeon® processor E5 family • Intel® Xeon® processor E3-1200 family • Intel® Xeon® processor E7-8800/4800/2800 product families • Intel® Core™ i7-3930K processor • 2nd generation Intel® Core™ i7-2xxx processor series • Intel® Core™ i3-2xxx processor series
Ivy Bridge microarchitecture	• Intel® Xeon® processor E7-8800/4800/2800 v2 product families • Intel® Xeon® processor E3-1200 v2 product family • 3rd generation Intel® Core™ processors
Ivy Bridge-E microarchitecture	• Intel® Xeon® processor E5-4600/2600/1600 v2 product families • Intel® Xeon® processor E5-2400/1400 v2 product families • Intel® Core™ i7-49xx Processor Extreme Edition
Haswell microarchitecture	• Intel® Xeon® processor E3-1200 v3 product family • 4th Generation Intel® Core™ processors
Haswell-E microarchitecture	• Intel® Xeon® processor E5-2600/1600 v3 product families • Intel® Core™ i7-59xx Processor Extreme Edition
Airmont microarchitecture	Intel® Atom® processor Z8000 series
Silvermont microarchitecture	Intel® Atom® processor Z3400 series
Broadwell microarchitecture	• Intel® Core™ M processor family • 5th generation Intel® Core™ processors • Intel® Xeon® processor D-1500 product family • Intel® Xeon® processor E5 v4 family
Skylake microarchitecture	• Intel® Xeon® Scalable processor family • Intel® Xeon® processor E3-1500m v5 product family • 6th generation Intel® Core™ processors
Kaby Lake microarchitecture	7th generation Intel® Core™ processors

1. For more about this architecture, visit: https://www.intel.com/content/www/us/en/architecture-and-technology/microarchitecture/intel-64-architecture-general.html

Table 1-1. (Contd.)Intel Processors Organized by Microarchitecture

Microarchitecture	Processor(s), Series, Product(s)
Goldmont microarchitecture	• Intel Atom® processor C series • Intel Atom® processor X series • Intel® Pentium® processor J series • Intel® Celeron® processor J series • Intel® Celeron® processor N series
Knights Landing microarchitecture	Intel® Xeon Phi™ Processor 3200, 5200, 7200 series
Goldmont Plus microarchitecture	• Intel® Pentium® Silver processor series • Intel® Celeron® processor J series • Intel® Celeron® processor N series
Coffee Lake microarchitecture	• Intel® Xeon® E processors • 8th generation Intel® Core™ processors • 9th generation Intel® Core™ processors
Knights Mill microarchitecture	Intel® Xeon® Phi™ Processor 7215, 7285, 7295 Series
Cascade Lake microarchitecture	2nd generation Intel® Xeon® Scalable processor family
Ice Lake microarchitecture	• Some of the 3rd generation Intel® Xeon® Scalable processor family • Some 10th generation Intel® Core™ processors
Comet Lake microarchitecture	Some 10th generation Intel® Core™ processors
Tiger Lake microarchitecture	Some 11th generation Intel® Core™ processors
Rocket Lake microarchitecture	Some 11th generation Intel® Core™ processors
Cooper Lake microarchitecture	Some of the 3rd generation Intel® Xeon® Scalable processor family
Gracemont microarchitecture	Intel N-Series processors
Alder Lake microarchitecture	12th generation Intel® Core™ processors
Raptor Lake microarchitecture	• 13th generation Intel® Core™ processors • 14th generation Intel® Core™ processors
Sapphire Rapids microarchitecture	4th generation Intel® Xeon® Scalable processor family
Emerald Rapids microarchitecture	5th generation Intel® Xeon® Scalable processor family
Granite Lake microarchitecture	6th generation Intel® Xeon® Scalable processor family
Meteor Lake microarchitecture	1st generation Intel® Core™ Ultra processors

1.4 ORGANIZATION OF THIS MANUAL

This manual is divided into two volumes. The first considers the optimization of newer products and technologies. Volume Two considers older technology that may not currently be supported.

1.4.1 CHAPTER SUMMARIES

1.4.1.1 Volume 1

- **Chapter 1: Introduction:** Defines the purpose and outlines the contents of this manual.
- **Chapter 2: Intel® 64 and IA-32 Processor Architectures:** Describes the microarchitecture of recent Intel 64 and IA-32 processor families, and other features relevant to software optimization.
- **Chapter 3: General Optimization Guidelines:** Describes general code development and optimization techniques that apply to all applications designed to take advantage of the common features of current Intel processors.
- **Chapter 4: Intel Atom® Processor Architecture:** Describes the microarchitecture of recent Intel Atom processor families, and other features relevant to software optimization.
- **Chapter 5: Coding for SIMD Architectures:** Describes techniques and concepts for using the SIMD integer and SIMD floating-point instructions provided by the MMX™ technology, Streaming SIMD Extensions, Streaming SIMD Extensions 2, Streaming SIMD Extensions 3, SSSE3, and SSE4.1.
- **Chapter 6: Optimizing for SIMD Integer Applications**: Provides optimization suggestions and common building blocks for applications that use the 128-bit SIMD integer instructions.
- **Chapter 7: Optimizing for SIMD Floating-point Applications:** Provides optimization suggestions and common building blocks for applications that use the single-precision and double-precision SIMD floating-point instructions.
- **Chapter 8: INT8 Deep Learning Inference:** Describes INT8 as a data type for Deep learning Inference on Intel technology. The document covers both AVX-512 implementations and implementations using the new Intel® DL Boost Instructions.
- **Chapter 9: Optimizing Cache Usage**: Describes how to use the PREFETCH instruction, cache control management instructions to optimize cache usage, and the deterministic cache parameters.
- **Chapter 10: Introducing Sub-NUMA Clustering:** Describes Sub-NUMA Clustering (SNC), a mode for improving average latency from last level cache (LLC) to local memory.
- **Chapter 11: Multicore and Intel® Hyper-Threading Technology:** Describes guidelines and techniques for optimizing multithreaded applications to achieve optimal performance scaling. Use these when targeting multicore processor, processors supporting Hyper-Threading Technology, or multiprocessor (MP) systems.
- **Chapter 12: Intel® Optane™ DC Persistent Memory:** Provides optimization suggestions for applications that use Intel® Optane™ DC Persistent Memory.
- **Chapter 13: 64-Bit Mode Coding Guidelines:** This chapter describes a set of additional coding guidelines for application software written to run in 64-bit mode.
- **Chapter 14: SSE4.2 and SIMD Programming for Text-Processing/Lexing/Parsing:** Describes SIMD techniques of using SSE4.2 along with other instruction extensions to improve text/string processing and lexing/parsing applications.
- **Chapter 15: Optimizations for Intel® AVX, FMA, and Intel® AVX2:** Provides optimization suggestions and common building blocks for applications that use Intel® Advanced Vector Extensions, FMA, and Intel® Advanced Vector Extensions 2 (Intel® AVX2).

- **Chapter 16: Intel Transactional Synchronization Extensions:** Tuning recommendations to use lock elision techniques with Intel Transactional Synchronization Extensions to optimize multi-threaded software with contended locks.
- **Chapter 16: Power Optimization for Mobile Usages:** This chapter provides background on power saving techniques in mobile processors and makes recommendations that developers can leverage to provide longer battery life.
- **Chapter 17: Software Optimization for Intel® AVX-512 Instructions:** Provides optimization suggestions and common building blocks for applications that use Intel® Advanced Vector Extensions 512.
- **Chapter 18: Intel® Advanced Vector Extensions 512-FP16 Instruction Set for Intel® Xeon® Processors: D**escribes the addition of the FP16 ISA for Intel AVX-512 to handle IEEE 754-2019 compliant half-precision floating-point operations.
- **Chapter 19: Cryptography & Finite Field Arithmetic Enhancements:** Describes the new instruction extensions designated for acceleration of cryptography flows and finite field arithmetic.
- **Chapter 20: Intel® Advanced Matrix Extensions (Intel® AMX): Describes best practices to** optimally code to the metal on Intel® Xeon® Processors based on Sapphire Rapids SP microarchitecture. It extends the public documentation on Optimizing DL code with DL Boost instructions.
- **Chapter 21: Intel® QuickAssist Technology (Intel® QAT):** Describes software development guidelines for the Intel® QuickAssist Technology (Intel® QAT) API. This API supports both the Cryptographic and Data Compression services.
- **Appendix A: Application Performance Tools:** Introduces tools for analyzing and enhancing application performance without having to write assembly code.
- **Appendix B: Using Performance Monitoring Events:** Provides information on the Top-Down Analysis Method and information on how to use performance events specific to the Intel Xeon processor 5500 series, processors based on Sandy Bridge microarchitecture, and Intel Core Solo and Intel Core Duo processors.
- **Appendix C: Intel Architecture Optimization with Large Code Pages:** Provides information on how the performance of runtimes can be improved by using large code pages.

1.4.1.2 Volume 2: Earlier Generations of Intel® 64 and IA-32 Processor Architectures

- **Chapter 1: Haswell Microarchitecture:** Describes the Haswell microarchitecture.
- **Chapter 2: Sandy Bridge Microarchitecture:** Describes the Sandy Bridge microarchitecture and associated considerations.
- **Chapter 3: Intel® Core™ Microarchitecture and Enhanced Intel® Core™ Microarchitecture:** Describes the Intel® Core™ and Enhanced Intel® Core ™microarchitectures and associated considerations.
- **Chapter 4: Nehalem Microarchitecture:** Describes the Sandy Bridge microarchitecture and associated considerations.
- **Chapter 5: Knights Landing Microarchitecture Optimization:** Describes the Sandy Bridge microarchitecture and associated considerations, including Multithreading and Intel® HyperThreading Technology (Intel® HT).
- **Chapter 6: Earlier Generations of Intel Atom® Microarchitecture and Software Optimization:** Describes the microarchitecture of earlier generations of processor families based on Intel Atom microarchitecture, and software optimization techniques targeting Intel Atom microarchitecture.

1.5 GLOSSARY

Table 1-2 provides definitions of commonly used terms throughout this volume.

Table 1-2. Term Definitions

Term	Description
Arithmetic formats	Sets of binary and decimal floating-point data, which consist of finite numbers (including signed zeros and subnormal numbers), infinities, and special "not a number" values.
CFP16	Complex-valued floating-point format comprising two FP16 values representing the real and imaginary values respectively. When used in SIMD, the individual real/imaginary values from each complex value are interleaved in the register.
Denormal	A subset of denormalized numbers that fill the underflow gap around zero in floating-point arithmetic.
FP16	Half precision 16-bit floating-point data format.
FP32	Single precision 32-bit floating-point data format.
FP64	Double precision 64-bit floating-point data format.
FFT	Fast Fourier Transform.
IEEE 754-2019	The current IEEE Standard for Floating-Point Arithmetic used in Intel® AVX-512 FP16 instructions.
Intel® AVX	Intel® Advanced Vector Extensions.
Intel® AVX2	Intel® Advanced Vector Extensions 2.
Intel® AVX-512	Intel® Advanced Vector Extensions 512.
Intel® AVX-512 FP16	ISA for handling half precision floating-point, added as an extension to Intel AVX-512.
Intrinsic	A function that can be called from a high-level language, like C/C++, which gives direct access to the underlying ISA. Intrinsics allow the programmer to bypass the compiler and directly specify that a particular instruction be used.
ISA	Instruction Set Architecture[1]: a part of the abstract model of a computer, which generally defines how software controls the CPU[2].
MMSE	Minimum Mean Squared Error.
NaN	Not A Number. A way to represent a value that is undefined or unrepresentable. For example, the square root of a negative number would generate a NaN value.
Normal	A floating-point number that can be represented without leading zeros in its significand.
SIMD	Single instruction, multiple data. A way of packing several data elements into a single container and operating on them all at once.[3]
SINR	Signal-to-Interference-plus-Noise Ratio.
SSE	SIMD Streaming Extensions[4].

Table 1-2. Term Definitions

Term	Description
µop	Also uop. Refers to **micro-operations** (micro-ops) and is usually identified in code as UOP. Micro-operations are detailed low-level instructions used in some designs to implement complex machine instructions (sometimes referred to as macro-instructions in this context).[5]

NOTES:

1. See Intel's Instruction Set Architecture landing page.
2. See Wikipedia.
3. See Chapter 5, "Coding for SIMD Architectures"
4. See Intel's Instruction Set Extensions Technology Support landing page.
5. See Wikipedia for a deep dive.

1.6 RELATED INFORMATION

For more information on the Intel® architecture, techniques, and the processor architecture terminology, the following are of particular interest. Intel publishes new and updated content continuously.

Table 1-3. Additional References in and Beyond this Document

Title	Description
Intel® 64 and IA-32 Architectures Software Developer's Manual	These manuals describe the architecture and programming environment of the Intel® 64 and IA-32 architectures. This links directly to the PDF containing all 4 columns of the content.
CPU Tuning and Optimization Guides	This page includes guides covering specific Processors and technologies.
Intel® 64 Architecture Processor Topology Enumeration	Covers the topology enumeration algorithm for single-socket to multiple-socket platforms using Intel® 64 and IA-32 processors.
Intel® Artificial Intelligence (Intel® AI) Solutions landing page	The official source for development using Intel® AI solutions supporting Deep Learning (DL) and Machine Learning (ML). Includes a section with documentation.
Support for Intel® Processors	Landing page for support information for Intel® processors including featured content, downloads, specifications, warranty, and community posts.
Get Started with Intel® Fortran Compiler Classic and Intel® Fortran Compiler	A guide to the basics of using Intel® Fortran Compilers: ifort and ifx. Please note: IFORT will be discontinued in October 2024.
Intel® C++ Compiler Classic (ICC) Developer Guide and Reference	Contains information about the Intel® C++ Compiler Classic (icc for Linux* and icl for Windows*) compiler and runtime environment.
Intel® Data Streaming Accelerator User Guide	

Table 1-3. Additional References in and Beyond this Document

Title	Description
Intel® Developer Zone Landing Page	The official source for developing on Intel® hardware and software. Includes documentation.
Intel® Developer Catalog	Find software and tools to develop and deploy solutions optimized for Intel® architecture.
Intel® Development Topics & Technologies landing page	A landing page devoted to everything from storage to computer vision (CV).
Intel® Distribution of OpenVino™ Toolkit landing page	The official source for the Intel® distribution of OpenVINO™, an open source toolkit that simplifies deployment. Includes a section with documentation.
Intel® Hyper-Threading Technology (Intel® HT Technology)	An overview of Intel® HT Technology. This links directly to the PDF.
Intel® In-Memory Analytics Accelerator Architecture Specification	Describes the architecture of the Intel® In-Memory Analytics Accelerator (Intel® IAA). This links directly to the PDF.
Intel® Instruction Set Extensions Technology Support	A landing page dedicated to all content related to supporting the Intel® ISE technologies. Includes the Intel® SSE4 Programming Reference.
Intel® oneAPI Data Analytics Library Landing Page	The official source for development using Intel® one API Data Analytics Library (oneDAL). Includes a section with documentation.
Intel® oneAPI DPD++/C++ Compiler	Intel® oneAPI DPC++/C++ Compile, a standards-based, cross-architecture compiler and update to both ifort and ifx.
Intel® QuickAssist Technology (Intel® QAT)	The official source for the Intel® QuickAssist Technology (Intel® QAT). Includes a section with documentation.
Intel® VTune™ Profiler User Guide	A comprehensive overview of the product functionality, tuning methodologies, workflows, and instructions to use the Intel® VTune™ Profiler performance analysis tool. This links directly to the PDF.
Intel® Xeon® Processors Technical Resources Page	A landing page including technical resources for all Intel® Xeon® Scalable processors.
C2C - False Sharing Detection in Linux Perf	An introduction to perf c2c in Linux.
Developing Multi-Threaded Applications: A Platform Consistent Approach	The objective of the Multithreading Consistency Guide is to provide guidelines for developing efficient multithreaded applications across Intel-based symmetric multiprocessors (SMP) and/or systems with Intel® Hyper-Threading Technology (Intel® HT). (2005)

2. Updates to Chapter 2

Change bars and **violet** text show changes to Chapter 2 of the Intel® 64 and IA-32 Architectures Optimization Reference Manual: **Intel® 64 and IA-32 Processor Architectures.**

Changes to this chapter:

- Section 2.1
  - Section 2.1.1: New section: **The Redwood Cove Microarchitecture.**
  - Figure 2-1: New additional image.
- Section 2.8: Added new heading: **Intel® 64 and IA-32 Instruction Best Practices.**
  - 2.8.1: New section: **Non-Privileged Instruction Serialization.**

CHAPTER 2 INTEL® 64 AND IA-32 PROCESSOR ARCHITECTURES

This chapter overviews features relevant to software optimization for current generations of Intel® 64 and IA-32 processors[1]. These features include:

- Microarchitectures that enable executing instructions with high throughput at high clock speeds, a high-speed cache hierarchy, and a high-speed system bus.
- Intel® Hyper-Threading Technology[2] (Intel® HT Technology) support.
- Intel 64 architecture on Intel 64 processors.
- Single Instruction Multiple Data (SIMD) instruction extensions: MMX™ technology, Streaming SIMD Extensions (Intel® SSE), Streaming SIMD Extensions 2 (Intel® SSE2), Streaming SIMD Extensions 3 (Intel® SSE3), Supplemental Streaming SIMD Extensions 3 (SSSE3), Intel® SSE4.1, and Intel® SSE4.2.
- Intel® Advanced Vector Extensions (Intel® AVX).
- Half-precision floating-point conversion and RDRAND.
- Fused Multiply Add Extensions.
- Intel® Advanced Vector Extensions 2 (Intel® AVX2).
- ADX and RDSEED.
- Intel® Advanced Vector Extensions 512 (Intel® AVX-512).
- Intel® Thread Director.

2.1 6TH GENERATION INTEL® XEON® SCALABLE PROCESSOR FAMILY

2.1.1 THE REDWOOD COVE MICROARCHITECTURE

The Redwood Cove microarchitecture is the successor of the Golden Cove microarchitecture. The Redwood Cove microarchitecture introduces the following enhancements:

- Improvements for larger code footprint workloads.
- Larger instruction cache: 32K→64K.
- Branch Hint x86 architecture extension.
- Code Software Prefetch x86 architecture extension (Granite Rapids only).
- Improved LSD coverage: the IDQ can hold 192 uops per logical processor in single-thread mode or 96 uops per thread when SMT is active.
- Improved branch prediction and reduced average branch misprediction recovery latency.
- New LD+OP and MOV+OP macro fusions.

1. Intel® Atom® processors are covered in Chapter 4, “Intel Atom® Processor Architectures.”
2. Intel® HT Technology requires a computer system with an Intel processor supporting hyper-threading and an Intel® HT Technology-enabled chipset, BIOS, and operating system. Performance varies depending on the hardware and software used.

- EXE: 3-cycle Floating Point multiplication.
- Improved SMT performance and efficiency.
- Improved load lock performance.
- New HW data prefetcher to recognize and prefetch the “Array of Pointers” pattern.
- Mid-level-cache size increased to 2MBs for Client.
- Improved Memory Bandwidth.
  - Increased number of outstanding misses (48→64 Deeper MLC miss queues).
  - LLC Page Prefetcher.
- AMX supports FP16 for AI/ML (Granite Rapids only).

2.1.1.1 Branch Hint

If the branch predictor has stored information about a conditional branch, the predictor should use this information to predict the branch. If the predictor does not have stored information, the predictor predicts the branch to be **not-taken**. This is usually, but not always, correct.

Starting with the Redwood Cove microarchitecture, if the predictor has no stored information about a branch, the branch has the Intel® SSE2 branch taken hint (i.e., instruction prefix 3EH),

When the codec decodes the branch, it flips the branch’s prediction from not-taken to taken. It then flushes the pipeline in front of it and steers this pipeline to fetch the **taken path** of the branch.

If the branch is taken when executed, the hint reduces the misprediction penalty by replacing an execution time pipeline flush and re-steer with a decode time of one. Alternatively, if the branch is not-taken, the hint increases the branch misprediction penalty by introducing a new decode time and a new execution time flush and re-steer. Consequently, the hint should not be added to a **not-taken branch** since this increases the program’s run time (see Section 2.1.1.3).

The hint is only used when the predictor does not have stored information about the branch. To avoid code bloat and reducing the instruction fetch bandwidth, don’t add the hint to a branch in hot code—for example, a branch inside a loop with a high iteration count—because the predictor will likely have stored information about that branch. Ideally, the hint should only be added to infrequently executed branches that are mostly taken, but identifying those branches may be difficult. Compilers are advised to add the hints as part of profile-guided optimization, where the one-sided execution path cannot be laid out as a fall-through. The Redwood Cove microarchitecture introduces new performance monitoring events to guide hint placement (see Section 2.1.1.3).

Judiciously adding hints can reduce the program’s run time by reducing the branch misprediction penalty. Hints are especially useful for large code footprint workloads where the number of branches exceeds the predictor's capacity.

Details include:

- The hint only applies to JCCs, not CXZ and LOOP/LOOPCC[1].
- The microarchitecture is obliged to decode the Intel® SSE2 branch not-taken hint (i.e., instruction prefix 2EH), but otherwise ignores it.
  - There’s no need to hint that a branch is not-taken since, by default, it’s predicted not-taken if the predictor doesn’t have stored information about it.
- An instruction has the taken hint if it has at least one instruction prefix 3EH.
  - **Example:** If an instruction has the 3EH and 2EH prefixes, regardless of order, it still has the taken the hint.

1. As specified in section 2.1.1 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 2A Chapter 2, “Instruction Format”.

2.1.1.2 Profiling Support for Branch Hints

The following performance monitoring events may be used to identify branches that may benefit from adding the previous hint:

- FRONTEND_RETIRED.MISP_ANT (Event Code = C6H, Unit Mask = 02H): Always Not-Taken conditional branch instructions retired that were miss-predicted.
- FRONTEND_RETIRED.ANY_ANT (Event Code = C6H, Unit Mask = 03H): Always Not-Taken conditional branch instructions retired.
- BR_INST_RETIRED.COND (Event Code = C4H, Unit Mask = 11H): conditional branch instructions retired.

Note that collecting Last Branch Records (LBR) when sampling on the last two events is instrumental to determining the polarity of the branch (e.g., was the branch mostly taken or mostly not taken).

2.1.1.3 New Redwood Cove Macro-Fusions

The Redwood Cove microarchitecture supports two new types of instruction macro-fusion: MOV-OP and LOAD-OP. The new macro-fusions are only supported out of the µop-Cache (not supported on the legacy decode pipeline).

With a new MOV+OP macro-fusion, a register MOV instruction can be fused with following an OP instruction to form the non-destructive source operation form: c ← *a* op *b*, preserving both source operands.

For example:

Table 2-1. MOV+OP Macro-Fusion Example

Class	Example	Macro-Fused To:
MOV+OP	mov rax, rbx sub rax, rcx	rax = sub rbx, rcx

The two instructions are macro-fused to a single micro-operation cached in the µop-Cache. The macro-fused operation takes a single slot in IDQ and during allocation, execution, and retirement.

Similarly, with a new LOAD+OP macro-fusion, a LOAD instruction could be macro-fused by following an OP instruction to form a fused load-op micro-operation, which often cannot be encoded within the x86 instruction set.

For example:

Table 2-2. LD+OP Macro-Fusion Example

Class	Example	Macro-Fused To:
LD+OP	mov eax, [mem] sub eax, ebx	eax = sub [mem], ebx[1]

NOTES:

1. Note that only sub reg, [mem] is encodable within x86 instruction set.

The Limitations of LOAD+OP Macro-Fusion

The following instruction pair can be macro-fused to the single DST:= OP [mem], SRC_REG micro-operation.

(1) MOV DST, [mem]

(2)OP DST, SRC_REG // SRC_REG cannot be immediate and must be different from DST

- LOAD and OP operations must be the same Operand Size 32/64 matching load data size.
- Load cannot be encoded using RIP-Relative form or have Index operand.
- Only integer and packed SIMD/floating point 128-bit vector operations are supported.
- The LOAD must be:
  - — An integer load operation without sign-extension (opcode 0x8B).
  - — SSE packed load instructions (movaps/pd, movups/pd, movdqa/u).
- No VEX or EVEX forms are supported.

2.1.1.4 Improved Memory BW

The Redwood Cove microarchitecture improves the peak memory bandwidth that a core can consume over the Golden Cove microarchitecture. This significantly increases the depth of the MLC miss queue (48→64 outstanding MLC misses) and introduces new hardware prefetcher algorithms.

2.1.1.5 Array of Pointers Prefetcher

An array of pointers is an array whose elements are pointers. The data associated with each element of the array is usually dynamically allocated elsewhere and can be accessed by dereferencing the pointer pointing to it. For example:

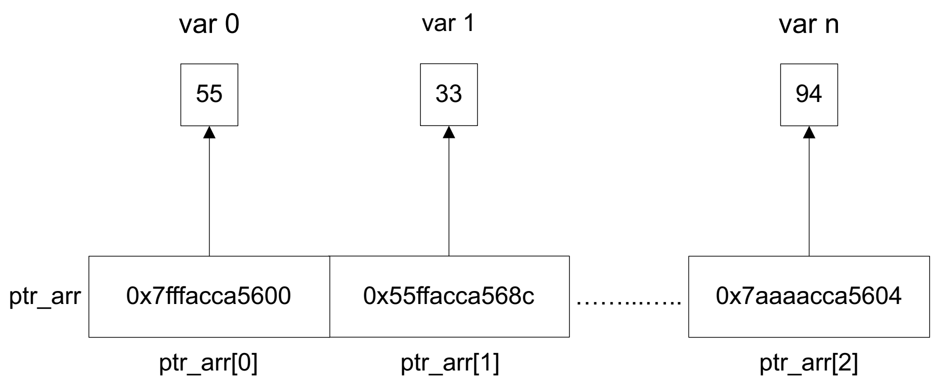


Figure 2-1. Layout of Array of Pointers Prefetchers

The Redwood Cove microarchitecture adds a new Array of Pointers (AOP) hardware prefetcher for detecting and prefetching an array of pointers references into the cache hierarchy. The AOP prefetcher treats the data prefetched for a constant stride load as a pointer and may issue prefetch requests to the memory addresses corresponding to the pointer's value.

Successful detection of the pattern allows AOP Prefetch to bring data to be accessed by the program in advance, saving costly cache misses.

To benefit most from the AOP Prefetching capabilities, we recommend the following:

The array of pointers pattern, in a nutshell, may look like this:

```
for (i=0; i<array_length; i++) {
    (1)mov rax, array[i]
    (2)mov rbx, [rax + offset]; data of rax used as a pointer
}
```

In the Redwood Cove microarchitecture, to benefit from AOP Prefetch, software must ensure that:

- 1st load operation is a regular integer load, naturally aligned to its data size (64-bit or 32-bit in legacy or compatibility mode)
- 2nd load operation depends upon the first load through the base operand and has no index operand.
  - The AOP Prefetcher does not track and would issue a prefetch ignoring the offset. Therefore, it is recommended to use small offsets or avoid using offsets at all.
  - The AOP Prefetcher avoids triggering prefetch if the offset exceeds 32 bytes.

Software that observes any issue with the AOP Prefetcher engine (for example, exceeded memory bandwidth due to redundant prefetch requests) can deactivate it through bit 7 of MSR 0x1A4 (MSR_PREFETCH_CTRL). The AOP Prefetch is a data-dependent prefetcher. Security implications of data-dependent prefetching are discussed separately at:

https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/data-dependent-prefetcher.html

2.1.1.6 LLC Page Prefetcher (LLCPP)

The LLC Page Prefetcher extends the Next-Page-Prefetcher algorithm, which detects a sequence of accesses that are about to reach the end of a page (either going up towards the next page in memory or down towards the previous page in memory) and triggers a linear prefetch to the start of the next predicted page. NPP's intends to prefetch the translation of the next page and initiate prefetching of the first few cache lines from that predicted page.

The LLC Page Prefetch enhances Next-Page Prefetcher in two significant ways:

- It issues prefetches two pages ahead in linear address space.
- The prefetch attempts to bring the entire 4Kb page to the last level cache using opportunistic IDI request slots without the DCU or MLC buffers.

LLCPP is effective when streaming through memory over a buffer greater than 8KB. Smaller buffer access might result in redundant prefetching to the last level cache.

Software that observes any issue with the LLC Page Prefetcher engine (for example, exceeded memory bandwidth due to redundant prefetch requests) can disable LLC Page Prefetcher through bit 6 of MSR 0x1A4 (MSR_PREFETCH_CTRL).

2.2 THE SAPPHIRE RAPIDS MICROARCHITECTURE

Intel processors based on Sapphire Rapids microarchitecture use Golden Cove cores and support the following additional features:

- Intel® Advanced Matrix Extensions (Intel® AMX) (Chapter 20).
- Intel® Advanced Vector Extensions 512 (Intel® AVX-512) (Chapter 18).
- Intel® Data Streaming Accelerator (Intel® DSA)[1].

1. Please see the Intel® DSA Specification and Intel® DSA User Guide.

- Intel® In-Memory Analytics Accelerator (Intel® IAA)[1].
- Intel® Quick Assist Technology (Intel® QAT)(Chapter 21)

2.2.1 4TH GENERATION INTEL® XEON® SCALABLE FAMILY OF PROCESSORS

Intel's fourth generation Xeon® Scalable Family of Processors changes from a single-die monolithic design to multi-die Tiles.

- The server products are scalable from dual-socket to eight-socket configurations.
- The I/O is increased with PCI Express 5.0, DDR5 memory, and Compute Express Link 1.1.
- Packaging includes a multi-die chip with up to 4 tiles. Each tile is a 400mm2 SoC, providing compute cores and I/O.
- Each tile contains 15 Golden Cove cores (see Section 2.4). Its memory controller provides two channels of DDR5, up to eight channels across four tiles, and 28 PCIe 5.0 lanes, up to a maximum of 112 across four tiles.

2.3 THE ALDER LAKE PERFORMANCE HYBRID ARCHITECTURE

The Alder Lake performance hybrid architecture combines two Intel architectures, combining the Golden Cove performant cores and the Gracemont efficient Atom cores onto a single SoC. For details on the Golden Cove microarchitecture, see Section 2.4. For details on the Crestmont microarchitecture, see Section 4.1

2.3.1 12TH GENERATION INTEL® CORE™ PROCESSORS SUPPORTING PERFORMANCE HYBRID ARCHITECTURE

12th Generation Intel® Core™ processors supporting performance hybrid architecture compriseup to eight Performance cores (P-cores) and eight Efficient cores (E-cores). These processors also include a 3MBs Last Level Cache (LLC) per IDI module, where a module is one P-core or four E-cores. It has a symmetrical ISA and comes in a variety of configurations.

P-cores provide single or limited thread performance, while E-cores help provide improved scaling and multithreaded efficiency. These processors' P-cores can also have Intel Hyper-Threading Technology enabled. When the operating system (OS) decides to schedule all processors, all cores can be active simultaneously.

A key OSV requirement for enabling hybrid is symmetric ISA across different core types in a performance hybrid architecture. In 12th Generation Intel Core processors supporting performance hybrid architecture, ISA is converged to a common baseline between the P-cores and E-cores. To maintain symmetric ISA, the E-cores do not support the following features: Intel AVX-512, Intel AVX-512 FP-16, and Intel® TSX. The E-cores do support Intel AVX2 and Intel AVX-VNNI.

2.3.2 HYBRID SCHEDULING

2.3.2.1 Intel® Thread Director

Intel® Thread Director monitors software in real-time, giving hints to the operating system's scheduler, allowing it to make more intelligent and data-driven decisions on thread scheduling. With Intel Thread Director, the hardware provides runtime feedback to the OS per thread based on various IPC performance characteristics in the form of:

1. Please see the Intel® IAA Specification.

- Dynamic performance and energy efficiency capabilities of P-cores and E-cores based on power/thermal limits.
- Idling hints when power and thermal are constrained.

Intel Thread Director was first introduced in desktop and mobile variants of the 12th generation Intel Core processor based on Alder Lake performance hybrid architecture.

A processor containing P-cores and E-cores with different performance characteristics challenges the operating system's scheduler. Additionally, different software threads see different performance ratios between the P-cores and E-cores. For example, the performance ratio between the P-cores and E-cores for highly vectorized floating-point code is higher than that for scalar integer code. So, when the operating system must make an optimal scheduling decision, it must be aware of the characteristics of the software threads that are candidates for scheduling. Suppose there are insufficient P-cores and a mix of software threads with different characteristics. In that case, the operating system should schedule those threads that benefit most from the P-cores onto those cores and schedule the others on the E-cores.

Intel Thread Director provides the necessary hint to the operating system about the characteristics of the software thread executing on each of the logical processors. The hint is dynamic and reflects the recent characteristics of the thread, i.e., it may change over time based on the dynamic instruction mix of the thread. The processor also considers microarchitecture factors to define the dynamic software thread characteristics.

Thread specific hardware support is enumerated via the CPUID instruction and enabled by the operating system via writing to configuration MSRs. The Intel Thread Director implementation on processors based on Alder Lake performance hybrid architecture defines four thread classes:

1. Non-vectorized integer or floating-point code.
2. Integer or floating-point vectorized code, excluding Intel® Deep Learning Boost (Intel® DL Boost) code.
3. Intel DL Boost code.
4. Pause (spin-wait) dominated code.

The dynamic code need not be 100% of the class definition. It should be large enough to be considered belonging to that class. Also, dynamic microarchitectural metrics such as consumed memory bandwidth or cache bandwidth may move software threads between classes. Example pseudo-code sequences for the Intel Thread Director classes available on processors based on Alder Lake performance hybrid architecture are provided in Examples 2-1 through 2-4.

Intel Thread Director also provides a table in system memory, only accessible to the operating system, that defines the P-core vs. E-core performance ratio per class. This allows the operating system to pick and choose the correct software thread for the correct logical processor.

In addition to the performance ratio between P-cores and E-cores, Intel Thread Director provides the energy efficiency ratio between those cores. The operating system can use this information when it prefers energy savings over maximum performance. For example, a background task such as indexing can be scheduled on the most energy-efficient core since its performance is less critical.

Example 2-1. Class 0 Pseudo-code Snippet

```
while (1)
{
    asm("xor rax, rax;"
          "add rax, 5;"
          "inc rax;"
    );
}
```

Example 2-2. Class 1 Pseudo-code Snippet

```
while (1)
{
    asm(“vfmaddsub132ps %ymm0, %ymm1, %ymm2;”
         “vfmaddsub213ps %ymm0, %ymm1, %ymm3;”
         “vfmaddsub231ps %ymm0, %ymm1, %ymm4;”
         “vfmaddsub132ps %ymm0, %ymm1, %ymm5;”
         “vfmaddsub213ps %ymm0, %ymm1, %ymm6;”
         “vfmaddsub231ps %ymm0, %ymm1, %ymm7;”
         “vfmaddsub132ps %ymm0, %ymm1, %ymm8;”
         “vfmaddsub213ps %ymm0, %ymm1, %ymm9;”
         “vfmaddsub231ps %ymm0, %ymm1, %ymm10;”
         “vfmaddsub132ps %ymm0, %ymm1, %ymm2;”
         “vfmaddsub213ps %ymm0, %ymm1, %ymm3;”
         “vfmaddsub231ps %ymm0, %ymm1, %ymm4;”
         “vfmaddsub132ps %ymm0, %ymm1, %ymm5;”
         “vfmaddsub213ps %ymm0, %ymm1, %ymm6;”
         “vfmaddsub231ps %ymm0, %ymm1, %ymm7;”
         “vfmaddsub132ps %ymm0, %ymm1, %ymm8;”
         “vfmaddsub213ps %ymm0, %ymm1, %ymm9;”
         “vfmaddsub231ps %ymm0, %ymm1, %ymm10;”
         “vfmaddsub132ps %ymm0, %ymm1, %ymm2;”
         “vfmaddsub213ps %ymm0, %ymm1, %ymm3;”
         “vfmaddsub231ps %ymm0, %ymm1, %ymm4;”
         “vfmaddsub132ps %ymm0, %ymm1, %ymm5;”
         “vfmaddsub213ps %ymm0, %ymm1, %ymm6;”
         “vfmaddsub231ps %ymm0, %ymm1, %ymm7;”
         “vfmaddsub132ps %ymm0, %ymm1, %ymm8;”
         “vfmaddsub213ps %ymm0, %ymm1, %ymm9;”
         “vfmaddsub231ps %ymm0, %ymm1, %ymm10;”
         “vfmaddsub132ps %ymm0, %ymm1, %ymm2;”
         “vfmaddsub213ps %ymm0, %ymm1, %ymm3;”
         “vfmaddsub231ps %ymm0, %ymm1, %ymm4;”
         “vfmaddsub132ps %ymm0, %ymm1, %ymm5;”
         “vfmaddsub213ps %ymm0, %ymm1, %ymm6;”
         “vfmaddsub231ps %ymm0, %ymm1, %ymm7;”
         “vfmaddsub132ps %ymm0, %ymm1, %ymm8;”

         “vfmaddsub213ps %ymm0, %ymm1, %ymm9;”
         “vfmaddsub231ps %ymm0, %ymm1, %ymm10;”
         “vfmaddsub132ps %ymm0, %ymm1, %ymm2;”
    );
}
```

Example 2-3. Class 2 Pseudo-code Snippet

```
while (1)
{
    __asm(
        vpdpbusd ymm2, ymm0, ymm1
        vpdpbusd ymm3, ymm0, ymm1
        vpdpbusd ymm4, ymm0, ymm1
        vpdpbusd ymm5, ymm0, ymm1
        vpdpbusd ymm6, ymm0, ymm1
        vpdpbusd ymm7, ymm0, ymm1
        vpdpbusd ymm8, ymm0, ymm1
        vpdpbusd ymm9, ymm0, ymm1
        vpdpbusd ymm10, ymm0, ymm1
        vpdpbusd ymm11, ymm0, ymm1
        vpdpbusd ymm12, ymm0, ymm1
        vpdpbusd ymm13, ymm0, ymm1
    );
}
```

Example 2-4. Class 3 Pseudo-code Snippet

```
while (1)
{
        asm("PAUSE;")
        asm("PAUSE;")
        asm("PAUSE;")
        asm("PAUSE;")
        asm("PAUSE;")
        asm("PAUSE;")
        asm("PAUSE;")
        asm("PAUSE;")
        asm("PAUSE;")
        asm("PAUSE;")
    );
}
```

2.3.2.2 Scheduling with Intel® Hyper-Threading Technology-Enabled on Processors Supporting x86 Hybrid Architecture

E-cores are designed to provide better performance than a logical P-core with both hardware sibling hyper-threads busy.

2.3.2.3 Scheduling with a Multi-E-Core Module

E-cores within an idle module help improve performance than E-cores in a busy module.

2.3.2.4 Scheduling Background Threads on x86 Hybrid Architecture

In most scenarios, background threads can leverage the scalability and multithread efficiency of E-cores.

2.3.3 RECOMMENDATIONS FOR APPLICATION DEVELOPERS

The following are recommendations when using processors supporting performance hybrid architecture:

- Stay up to date on updates on operating systems and optimized libraries.
- Software must avoid setting hard affinities on either threads or processes to allow the operating system to provide the optimal core selection for Intel Hybrid.
- Software should replace active spin-waits with lightweight waits, ideally using the new UMWAIT/TPAUSE and older PAUSE instructions. This will allow the scheduler to get better hints on time spinning.
- Software can utilize the Windows Power Throttling information using process and thread information APIs to give the scheduler hints on the Quality of Service (QoS) required for a particular thread or process, improving performance and energy efficiency.
- Leverage Windows frameworks and media APIs for multimedia application development. Windows Media Foundation framework is optimized for hybrid architecture, enabling media applications to run efficiently while preventing glitches.
- The Windows IrqPolicyMachineDefault policy enables Windows to target interrupts to the right core optimally, and more so on hybrid architecture.

For additional recommendations and information on performance hybrid architecture, refer to the white papers on the Performance Hybrid Architecture page.

2.4 THE GOLDEN COVE MICROARCHITECTURE

The Golden Cove microarchitecture is the successor of the Ice Lake microarchitecture. The Golden Cove microarchitecture introduces the following enhancements:

- Wider machine: 5→6 wide allocation, 10→12 execution ports, and 4→8 wide retirement.
- Significant increases in the size of key structures enable deeper OOO execution and expose more instruction-level parallelism.
- Greater capabilities per execution port, e.g., 5th integer ALU execution ports with expanded capability and a new fast floating-point adder.
- Intel® Advanced Matrix Extensions (Intel® AMX)[1]: Built-in integrated Tiled Matrix Multiplication / Machine Learning Accelerator.
- Improved branch prediction.
- Improvements for large code footprint workloads, for example, larger branch prediction structures, enhanced code prefetcher, and larger instruction TLB.
- Wider fetch: legacy decode pipeline fetch bandwidth increase to 32B/cycles, 4→6 decoders, increased micro-op cache size, and increased micro-op cache bandwidth.

1. Intel AMX are unavailable on client parts.

- Maximum load bandwidth increased from two loads/cycle to three loads/cycle.
- Larger 4K Pages DTLB, increase of outstanding Page Miss handlers.
- Increase of outstanding misses (16 FB, 32⟶48 Deeper MLC miss queues).
- Enhanced data prefetchers for increased memory parallelism.
- Mid-level cache size increased to 2MB on server parts; remains 1.25MB on client parts.

2.4.1 GOLDEN COVE MICROARCHITECTURE OVERVIEW

The basic pipeline functionality of the Golden Cove microarchitecture is depicted in Figure 2-2.

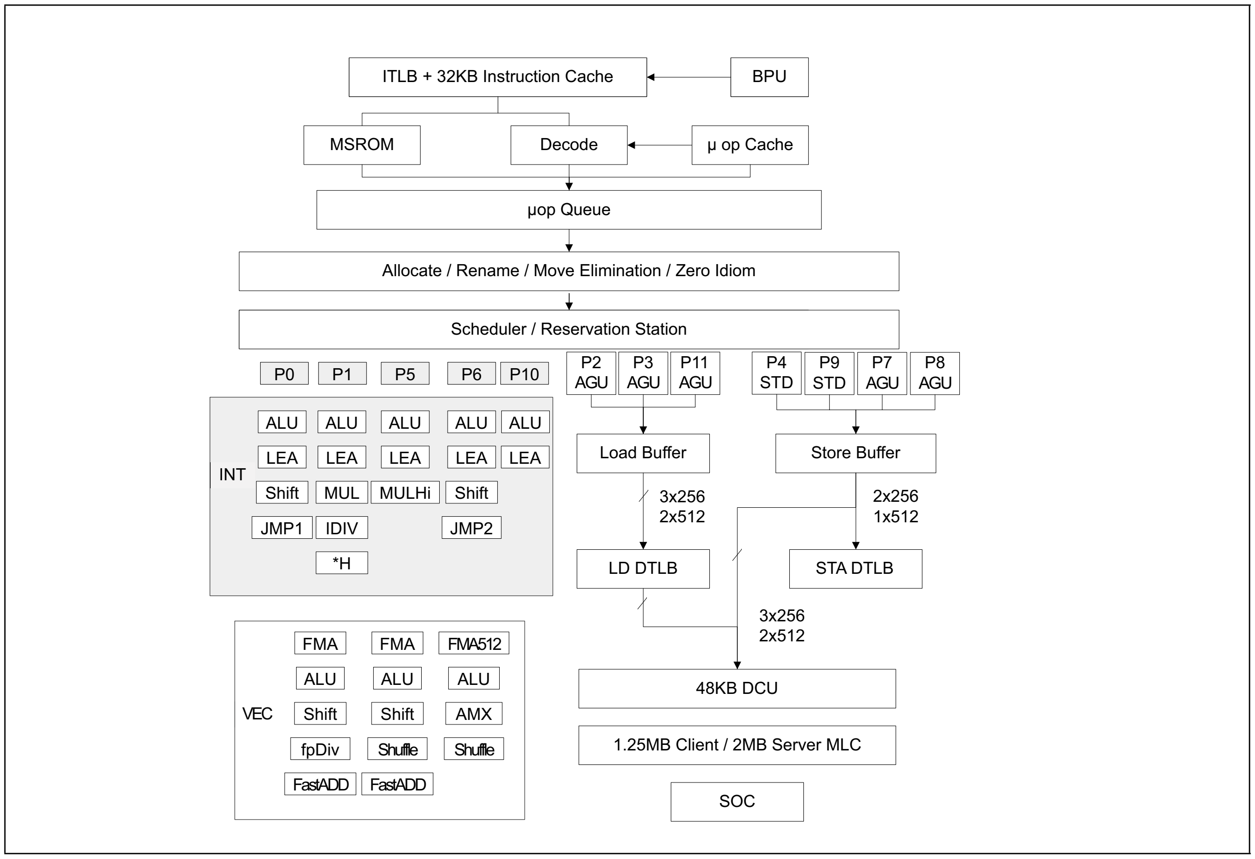


Figure 2-2. Processor Core Pipeline Functionality of the Golden Cove Microarchitecture

The Golden Cove front end is depicted in Figure 2-3. The front end is built to feed the wider and deeper out-of-order core:

- Legacy decode pipeline fetch bandwidth increased from 16 to 32 bytes/cycle.
- The number of decoders increased from four to six, allowing the decode of up to six instructions per cycle.
- The micro-op cache size increased, and its bandwidth increased to deliver up to eight micro-ops per cycle.
- Improved branch prediction.

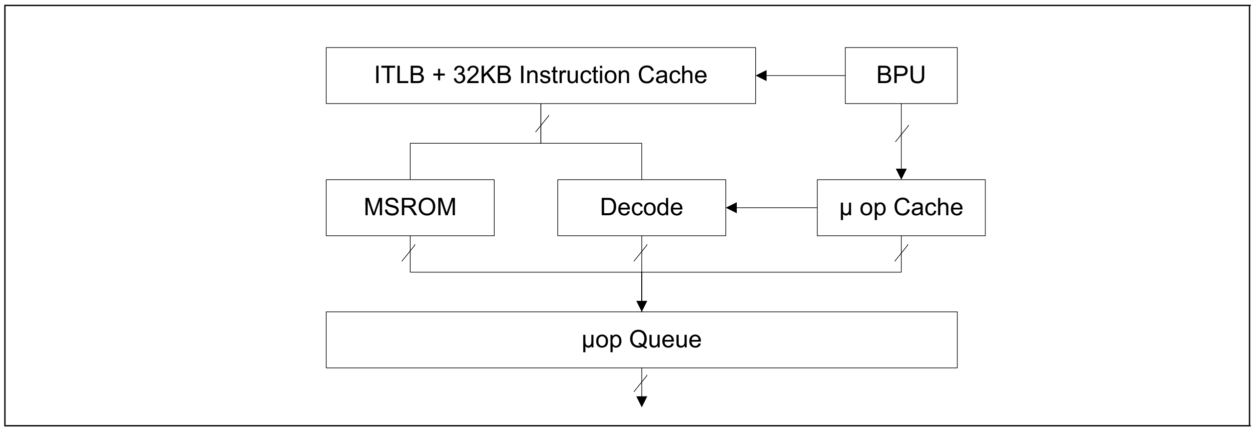


Figure 2-3. Processor Front End of the Golden Cove Microarchitecture

Improvements for large code footprint workloads:

- Double the size of the instruction TLB: 128⟶256 entries for 4K pages, 16⟶32 entries for 2M/4M pages.
- Bigger branch prediction structures.
- Enhanced code prefetcher.
- Improved LSD coverage.
- The IDQ can hold 144 uops per logical processor in single-thread mode, or 72 uops per thread when SMT is active.

Additional improvements include:

- The significant increase in the size of key buffer structures to enable deeper OOO execution and expose more instruction-level parallelism.
- Wider machine:
  — Wider allocation (5⟶6 uops per cycle) and retirement (4⟶8 uops per cycle) width.
  — Increased execution ports (10⟶12).
  — Greater capabilities per execution port.

Table 2-3 summarizes the OOO engine's capability to dispatch different types of operations to ports.

Table 2-3. Dispatch Port and Execution Stacks of the Golden Cove Microarchitecture

Port 0	Port 1[1]	Port 2	Port 3	Port 4	Port 5[2]	Port 6	Ports 7, 8	Port 9	Port 10	Port 11
INT ALU LEA INT Shift Jump1	INT ALU[3] LEA INT Mul INT Div	Load	Load	Store Data	INT ALU LEA INT MUL Hi	INT ALU LEA INT Shift Jump2	Store Address	Store Data	INT ALU LEA	Load

Table 2-3. Dispatch Port and Execution Stacks of the Golden Cove Microarchitecture (Contd.)

Port 0	Port 1[1]	Port 2	Port 3	Port 4	Port 5[2]	Port 6	Ports 7, 8	Port 9	Port 10	Port 11
FMA Vec ALU Vec Shift FP Div	FMA* Fast Adder* Vec ALU* Vec Shift* Shuffle *				FMA** Fast Adder Vec ALU Shuffle					

NOTES:

1. "*" in this table indicates that these features are unavailable for 512-bit vectors.
2. "**" in this table indicates that these features are unavailable for 512-bit vectors in Client parts.
3. The Golden Cove microarchitecture implemented performance improvements requiring constraint of the micro-ops which use *H partial registers (i.e. AH, BH, CH, DH). See Section 3.5.2.3 for more details.

Table 2-4 lists execution units and common representative instructions that rely on these units.

Throughput improvements across the Intel® SSE, Intel AVX, and general-purpose instruction sets are related to the number of units for the respective operations, and the varieties of instructions that execute using a particular unit.

Table 2-4. Golden Cove Microarchitecture Execution Units and Representative Instructions[1]

Execution Unit	# of Unit	Instructions
ALU	5[2]	add, and, cmp, or, test, xor, movzx, movsx, mov, (v)movdqu, (v)movdqa, (v)movap*, (v)movup*
SHFT	2[3]	sal, shl, rol, adc, sarx, adcx, adox, etc.
Slow Int	1	mul, imul, bsr, rcl, shld, mulx, pdep, etc.
BM	2	andn, bextr, blsi, blsmsk, bzhi, etc.
Vec ALU	2x256-bit 1x512-bit	(v)add, (v)cmp. (v)max, (v)min, (v)sub, (v)cvtps2dq, (v)cvtdq2ps, (v)cvtsd2sl, (v)cvtss2sl
	3x256-bit 2x512-bit	(v)pand, (v)por, (v)pxor, (v)movq, (v)movq, (v)movap*, (v)movup*, (v)andp*, (v)orp*, (v)paddb/w/d/q, (v)blendv*, (v)blendp*, (v)pblendd
Vec_Shft	2x256-bit 1x512-bit	(v)psllv*, (v)psrlv*, vector shift count in imm8
VEC Add (in VEC FMA)	2x256-bit 1x512-bit	(v)add*, (v)cmp*, (v)max*, (v)min*, (v)sub*, (v)padds*, (v)paddus*, (v)psign, (v)pabs, (v)pavgb, (v)pcmpeq*, (v)pmax, (v)cvtps2dq, (v)cvtdq2ps, (v)cvtsd2si, (v)cvtss2si
VEC Fast Add	2x256-bit 1x512-bit	(v)add*, (v)addsub*, (v)sub*

Table 2-4. Golden Cove Microarchitecture Execution Units and Representative Instructions[1]

Execution Unit	# of Unit	Instructions
Shuffle	2x256-bit 1x512-bit	(v)shufp*, vperm*, (v)pack*, (v)unpck*, (v)punpck*, (v)pshuf*, (v)pslldq, (v)alignr, (v)pmovzx*, vbroadcast*, (v)pslldq, (v)psrldq, (v)pblendw (new cross lane shuffle on both ports)
Vec Mul/FMA	2x256-bit (1 or 2)x512-bit	(v)mul*, (v)pmul*, (v)pmadd*
SIMD Misc	1	STTNI, (v)pclmulqdq, (v)psadw, vector shift count in xmm
FP Mov	1	(v)movsd/ss, (v)movd gpr
DIVIDE	1	divp*, divs*, vdiv*, sqrt*, vsqrt*, rcp*, vrcp*, rsqrt*, idiv

NOTES:

1. Execution unit mapping to MMX instructions are not covered in this table. See Section 15.16.5 on MMX instruction throughput remedy.
2. The Golden Cove microarchitecture implemented performance improvements requiring constraint of the micro-ops which use *H partial registers (i.e. AH, BH, CH, DH). See Section 3.5.2.3 for more details.
3. Ibid.

Table 2-5 describes bypass delay in cycles between producer and consumer operations.

Table 2-5. Bypass Delay Between Producer and Consumer Micro-Ops

	TO [EU/PORT/Latency]						
FROM [EU/Port/Latency]	SIMD/0,1/1	FMA/0,1/4	MUL/0,1/4	Fast Adder/1,5/3	SIMD/5/1,3	SHUF/1,5/1,3	V2I/0/3
SIMD/0,1/1	0	1	1	1	0	0	0
FMA/0,1/4	1	0	1	0	0	0	0
MUL/0,1/4	1	0	1	0	0	0	0
Fast Adder/0,1/3	1	0	1	-1	0	0	0
SIMD/5/1,3	0	1	1	1	0	0	0
SHUF/1,5/1,3	0	0	1	0	0	0	0
V2I/0/3	0	0	1	0	0	0	0
I2V/5/1	0	1	1	0	0	0	0

The attributes relevant to the producer/consumer micro-ops for bypass are a triplet of abbreviation/one or more port number/latency cycle of the uop. For example:

- "SIMD/0,1/1" applies to a 1-cycle vector SIMD uop dispatched to either port 0 or port 1.
- "SIMD/5/1,3" applies to either a 1-cycle or 3-cycle non-shuffle uop dispatched to port 5.

- “V2I/0/3” applies to a three-cycle vector-to-integer uop dispatched to port 0.
- “I2V/5/1” applies to a one-cycle integer-to-vector uop dispatched to port 5.
- “Fast Adder/1,5/3” applies to either a three-cycle 256-bit uop dispatched to port 1 or port 5, or a 512-bit uop dispatched to port 5. This operation supports two cycles back-to-back between a pair of Fast Adder operations.

A new Fast Adder[1] unit is added as 512-bit on port 5 in the VEC stack and as 256-bit on ports 1 and 5. The Fast Adder performs floating-point ADD/SUB operations in three cycles.

Back-to-back ADD/SUB operations, both executed on the Fast Adder unit, perform the operations in two cycles.

- In 128/256-bit, back-to-back ADD/SUB operations executed on the Fast Adder unit perform the operations in two cycles.
- In 512-bit, back-to-back ADD/SUB operations are executed in two cycles if both operations use the Fast Adder unit on port 5.

The following instructions are executed by the Fast Adder unit:

- (V)ADDSUBSS/SD/PS/PD
- (V)ADDSS/SD/PS/PD
- (V)SUBSS/SD/PS/PD

2.4.1.1 Cache Subsystem and Memory Subsystem

Changes in the cache subsystem and memory subsystem within the Golden Cove microarchitecture include:

- Maximum load bandwidth increased from two to three loads per cycle. Bandwidth of Intel AVX-512 loads, Intel AMX loads, and MMX/x87 loads remain at a maximum of two loads per cycle.
- Simultaneous handling of more loads and stores enabled by enlarged buffers.
- Number of entries for 4K pages in the load DTLB increased from 64 to 96.
- The Page Miss handler can handle up to four D-side page walks in parallel instead of two.
- Increased outstanding DCU and MLC misses.
- Enhanced data prefetchers for increased memory parallelism.
- Partial store forwarding allowing forwarding data from store to load also when only part of the load was covered by the store (in case the load's offset matches the store's offset).

2.4.1.2 Avoiding Destination False Dependency

Some SIMD instructions incur false dependencies on the destination operand. The following instructions are affected:

- VFMULCSH, VFMULCPH
- VFCMULCSH, VFCMULCPH
- VPERMD, VPERMQ, VPERMPS, VPERMPD
- VRANGE[SS,PS,SD,PD]
- VGETMANTSH, VGETMANTSS, VGETMANTSD
- VGETMANTPS, VGETMANTPD (memory versions only)
- VPMULLQ

1. The Fast Adder unit is unavailable on 512-bit vectors in Client parts.

Recommendation: Use dependency breaking zero idioms on the destination register before the affected instructions to avoid potential slowdown from the false dependency.

Example 2-5. Breaking False Dependency through Zero Idiom

Code with False Dependency Impact	Mitigation: Break False Dependency with Zero Idiom

vaddps zmm3, zmm4, zmm5
vmovaps [rsi], zmm3
vfmulcph zmm3, zmm2, zmm1 ;False dependency on zmm3.
//Will not execute out-of-order until vaddps writes zmm3.
	vaddps zmm3, zmm4, zmm5
	vmovaps [rsi], zmm3
	vpxord zmm3, zmm3, zmm3
	//Dependency-breaking zero idiom.
	vfmulcph zmm3, zmm2, zmm1
	//Execute out-of-order
	without waiting for vaddps result.

2.5 ICE LAKE CLIENT MICROARCHITECTURE

The Ice Lake client microarchitecture introduces the following new features that allow optimizations of applications for performance and power consumption:

- Targeted vector acceleration.
- Crypto acceleration.
- Intel® Software Guard Extensions (Intel® SGX) enhancements.
- Cache line writeback instruction (CLWB).

2.5.1 ICE LAKE CLIENT MICROARCHITECTURE OVERVIEW

The Ice Lake client microarchitecture builds on the successes of the Skylake client microarchitecture. The basic pipeline functionality of the Ice Lake Client microarchitecture is depicted in Figure 2-4.

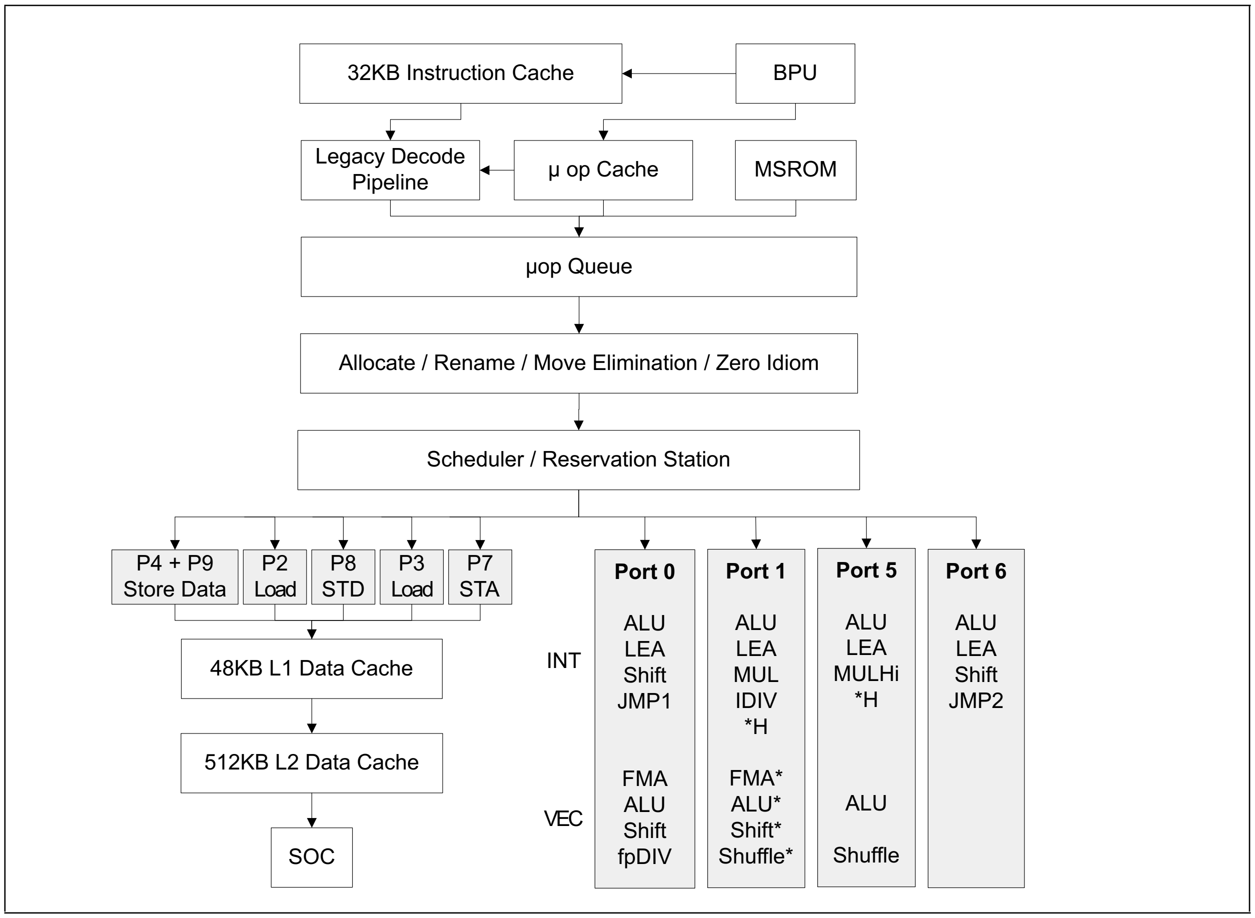


Figure 2-4. Processor Core Pipeline Functionality of the Ice Lake Client Microarchitecture[1]

NOTES:

1. "*" in the figure above indicates these features are unavailable for 512-bit vectors.
2. "INT" represents GPR scalar instructions.
3. "VEC" represents floating-point and integer vector instructions.
4. "MULHi" produces the upper 64 bits of the result of an iMul operation that multiplies two 64-bit registers and places the result into two 64-bits registers.
5. The "Shuffle" on port 1 is new, and supports only in-lane shuffles that operate within the same 128-bit sub-vector.
6. The "IDIV" unit on port 1 is new, and performs integer divide operations at a reduced latency.
7. The Golden Cove microarchitecture implemented performance improvements requiring constraint of the micro-ops which use *H partial registers (i.e. AH, BH, CH, DH). See Section 3.5.2.3 for more details.

The Ice Lake client microarchitecture introduced the following features:

- Significant increase in size of key structures enable deeper OOO execution.
- Wider machine: 4 ⟶ 5 wide allocation, 8 ⟶ 10 execution ports.
- Intel AVX-512 (new for client processors): 512-bit vector operations, 512-bit loads and stores to memory, and 32 new 512-bit registers.
- Greater capabilities per execution port (e.g., SIMD shuffle, LEA), reduced latency Integer Divider.
- 2×BW for AES-NI peak throughput for existing binaries (microarchitectural).

- Rep move string acceleration.
- 50% increase in size of the L1 data cache.
- Reduced effective load latency.
- 2×L1 store bandwidth: 1 $\longrightarrow$ 2 stores per cycle.
- Enhanced data prefetchers for increased memory parallelism.
- Larger 2nd level TLB.
- Larger uop cache.
- Improved branch predictor.
- Large page ITLB size in single thread mode doubled.
- Larger L2 cache.

The Ice Lake client microarchitecture supports flexible integration of multiple processor cores with a shared uncore sub-system consisting of some components including a ring interconnect to multiple slices of L3, processor graphics, integrated memory controller, interconnect fabrics, and more.

2.5.1.1 The Front End

The front end changes in Ice Lake Client microarchitecture included:

- Improved branch predictor.
- Large page ITLB in single thread mode increased from 8 to 16 entries.
- Larger uop cache.
- The IDQ can hold 70 uops per logical processor vs. 64 uops per logical processor in previous generations when two sibling logical processors in the same core are active (2×70 vs. 2×64 per core). If only one logical processor is active in the core, the IDQ can hold 70 uops vs. 64 uops.
- The LSD in the IDQ can detect loops of up to 70 uops per logical processor irrespective single thread or multi thread operation.

2.5.1.2 The Out of Order and Execution Engines

The Out of Order and execution engines changes in Ice Lake client microarchitecture include:

- A significant increase in size of reorder buffer, load buffer, store buffer, and reservation stations enable deeper OOO execution and higher cache bandwidth.
- Wider machine: 4 $\longrightarrow$ 5 wide allocation, 8 $\longrightarrow$ 10 execution ports.
- Greater capabilities per execution port (e.g., SIMD shuffle, LEA).
- Reduced latency Integer Divider.
- A new iDIV unit was added, significantly reducing the latency and improving the throughput of integer divide operations.

Table 2-6 summarizes the OOO engine's capability to dispatch different types of operations to ports.

Table 2-6. Dispatch Port and Execution Stacks of the Ice Lake Client Microarchitecture

Port 0	Port 1[1]	Port 2	Port 3	Port 4	Port 5	Port 6	Port 7	Port 8	Port 9
INT ALU LEA INT Shift Jump1	INT ALU LEA INT Mul INT Div	Load	Load	Store Data	INT ALU LEA INT MUL Hi	INT ALU LEA INT Shift Jump2	Store Address	Store Address	Store Data
FMA Vec ALU Vec Shift FP Div	FMA* Vec ALU* Vec Shift* Vec Shuffle*				Vec ALU Vec Shuffle				

NOTES:

1. “*” in this table indicates these features are unavailable for 512-bit vectors.

Table 2-7 lists execution units and common representative instructions that rely on these units.

Throughput improvements across the Intel SSE, Intel AVX, and general-purpose instruction sets are related to the number of units for the respective operations, and the varieties of instructions that execute using a particular unit.

Table 2-7. Ice Lake Client Microarchitecture Execution Units and Representative Instructions[1]

Execution Unit	# of Unit	Instructions
ALU	4	add, and, cmp, or, test, xor, movzx, movsx, mov, (v)movdqu, (v)movdqa, (v)movap*, (v)movup*
SHFT	2	sal, shl, rol, adc, sarx, adcx, adox, etc.
Slow Int	1	mul, imul, bsr, rcl, shld, mulx, pdep, etc.
BM	2	andn, bextr, blsi, blsmsk, bzhi, etc.
Vec ALU	3	(v)pand, (v)por, (v)pxor, (v)movq, (v)movq, (v)movap*, (v)movup*, (v)andp*, (v)orp*, (v)paddb/w/d/q, (v)blendv*, (v)blendp*, (v)pblendd
Vec_Shft	2	(v)psllv*, (v)psrlv*, vector shift count in imm8
Vec Add	2	(v)addp*, (v)cmpp*, (v)max*, (v)min*, (v)padds*, (v)paddus*, (v)psign, (v)pabs, (v)pavgb, (v)pcmpeq*, (v)pmax, (v)cvtps2dq, (v)cvtdq2ps, (v)cvtsd2si, (v)cvtss2si
Shuffle	2	(v)shufp*, vperm*, (v)pack*, (v)unpck*, (v)punpck*, (v)pshuf*, (v)pslldq, (v)alignr, (v)pmovzx*, vbroadcast*, (v)pslldq, (v)psrldq, (v)pblendw
Vec Mul	2	(v)mul*, (v)pmul*, (v)pmadd*

Table 2-7. Ice Lake Client Microarchitecture Execution Units and Representative Instructions[1]

Execution Unit	# of Unit	Instructions
SIMD Misc	1	STTNI, (v)pclmulqdq, (v)psadw, vector shift count in xmm
FP Mov	1	(v)movsd/ss, (v)movd gpr
DIVIDE	1	divp*, divs*, vdiv*, sqrt*, vsqrt*, rcp*, vrcp*, rsqrt*, idiv

NOTES:

1. Execution unit mapping to MMX instructions are not covered in this table. See Section 15.16.5 on MMX instruction throughput remedy.

Table 2-8 describes bypass delay in cycles between producer and consumer operations.

Table 2-8. Bypass Delay Between Producer and Consumer Micro-ops

	TO [EU/PORT/Latency]						
FROM [EU/Port/Latency]	SIMD/0,1/ 1	FMA/0,1/ 4	VIMUL/0,1/ 4	SIMD/5/1, 3	SHUF/5/1, 3	V2I/0/ 3	I2V/5/1
SIMD/0,1/1	0	1	1	0	0	0	NA
FMA/0,1/4	1	0	1	0	0	0	NA
VIMUL/0,1/4	1	0	1	0	0	0	NA
SIMD/5/1,3	0	1	1	0	0	0	NA
SHUF/5/1,3	0	0	1	0	0	0	NA
V2I/0/3	0	0	1	0	0	0	NA
I2V/5/1	0	1	1	0	0	0	NA

The attributes that are relevant to the producer/consumer micro-ops for bypass are a triplet of abbreviation/one or more port number/latency cycle of the uop. For example:

- “SIMD/0,1/1” applies to 1-cycle vector SIMD uop dispatched to either port 0 or port 1.
- “SIMD/5/1,3” applies to either a 1-cycle or 3-cycle non-shuffle uop dispatched to port 5.
- “V2I/0/3” applies to a 3-cycle vector-to-integer uop dispatched to port 0.
- “I2V/5/1” applies to a 1-cycle integer-to-vector uop to port 5.

2.5.1.3 Cache and Memory Subsystem

The cache hierarchy changes in Ice Lake Client microarchitecture include:

- 50% increase in size of the L1 data cache.
- 2×L1 store bandwidth: 3 $\longrightarrow$ 4 AGUs, 1 $\longrightarrow$ 2 store data.
- Simultaneous handling of more loads and stores enabled by enlarged buffers.
- Higher cache bandwidth compared to previous generations.

- Larger 2nd level TLB: 1.5K entries → 2K entries.
- Enhanced data prefetchers for increased memory parallelism.
- L2 cache size increased from 256KB to 512KB.
- L2 cache associativity increased from 4 ways to 8 ways.
- Significant reduction in effective load latency.

Table 2-9. Cache Parameters of the Ice Lake Client Microarchitecture

Level	Capacity / Associativity	Line Size (bytes)	Latency [1] (cycles)	Peak Bandwidth (bytes/cycles)	Sustained Bandwidth (bytes/cycles)	Update Policy
First Level (DCU)	48KB/8	64	5	2×64B loads + 1x64B or 2x32B stores	Same as peak	Writeback
Second Level (MLC)	512KB/8	64	13	64	48	Writeback
Third Level (LLC)	Up to 2MB per core/up to 16 ways	64	xx[2]	32	21	Writeback

NOTES:
1. Software-visible latency/bandwidth will vary depending on access patterns and other factors.
2. This number depends on core count.

The TLB hierarchy consists of dedicated level one TLB for instruction cache, TLB for L1D, shared L2 TLB for 4K and 4MB pages and a dedicated L2 TLB for 1GB pages.

Table 2-10. TLB Parameters of the Ice Lake Client Microarchitecture

Level	Page Size	Entries ST	Per-thread Entries MT Latency	Associativity
Instruction	4KB	128	64	8
Instruction	2MB/4MB	16	8	8
First Level Data (loads)	4KB	64	64 competitively shared	4
First Level Data (loads)	2MB/4MB	32	32 competitively shared	4
First Level Data (loads)	1GB	8	8 competitively shared	8

Table 2-10. TLB Parameters of the Ice Lake Client Microarchitecture (Contd.)

Level	Page Size	Entries ST	Per-thread Entries MT Latency	Associativity
First Level Data (stores)	Shared for all page sizes	16	16 competitively shared	16
Second Level	Shared for all page sizes	2048[1]	2048 competitively shared	16

NOTES:

1. 4K pages can use all 2048 entries. 2/4MB pages can use 1024 entries (in 8 ways), sharing them with 4K pages. 1GB pages can use the other 1024 entries (in 8 ways), also sharing them with 4K pages.

Paired Stores

Ice Lake Client microarchitecture includes two store pipelines in the core, with the following features:

- Two dedicated AGU for LDs on ports 2 and 3.
- Two dedicated AGU for STAs on ports 7 and 8.
- Two fully featured STA pipelines.
- Two 256-bit wide STD pipelines (Intel AVX-512 store data takes two cycles to write).
- Second senior store pipeline to the DCU via store merging.

Ice Lake Client microarchitecture can write two senior stores to the cache in a single cycle if these two stores can be paired together. That is:

- The stores must be to the same cache line.
- Both stores are of the same memory type, WB or USWC.
- None of the stores cross cache line or page boundary.

To maximize performance from the second store port try to:

- Align store operations whenever possible.
- Place consecutive stores in the same cache line (not necessarily as adjacent instructions).

As seen in Example 2-6, it is important to take into consideration all stores, explicit or not.

Example 2-6. Considering Stores

Stores are Paired Across Loop Iterations	Stores Not Paired Due to Stack Update in Between

Loop:
    compute reg
    …
    store [X], reg
    add X, 4
    jmp Loop ; stores from different iterations of the loop can be paired all together because they usually would be same line
	Loop:
	    call function to compute reg
	    …
	    store [X], reg
	    add X, 4
	    jmp Loop ; stores from different iterations of the loop cannot be paired anymore because of the call store to stack
	; the call is disturbing pairing

In some cases it is possible to rearrange the code to achieve store pairing. Example 2-7 provides details.

Example 2-7. Rearranging Code to Achieve Store Pairing

Stores to Different Cache Lines - Not Paired	Unrolling May Solve the Problem
Loop:	Loop:
... compute ymm1 ...	... compute ymm1 ...
vmovaps [x], ymm1	vmovaps [x], ymm1
... compute ymm2 ...	... compute new ymm1 ...
vmovaps [y], ymm2	vmovaps [x+32], ymm1
add x, 32	... compute ymm2 ...
add y, 32	vmovaps [y], ymm2
jmp Loop	... compute new ymm2 ...
	vmovaps [y+32], ymm2
//this loop cannot pair any store because of alternating store to different cache lines [x] and [y]	add x, 64
	add y, 64
	jmp Loop
	//the loop was unrolled 2 times and stores rearranged to make sure two stores to the same cache line are placed one after another. Now stores to addresses [x] and [x+32] are to the same cache line and could be paired together and executed in same cycle.

2.5.1.4 Fast Store Forwarding Prediction (FSFP)

This section includes recommendations for effective use of Fast Store Forwarding Prediction (FSFP) introduced in Ice Lake microarchitecture. Extrapolated from previous behavior, FSFP enables the processor to predict that a store will forward data to a younger load and optimize that case. The optimization allows the load to complete using the data of predicted store but without accessing the memory. Only integer loads support FSFP in the Ice Lake microarchitecture.

The Fast Store Forwarding Prediction has limitations. to maximize performance gain on Ice Lake microarchitecture it is recommended to follow these recommendations:

- Only loads and stores without Index (encoded with no SIB byte) are supported. LEA operation can be used to avoid Index register usage during memory address computations.
- Loads and stores using RIP-relative addressing do not support FSFP. We recommend using the LEA operation to pre-compute address to enable FSFP for such cases.
- Loads and stores operating with 16-bit General Purpose Registers (AX/BX/CX/DX and etc) or *H 8-bit registers do not support FSFP optimization. We recommend using **movzx** instruction instead of unsupported registers.

Example 2-8. FSFP Optimization

Slow Version Not Enabling PFSP	Enabling FSFP Using LEA Operation

Loop:
    mov r10,[rsi+r8*8]
    inc qword[rdi+r10*8]
    mov r11,[rsi+r8*8]
    inc r8
    inc qword[rdi+r11*8]
… jmp Loop
	Loop:
	    mov r10,[rsi+r8*8]
	    lea r12,[rdi+r10*8] ; using LEA to avoid
	    ;Index register
	for ;inc below
	    inc qword[r12]
	    mov r11,[rsi+r8*8]
	    inc r8
	    lea r13,[rdi+r11*8] ; another similar case
	    inc qword [r13]
	…. jmp Loop

2.5.1.5 New Instructions

New instructions and architectural changes in Ice Lake Client microarchitecture are listed below. Actual support may be product dependent.

- Crypto acceleration
  - SHA NI for acceleration of SHA1 and SHA256 hash algorithms.
  - Big-Number Arithmetic (IFMA): VPMADD52 - two new instructions for big number multiplication for acceleration of RSA vectorized SW and other Crypto algorithms (Public key) performance.
  - Galois Field New Instructions (GFNI) for acceleration of various encryption algorithms, error correction algorithms, and bit matrix multiplications.
  - Vector AES and Vector Carry-less Multiply (PCLMULQDQ) instructions to accelerate AES and AES-GCM.
- Security Technologies
  - Intel® SGX enhancements to improve usability and applicability: EDMM, multi-package server support, support for VMM memory oversubscription, performance, larger secure memory.
- Sub Page protection for better performance of security VMMs.
- Targeted Acceleration
  - Vector Bit Manipulation Instructions: VBMI1 (permutes, shifts) and VBMI2 (Expand, Compress, Shifts)- used for columnar database access, dictionary based decompression, discrete mathematics, and data-mining routines (bit permutation and bit-matrix-multiplication).
  - VNNI with support for integer 8 and 16 bits data types- CNN/ML/DL acceleration.
  - Bit Algebra (POPCNT, Bit Shuffle).
  - Cache line writeback instruction (CLWB) enables fast cache-line update to memory, while retaining clean copy in cache.
- Platform analysis features for more efficient performance software tuning and debug.
  - AnyThread removal.
  - 2x general counters (up to 8 per-thread).
  - Fixed Counter 3 for issue slots.

— New performance metrics for built-in support for Level 1 Top-Down method (% of Issue slots that are front-end bound, back-end bound, bad speculation, retiring) while leaving the 8 general purpose counters free for software use.

2.5.1.6 Ice Lake Client Microarchitecture Power Management

Processors based on Ice Lake client microarchitecture are the first client processors whose cores may execute at a different frequency from one another. The frequency is selected based on the specific instruction mix; the type, width and number of vector instructions of the program that executes on each core, the ratio between active time and idle time of each core, and other considerations such as how many cores share similar characteristics.

Most of the power management features of Skylake Server Microarchitecture (see Section 2.6) is applicable to Ice Lake Client microarchitecture as well. The main differences are the following:

- The typical P0n max frequency difference between Intel® Advanced Vector Extensions (Intel® AVX-512) and Intel® Advanced Vector Extensions 2 (Intel® AVX2) on Ice Lake Client microarchitecture is much lower than on Skylake Server microarchitecture. Therefore, the negative impact on overall application performance is much smaller.
- All processors based on Ice Lake Client microarchitecture contain a single 512-bit FMA unit, whereas some of the processors based on Skylake Server microarchitecture contain two such units. Both processors contain two 256-bit FMA units. The power consumed by Ice Lake Client FMA units is the same, whereas on Skylake Server the 512-bit units consume twice as much.

Compute heavy workloads, especially those that span multiple Ice Lake client cores, execute at a lower frequency than P0n, under Intel AVX-512 and under Intel AVX2 instruction sets, due to power limitations. In this scenario, Intel AVX-512 architecture, which requires less dynamic instructions to complete the same task than Intel AVX2 architecture, consumes less power and thus may achieve higher frequency. The net result may be higher performance due to the shorter path length and a bit higher frequency.

There are still some cases where coding to the Intel AVX-512 instruction set yields lower performance than when coding to the Intel AVX2 instruction set. Sometimes it is due to microarchitecture artifacts of longer vectors, in other cases the natural vectors are just not long enough. Most compilers are still maturing their Intel AVX-512 support, and it may take them a few more years to generate optimal code.

The general recommendation in the Skylake Server Power Management section (see Section 2.6.3) still holds. Developers should code to the Intel AVX-512 instruction set and compare the performance to their Intel AVX2 workload on Ice Lake client microarchitecture, before making the decision to proceed with a complete port.

2.6 SKYLAKE SERVER MICROARCHITECTURE

The Intel® Xeon® Processor scalable processors based on the Skylake microarchitecture can be identified using CPUID's DisplayFamily_DisplayModel signature, which can be found in Table 2-1 of CHAPTER 2 of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 4.

The Skylake Server microarchitecture introduces the following new features[1] that allow you to optimize your application for performance and power consumption.

- A new core based on the Skylake Server microarchitecture with process improvements based on the Kaby Lake microarchitecture.
- Intel AVX-512 support.
- More cores per socket (max 28 vs. max 22).
- 6 memory channels per socket in Skylake microarchitecture vs. 4 in the Broadwell microarchitecture.

1. Some features may not be available on all products.

- Bigger L2 cache, smaller non inclusive L3 cache.
- Intel® Optane™ support.
- Intel® Omni-Path Architecture (Intel® OPA).
- Sub-NUMA Clustering (SNC) support.

The gray rectangles in Figure 2-5 represent features different in Skylake Server microarchitecture compared to Skylake microarchitecture for client;

- A 1MB L2 cache.
- An additional Intel AVX-512 FMA unit on port 5 which is available on some parts.

In this figure:

- INTEGER represents GPR scalar instructions.
- VEC represents floating point and integer vector instructions.
- SLOW LEA represents a lea with two registers and displacement, all other lea versions considered as FAST LEA.
- BRANCH1 is the primary branch and more capable than BRANCH2.

Since port 0 and port 1 are 256-bits wide, Intel AVX-512 operations that will be dispatched to port 0 will execute on port 0 and port 1; however, other operations such as **LEA** can still execute on port 1 in parallel. See the red block in Figure 2-9 for the fusion of ports 0 and 1.

Notice that, unlike Skylake microarchitecture for client, the Skylake Server microarchitecture has its front end loop stream detector (LSD) disabled.

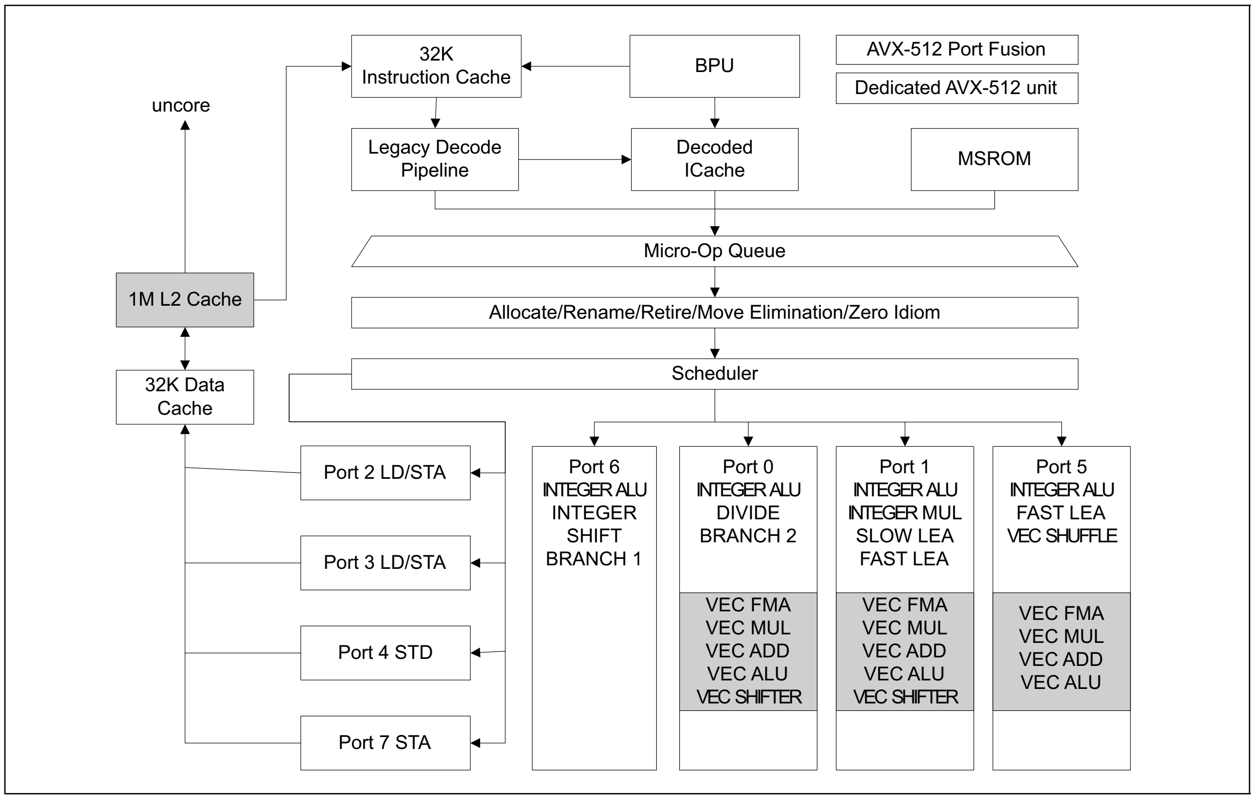

32K Instruction Cache
BPU
AVX-512 Port Fusion
Dedicated AVX-512 unit
uncore
Legacy Decode Pipeline
Decoded ICache
MSROM
Micro-Op Queue
1M L2 Cache
Allocate/Rename/Retire/Move Elimination/Zero Idiom
32K Data Cache
Scheduler
Port 2 LD/STA
Port 3 LD/STA
Port 4 STD
Port 7 STA
Port 6 INTEGER ALU INTEGER SHIFT BRANCH 1
Port 0 INTEGER ALU DIVIDE BRANCH 2 VEC FMA VEC MUL VEC ADD VEC ALU VEC SHIFTER
Port 1 INTEGER ALU INTEGER MUL SLOW LEA FAST LEA VEC FMA VEC MUL VEC ADD VEC ALU VEC SHIFTER
Port 5 INTEGER ALU FAST LEA VEC SHUFFLE VEC FMA VEC MUL VEC ADD VEC ALU

Figure 2-5. Processor Core Pipeline Functionality of the Skylake Server Microarchitecture

2.6.1 SKYLAKE SERVER MICROARCHITECTURE CACHE

Intel Xeon scalable processors based on Skylake server microarchitecture has significant changes in core and uncore architecture to improve performance and scalability of several components compared with the previous generation of the Intel Xeon processors based on the Broadwell microarchitecture.

2.6.1.1 Larger Mid-Level Cache

Skylake server microarchitecture implements a mid-level (L2) cache of 1 MB capacity with a minimum load-to-use latency of 14 cycles. The mid-level cache capacity is four times larger than the capacity in previous Intel Xeon processor family implementations. The line size of the mid-level cache is 64B and it is 16-way associative. The mid-level cache is private to each core.

Software that has been optimized to place data in mid-level cache may have to be revised to take advantage of the larger mid-level cache available in Skylake server microarchitecture.

2.6.1.2 Non-Inclusive Last Level Cache

The last level cache (LLC) in Skylake is a non-inclusive, distributed, shared cache. The size of each of the banks of last level cache has shrunk to 1.375 MBs per bank. Because of the non-inclusive nature of the last level cache, blocks that are present in the mid-level cache of one of the cores may not have a copy resident in a bank of last level cache. Based on the access pattern, size of the code and data accessed, and sharing behavior between cores for a cache block, the last level cache may appear as a victim cache of the mid-level cache and the aggregate cache capacity per core may appear to be a combination of the private mid-level cache per core and a portion of the last level cache.

2.6.1.3 Skylake Server Microarchitecture Cache Recommendations

A high-level comparison between Skylake server microarchitecture cache and the previous generation Broadwell microarchitecture cache is available in the table below.

Table 2-11. Cache Comparison Between Skylake Microarchitecture and Broadwell Microarchitecture

Cache level	Category	Broadwell Microarchitecture	Skylake Server Microarchitecture
L1 Data Cache Unit (DCU)	Size [KB]	32	32
	Latency [cycles]	4-6	4-6
	Max bandwidth [bytes/cycles]	96	192
	Sustained bandwidth [bytes/cycles]	93	133
	Associativity [ways]	8	8

Table 2-11. Cache Comparison Between Skylake Microarchitecture and Broadwell Microarchitecture (Contd.)

Cache level	Category	Broadwell Microarchitecture	Skylake Server Microarchitecture
L2 Mid-level Cache (MLC)	Size [KB]	256	1024 (1MB)
	Latency [cycles]	12	14
	Max bandwidth [bytes/cycles]	32	64
	Sustained bandwidth [bytes/cycles]	25	52
	Associativity [ways]	8	16
L3 Last-level Cache (LLC)	Size [MB]	Up to 2.5 per core	up to 1.375[1] per core
	Latency [cycles]	50-60	50-70
	Max bandwidth [bytes/cycles]	16	32
	Sustained bandwidth [bytes/cycles]	14	15

NOTES:

1. Some Skylake server parts have some cores disabled and hence have more than 1.375 MBs per core of L3 cache.

The figure below shows how Skylake server microarchitecture shifts the memory balance from shared-distributed with high latency, to private-local with low latency.

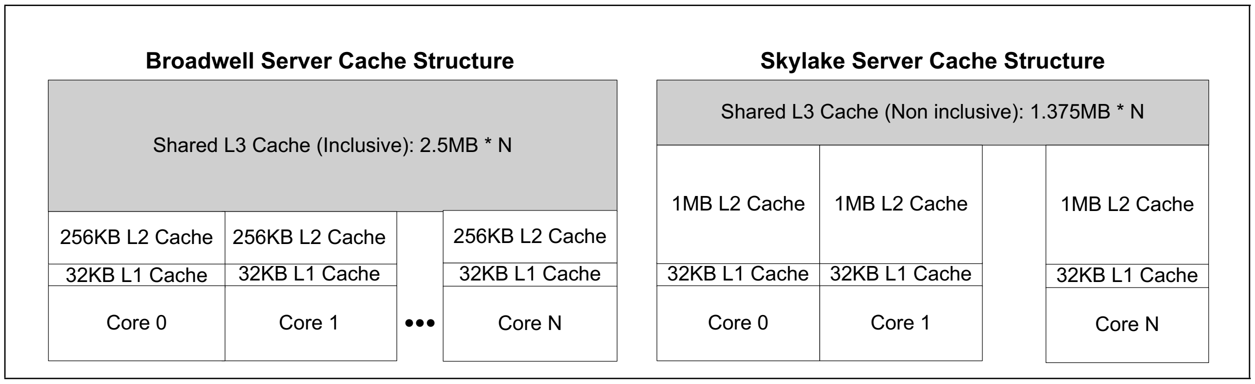


Figure 2-6. Broadwell Microarchitecture and Skylake Server Microarchitecture Cache Structures

The potential performance benefit from the cache changes is high, but software will need to adapt its memory tiling strategy to be optimal for the new cache sizes.

Recommendation: Rebalance application shared and private data sizes to match the smaller, non-inclusive L3 cache, and larger L2 cache.

Choice of cache blocking should be based on application bandwidth requirements and changes from one application to another. Having four times the L2 cache size and twice the L2 cache bandwidth compared to the previous generation Broadwell microarchitecture enables some applications to block to L2 instead of L1 and thereby improves performance.

Recommendation: Consider blocking to L2 on Skylake Server microarchitecture if L2 can sustain the application’s bandwidth requirements.

The change from inclusive last level cache to non-inclusive means that the capacity of mid-level and last level cache can now be added together. Programs that determine cache capacity per core at run time should now use a combination of mid-level cache size and last level cache size per core to estimate the effective cache size per core. Using just the last level cache size per core may result in non-optimal use of available on-chip cache; see Section 2.6.2 for details.

Recommendation: In case of no data sharing, applications should consider cache capacity per core as L2 and L3 cache sizes and not only L3 cache size.

2.6.2 NON-TEMPORAL STORES ON SKYLAKE SERVER MICROARCHITECTURE

Because of the change in the size of each bank of last level cache on Skylake server microarchitecture, if an application, library, or driver only considers the last level cache to determine the size of on-chip cache-per-core, it may see a reduction with Skylake server microarchitecture and may use non-temporal store with smaller blocks of memory writes. Since non-temporal stores evict cache lines back to memory, this may result in an increase in subsequent cache misses and memory bandwidth demands on Skylake Server microarchitecture, compared to the previous Intel Xeon processor family.

Also, because of a change in the handling of accesses resulting from non-temporal stores by Skylake Server microarchitecture, the resources within each core remain busy for a longer duration compared to similar accesses on the previous Intel Xeon processor family. As a result, if a series of such instructions are executed, there is a potential that the processor may run out of resources and stall, thus limiting the memory write bandwidth from each core.

The increase in cache misses due to overuse of non-temporal stores and the limit on the memory write bandwidth per core for non-temporal stores may result in reduced performance for some applications.

To avoid the performance condition described above with Skylake server microarchitecture, include mid-level cache capacity per core in addition to the last level cache per core for applications, libraries, or drivers that determine the on-chip cache available with each core. Doing so optimizes the available on-chip cache capacity on Skylake server microarchitecture as intended, with its non-inclusive last level cache implementation.

2.6.3 SKYLAKE SERVER POWER MANAGEMENT

This section describes the interaction of Skylake Server's Power Management and its Vector ISA.

Skylake Server microarchitecture dynamically selects the frequency at which each of its cores executes. The selected frequency depends on the instruction mix; the type, width, and number of vector instructions that execute over a given period of time. The processor also takes into account the number of cores that share similar characteristics.

Intel® Xeon® processors based on Broadwell microarchitecture work similarly, but to a lesser extent since they only support 256-bit vector instructions. Skylake Server microarchitecture supports Intel® AVX-512 instructions, which can potentially draw more current and more power than Intel® AVX2 instructions.

The processor dynamically adjusts its maximum frequency to higher or lower levels as necessary, therefore a program might be limited to different maximum frequencies during its execution.

Table 2-12 includes information about the maximum Intel® Turbo Boost technology core frequency for each type of instruction executed. The maximum frequency (P0n) is an array of frequencies which depend on the number of cores within the category. The more cores belonging to a category at any given time, the lower the maximum frequency.

Table 2-12. Maximum Intel® Turbo Boost Technology Core Frequency Levels

Level	Category	Frequency Level	Max Frequency (P0n)	Instruction Types
0	Intel® AVX2 light instructions	Highest	Max	Scalar, AVX128, SSE, Intel® AVX2 w/o FP or INT MUL/FMA
1	Intel® AVX2 heavy instructions + Intel® AVX-512 light instructions	Medium	Max Intel® AVX2	Intel® AVX2 FP + INT MUL/FMA, Intel® AVX-512 without FP or INT MUL/FMA
2	Intel® AVX-512 heavy instructions	Lowest	Max Intel® AVX-512	Intel® AVX-512 FP + INT MUL/FMA

For per SKU max frequency details (reference figure 1-15), refer to the Intel® Xeon® Scalable Processor Family Technical Resources page.

Figure 2-7 is an example for core frequency range in a given system where each core frequency is determined independently based on the demand of the workload.

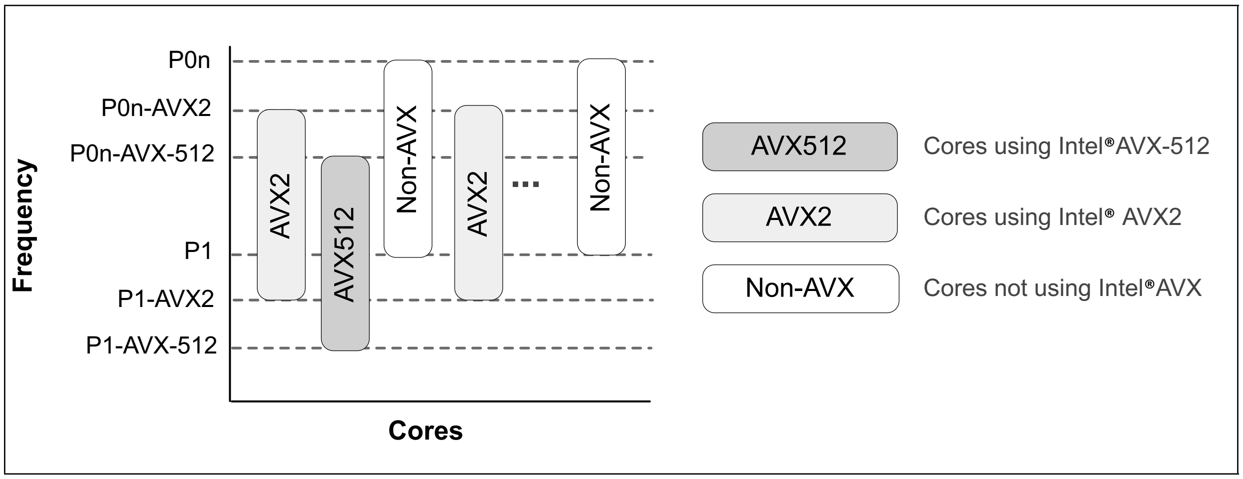


Figure 2-7. Mixed Workloads

The following performance monitoring events can be used to determine how many cycles were spent in each of the three frequency levels.

- CORE_POWER.LVL0_TURBO_LICENSE: Core cycles where the core was running in a manner where the maximum frequency was P0n.
- CORE_POWER.LVL1_TURBO_LICENSE: Core cycles where the core was running in a manner where the maximum frequency was P0n-AVX2.
- CORE_POWER.LVL2_TURBO_LICENSE: Core cycles where the core was running in a manner where the maximum frequency was P0n-AVX-512.

When the core requests a higher license level than its current one, it takes the PCU up to 500 micro-seconds to grant the new license. Until then the core operates at a lower peak capability. During this time period the PCU evaluates how many cores are executing at the new license level and adjusts their frequency as necessary, potentially lowering the frequency. Cores that execute at other license levels are not affected.

A timer of approximately 2ms is applied before going back to a higher frequency level. Any condition that would have requested a new license resets the timer.

NOTES

A license transition request may occur when executing instructions on a mis-speculated path.

A large enough mix of Intel AVX-512 light instructions and Intel AVX2 heavy instructions drives the core to request License 2, despite the fact that they usually map to License 1. The same is true for Intel AVX2 light instructions and Intel SSE heavy instructions that may drive the core to License 1 rather than License 0. For example, The Intel® Xeon® Platinum 8180 processor moves from license 1 to license 2 when executing a mix of 110 Intel AVX-512 light instructions and 20 256-bit heavy instructions over a window of 65 cycles.

Some workloads do not cause the processor to reach its maximum frequency as these workloads are bound by other factors. For example, the LINPACK benchmark is power limited and does not reach the processor's maximum frequency. The following graph shows how frequency degrades as vector width grows, but, despite the frequency drop, performance improves. The data for this graph was collected on an Intel Xeon Platinum 8180 processor.

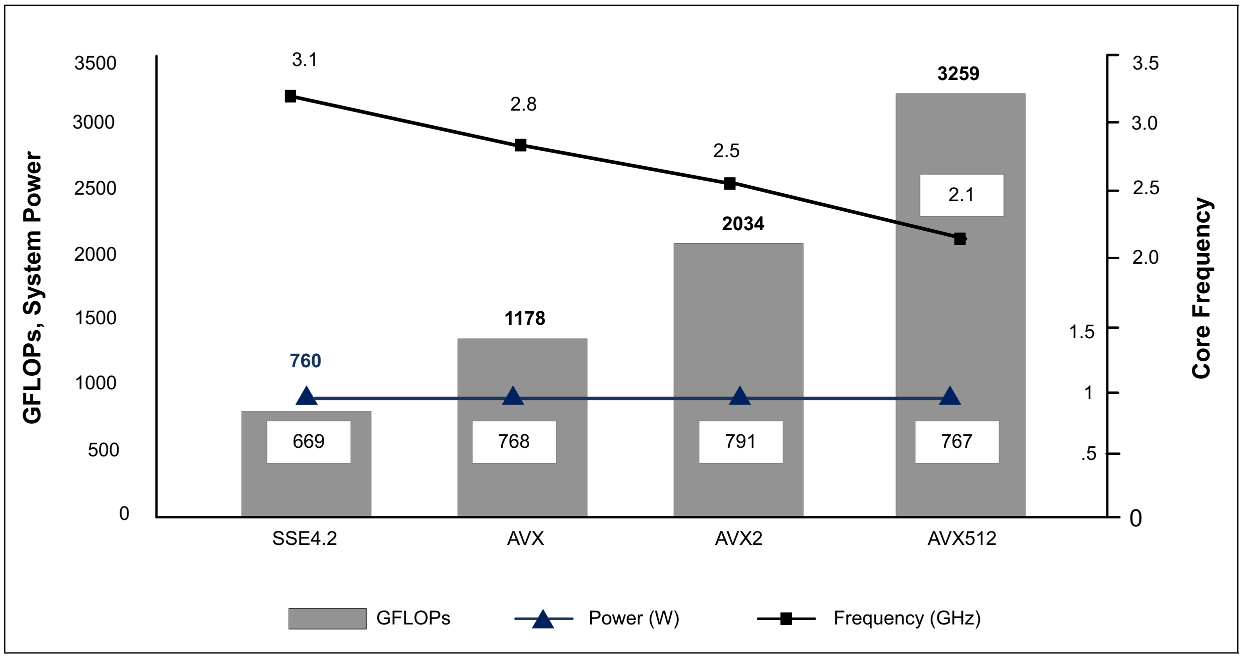


Figure 2-8. LINPACK Performance

Workloads that execute Intel AVX-512 instructions as a large proportion of their whole instruction count can gain performance compared to Intel AVX2 instructions, even though they may operate at a lower frequency. For example, maximum frequency bound Deep Learning workloads that target Intel AVX-512 heavy instructions at a very high percentage can gain 1.3x-1.5x performance improvement vs. the same workload built to target Intel AVX2 (both operating on Skylake Server microarchitecture).

It is not always easy to predict whether a program's performance will improve from building it to target Intel AVX-512 instructions. Programs that enjoy high performance gains from the use of xmm or ymm registers may expect performance improvement by moving to the use of zmm registers. However, some programs that use zmm registers may not gain as much, or may even lose performance. It is recommended to try multiple build options and measure the performance of the program.

Recommendation: To identify the optimal compiler options to use, build the application with each of the following set of options and choose the set that provides the best performance.

- -xCORE-AVX2 **-mtune=skylake-avx512** (Linux* and macOS*)
  /QxCORE-AVX2 **/tune=skylake-avx512** (Windows*)
- -xCORE-AVX512 **-qopt-zmm-usage=low** (Linux* and macOS*)
  /QxCORE-AVX512 **/Qopt-zmm-usage:low** (Windows*)
- -xCORE-AVX512 **-qopt-zmm-usage=high** (Linux* and macOS*)
  /QxCORE-AVX512 **/Qopt-zmm-usage:high** (Windows*)

See Section 17.26 for more information about these options.

The GCC Compiler has the option -mprefer-vector-width=none|128|256|512 to control vector width preference. While -march=skylake-avx512 is designed to provide the best performance for the Skylake Server microarchitecture some programs can benefit from different vector width preferences. To identify the optimal compiler options to use, build the application with each of the following set of options and choose the set that provides the best performance. -mprefer-vector-width=256 is the default for skylake-avx512.

- -march=skylake -mtune=skylake-avx512
- -march=skylake-avx512
- -march=skylake-avx512 -mprefer-vector-width=512

Clang/LLVM is currently implementing the option -mprefer-vector-width=none|128|256|512, similar to GCC. To identify the optimal compiler options to use, build the application with each of the following set of options and choose the set that provides the best performance.

- -march=skylake -mtune=skylake-avx512
- -march=skylake-avx512 (plus -mprefer-vector-width=256, if available)
- -march=skylake-avx512 (plus -mprefer-vector-width=512, if available)

2.7 SKYLAKE CLIENT MICROARCHITECTURE

The Skylake client microarchitecture builds on the successes of the Haswell and Broadwell microarchitectures. The basic pipeline functionality of the Skylake client microarchitecture is depicted in Figure 2-9.

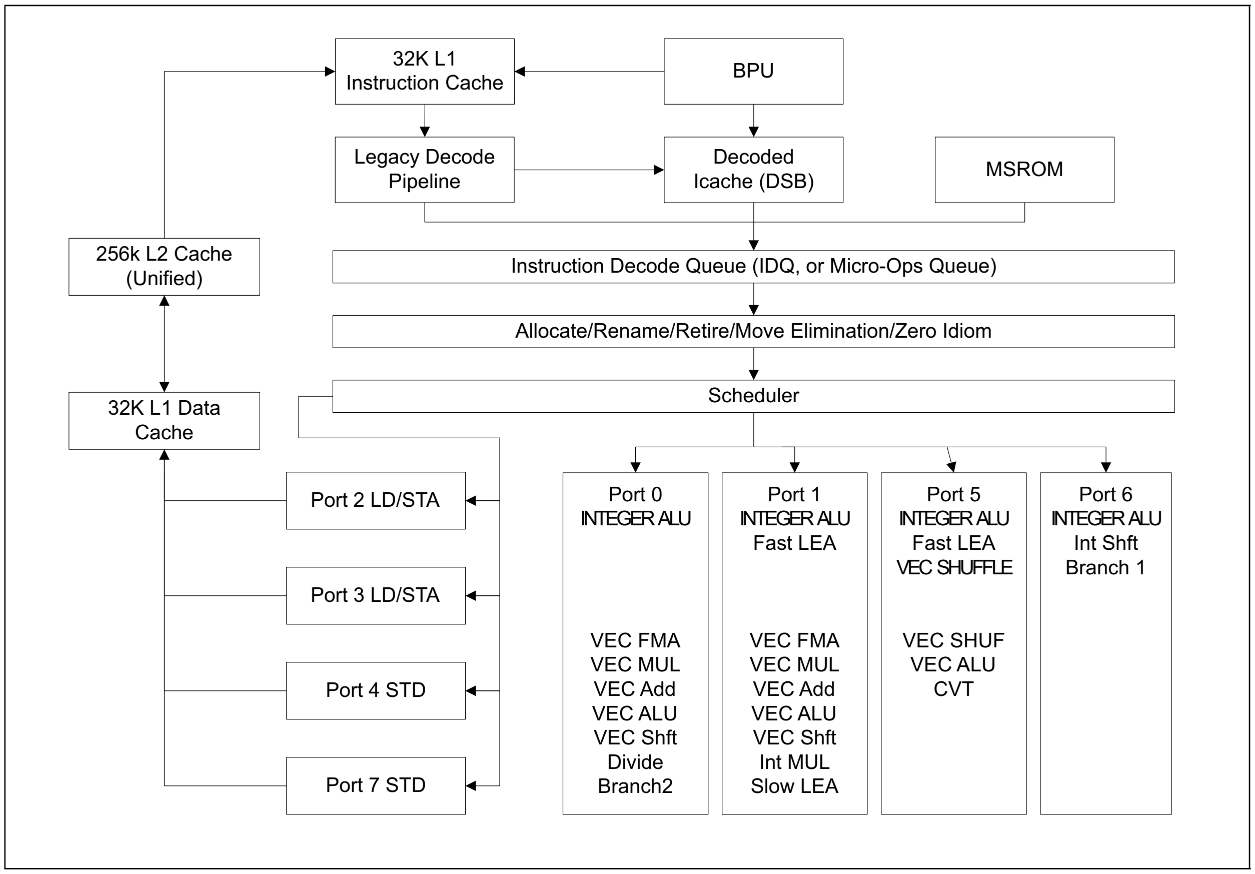

32K L1 Instruction Cache
BPU
Legacy Decode Pipeline
Decoded Icache (DSB)
MSROM
256k L2 Cache (Unified)
Instruction Decode Queue (IDQ, or Micro-Ops Queue)
Allocate/Rename/Retire/Move Elimination/Zero Idiom
Scheduler
32K L1 Data Cache
Port 2 LD/STA
Port 3 LD/STA
Port 4 STD
Port 7 STD
Port 0 INTEGER ALU VEC FMA VEC MUL VEC Add VEC ALU VEC Shft Divide Branch2
Port 1 INTEGER ALU Fast LEA VEC FMA VEC MUL VEC Add VEC ALU VEC Shft Int MUL Slow LEA
Port 5 INTEGER ALU Fast LEA VEC SHUFFLE VEC SHUF VEC ALU CVT
Port 6 INTEGER ALU Int Shft Branch 1

Figure 2-9. CPU Core Pipeline Functionality of the Skylake Client Microarchitecture

The Skylake Client microarchitecture offers the following enhancements:

- Larger internal buffers to enable deeper OOO execution and higher cache bandwidth.
- Improved front end throughput.
- Improved branch predictor.
- Improved divider throughput and latency.
- Lower power consumption.
- Improved SMT performance with Hyper-Threading Technology.
- Balanced floating-point ADD, MUL, FMA throughput and latency.

The microarchitecture supports flexible integration of multiple processor cores with a shared uncore sub-system consisting of a number of components including a ring interconnect to multiple slices of L3 cache (an off-die L4 is optional), processor graphics, integrated memory controller, interconnect fabrics, etc.

2.7.1 THE FRONT END

The front end in the Skylake Client microarchitecture provides the following improvements over previous generation microarchitectures:

- Legacy Decode Pipeline delivery of 5 uops per cycle to the IDQ compared to 4 uops in previous generations.
- The DSB delivers 6 uops per cycle to the IDQ compared to 4 uops in previous generations.
- The IDQ can hold 64 uops per logical processor vs. 28 uops per logical processor in previous generations when two sibling logical processors in the same core are active (2x64 vs. 2x28 per core). If only one logical processor is active in the core, the IDQ can hold 64 uops (64 vs. 56 uops in ST operation).
- The LSD in the IDQ can detect loops up to 64 uops per logical processor irrespective ST or SMT operation.
- Improved Branch Predictor.

2.7.2 THE OUT-OF-ORDER EXECUTION ENGINE

The Out of Order and execution engine changes in Skylake Client microarchitecture include:

- Larger buffers enable deeper OOO execution compared to previous generations.
- Improved throughput and latency for divide/sqrt and approximate reciprocals.
- Identical latency and throughput for all operations running on FMA units.
- Longer pause latency enables better power efficiency and better SMT performance resource utilization.

Table 2-13 summarizes the OOO engine's capability to dispatch different types of operations to various ports.

Table 2-13. Dispatch Port and Execution Stacks of the Skylake Client Microarchitecture

Port 0	Port 1	Port 2, 3	Port 4	Port 5	Port 6	Port 7
ALU, Vec ALU	ALU, Fast LEA, Vec ALU	LD STA	STD	ALU, Fast LEA, Vec ALU,	ALU, Shft,	STA
Vec Shft, Vec Add,	Vec Shft, Vec Add,			Vec Shuffle,	Branch1	
Vec Mul, FMA,	Vec Mul, FMA					
DIV,	Slow Int					
Branch2	Slow LEA					

Table 2-14 lists execution units and common representative instructions that rely on these units. Throughput improvements across the SSE, AVX and general-purpose instruction sets are related to the number of units for the respective operations, and the varieties of instructions that execute using a particular unit.

Table 2-14. Skylake Client Microarchitecture Execution Units and Representative Instructions[1]

Execution Unit	# of Unit	Instructions
ALU	4	add, and, cmp, or, test, xor, movzx, movsx, mov, (v)movdqu, (v)movdqa, (v)movap*, (v)movup*

Table 2-14. Skylake Client Microarchitecture Execution Units and Representative Instructions[1]

Execution Unit	# of Unit	Instructions
SHFT	2	sal, shl, rol, adc, sarx, adcx, adox, etc.
Slow Int	1	mul, imul, bsr, rcl, shld, mulx, pdep, etc.
BM	2	andn, bextr, blsi, blsmsk, bzhi, etc
Vec ALU	3	(v)pand, (v)por, (v)pxor, (v)movq, (v)movq, (v)movap*, (v)movup*, (v)andp*, (v)orp*, (v)paddb/w/d/q, (v)blendv*, (v)blendp*, (v)pblendd
Vec_Shft	2	(v)psllv*, (v)psrlv*, vector shift count in imm8
Vec Add	2	(v)addp*, (v)cmpp*, (v)max*, (v)min*, (v)padds*, (v)paddus*, (v)psign, (v)pabs, (v)pavgb, (v)pcmpeq*, (v)pmax, (v)cvtps2dq, (v)cvtdq2ps, (v)cvtsd2si, (v)cvtss2si
Shuffle	1	(v)shufp*, vperm*, (v)pack*, (v)unpck*, (v)punpck*, (v)pshuf*, (v)pslldq, (v)alignr, (v)pmovzx*, vbroadcast*, (v)pslldq, (v)psrldq, (v)pblendw
Vec Mul	2	(v)mul*, (v)pmul*, (v)pmadd*,
SIMD Misc	1	STTNI, (v)pclmulqdq, (v)psadw, vector shift count in xmm,
FP Mov	1	(v)movsd/ss, (v)movd gpr,
DIVIDE	1	divp*, divs*, vdiv*, sqrt*, vsqrt*, rcp*, vrcp*, rsqrt*, idiv

NOTES:

1. Execution unit mapping to MMX instructions are not covered in this table. See Section 15.16.5 on MMX instruction throughput remedy.

A significant portion of the Intel SSE, Intel AVX and general-purpose instructions also have latency improvements. Appendix C lists the specific details. Software-visible latency exposure of an instruction sometimes may include additional contributions that depend on the relationship between micro-ops flows of the producer instruction and the micro-op flows of the ensuing consumer instruction. For example, a two-uop instruction like VPMULLD may experience two cumulative bypass delays of 1 cycle each from each of the two micro-ops of VPMULLD.

Table 2-15 describes the bypass delay in cycles between a producer uop and the consumer uop. The left-most column lists a variety of situations characteristic of the producer micro-op. The top row lists a variety of situations characteristic of the consumer micro-op.

Table 2-15. Bypass Delay Between Producer and Consumer Micro-ops

	SIMD/0,1/1	FMA/0,1/4	VIMUL/0,1/4	SIMD/5/1,3	SHUF/5/1,3	V2I/0/3	I2V/5/1
SIMD/0,1/1	0	1	1	0	0	0	NA
FMA/0,1/4	1	0	1	0	0	0	NA
VIMUL/0,1/4	1	0	1	0	0	0	NA
SIMD/5/1,3	0	1	1	0	0	0	NA

Table 2-15. Bypass Delay Between Producer and Consumer Micro-ops (Contd.)

	SIMD/0,1/1	FMA/0,1/4	VIMUL/0,1/4	SIMD/5/1,3	SHUF/5/1,3	V2I/0/3	I2V/5/1
SHUF/5/1,3	0	0	1	0	0	0	NA
V2I/0/3	NA	NA	NA	NA	NA	NA	NA
I2V/5/1	0	0	1	0	0	0	NA

The attributes that are relevant to the producer/consumer micro-ops for bypass are a triplet of abbreviation/one or more port number/latency cycle of the uop. For example:

- "SIMD/0,1/1" applies to 1-cycle vector SIMD uop dispatched to either port 0 or port 1.
- "VIMUL/0,1/4" applies to 4-cycle vector integer multiply uop dispatched to either port 0 or port 1.
- "SIMD/5/1,3" applies to either 1-cycle or 3-cycle non-shuffle uop dispatched to port 5.

2.7.3 CACHE AND MEMORY SUBSYSTEM

The cache hierarchy of the Skylake Client microarchitecture has the following enhancements:

- Higher Cache bandwidth compared to previous generations.
- Simultaneous handling of more loads and stores enabled by enlarged buffers.
- Processor can do two page walks in parallel compared to one in Haswell microarchitecture and earlier generations.
- Page split load penalty down from 100 cycles in previous generation to 5 cycles.
- L3 write bandwidth increased from 4 cycles per line in previous generation to 2 per line.
- Support for the CLFLUSHOPT instruction to flush cache lines and manage memory ordering of flushed data using SFENCE.
- Reduced performance penalty for a software prefetch that specifies a NULL pointer.
- L2 associativity changed from 8 ways to 4 ways.

Table 2-16. Cache Parameters of the Skylake Client Microarchitecture

Level	Capacity / Associativity	Line Size (bytes)	Fastest Latency[1]	Peak Bandwidth (bytes/cyc)	Sustained Bandwidth (bytes/cyc)	Update Policy
First Level Data	32 KB/ 8	64	4 cycle	96 (2x32B Load + 1*32B Store)	~81	Writeback
Instruction	32 KB/8	64	N/A	N/A	N/A	N/A
Second Level	256KB/4	64	12 cycle	64	~29	Writeback
Third Level (Shared L3)	Up to 2MB per core/Up to 16 ways	64	44	32	~18	Writeback

NOTES:

1. Software-visible latency will vary depending on access patterns and other factors.

The TLB hierarchy consists of dedicated level one TLB for instruction cache, TLB for L1D, plus unified TLB for L2. The partition column of Table 2-17 indicates the resource sharing policy when Hyper-Threading Technology is active.

Table 2-17. TLB Parameters of the Skylake Client Microarchitecture

Level	Page Size	Entries	Associativity	Partition
Instruction	4KB	128	8 ways	dynamic
Instruction	2MB/4MB	8 per thread		fixed
First Level Data	4KB	64	4	fixed
First Level Data	2MB/4MB	32	4	fixed
First Level Data	1GB	4	4	fixed
Second Level	Shared by 4KB and 2/4MB pages	1536	12	fixed
Second Level	1GB	16	4	fixed

2.7.4 PAUSE LATENCY IN SKYLAKE CLIENT MICROARCHITECTURE

The PAUSE instruction is typically used with software threads executing on two logical processors located in the same processor core, waiting for a lock to be released. Such short wait loops tend to last between tens and a few hundreds of cycles, so performance-wise it is better to wait while occupying the CPU than yielding to the OS. When the wait loop is expected to last for thousands of cycles or more, it is preferable to yield to the operating system by calling an OS synchronization API function, such as WaitForSingleObject on Windows* OS or futex on Linux.

The PAUSE instruction is intended to:

- Temporarily provide the sibling logical processor (ready to make forward progress exiting the spin loop) with competitively shared hardware resources. The competitively-shared microarchitectural resources that the sibling logical processor can utilize in the Skylake Client microarchitecture are listed below.
  - Front end slots in the Decode ICache, LSD and IDQ.
  - Execution slots in the RS.
- Save power consumed by the processor core compared with executing equivalent spin loop instruction sequence in the following configurations.
  - One logical processor is inactive (e.g., entering a C-state).
  - Both logical processors in the same core execute the PAUSE instruction.
  - HT is disabled (e.g. using BIOS options).

The latency of the PAUSE instruction in prior generation microarchitectures is about 10 cycles, whereas in Skylake Client microarchitecture it has been extended to as many as 140 cycles.

The increased latency (allowing more effective utilization of competitively-shared microarchitectural resources to the logical processor ready to make forward progress) has a small positive performance impact of 1-2% on highly threaded applications. It is expected to have negligible impact on less threaded applications if forward progress is not blocked executing a fixed number of looped PAUSE instructions. There's also a small power benefit in 2-core and 4-core systems.

As the PAUSE latency has been increased significantly, workloads that are sensitive to PAUSE latency will suffer some performance loss.

The following is an example of how to use the PAUSE instruction with a dynamic loop iteration count.

Notice that in the Skylake Client microarchitecture the RDTSC instruction counts at the machine's guaranteed P1 frequency independently of the current processor clock (see the INVARIANT TSC property), and therefore, when running in Intel® Turbo-Boost-enabled mode, the delay will remain constant, but the number of instructions that could have been executed will change.

Use Poll Delay function in your lock to wait a given amount of guaranteed P1 frequency cycles, specified in the "clocks" variable.

Example 2-9. Dynamic Pause Loop Example

```
#include <x86intrin.h>
#include <stdint.h>

/* A useful predicate for dealing with timestamps that may wrap.
 Is a before b? Since the timestamps may wrap, this is asking whether it's
 shorter to go clockwise from a to b around the clock-face, or anti-clockwise.
 Times where going clockwise is less distance than going anti-clockwise
 are in the future, others are in the past. e.g. a = MAX-1, b = MAX+1 (=0),
 then a > b (true) does not mean a reached b; whereas signed(a) = -2,
 signed(b) = 0 captures the actual difference */

static inline bool before(uint64_t a, uint64_t b)
{
   return ((int64_t)b - (int64_t)a) > 0;
}

void pollDelay(uint32_t clocks)
{
   uint64_t endTime = _rdtsc()+ clocks;

   for (; before(_rdtsc(), endTime); )
      _mm_pause();
}
```

For contended spinlocks of the form shown in the baseline example below, we recommend an exponential back off when the lock is found to be busy, as shown in the improved example, to avoid significant performance degradation that can be caused by conflicts between threads in the machine. This is more important as we increase the number of threads in the machine and make changes to the architecture that might aggravate these conflict conditions. In multi-socket Intel server processors with shared memory, conflicts across threads take much longer to resolve as the number of threads contending for the same lock increases. The exponential back off is designed to avoid these conflicts between the threads thus avoiding the potential performance degradation. Note that in the example below, the number of PAUSE instructions are increased by a factor of 2 until some MAX_BACKOFF is reached which is subject to tuning.

Example 2-10. Contended Locks with Increasing Back-off Example

```
/*******************/
/*Baseline Version */
/*******************/

// atomic {if (lock == free) then change lock state to busy}
while (cmpxchg(lock, free, busy) == fail)
{
   while (lock == busy)
   {
     __asm__ (“pause”);
   }
}

/*******************/
/*Improved Version */
/*******************/

int mask = 1;
int const max = 64; //MAX_BACKOFF
while (cmpxchg(lock, free, busy) == fail)
{
  while (lock == busy)
  {
     for (int i=mask; i; --i){
        __asm__ (“pause”);
     }
       mask = mask < max ? mask<<1 : max;
  }
}
```

2.8 INTEL® 64 AND IA-32 INSTRUCTION BEST PRACTICES

2.8.1 NON-PRIVILEGED INSTRUCTION SERIALIZATION

Software may be required to serialize the execution pipeline of the current processor for various reasons in parallel execution environments. Several defined instructions, privileged and non-privileged, are classified as serializing instructions.

The concept of serializing instructions was introduced into the IA-32 architecture with the Pentium processor to support parallel instruction execution. Serializing instructions have no meaning for the Intel486 and earlier processors that do not implement parallel instruction execution. It is important to note that executing serializing instructions on P6 and more recent processor families constrains speculative execution because the results of speculatively executed instructions are discarded.

These instructions ensure the processor completes all modifications to flags, registers, and memory by previous instructions and drains all buffered writes to memory before the next instruction is fetched and executed. The non-privileged serialization instructions are:

- SERIALIZE
- CPUID
- IRET
- RSM

The SERIALIZE instruction was introduced in the Sapphire Rapids and Alder Lake platforms as a purpose-specific method of providing serialization to supersede the current typical usages such as CPUID.(EAX=0H). For example, CPUID usage for serialization has issues such that registers [EAX, EBX, ECX, EDX] are modified and, when executed on top of a VMM, will always incur the latency of a VM exit/VM entry round trip. SERIALIZE does not modify registers, arithmetic flags, or memory and does not incur a VM exit. The **SERIALIZE i**nstruction is enumerated via **CPUID.(EAX=07H,ECX=0):EDX[14]=1**, and software must verify support before usage.

Software that uses CPUID for serialization is recommended to use **Leaf 0 [CPUID.(EAX=0H)] CPUID**. CPUID leaves have variable performance with Leaf 0 providing the lowest latency when executed natively.

2.8.2 INTEL® HYPER-THREADING TECHNOLOGY (INTEL® HT TECHNOLOGY)

Intel® Hyper-Threading Technology (Intel® HT Technology) enables software to take advantage of task-level, or thread-level parallelism by providing multiple logical processors within a physical processor package, or within each processor core in a physical processor package. In its first implementation in the Intel® Xeon® processor, Intel HT Technology makes a single physical processor (or a processor core) appear as two or more logical processors.

Most Intel Architecture processor families support Intel HT Technology with two logical processors in each processor core, or in a physical processor in early implementations. The rest of this section describes features of the early implementation of Intel HT Technology. Most of the descriptions also apply to later implementations supporting two logical processors. The microarchitecture sections in this chapter provide additional details to individual microarchitecture and enhancements to Intel HT Technology.

The two logical processors each have a complete set of architectural registers while sharing one single physical processor's resources. By maintaining the architecture state of two processors, an Intel HT Technology-capable processor looks like two processors to software, including operating system and application code.

By sharing resources needed for peak demands between two logical processors, Intel HT Technology is well suited for multiprocessor systems to provide an additional performance boost in throughput when compared to traditional MP systems.

Figure 2-10 shows a typical bus-based symmetric multiprocessor (SMP) based on processors supporting Intel HT Technology. Each logical processor can execute a software thread, allowing a maximum of two software threads to execute simultaneously on one physical processor. The two software threads execute simultaneously, meaning that in the same clock cycle an “add” operation from logical processor 0 and another “add” operation and load from logical processor 1 can be executed simultaneously by the execution engine.

In the first implementation of Intel HT Technology, the physical execution resources are shared and the architecture state is duplicated for each logical processor. This minimizes the die area cost of implementing Intel HT Technology while still achieving performance gains for multithreaded applications or multitasking workloads.

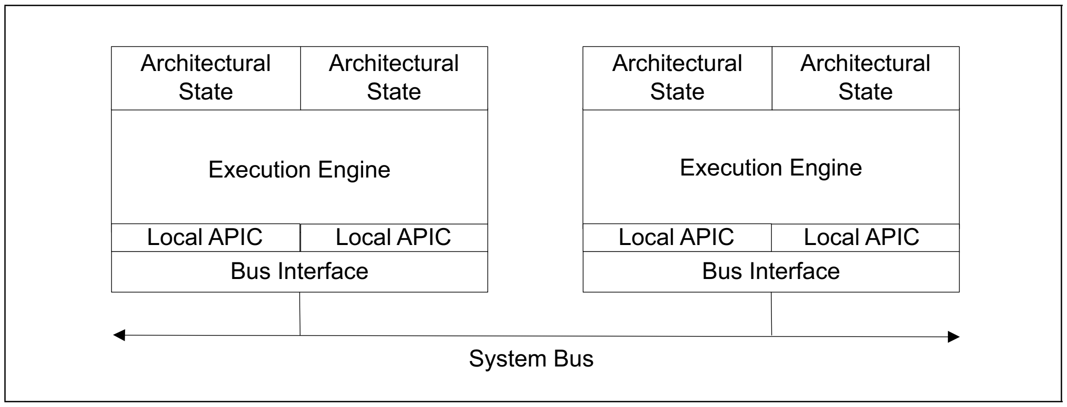


Figure 2-10. Intel® Hyper-Threading Technology on an SMP System

The performance potential due to Intel HT Technology is due to:

- The fact that operating systems and user programs can schedule processes or threads to execute simultaneously on the logical processors in each physical processor.
- The ability to use on-chip execution resources at a higher level than when only a single thread is consuming the execution resources; higher level of resource utilization can lead to higher system throughput.

2.8.2.1 Processor Resources and Intel® HT Technology

Most microarchitecture resources in a physical processor are shared between the logical processors. Only a few small data structures were replicated for each logical processor. This section describes how resources are shared, partitioned or replicated.

Replicated Resources

The architectural state is replicated for each logical processor. The architecture state consists of registers that are used by the operating system and application code to control program behavior and store data for computations. This state includes the eight general-purpose registers, the control registers, machine state registers, debug registers, and others. There are a few exceptions, most notably the memory type range registers (MTRRs) and the performance monitoring resources. For a complete list of the architecture state and exceptions, see the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 3A, 3B, 3C, & 3D.

Other resources such as instruction pointers and register renaming tables were replicated to simultaneously track execution and state changes of the two logical processors. The return stack predictor is replicated to improve branch prediction of return instructions.

In addition, a few buffers (for example, the two-entry instruction streaming buffers) were replicated to reduce complexity.

Partitioned Resources

Several buffers are shared by limiting the use of each logical processor to half the entries. These are referred to as partitioned resources. Reasons for this partitioning include:

- Operational fairness.
- Permitting the ability to allow operations from one logical processor to bypass operations of the other logical processor that may have stalled.

For example: a cache miss, a branch misprediction, or instruction dependencies may prevent a logical processor from making forward progress for some number of cycles. The partitioning prevents the stalled logical processor from blocking forward progress.

In general, the buffers for staging instructions between major pipe stages are partitioned. These buffers include µop queues after the execution trace cache, the queues after the register rename stage, the reorder buffer which stages instructions for retirement, and the load and store buffers.

In the case of load and store buffers, partitioning also provided an easier implementation to maintain memory ordering for each logical processor and detect memory ordering violations.

Shared Resources

Most resources in a physical processor are fully shared to improve the dynamic utilization of the resource, including caches and all the execution units. Some shared resources which are linearly addressed, like the DTLB, include a logical processor ID bit to distinguish whether the entry belongs to one logical processor or the other.

2.8.2.2 Microarchitecture Pipeline and Intel® HT Technology

This section describes the Intel HT Technology microarchitecture and how instructions from the two logical processors are handled between the front end and the back end of the pipeline.

Although instructions originating from two programs or two threads execute simultaneously and not necessarily in program order in the execution core and memory hierarchy, the front end and back end contain several selection points to select between instructions from the two logical processors. All selection points alternate between the two logical processors unless one logical processor cannot make use of a pipeline stage. In this case, the other logical processor has full use of every cycle of the pipeline stage. Reasons why a logical processor may not use a pipeline stage include cache misses, branch mispredictions, and instruction dependencies.

2.8.2.3 Execution Core

The core can dispatch up to six µops per cycle, provided the µops are ready to execute. Once the µops are placed in the queues waiting for execution, there is no distinction between instructions from the two logical processors. The execution core and memory hierarchy is also oblivious to which instructions belong to which logical processor.

After execution, instructions are placed in the re-order buffer. The re-order buffer decouples the execution stage from the retirement stage. The re-order buffer is partitioned so each uses half the entries.

2.8.2.4 Retirement

The retirement logic tracks when instructions from the two logical processors are ready to be retired. It retires the instruction in program order for each logical processor by alternating between the two logical processors. If one logical processor is not ready to retire any instructions, then all retirement bandwidth is dedicated to the other logical processor.

Once stores have retired, the processor must write the store data into the level-one data cache. Selection logic alternates between the two logical processors to commit store data to the cache.

2.8.3 SIMD TECHNOLOGY

SIMD computations (see Figure 2-11) were introduced to the architecture with MMX technology. MMX technology allows SIMD computations to be performed on packed byte, word, and doubleword integers. The integers are contained in a set of eight 64-bit registers called MMX registers (see Figure 2-12).

Earlier processors extended the SIMD computation model with the introduction of the Streaming SIMD Extensions (SSE). SSE allows SIMD computations to be performed on operands that contain four packed single-precision floating-point data elements. The operands can be in memory or in a set of eight 128-bit XMM registers (see Figure 2-12). SSE also extended SIMD computational capability by adding additional 64-bit MMX instructions.

Figure 2-11 shows a typical SIMD computation. Two sets of four packed data elements (X1, X2, X3, and X4, and Y1, Y2, Y3, and Y4) are operated on in parallel, with the same operation being performed on each corresponding pair of data elements (X1 and Y1, X2 and Y2, X3 and Y3, and X4 and Y4). The results of the four parallel computations are sorted as a set of four packed data elements.

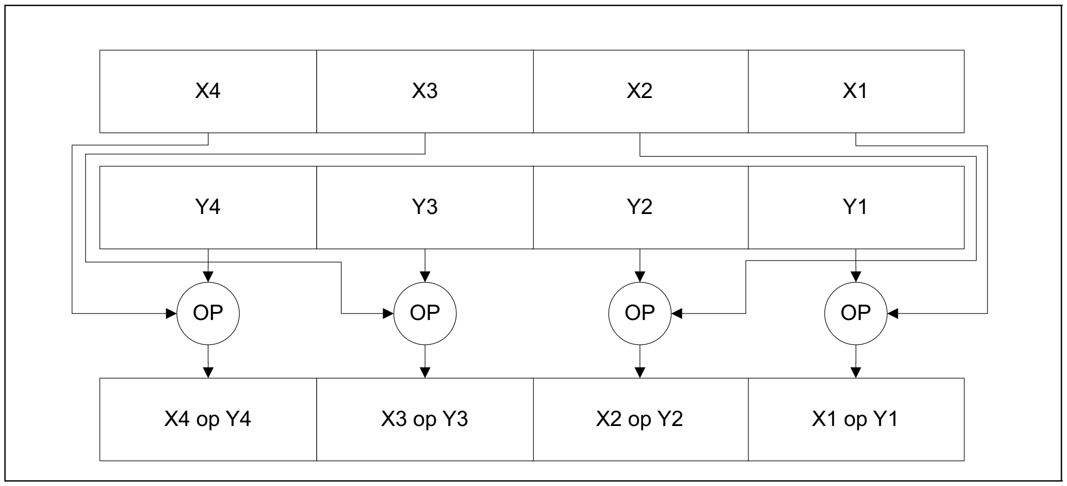


Figure 2-11. Typical SIMD Operations

The Pentium 4 processor further extended the SIMD computation model with the introduction of Streaming SIMD Extensions 2 (SSE2), Streaming SIMD Extensions 3 (SSE3), and Intel Xeon processor 5100 series introduced Supplemental Streaming SIMD Extensions 3 (SSSE3).

SSE2 works with operands in either memory or in the XMM registers. The technology extends SIMD computations to process packed double-precision floating-point data elements and 128-bit packed integers. There are 144 instructions in SSE2 that operate on two packed double-precision floating-point data elements or on 16 packed byte, 8 packed word, 4 doubleword, and 2 quadword integers.

SSE3 enhances x87, SSE and SSE2 by providing 13 instructions that can accelerate application performance in specific areas. These include video processing, complex arithmetics, and thread synchronization. SSE3 complements SSE and SSE2 with instructions that process SIMD data asymmetrically, facilitate horizontal computation, and help avoid loading cache line splits. See Figure 2-12.

SSSE3 provides additional enhancement for SIMD computation with 32 instructions on digital video and signal processing.

SSE4.1, SSE4.2 and AESNI are additional SIMD extensions that provide acceleration for applications in media processing, text/lexical processing, and block encryption/decryption.

The SIMD extensions operates the same way in Intel 64 architecture as in IA-32 architecture, with the following enhancements:

- 128-bit SIMD instructions referencing XMM register can access 16 XMM registers in 64-bit mode.
- Instructions that reference 32-bit general purpose registers can access 16 general purpose registers in 64-bit mode.

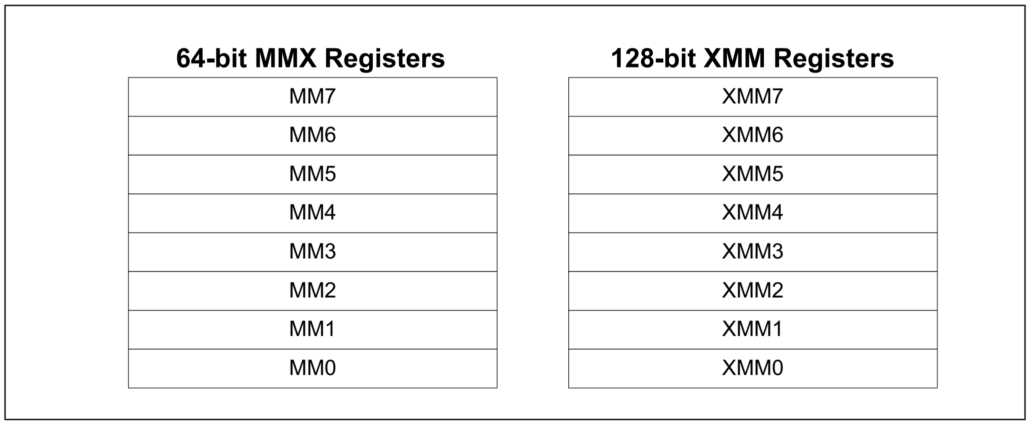


Figure 2-12. SIMD Instruction Register Usage

SIMD improves the performance of 3D graphics, speech recognition, image processing, scientific applications and applications that have the following characteristics:

- Inherently parallel.
- Recurring memory access patterns.
- Localized recurring operations performed on the data.
- Data-independent control flow.

2.8.4 SUMMARY OF SIMD TECHNOLOGIES AND APPLICATION LEVEL EXTENSIONS

SIMD floating-point instructions fully support the IEEE Standard 754 for Binary Floating-Point Arithmetic. They are accessible from all IA-32 execution modes: protected mode, real address mode, and Virtual 8086 mode.

SSE, SSE2, and MMX technologies are architectural extensions. Existing software will continue to run correctly, without modification on Intel microprocessors that incorporate these technologies. Existing software will also run correctly in the presence of applications that incorporate SIMD technologies.

SSE and SSE2 instructions also introduced cacheability and memory ordering instructions that can improve cache usage and application performance.

For more on SSE, SSE2, SSE3 and MMX technologies, see the following chapters in the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1

- Chapter 9, "Programming with Intel® MMX™ Technology."
- Chapter 10, "Programming with Intel® Streaming SIMD Extensions (Intel® SSE)."
- Chapter 11, "Programming with Intel® Streaming SIMD Extensions 2 (Intel® SSE2)."
- Chapter 12, "Programming with Intel® SSE3, SSSE3, Intel® SSE4, and Intel® AES-NI."
- Chapter 14, "Programming with Intel® AVX, FMA, and Intel® AVX2."
- Chapter 15, "Programming with Intel® AVX-512."

2.8.4.1 MMX™ Technology

MMX Technology introduced:

- 64-bit MMX registers.
- Support for SIMD operations on packed byte, word, and doubleword integers.

Recommendation: Integer SIMD code written using MMX instructions should consider more efficient implementations using SSE/Intel AVX instructions.

2.8.4.2 Streaming SIMD Extensions

Streaming SIMD extensions introduced:

- 128-bit XMM registers.
- 128-bit data type with four packed single-precision floating-point operands.
- Data prefetch instructions.
- Non-temporal store instructions and other cacheability and memory ordering instructions.
- Extra 64-bit SIMD integer support.

SSE instructions are useful for 3D geometry, 3D rendering, speech recognition, and video encoding and decoding.

2.8.4.3 Streaming SIMD Extensions 2

Streaming SIMD extensions 2 add the following:

- 128-bit data type with two packed double-precision floating-point operands.
- 128-bit data types for SIMD integer operation on 16-byte, 8-word, 4-doubleword, or 2-quadword integers.
- Support for SIMD arithmetic on 64-bit integer operands.
- Instructions for converting between new and existing data types.
- Extended support for data shuffling.
- Extended support for cacheability and memory ordering operations.

SSE2 instructions are useful for 3D graphics, video decoding/encoding, and encryption.

2.8.4.4 Streaming SIMD Extensions 3

Streaming SIMD extensions 3 add the following:

- SIMD floating-point instructions for asymmetric and horizontal computation.
- A special-purpose 128-bit load instruction to avoid cache line splits.
- An x87 FPU instruction to convert to integer independent of the floating-point control word (FCW).
- Instructions to support thread synchronization.

SSE3 instructions are useful for scientific, video and multi-threaded applications.

2.8.4.5 Supplemental Streaming SIMD Extensions 3

The Supplemental Streaming SIMD Extensions 3 introduces 32 new instructions to accelerate eight types of computations on packed integers. These include:

- 12 instructions that perform horizontal addition or subtraction operations.

- 6 instructions that evaluate the absolute values.
- 2 instructions that perform multiply and add operations and speed up the evaluation of dot products.
- 2 instructions that accelerate packed-integer multiply operations and produce integer values with scaling.
- 2 instructions that perform a byte-wise, in-place shuffle according to the second shuffle control operand.
- 6 instructions that negate packed integers in the destination operand if the signs of the corresponding element in the source operand is less than zero.
- 2 instructions that align data from the composite of two operands.

2.8.5 SSE4.1

SSE4.1 introduces 47 new instructions to accelerate video, imaging and 3D applications. SSE4.1 also improves compiler vectorization and significantly increase support for packed dword computation. These include:

- Two instructions perform packed dword multiplies.
- Two instructions perform floating-point dot products with input/output selects.
- One instruction provides a streaming hint for WC loads.
- Six instructions simplify packed blending.
- Eight instructions expand support for packed integer MIN/MAX.
- Four instructions support floating-point round with selectable rounding mode and precision exception override.
- Seven instructions improve data insertion and extractions from XMM registers
- Twelve instructions improve packed integer format conversions (sign and zero extensions).
- One instruction improves SAD (sum absolute difference) generation for small block sizes.
- One instruction aids horizontal searching operations of word integers.
- One instruction improves masked comparisons.
- One instruction adds qword packed equality comparisons.
- One instruction adds dword packing with unsigned saturation.

2.8.5.1 SSE4.2

SSE4.2 introduces 7 new instructions. These include:

- A 128-bit SIMD integer instruction for comparing 64-bit integer data elements.
- Four string/text processing instructions providing a rich set of primitives, these primitives can accelerate:
  - Basic and advanced string library functions from strlen, strcmp, to strcspn.
  - Delimiter processing, token extraction for lexing of text streams.
  - Parser, schema validation including XML processing.
- A general-purpose instruction for accelerating cyclic redundancy checksum signature calculations.
- A general-purpose instruction for calculating bit count population of integer numbers.

2.8.5.2 AESNI and PCLMULQDQ

AESNI introduces seven new instructions, six of them are primitives for accelerating algorithms based on AES encryption/decryption standard, referred to as AESNI.

The PCLMULQDQ instruction accelerates general-purpose block encryption, which can perform carry-less multiplication for two binary numbers up to 64-bit wide.

Typically, algorithm based on AES standard involve transformation of block data over multiple iterations via several primitives. The AES iteration.

AES encryption involves processing 128-bit input data (plain text) through a finite number of iterative operation, referred to as “AES round”, into a 128-bit encrypted block (ciphertext). Decryption follows the reverse direction of iterative operation using the “equivalent inverse cipher” instead of the “inverse cipher”.

The cryptographic processing at each round involves two input data, one is the “state”, the other is the “round key”. Each round uses a different “round key”. The round keys are derived from the cipher key using a “key schedule” algorithm. The “key schedule” algorithm is independent of the data processing of encryption/decryption, and can be carried out independently from the encryption/decryption phase.

The AES extensions provide two primitives to accelerate AES rounds on encryption, two primitives for AES rounds on decryption using the equivalent inverse cipher, and two instructions to support the AES key expansion procedure.

2.8.5.3 Intel® Advanced Vector Extensions (Intel® AVX)

Intel® Advanced Vector Extensions (Intel® AVX) offers comprehensive architectural enhancements over previous generations of Streaming SIMD Extensions. Intel AVX introduces the following architectural enhancements:

- Support for 256-bit wide vectors and SIMD register set.
- 256-bit floating-point instruction set enhancement with up to 2X performance gain relative to 128-bit Streaming SIMD extensions.
- Instruction syntax support for generalized three-operand syntax to improve instruction programming flexibility and efficient encoding of new instruction extensions.
- Enhancement of legacy 128-bit SIMD instruction extensions to support three-operand syntax and to simplify compiler vectorization of high-level language expressions.
- Support flexible deployment of 256-bit AVX code, 128-bit AVX code, legacy 128-bit code and scalar code.

Intel AVX instruction set and 256-bit register state management detail are described in Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volumes 2A, 2B, 2C, & 2D. Optimization techniques for Intel AVX are discussed in Chapter 15, “Programming with Intel® AVX-512.”

2.8.5.4 Half-Precision Floating-Point Conversion (F16C)

VCVTPH2PS and VCVTPS2PH are two instructions supporting half-precision floating-point data type conversion to and from single-precision floating-point data types. These two instruction extends on the same programming model as Intel AVX.

2.8.5.5 RDRAND

The RDRAND instruction retrieves a random number supplied by a cryptographically secure, deterministic random bit generator (DBRG). The DBRG is designed to meet NIST SP 800-90A standard.

2.8.5.6 Fused-Multiply-ADD (FMA) Extensions

FMA extensions enhances Intel AVX with high-throughput, arithmetic capabilities covering fused multiply-add, fused multiply-subtract, fused multiply add/subtract interleave, signed-reversed multiply on fused multiply-add and multiply-subtract operations. FMA extensions provide 36 256-bit floating-point instructions to perform computation on 256-bit vectors and additional 128-bit and scalar FMA instructions.

2.8.5.7 Intel® Advanced Vector Extensions 2 (Intel® AVX2)

Intel® AVX2 extends Intel AVX by promoting most of the 128-bit SIMD integer instructions with 256-bit numeric processing capabilities. Intel AVX2 instructions follow the same programming model as AVX instructions.

In addition, Intel AVX2 provide enhanced functionalities for broadcast/permute operations on data elements, vector shift instructions with variable-shift count per data element, and instructions to fetch non-contiguous data elements from memory.

2.8.5.8 General-Purpose Bit-Processing Instructions

The fourth generation Intel Core processor family introduces a collection of bit processing instructions that operate on the general purpose registers. Most of these instructions uses the VEX-prefix encoding scheme to provide non-destructive source operand syntax.

There instructions are enumerated by three separate feature flags reported by CPUID. For details, see Section 5.1 of Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1 and chapters 3, 4 and 5 of the Intel® 64 and IA-32 Architectures Software Developer's Manual, Volumes 2A, 2B, 2C, & 2D.

2.8.5.9 RDSEED

The RDSEED instruction retrieves a random number supplied by a cryptographically secure, enhanced deterministic random bit generator Enhanced NRBG). The NRBG is designed to meet the NIST SP 800-90B and NIST SP 800-90C standards.

2.8.5.10 ADCX and ADOX Instructions

The ADCX and ADOX instructions, in conjunction with MULX instruction, enable software to speed up calculations that require large integer numerics.

3. Updates to Chapter 4

Change bars and **violet** text show changes to Chapter 4 of the *Intel® 64 and IA-32 Architectures Optimization Resource Manual: **Intel Atom® Processor Architectures***.

Changes to this chapter:

- Section 4.1: All content updated to Crestmont microarchitecture from Gracemont microarchitecture.
  - This includes new instructions and updated features and values.
  - Figures 4-1 to 4-3 were updated with Crestmont information.
- Section 4.2:
  - Figure 4-4 was edited with style guidelines.

CHAPTER 4 INTEL ATOM® PROCESSOR ARCHITECTURES

This chapter gives an overview of features relevant to software optimization for current generations of Intel Atom® processors.

4.1 THE CRESTMONT MICROARCHITECTURE

The Crestmont microarchitecture builds on the success of the Gracemont microarchitecture. Listed below are some of the many enhancements provided by the Gracemont microarchitecture.

- Increased Branch Prediction Bandwidth (128B/cycle max from 32B/cycle on Gracemont).
- Larger Branch Target Buffer (6K entry from 5K) with Enhanced Path Based Branch Prediction.
- Wider allocation width (6-wide from 5-wide).
- Larger second-level TLB and larger dedicated 1GB page TLB.
- 48-bit VA with 52-bit PA used for MKTME keys.
- 2x SIMD integer multiply units, faster integer divide units.
- VEX-based AVX-NE-CONVERT convert, AVX-VNNI-INT8 and AVX-IFMA ISA extension.
- ECC protected Data Cache (in server products).
- Linear address masking (LAM)[1], Linear Address Space Separation (LASS)[2], Secure Arbitration Mode (SEAM), and Trust Domain Extensions (TDX) ISA extensions.
- Performance Monitoring enhancements include eight general-purpose counters (from six), precise distribution support for general-purpose counter 1 (totaling three counters), timed PEBS support, LBR event logging support, and multiple new events.

1. Not available in the Meteor Lake microarchitecture.
2. *ibid*.

4.1.1 CRESTMONT MICROARCHITECTURE OVERVIEW

The basic pipeline functionality of the Crestmont microarchitecture is depicted in Figure 4-1.

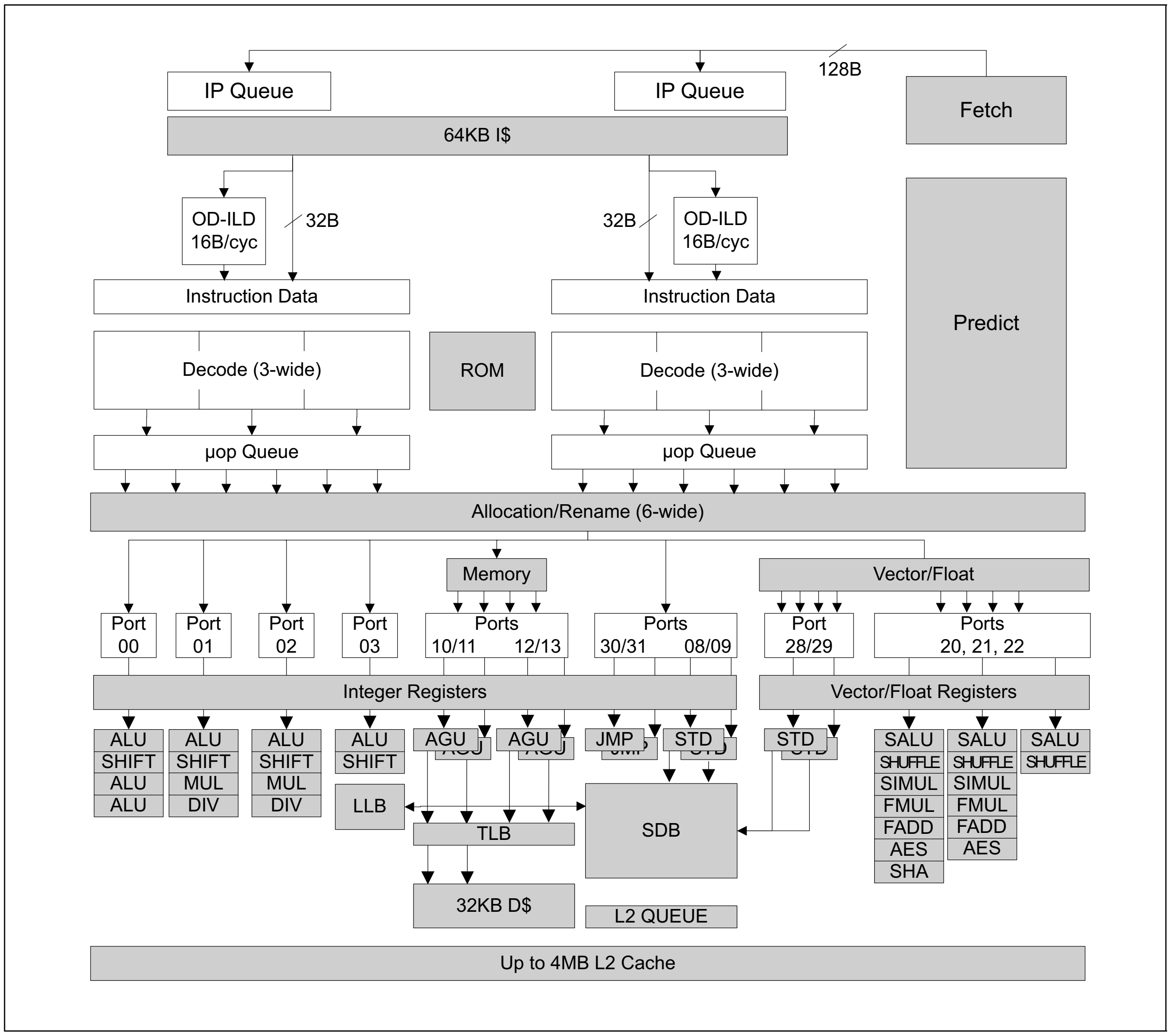


Figure 4-1. Processor Core Pipeline Functionality of the Crestmont Microarchitecture

The Crestmont microarchitecture supports flexible integration of multiple processor cores with a shared un-core subsystem consisting of a number of components including a ring interconnect to multiple slices of L3, processor graphics, integrated memory controller, interconnect fabrics, and more.

4.1.2 PREDICT AND FETCH

Crestmont features a front end with up to 128 bytes per cycle prediction. This is dramatically larger than the 32B/cycle from the Gracemont generation. To process all 128 bytes, the address must be aligned such that an even 64B line plus an odd 64B line can be processed together based on bit 7 of the linear address. If there is a taken branch to an odd 64B line, that prediction cycle will process only the 64B line. 128B/cycle processing will resume the following cycle, assuming no taken branches were detected.

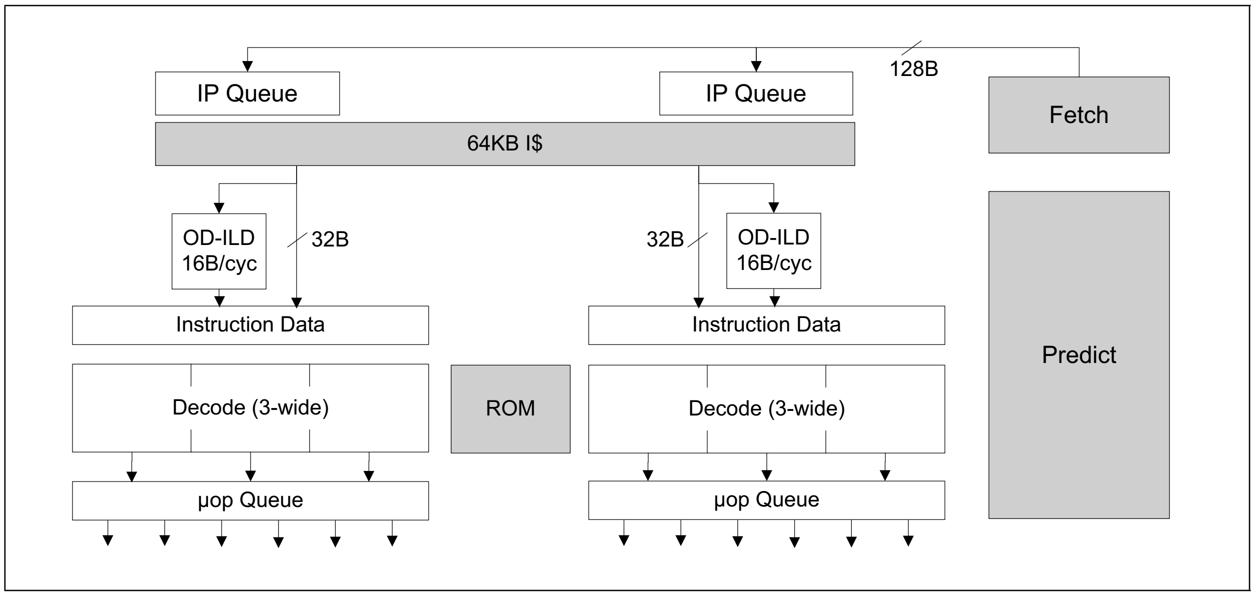


Figure 4-2. Front-End Pipeline Functionality of the Crestmont Microarchitecture

Each cycle, the predicted IP is sent down the instruction fetch pipeline. These predictions can look up the Instruction TLB (ITLB) and the instruction cache tag to determine the physical address and instruction cache hit or miss. Upon successful translation, and depending on resource availability, these accesses are stored into the instruction pointer (IP) queues. This enables the decoupling instruction cache hit/miss from delivering raw instruction bytes to the rest of the front end. In the case of an instruction cache miss, the IP queue holds the address but signals that the data cannot be read until it is returned from the memory subsystem. The stream of IPs generated at fetch can handle up to eight concurrent instruction cache misses. There are two independent IP queues, each with its instruction data buffers. Combined with their associated decoders, these are referred to as clusters. For each taken branch or inserted toggle point, the prediction will toggle back and forth between each IP queue and cluster. This toggling enables out-of-order decode, which is the key feature that enables this microarchitecture to fetch and decode up to 6 variable length x86 instructions per cycle.

Performance debug of prediction or fetch can be done utilizing the front-end bound events in the top-down category of performance monitoring events[1]. Front-end bound events count slots at allocation only when slots are available, but no µops are present. If bubbles caused by the three-cycle predictor percolate to allocation, for example, these will be represented by TOPDOWN_FE_BOUND.BRANCH_RESTEER. You can precisely tag the instruction following such a bubble via FRONTEND_RETIRED.BRANCH_RESTEER. If the predictor failed to cache a branch target and redirection occurred during decode, those slots are counted by TOPDOWN_FE_BOUND.BRANCH_DETECT. If µops are not delivered due to misses in the Instruction Cache or Instruction TLB, these appear as TOPDOWN_FE_BOUND.ICACHE and TOPDOWN_FE_BOUND.ITLB, respectively. Like BRANCH_RESTEER, all front-end bound slot-based accounting can be tracked precisely via the corresponding FRONTEND_RETIRED set of events. The instruction code can often be

1. Please see https://perfmon-events.intel.com.

rearranged to optimize such a bottleneck away. Multiple event classes can be tracked simultaneously (e.g., mark both ICACHE and ITLB events) on the same general-purpose performance counter or with different events across multiple performance counters.

Sometimes, a code loop is too short and/or poorly aligned within the cache to enable the machine to decode sufficiently fast. In this situation you could be fetching every cycle and never inserting bubbles, but still unable to keep the back-end fed. When this happens, the event class that detects this is TOPDOWN_FE_BOUND.OTHER. The “other” event class catches front-end bound behavior that cannot be pinpointed to other specific sources.

4.1.3 DYNAMIC LOAD BALANCING

Because Crestmont increased the prediction rate in many cases, it is expected to report fewer FE_BOUND.OTHER cases compared to Gracemont in general. However, since Crestmont also increased the allocation width by 20%, there are now more SLOTS that must be accounted for. Additional toggle points can be created based on internal heuristics when the hardware detects long basic blocks. These toggle points are added to the predictors, guiding the machine to toggle within the basic block and keeping both decode clusters busy.

4.1.4 INSTRUCTION DECODE AND THE ON-DEMAND INSTRUCTION LENGTH DECODER (OD-ILD)

Crestmont maintains the Gracemont on-demand instruction length decoding blocks. These blocks are typically only active when new instruction bytes are brought into the instruction cache due to a miss. When this happens, two extra cycles are added to the fetch pipeline of the affected decode cluster to generate pre-decode bits on the fly. Each block process is done across 16 bytes per cycle. With clustering, this means Crestmont is capable of 32 bytes per cycle across the two independent OD-ILDs (as in Gracemont).

Like Gracemont, Crestmont has two 3-wide instruction decode clusters capable of 6 instruction decode per cycle. Each instruction decoder generates a single µop yet can generate most x86 code as measured by dynamic instruction count. Load-op-stores, complicated addressing forms, Control Enforcement Technology (CET) instructions, and many more types are generated in a single internal µop format. Each decoder is also capable of detecting a microcode entry point. The most common short microcode flows can be executed out of order between the clusters, enabling additional performance. All µops are written into two parallel µop queues (one queue per 3-wide cluster), designed to allow the front and back end of the core to execute independently. The allocation and rename pipeline reads both µop queues in parallel and returns the instruction stream for register renaming and resource allocation.

The low-level characteristics of the microarchitecture within each decode cluster have remained identical since the Tremont microarchitecture[1].

If you are doing performance debugging and think load balancing or other decode restrictions may be an issue, this will often be indicated by TOPDOWN_FE_BOUND.DECODE. If the decoder struggled due to not having the correct pre-decode bits OR there were too many prefixes or escapes on the instructions, this would be represented by TOPDOWN_FE_BOUND.PREDECODE. If the machine is stuck waiting on lengthy microcode sequences, this will be represented by TOPDOWN_FE_BOUND.CISC. As with all other allocation slot-based FE_BOUND events, there are corresponding FRONTEND_RETIRED events that mark an instruction **after** the designated event class occurred. However, there is a difference in how this is reported for CISC events. As slot-based bottlenecks due to executing long microcode instructions are typically seen “within” an instruction, FRONTEND_RETIRED.CISC will often tag the CISC instruction itself, not the following instruction. When microcode is invoked to handle external interrupts, faults, traps, or other types of assists, FRONTEND_RETIRED.CISC will mark the next instruction that follows.

1. Refer to Chapter 6, “Earlier Generations of Intel Atom® Microarchitecture and Software Optimization” in the Intel® 64 and IA-32 Architectures Optimization Reference Manual Documentation Volume 2: Earlier Generations of Intel® 64 and IA-32 Processor Architectures, Throughput, and Latency.

4.1.5 ALLOCATION AND RETIREMENT

Crestmont can allocate up to six µops per cycle, compared to five µops in Gracemont. Allocation reads the µop queues of all front-end clusters simultaneously and generates an in-order stream splicing across clustering boundaries within the same cycle as necessary. In some cases, there can be an expansion between the format inside the µop queue and the format allocated to the machine. For example, for a 256-bit AVX instruction, the front-end decodes the instruction as a single µop, subdivided into 128-bit operations at allocation time. In this case, two allocation lanes allocate the two 128-bit halves of the instruction. The most common µops that use this method besides 256-bit AVX µops are integer µops that require multiple logical register destinations, like integer multiplies and divides. Another example is PUSH memory, which loads a value from memory from one address, stores the value into memory at the location of the stack pointer, and updates the stack pointer. MOV elimination, NOP detection, idiom detection (for example, XOR, a register by itself, producing all zeros), and memory renaming are performed at allocation time. This can reduce dependency chains and, in some situations, eliminate µops from execution.

For the 256-entry retirement buffer, retirement can be up to eight instructions per cycle. Retirement is wider than allocation to improve performance for store deallocation and other less common flushing conditions.

4.1.6 THE OUT-OF-ORDER AND EXECUTION ENGINES

The Out-of-Order and execution engines in the Crestmont microarchitecture remain mostly unchanged from the Gracemont microarchitecture with enhancements in:

- Integer divide units.
- Double SIMD integer multiply units supporting VNNI and new INT8 VNNI for 2x throughput.
- Support for AVX-IFMA in FMUL unit.

The execution pipeline functionality of the Crestmont microarchitecture is shown in Figure 4-3.

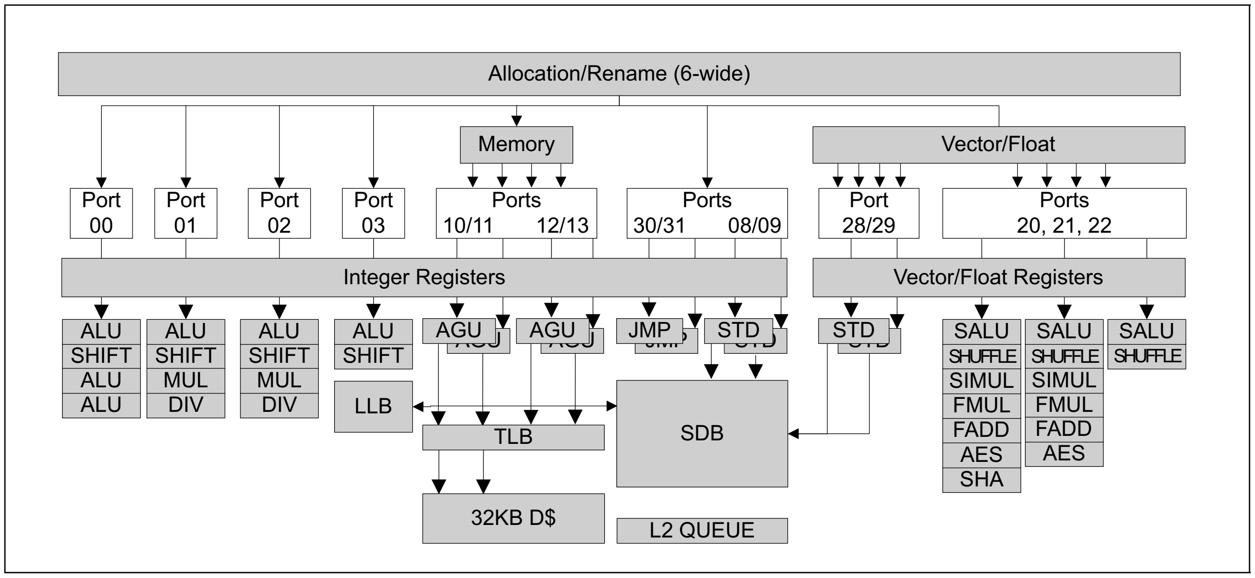


Figure 4-3. Execution Pipeline Functionality of the Crestmont Microarchitecture

Allocation delivers µops to three types of structures. Each µop is written into one or more of five reservation stations for pure integer operations. These hold instructions, track their dependencies, and schedule them for execution.

- Four are for ALU operations, labeled ports 00 to 03.

    - These execution units are mostly symmetric for single-cycle operations.
- Two of the four ports (01 and 02) can execute longer latency operations like multiplies and divides.
- The fifth integer reservation station holds *jump* and *store data* operations.
    - This structure is banked and can schedule two µops of each type every cycle
        - Two store data on ports 08 and 09.
        - Two jumps on ports 30 and 31.
- Complex instructions like an ADD, where one source and the destination are in memory, are decoded by the front end and allocated as a single µop.

The Crestmont microarchitecture can allocate five to six instructions like these per cycle. However, such µops separate into multiple pieces as they enter the back end. In this example, this single complex µop generates:

- A load.
- An add.
- A store address operation.
- A store data operation.

These pieces execute independently in the out-of-order machine, requiring four different dispatch ports.

Load Effective Address Operations (LEAs) are special and deserve extra attention.

The ALU ports are optimized to execute standard two-source arithmetic/logical operations, while the AGUs are optimized to handle the complexities of x86 memory addressing.

- LEAs are ALU operations that can have the same complex characteristics as AGU operations.
- LEAs without a scaled index and with only two sources among base, index, and displacement execute as a normal ALU operation on any port (00 through 03).
- LEAs with three sources fracture into two operations and take an additional cycle of latency.
- LEAs with a scaled index without a displacement execute as a single operation but are statically bound to port 02.

Allocation can also write into a memory queue.

This FIFO queue enables deeper buffering of the microarchitecture at a very low implementation cost. The memory queue can then write into a unified reservation station that holds load and store address generation operations. This reservation station can generate two load (ports 10 and 11) and store address calculations (ports 12 and 13)per cycle. The memory queue also writes the load and stores µops into the memory subsystem to perform translation and data cache access.

Finally, allocation can write the vector queue. This is where all vector SIMD and floating-point ALU operations go. This FIFO queue can then write into a unified reservation with three scheduling pipelines (ports 20, 21, and 22) or a store data reservation station capable of dispatching two store data per cycle (ports 28 and 29). The vector unit can execute any combination of two floating-point multiplies, adds, or multiply-add operations. This enables a peak of sixteen single-precision or eight double-precision FLOPS per cycle. It can also execute up to three SIMD integer ALU or shuffle operations along with dedicated AES and SHA units.

4.1.7 CACHE AND MEMORY SUBSYSTEM

The cache and memory subsystem remain largely unchanged from Gracemont other than the following enhancements:

- Enhanced store-to-load forwarding to cover more partial forwarding conditions.
- Larger second-level TLB.

- Larger and dedicated 1GB page TLB.
- Faster handling of locked loads.
- Support for Page Modification Logging and LAM.
- Faster eviction protocol.
- Dead Block Prediction to optimize last-level cache usage in some SoCs.
- Server SKU-specific features, including support for “end-to-end” parity and ECC in the L1 data cache.
- New performance monitoring events.
- Enhancements to the Memory Bandwidth Enforcement (MBE) QoS feature.

The Crestmont microarchitecture’s memory subsystem is designed to handle two 16 byte loads and two 16 byte stores per cycle, providing simultaneous 32 bytes of read bandwidth and 32 bytes of write bandwidth per cycle. The load-to-use latency for loads is typically four cycles. Suppose you are doing a pointer-chasing operation where the computed address results from a single prior load and a positive displacement of no more than +1023. In that case, the load-to-use latency observed can be reduced to three cycles. The L1 data cache is dual-ported to eliminate potential bank conflicts.

Memory disambiguation is supported, which allows loads to execute while older stores have unresolved addresses. Loads that forward from stores can do so in the same load to use latency as cache hits for cases where the store’s address is known, and the store data is available. Precise blocking and scheduling are done for cases where the store address or data is not immediately available and the hardware has determined that these are likely to be related addresses.

Address translations are performed through the first level DTLB, which is fully associative. On Crestmont, 2MB translations are natively cached within the first-level DTLB. The DTLB is backed by two second-level TLB (STLB) structures shared between code and data requests. The main STLB is 3072 entries 6-way set associative and caches 4KB and 2MB translations. Additionally, Crestmont has a 16-entry fully associative structure for 1 GB translations. STLB misses are sent to the page miss handler (PMH) which is pipelined such that it can perform up to four walks in parallel.

Table 4-1. Paging Cache Parameters of the Crestmont Microarchitecture

Level	Entries	Associativity	Architectural Page Size	Cached Translation Size
ITLB	64	Fully associative	All	4KB, 256KB
DTLB	32	Fully associative	All	4KB, 2MB
STLB	3072	6-Way	4K/2M/4M	4KB, 2MB
STLB	16	Fully associative	1GB	1GB

There are three independent L1 prefetchers. One does a simple next-line fetch on DL1 load misses. An instruction pointer-based prefetcher capable of detecting striding access patterns of various sizes. This prefetcher works in the linear address space; it can, therefore, cross page boundaries and start translations for TLB misses. The final prefetcher is a next-page prefetcher that detects accesses likely to cross a page boundary and starts the access early. L1 data misses generated by these prefetchers communicate additional information to the L2 prefetchers, which helps them work together.

The L2 cache delivers 64 bytes of data per cycle at a latency of 17 cycles, and that bandwidth is shared amongst 4 cores. The L2 cache subsystem also contains multiple prefetchers, including a streaming prefetcher that detects striding access patterns. An additional L2 prefetcher attempts to detect more complicated access patterns. These prefetches can also be generated such that they only fill the LLC but do not fill into the L2 to help reduce DRAM latency.

The L2 cache subsystem of a single 4-core module can have 64 requests and 32 L2 data evictions outstanding on the fabric. To ensure fairness, these are competitively shared amongst the cores with per-core reservations.

4.1.8 CRESTMONT NEW INSTRUCTION SUPPORT

4.1.8.1 AVX-NE-CONVERT Instructions

The AVX-NE-CONVERT includes an instruction set that converts low-precision floating points like BF16/FP16 to high-precision floating point FP32. It can also convert FP32 elements to BF16. These instructions allow the platform to have improved AI capabilities and better compatibility. Crestmont implements BF16/FP16 up-conversion in the load pipelines, which do not consume FPC execution resources. This allows for low latency conversions with simultaneous floating point compute.

4.1.8.2 AVX-IFMA

AVX-IFMA includes two instructions, VPMADD52LUQ and VPMADD52HUQ. They are designed to accelerate Big Integer Arithmetic (BIA). These instructions can multiply eight 52-bit unsigned integers residing in YMM registers, produce the low (VPMADD52LUQ) and high (VPMADD52HUQ) halves of the 104-bit products, and add the results to 64-bit accumulators (i.e., SIMD elements), placing them in the destination register.

4.1.8.3 AVX-VNNI-INT8 Instructions

For more flexibility within a convolutional neural network for INT8 sofrware, add all signed/unsigned INT8 data type support combinations to VNNI.

4.1.9 LEGACY INTEL® AVX1/INTEL® AVX2 INSTRUCTION SUPPORT

The Crestmont microarchitecture continues to support Intel® AVX and Intel® AVX2 instructions.

Most 256-bit Intel AVX and Intel AVX2 instructions are decoded as a single instruction and stored as a single µop in the front-end pipeline. To execute 256-bit instructions on native 128-bit vector execution and load data paths, most 256-bit µops are further subdivided into two independent 128-bit µops at allocation before insertion into the MEC and FPC reservation stations. These two independent µops are usually assigned to different execution ports, so both may execute in parallel. In general, 256-bit µops consume twice the allocation, execution, and retirement resources compared to 128-bit µops.

While most 256-bit Intel AVX2 instructions can be decomposed into two independent 128-bit micro-operations, a subset of Intel AVX2 instructions, known as cross-lane operations, can only compute the result for an element by utilizing one or more sources belonging to other elements. For example, when some or all of the upper 128-bit result [255:128] depends upon one or all of a lower element segment [127:0]. For example, when some or all of the upper 128-bit result [255:128] depends on one or all of a lower element segment [127:0]. These 256-bit cross-lane instructions execute with longer latency and/or reduced throughput than their 256-bit non-cross-lane counterparts.

4.1.9.1 256-bit Permute Operations

The instructions listed below use more operand sources than can be natively supported by a single reservation station within these microarchitectures. They are decomposed into two µops, where the first µop resolves a subset of operand dependencies across two cycles. The dependent second µop executes the 256-bit operation by using a single 128-bit execution port for two consecutive cycles with a five-cycle latency for a total latency of seven cycles.

• **VPERM2I128**	ymm1, ymm2, ymm3/m256, imm8
• **VPERM2F128**	ymm1, ymm2, ymm3/m256, imm8
• **VPERMPD**	ymm1, ymm2/m256, imm8
• **VPERMPS**	ymm1, ymm2, ymm3/m256
• **VPERMD**	ymm1, ymm2, ymm3/m256
• **VPERMQ**	ymm1, ymm2/m256, imm8

4.1.9.2 256-bit Broadcast with 128-bit Memory Operand

The memory versions of the broadcast instructions listed below have a single 128-bit or less memory source operand, a single SIMD ALU µop, and a load operand. The register version of the same instructions is decomposed into two SIMD ALU µops.

Operation portion latency is one cycle in addition to load operation latency.

• **VBROADCASTSD**	ymm1, m64
• **VBROADCASTSS**	ymm1, m32

4.1.9.3 256-bit Insertion, Up-Conversion Instructions with 128-bit Memory Operand

The memory versions of the instructions listed below have a single 128-bit or less memory source operand. They are decomposed into two µops. However, the second micro-operation for the memory version depends on the first micro-operation, while the second micro-operation of the register version of the same instruction does not. The register version of the same instructions can execute the upper and lower 128-bit segments in parallel.

Operation portion latency is two cycles in addition to load operation latency for the 256-bit insert, packed move with zero, and sign extension instructions listed below.

• **VPMOVZX**	ymm1, m128/64/32
• **VPMOVSX**	ymm1, m128/64/32
• **VINSERTI128**	ymm1, ymm2, m128, imm8
• **VINSERTF128**	ymm1, ymm2, m128, imm8

Operation portion latency is six cycles in addition to load operation latency for the up-convert instructions listed below.

• **VCVTPS2PD**	ymm1, m128
• **VCVTDQ2PD**	ymm1, m128
• **VCVTPH2PS**	ymm1, m128

4.1.9.4 256-bit Variable Blend Instructions

The **VBLENDVPD** and **VBLENDVPS** instructions listed below are implemented as a micro-coded flow. Throughput is one every four cycles, and latency is three cycles.

• **VBLENDVPD**	ymm1, ymm2, ymm3/m256, ymm4
• **VBLENDVPS**	ymm1, ymm2, ymm3/m256, ymm4

4.1.9.5 256-bit Vector TEST Instructions

The 256-bit vector **TEST** instructions listed below are decomposed into two µops with dependence between them. The operation result is written in the GPR arithmetic flags. Throughput is one per cycle, and latency is seven cycles.

• **VTESTPS**	ymm1, ymm2/m256
• **VTESTPD**	ymm1, ymm2/m256
• **VPTEST**	ymm1, ymm2/m256

4.1.9.6 The GATHER Instruction

The **VGATHER** instructions are implemented as micro-coded flow. Latency is ~50 cycles.

4.1.9.7 Masked Load and Store Instructions

The throughput of 256-bit VMASKMOV load and store is once every two cycles and that of the 128-bit VMASKMOV load and store is one per cycle.

A masked load or store with a masked element may encounter performance degradation if the masked element memory access causes an exception or a fault.

4.1.9.8 ADX Instructions

ADX instructions are supported. **ADCX** and **ADOX** are partial arithmetic flag-updating instructions. Intel Core microarchitecture renames and tracks arithmetic flags differently than Intel Atom. The carry flag (CF), overflow flag (OF), and other flags (ZF, AF, PF, SF) are renamed as independent registers on Core while they remain as a single register on Atom.

Unless there is a non-flag consuming full flag updating instruction between ADCX/ADOX instructions, on Crestmont, there is an operand dependency between the ADCX and ADOX instructions as the arithmetic flag register is a source operand of both. As this dependence between ADCX and ADOX instructions does not exist in the Intel Core microarchitecture, hand-tuned binaries exploiting this parallelism exist. While Crestmont supports the ISA, the parallelism will be lower on this microarchitecture.

4.2 THE TREMONT MICROARCHITECTURE

The Tremont microarchitecture builds on the success of the Goldmont Plus microarchitecture and provides the following enhancements:

- Enhanced branch prediction unit.
  - Increased capacity with improved path-based conditional and indirect prediction.
  - New committed Return Stack Buffer.
- Novel clustered 6-wide out-of-order front-end fetch and decode pipeline.
  - Banked ICache with dual 16B reads.
  - Two 3-wide decode clusters enabling up to 6 instructions per cycle.
- Deeper back-end out-of-order windows.
- 32KB data cache.
- Larger load and store buffers.

- Dual generic load and store execution pipes capable of two loads, two stores, or one load and store per cycle.
- Dedicated integer and vector integer/floating point store data ports.
- New and improved cryptography.
  - New Galois-field instructions (GFNI).
  - Dual AES units.
  - Enhanced SHA-NI implementation.
  - Faster PCLMULQDQ.
- Support for user-level low-power and low-latency spin-loop instructions UMWAIT/UMONITOR and TPAUSE.

4.2.1 TREMONT MICROARCHITECTURE OVERVIEW

The basic pipeline functionality of the Tremont microarchitecture is depicted in Figure 4-4.

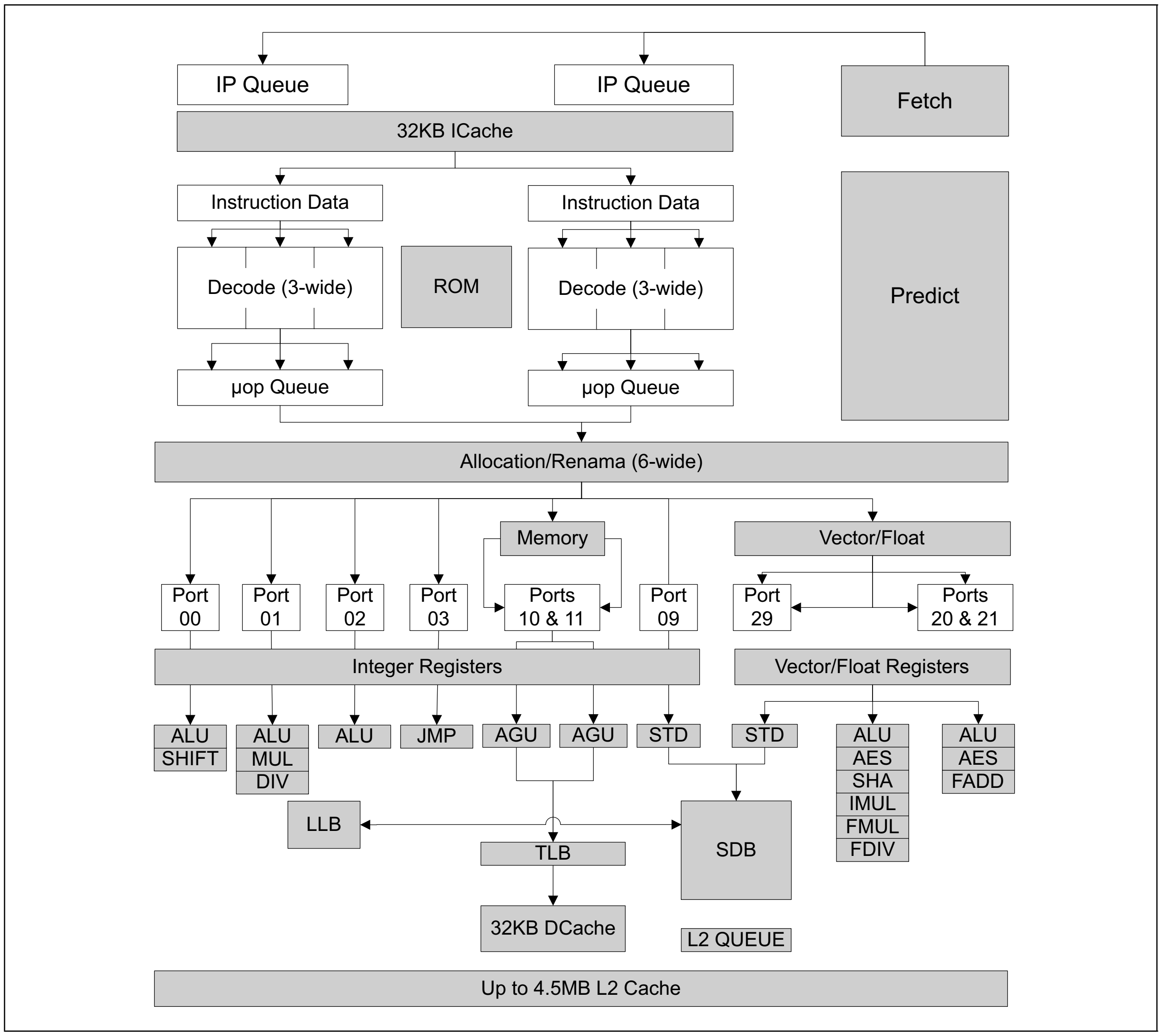


Figure 4-4. Processor Core Pipeline Functionality of the Tremont Microarchitecture

The Tremont microarchitecture supports flexible integration of multiple processor cores with a shared uncore sub-system consisting of several components, including a ring interconnect to multiple slices of L3, processor graphics, integrated memory controller, interconnect fabrics, and more.

4.2.2 THE FRONT END

Tremont microarchitecture introduces parallel out-of-order instruction decode. Instruction pointers access the ITLB, check the ICache tag array, and access the branch predictor. When the branch predictor produces a taken branch target, the new block of code advances the decode cluster assignment.

Tremont microarchitecture has a 32B predict pipeline that feeds dual 3-wide decode clusters capable of 6 instruction decode per cycle. Each cluster can access a banked 32KB instruction cache at 16B/cycle for a maximum of 32B/cycle. Due to differences in the number of instructions per block and other decode latency differences, younger blocks of code can decode before older blocks. At the end of each decode cluster is a queue of decoded instructions (µop queue).

The allocation and rename pipeline reads both µop queues in parallel and puts the instruction stream back in order for register renaming and resource allocation. Whereas increasing decode width for x86 traditionally requires exponential resources and triggers efficiency loss, clustering allows for x86 decode to be built with linear resources and little efficiency loss.

As the clustering algorithm is dependent on the ability to predict taken branches within the branch predictor, very long assembly sequences that lack taken branches (long unrolled code utilizing the floating point unit, for example) can be bottlenecked due to being unable to utilize both decode clusters simultaneously. Inserting unconditional JMP instructions to the next sequential instruction pointer at intervals between 16 to 32 instructions may relieve this bottleneck if encountered.

While Tremont microarchitecture did not build a dynamic mechanism to load balance the decode clusters, future generations of Intel Atom processors will include hardware to recognize and mitigate these cases without the need for explicit insertions of taken branches into the assembly code.

In addition to the novel clustered decode scheme, Tremont microarchitecture enhanced the branch predictor and doubled the size of the L2 Predecode cache from 64KB on the Goldmont Plus microarchitecture to 128 KB.

The low level characteristics of the microarchitecture within each decode cluster remain the same as in the Goldmont Plus microarchitecture. For example, instructions should avoid more than 4 Bytes of prefixes and escapes.

4.2.3 THE OUT-OF-ORDER AND EXECUTION ENGINES

The Out of Order and execution engines changes in the Tremont microarchitecture include:

- A significant increase in size of
  - Reorder buffer.
  - Load buffer.
  - Store buffer.
  - Reservation stations which enable deeper OOO execution and higher cache bandwidth.
- **Wider machine:** 8 $\longrightarrow$ 10 execution ports.
- Greater capabilities per execution port.

Table 4-2 summarizes the OOO engine's capability to dispatch different types of operations to ports.

Table 4-2. Dispatch Port and Execution Stacks of the Tremont Microarchitecture

Port 00 INT	Port 01 INT	Port 02 INT	Port 08 INT	Port 09 INT	Port 10	Port 11	Port 20 FP/VEC	Port 21 FP/VEC	Port 29 FP/VEC
ALU LEA[1] Shift	ALU LEA[2] Bit Ops IMUL IDIV POPCNT CRC32	ALU LEA[3]	JUMP	Store Data	Load Store Address	Load Store Address	ALU AES SHA-RND FMUL FDIV Shuffle Shift SIMUL GFNI Converts	ALU AES SHA-MSG FADD Shuffle	Store Data

NOTES:

1. LEAs without a scaled index and only two sources (among base, index, and displacement inputs) execute as one operation on any ALU port (00, 01, or 02).
2. LEAs with three sources fracture into two operations and take an additional cycle of latency. Index consuming portion, regardless of scale value, will bind to port 02 while second operation binds to either port 00 or 01.
3. LEAs with a scaled index but without a displacement execute as one operation on port 02.

4.2.4 CACHE AND MEMORY SUBSYSTEM

The cache hierarchy changes in Tremont microarchitecture include:

- 33% increase in size of the L1 data cache from 24KB to 32KB.
- 2×L1 load bandwidth:
  - 1 dedicated load port
  - 2 generic AGUs, shared between loads and stores.
- 2×L1 store bandwidth:
  - 1 dedicated store port
  - 2 generic AGUs, shared between loads and stores.
- Simultaneous handling of more loads and stores enabled by enlarged buffers.
- Maintains a 3-cycle load-to-use latency.
- Larger 2nd level TLB:
  - 512 4K entries → 1K 4K entries
  - 32 2M/4M entries → 64 2M/4M entries
- L2 cache size from 1MBs to 4.5MBs depending on SoC design choice:
  - The L2 size on Snow Ridge products is 4.5MBs, whereas the L2 size on Lakefield products is 1.5MBs.

The TLB hierarchy consists of a dedicated level one TLB for instruction cache and data cache with a shared second-level TLB for all page translations.

Table 4-3. Cache Parameters of the Tremont Microarchitecture

Level	Page Size	Entries	Associativity
Instruction	4KB/2M/4M[1]	48	Fully associative
First Level Data (loads and stores)	4KB/2M/4M[2]	32	Fully associative
Second Level	4KB	1024	4
Second Level	2M/4M	64	4

NOTES:

1. The first level instruction TLB (ITLB) caches small and large page translations but large pages are cached as 256KB regions per ITLB entry.
2. The first level data TLB (uTLB) caches small and large page translations but large pages are fully fractured into 4KB regions per uTLB entry.

4.2.5 NEW INSTRUCTIONS

New instructions and architectural changes in Tremont microarchitecture are listed below. Actual support may be product dependent.

- Galois Field New Instructions (GFNI) for
  - Acceleration of various encryption algorithms,
  - Error correction algorithms, and
  - Bit matrix multiplications.
- UMWAIT/UMONITOR/TPAUSE instructions enable power savings in user level spin loops.
- Cache line writeback instruction (CLWB) enables fast cache-line update to memory, while retaining clean copy in cache.
- Performance debugging benefits can be realized from the Tremont microarchitecture skidless PEBS implementation on both PMC0 and the fixed instruction counter. This enables a precise distribution via sampling on instructions and/or any of the precise general purpose events. As PEBS is triggered on the event **after** the overflow is signaled, counters should be programmed to large numbers that are (PRIME-1).

4.2.6 TREMONT MICROARCHITECTURE POWER MANAGEMENT

Tremont microarchitecture supports many of the same features as those found on the Ice Lake Client microarchitecture. Processors based on Tremont microarchitecture are the first Intel Atom processors with support for Intel® Speed Shift Technology. Power management features sometimes differ depending on the needs of the SoC.

4. Updates to Chapter 17

Change bars and **violet** text show changes to **Chapter 17** of the Intel® 64 and IA-32 Architectures Optimization Reference Manual: **Software Optimization for Intel® AVX-512 Instructions**

Changes to this chapter:

- Replaced Figure 17-1 with Table 17-1.
- Section 17.2
  - Added title

CHAPTER 17 SOFTWARE OPTIMIZATION FOR INTEL® AVX-512 INSTRUCTIONS

As part of the family of Intel® Accelerator Engines in Intel® Xeon® Scalable processors, Intel® Advanced Vector Extensions 512 (Intel® AVX-512) provides built-in acceleration for demanding workloads that involve heavy vector-based processing. They are the following set of 512-bit instruction set extensions:

- Intel® AVX-512 Foundation (F)
  - 512-bit vector width.
  - 32 512-bit long vector registers.
  - Data expand and data compress instructions.
  - Ternary logic instruction.
  - 8 new 64-bit long mask registers.
  - Two source cross-lane permute instructions.
  - Scatter instructions.
  - Embedded broadcast/rounding.
  - Transcendental support.
- Intel® AVX-512 Conflict Detection Instructions (CD)
- Intel® AVX-512 Exponential and Reciprocal Instructions (ER)
- Intel® AVX-512 Prefetch Instructions (PF)
- Intel® AVX-512 Byte and Word Instructions (BW)
- Intel® AVX-512 Double Word and Quad Word Instructions (DQ)
  - New QWORD and Compute and Convert Instructions.
- Intel® AVX-512 Vector Length Extensions (VL)

Performance reports in this chapter are based on Data Cache Unit (DCU) resident data measurements on the Skylake Server System with Intel® Turbo-Boost technology disabled, Intel® SpeedStep® Technology disabled, core and uncore frequency set to 1.8GHz, unless otherwise specified. This fixed frequency configuration is used to isolate code change impacts from other factors.

Table 17-1. Intel® AVX-512 Feature Flags Across Intel® Xeon® Processor Generations

Intel® Xeon® Scalable Processors	Intel® Core™ Processors	2nd Generation Intel® Xeon® Scalable Processors	3rd Generation Intel® Xeon® Scalable Processors	4th/5th Generation Intel® Xeon® Scalable Processors
AVX/AVX2	AVX/AVX2	AVX/AVX2	AVX/AVX2	AVX/AVX2
AVX512F, AVX512CD, AVX512BW, AVX512DQ	AVX512F, AVX512CD, AVX512BW, AVX512DQ	AVX512F, AVX512CD, AVX512BW, AVX512DQ	AVX512F, AVX512CD, AVX512BW, AVX512DQ	AVX512F, AVX512CD, AVX512BW, AVX512DQ
NA	AVX512_VBMI, AVX512_IFMA	AVX512_VBMI, AVX512_IFMA	AVX512_VBMI, AVX512_IFMA	AVX512_VBMI, AVX512_IFMA

Table 17-1. (Contd.)Intel® AVX-512 Feature Flags Across Intel® Xeon® Processor Generations

Intel® Xeon® Scalable Processors	Intel® Core™ Processors	2nd Generation Intel® Xeon® Scalable Processors	3rd Generation Intel® Xeon® Scalable Processors	4th/5th Generation Intel® Xeon® Scalable Processors
NA	NA	AVX512_VNNI	AVX512_VNNI	AVX512_VNNI
NA	NA	NA	AVX512_BF16	AVX512_BF16
NA	NA	NA	NA	AVX512_VPOPCNTD, AVX512_VBM12, VAES, GFNI, VPCLMULQDQ, AVX512_BITALG
NA	NA	NA	NA	AVX512_FP16

17.1 BASIC INTEL® AVX-512 VS. INTEL® AVX2 CODING

In most cases, the main performance driver for Intel AVX-512 will be the 512-bit register width. This section demonstrates the similarity and differences between basic Intel AVX2 and Intel AVX-512 code and explains how to convert code from Intel AVX2 to Intel AVX-512 easily. The first sub section demonstrates the conversion of intrinsic code and the second sub-section of assembly code. The following sections highlight advanced aspects that require consideration and treatment when doing such conversions.

The examples in the following subsections implement a Cartesian coordinate system rotation. A point in a Cartesian coordinate system is described by the pair (x,y). The following picture demonstrates a Cartesian rotation of (x,y) by angle θ to (x',y').

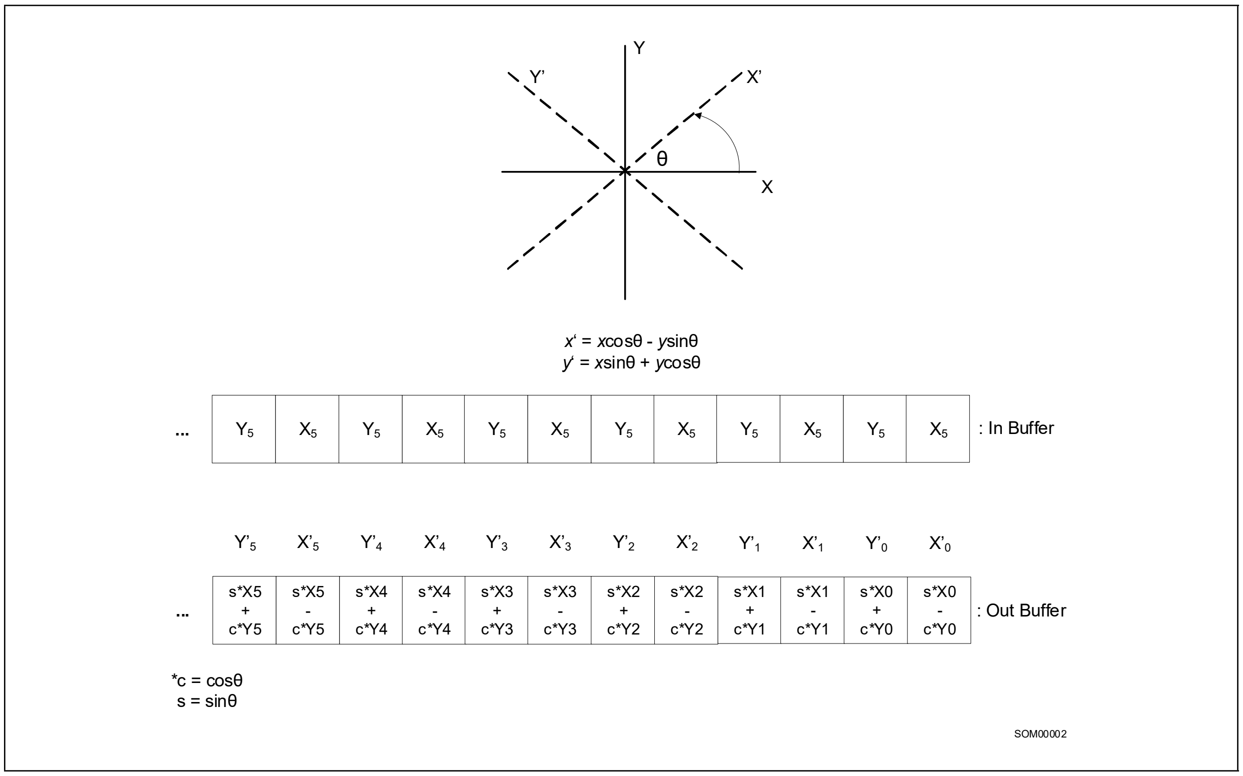

Y
Y'
X'
θ
X
x' = xcosθ - ysinθ
y' = xsinθ + ycosθ
... Y5 X5 Y5 X5 Y5 X5 Y5 X5 Y5 X5 Y5 X5 : In Buffer
Y'5 X'5 Y'4 X'4 Y'3 X'3 Y'2 X'2 Y'1 X'1 Y'0 X'0
... s*X5 + c*Y5 | s*X5 - c*Y5 | s*X4 + c*Y4 | s*X4 - c*Y4 | s*X3 + c*Y3 | s*X3 - c*Y3 | s*X2 + c*Y2 | s*X2 - c*Y2 | s*X1 + c*Y1 | s*X1 - c*Y1 | s*X0 + c*Y0 | s*X0 - c*Y0 : Out Buffer
*c = cosθ
s = sinθ
SOM00002

Figure 17-1. Cartesian Rotation

17.1.1 INTRINSIC CODING

The following comparison of Intel AVX2 and Intel AVX-512 shows how to convert a simple intrinsic Intel AVX2 code sequence to Intel AVX-512. This example demonstrates the Intel AVX Instruction format, 64 byte ZMM registers, dynamic and static memory allocation with data alignment of 64bytes, and the C data type representing 16 floating point elements in a ZMM register. Follow these guidelines when doing this transformation.

- Align statically and dynamically allocated buffers to 64-bytes.
- Use a double supplemental buffer size for constants.
- Change __mm256_ intrinsic name prefix with __mm512_.
- Change variable data types names from __m256 to __m512.

- Divide by 2 iteration count (double stride length).

Example 17-1. Cartesian Coordinate System Rotation with Intrinsics

Intel® AVX2 Intrinsics Code	Intel® AVX-512 Intrinsics Code
	#include <immintrin.h>
#include <immintrin.h>	int main()
int main()	{
{	int len = 3200;
int len = 3200;	//Dynamic memory allocation with 64byte
//Dynamic memory allocation with 32byte	//alignment
//alignment	float* pInVector = (float *)
float* pInVector = (float *)	_mm_malloc(len*sizeof(float),64);
_mm_malloc(len*sizeof(float),32);	float* pOutVector = (float *)
float* pOutVector = (float *)	_mm_malloc(len*sizeof(float),64);
_mm_malloc(len*sizeof(float),32);	
	//init data
//init data	for (int i=0; i<len; i++)
for (int i=0; i<len; i++)	pInVector[i] = 1;
pInVector[i] = 1;	
	float cos_theta = 0.8660254037;
float cos_theta = 0.8660254037;	float sin_theta = 0.5;
float sin_theta = 0.5;	
	//Static memory allocation of 16 floats with 64byte
//Static memory allocation of 8 floats with 32byte	alignments
alignments	__declspec(align(64)) float cos_sin_theta_vec[16] =
__declspec(align(32)) float cos_sin_theta_vec[8] =	{cos_theta, sin_theta, cos_theta, sin_theta, cos_theta,
{cos_theta, sin_theta, cos_theta, sin_theta, cos_theta,	sin_theta, cos_theta, sin_theta cos_theta, sin_theta,
sin_theta, cos_theta, sin_theta};	cos_theta, sin_theta, cos_theta, sin_theta, cos_theta,
	sin_theta};
	__declspec(align(64)) float sin_cos_theta_vec[16] =
__declspec(align(32)) float sin_cos_theta_vec[8] =	{sin_theta, cos_theta, sin_theta, cos_theta, sin_theta,
{sin_theta, cos_theta, sin_theta, cos_theta, sin_theta,	cos_theta, sin_theta, cos_theta, sin_theta, cos_theta,
cos_theta, sin_theta, cos_theta};	sin_theta, cos_theta, sin_theta, cos_theta, sin_theta,
	cos_theta};
//__m256 data type represents a Ymm	
// register with 8 float elements	//__m512 data type represents a Zmm
__m256 Ymm_cos_sin =	// register with 16 float elements
_mm256_load_ps(cos_sin_theta_vec);	__m512 Zmm_cos_sin =
	_mm512_load_ps(cos_sin_theta_vec);

Example 17-1. Cartesian Coordinate System Rotation with Intrinsics (Contd.)

Intel® AVX2 Intrinsics Code	Intel® AVX-512 Intrinsics Code

//Intel® AVX2 256bit packed single load
__m256 Ymm_sin_cos =
_mm256_load_ps(sin_cos_theta_vec);
 	
__m256 Ymm0, Ymm1, Ymm2, Ymm3;
//processing 16 elements in an unrolled
//twice loop
for(int i=0; i<len; i+=16)
{
Ymm0 = _mm256_load_ps(pInVector+i);
Ymm1 = _mm256_moveldup_ps(Ymm0);
Ymm2 = _mm256_movehdup_ps(Ymm0);
Ymm2 = _mm256_mul_ps(Ymm2,Ymm_sin_cos);
Ymm3 =
_mm256_fmaddsub_ps(Ymm1,Ymm_cos_sin,Ymm2);
_mm256_store_ps(pOutVector + i,Ymm3);
 	
Ymm0 = _mm256_load_ps(pInVector+i+8);
Ymm1 = _mm256_moveldup_ps(Ymm0);
Ymm2 = _mm256_movehdup_ps(Ymm0);
Ymm2 = _mm256_mul_ps(Ymm2, Ymm_sin_cos);
Ymm3 =
_mm256_fmaddsub_ps(Ymm1,Ymm_cos_sin,Ymm2);
_mm256_store_ps(pOutVector+i+8,Ymm3);
}
 	
_mm_free(pInVector);
_mm_free(pOutVector);
 	
return 0;
}
	//Intel® AVX-512 512bit packed single load
	__m512 Zmm_sin_cos =
	_mm512_load_ps(sin_cos_theta_vec);
	__m512 Zmm0, Zmm1, Zmm2, Zmm3;
	//processing 32 elements in an unrolled
	//twice loop
	for(int i=0; i<len; i+=32)
	{
	Zmm0 = _mm512_load_ps(pInVector+i);
	Zmm1 = _mm512_moveldup_ps(Zmm0);
	Zmm2 = _mm512_movehdup_ps(Zmm0);
	Zmm2 = _mm512_mul_ps(Zmm2,Zmm_sin_cos);
	Zmm3 =
	_mm512_fmaddsub_ps(Zmm1,Zmm_cos_sin,Zmm2);
	_mm512_store_ps(pOutVector + i,Zmm3);
	
	Zmm0 = _mm512_load_ps(pInVector+i+16);
	Zmm1 = _mm512_moveldup_ps(Zmm0);
	Zmm2 = _mm512_movehdup_ps(Zmm0);
	Zmm2 = _mm512_mul_ps(Zmm2, Zmm_sin_cos);
	Zmm3 =
	_mm512_fmaddsub_ps(Zmm1,Zmm_cos_sin,Zmm2);
	_mm512_store_ps(pOutVector+i+16,Zmm3);
	}
	_mm_free(pInVector);
	_mm_free(pOutVector);
	
	return 0;
	}

Baseline	Speedup: 1.95x

17.1.2 ASSEMBLY CODING

Similar to the intrinsic porting guidelines, assembly porting guidelines are listed below:

- Align statically and dynamically allocated buffers to 64-bytes.
- Double the supplemental buffer sizes if needed.
- Add a “v” prefix to instruction names.
- Change register names from ymm to zmm.
- Divide the iteration count by two (or double stride length).

Example 17-2. Cartesian Coordinate System Rotation with Assembly

Intel® AVX2 Assembly Code	Intel® AVX-512 Assembly Code
#include <immintrin.h>	#include <immintrin.h>
int main()	int main()
{	{
int len = 3200;	int len = 3200;
//Dynamic memory allocation with 32byte alignment	//Dynamic memory allocation with 64byte alignment
float* pInVector = (float *)	float* pInVector = (float *)
_mm_malloc(len*sizeof(float),32);	_mm_malloc(len*sizeof(float),64);
float* pOutVector = (float *)	float* pOutVector = (float *)
_mm_malloc(len*sizeof(float),32);	_mm_malloc(len*sizeof(float),64);
//init data	//init data
for (int i=0; i<len; i++)	for (int i=0; i<len; i++)
pInVector[i] = 1;	pInVector[i] = 1;
float cos_theta = 0.8660254037;	float cos_theta = 0.8660254037;
float sin_theta = 0.5;	float sin_theta = 0.5;
//Static memory allocation of 8 floats with 32byte	//Static memory allocation of 16 floats with 64byte
alignments	alignments
__declspec(align(32)) float cos_sin_theta_vec[8] =	__declspec(align(64)) float cos_sin_theta_vec[16] =
{cos_theta, sin_theta,	{cos_theta,
cos_theta, sin_theta, cos_theta, sin_theta, cos_theta,	sin_theta, cos_theta, sin_theta, cos_theta, sin_theta,
sin_theta};	cos_theta, sin_theta, cos_theta, sin_theta, cos_theta,
	sin_theta, cos_theta, sin_theta, cos_theta, sin_theta};
	__declspec(align(64)) float sin_cos_theta_vec[16] =
__declspec(align(32)) float sin_cos_theta_vec[8] =	{sin_theta, cos_theta, sin_theta, cos_theta, sin_theta,
{sin_theta, cos_theta, sin_theta, cos_theta, sin_theta,	cos_theta, sin_theta cos_theta, sin_theta, cos_theta,
cos_theta, sin_theta, cos_theta};	sin_theta, cos_theta, sin_theta, cos_theta, sin_theta,
	cos_theta};
__asm	__asm
{	{
mov rax,pInVector	mov rax,pInVector
mov r8,pOutVector	mov r8,pOutVector
// Load into a ymm register of 32 bytes	// Load into a zmm register of 64 bytes
vmovups ymm3, ymmword ptr[cos_sin_theta_vec]	vmovups zmm3, zmmword ptr[cos_sin_theta_vec]
vmovups ymm4, ymmword ptr[sin_cos_theta_vec]	vmovups zmm4, zmmword ptr[sin_cos_theta_vec]
mov edx, len	mov edx, len
shl edx, 2	shl edx, 2
xor ecx, ecx	xor ecx, ecx

Example 17-2. Cartesian Coordinate System Rotation with Assembly (Contd.)

Intel® AVX2 Assembly Code	Intel® AVX-512 Assembly Code

```
loop1:
  vmovsldup ymm0, [rax+rcx]
  vmovshdup ymm1, [rax+rcx]
  vmulps ymm1, ymm1, ymm4
  vfmaddsub213ps ymm0, ymm3, ymm1
  // 32 byte store from a ymm register
  vmovaps [r8+rcx], ymm0

vmovsldup ymm0, [rax+rcx+32]
  vmovshdup ymm1, [rax+rcx+32]
  vmulps ymm1, ymm1, ymm4
  vfmaddsub213ps ymm0, ymm3, ymm1
  // offset 32 bytes from previous store
  vmovaps [r8+rcx+32], ymm0

  // Processed 64bytes in this loop
  // (the code is unrolled twice)
  add ecx, 64
  cmp ecx, edx
  jl loop1
 }

  _mm_free(pInVector);
  _mm_free(pOutVector);

  return 0;
}
```
	```
	loop1:
	  vmovsldup zmm0, [rax+rcx]
	  vmovshdup zmm1, [rax+rcx]
	  vmulps zmm1, zmm1, zmm4
	  vfmaddsub213ps zmm0, zmm3, zmm1
	  // 64 byte store from a zmm register
	  vmovaps [r8+rcx], zmm0

	vmovsldup zmm0, [rax+rcx+64]
	  vmovshdup zmm1, [rax+rcx+64]
	  vmulps zmm1, zmm1, zmm4
	  vfmaddsub213ps zmm0, zmm3, zmm1
	  // offset 64 bytes from previous store
	  vmovaps [r8+rcx+64], zmm0

	  // Processed 128bytes in this loop
	  // (the code is unrolled twice)
	  add ecx, 128
	  cmp ecx, edx
	  jl loop1
	 }

	  _mm_free(pInVector);
	  _mm_free(pOutVector);

	  return 0;
	}
	```

Baseline	Speedup: 1.95x

17.2 MASKING

Intel AVX-512 instructions which use the Extended VEX coding scheme (EVEX) encode a predicate operand to conditionally control per-element computational operation and update the result to the destination operand. The predicate operand is known as the opmask register. The opmask is a set of eight architectural registers, 64 bits each. From this set of 8 architectural registers, only k1 through k7 can be addressed as the predicate operand; k0 can be used as a regular source or destination but cannot be encoded as a predicate operand.

A predicate operand can be used to enable memory fault-suppression for some instructions with a memory source operand.

As a predicate operand, the opmask registers contain one bit to govern the operation / update of each data element of a vector register. Masking is supported on Skylake microarchitecture for instructions with all data sizes:

- Byte (int8)
- Word (int16)
- Single-precision floating-point (float32)
- Integer doubleword (int32)

- Double precision floating-point (float64)
- Integer quadword (int64)

Therefore, a vector register holds either 8, 16, 32 or 64 elements; accordingly, the length of a vector mask register is 64 bits. Masking on Skylake microarchitecture is also enabled for all vector length values: 128-bit, 256-bit and 512-bit. Each instruction accesses only the number of least significant mask bits needed based on its data type and vector length. For example, Intel AVX-512 instructions operating on 64-bit data elements with a 512-bit vector length, only use the 8 (i.e., 512/64) least significant bits of the opmask register.

An opmask register affects an Intel AVX-512 instruction at per-element granularity. So, any numeric or non-numeric operation of each data element and per-element updates of intermediate results to the destination operand are predicated on the corresponding bit of the opmask register.

An opmask serving as a predicate operand in Intel AVX-512 has the following properties:

- The instruction's operation is only performed for an element if the corresponding opmask bit is set. This implies that no exception or violation can be caused by an operation on a masked-off element. Consequently, no MXCSR exception flag is updated as a result of a masked-off operation.
- A destination element is not updated with the result of the operation if the corresponding writemask bit is not set. Instead, the destination element value may be preserved (merging-masking) or zeroed out (zeroing-masking).
- For some instructions with a memory operand, memory faults are suppressed for elements with a mask bit of 0.

Note that this feature provides a powerful construct to implement control-flow predication, since the mask provides a merging behavior for Intel AVX-512 vector register destinations. As an alternative the masking can be used for zeroing instead of merging, so that the masked out elements are updated with 0 instead of preserving the old value. The zeroing behavior removes the implicit dependency on the old value when it is not needed.

Most instructions with masking enabled accept both forms of masking. Instructions that must have EVEX.aaa bits different than 0 (gather and scatter) and instructions that write to memory, only accept merging-masking.

The per-element destination update rule also applies when the destination operand is a memory location. Vectors are written on a per element basis, based on the opmask register used as a predicate operand.

The value of an opmask register can be:

- Generated as a result of a vector instruction (CMP, FPCLASS, etc.).
- Loaded from memory.
- Loaded from GPR register.
- Modified by mask-to-mask operations.

17.2.1 MASKING EXAMPLE

The masked instructions conditionally operate with packed data elements, depending on the mask bits associated with each data element. The mask bit for each data element is the corresponding bit in the mask register.

When performing a mask instruction, the returned value is 0 for elements which have a corresponding mask value of 0. The corresponding value in the destination register depends on the zeroing flag:

- If the flag is set, the memory location is filled with zeros.
- If the flag is not set, the values in memory location can are preserved.

The following figures show an example for a mask move from one register to another when using merging masking.

```
vmovaps  zmm1 {k1},  zmm0
```

The destination register before instruction execution is shown in Figure 17-2, 17-3 and 17-4.

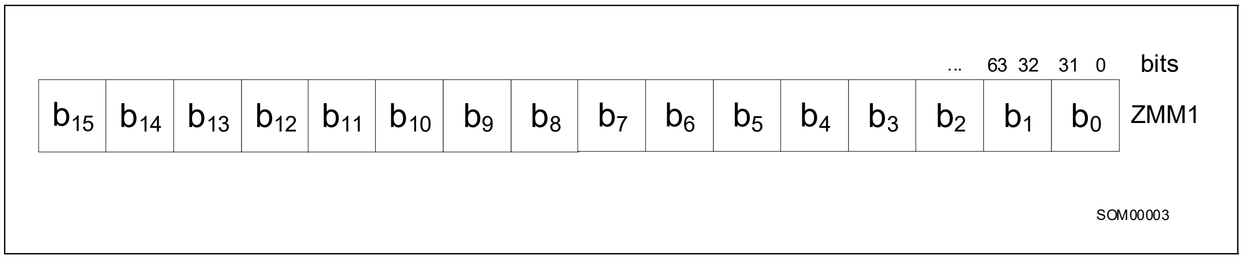


Figure 17-2. Mask Move When Using Merging Masking

The operation is as follows.

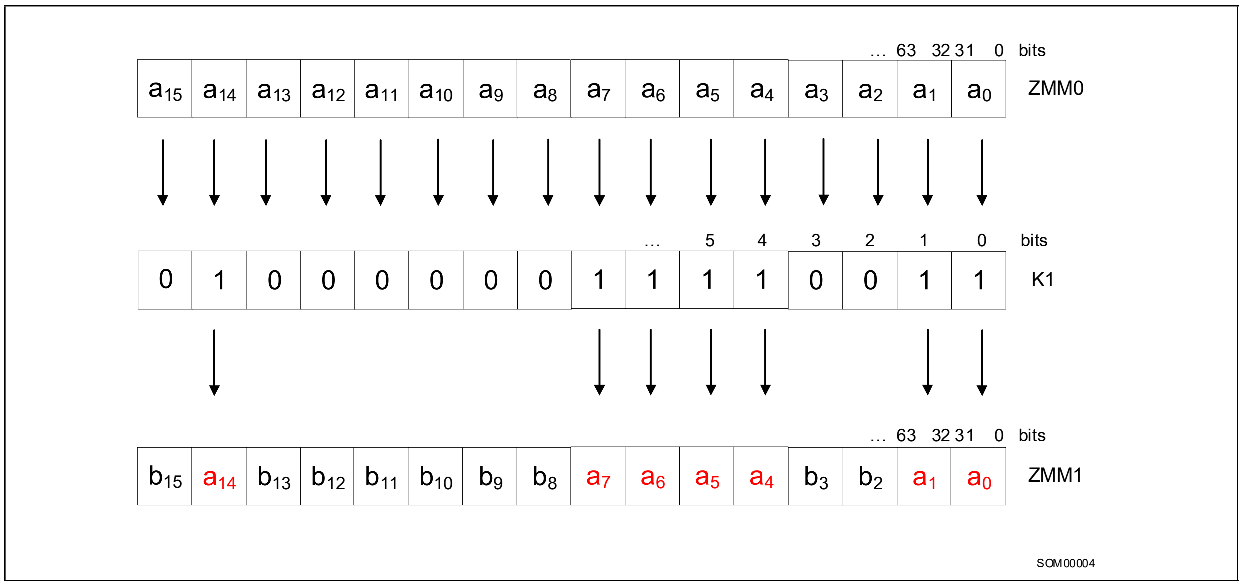


Figure 17-3. Mask Move Operation When Using Merging Masking

The result of the execution with zeroing masking is (notice the {z} in the instruction):

vmovaps zmm1 {k1}{z}, zmm0

.

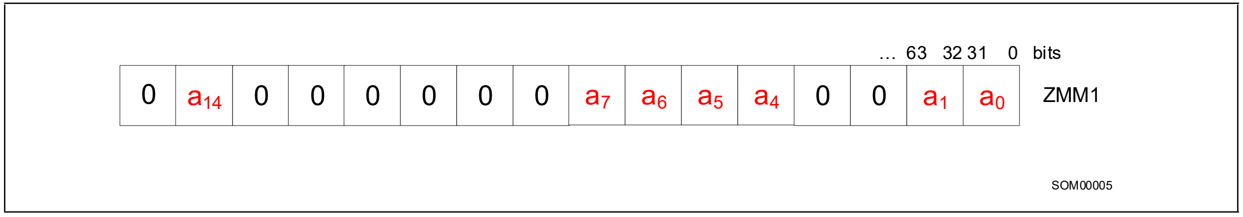


Figure 17-4. Result of Execution with Zeroing Masking

Notice that merging masking operations has a dependency on the destination, but zeroing masking is free of such dependency.

The following example shows how masking could be done with Intel AVX-512 in contrast to Intel AVX2.

C Code:

```
const int N = miBufferWidth;
const double* restrict a = A;
const double* restrict b = B;
double* restrict c = Cref;

for (int i = 0; i < N; i++){
        double res = b[i];
        if(a[i] > 1.0){
            res = res * a[i];
    }
        c[i] = res;
}
```

Example 17-3. Masking with Intrinsics

Intel® AVX2 Intrinsics Code	Intel® AVX-512 Intrinsics Code

```
  for (int i = 0; i < N; i+=32){
    __m256d aa, bb, mask;
    #pragma unroll(8)
    for (int j = 0; j < 8; j++){
      aa  = _mm256_loadu_pd(a+i+j*4);
      bb  = _mm256_loadu_pd(b+i+j*4);
      mask =
_mm256_cmp_pd(_mm256_set1_pd(1.0), aa, 1);
      aa  = _mm256_and_pd(aa, mask); // zero the false
values
      aa  = _mm256_mul_pd(aa, bb);
      bb  = _mm256_blendv_pd(bb, aa, mask);
      _mm256_storeu_pd(c+4*j, bb);
    }

    c += 32;
  }
```
	```
	  for (int i = 0; i < N; i+=32){
	    __m512d aa, bb;
	    __mmask8 mask;
	    #pragma unroll(4)
	    for (int j = 0; j < 4; j++){
	      aa  = _mm512_loadu_pd(a+i+j*8);
	      bb  = _mm512_loadu_pd(b+i+j*8);
	      mask =
	_mm512_cmp_pd_mask(_mm512_set1_pd(1.0), aa, 1);
	      bb  = _mm512_mask_mul_pd(bb, mask, aa, bb);
	      _mm512_storeu_pd(c+8*j, bb);
	    }

	    c += 32;
	  }
	```

Baseline	Speedup: 2.9x

Example 17-4. Masking with Assembly

Intel® AVX2 Assembly Code	Intel® AVX-512 Assembly Code
mov rax, a	
mov r11, b	
mov r8, N	
shr r8, 5	
mov rsi, c	
xor rcx, rcx	mov rax, a
xor r9, r9	mov r11, b
	mov r8, N
loop:	shr r8, 5
vmovupd ymm1, ymmword ptr [rax+rcx*8]	mov rsi, c
inc r9d	
vmovupd ymm6, ymmword ptr [rax+rcx*8+0x20]	xor rcx, rcx
vmovupd ymm2, ymmword ptr [r11+rcx*8]	xor r9, r9
vmovupd ymm7, ymmword ptr [r11+rcx*8+0x20]	mov rdi, 1
vmovupd ymm11, ymmword ptr [rax+rcx*8+0x40]	cvtsi2sd xmm8, rdi
vmovupd ymm12, ymmword ptr [r11+rcx*8+0x40]	vbroadcastsd zmm8, xmm8
vcmppd ymm4, ymm0, ymm1, 0x1	
vcmppd ymm9, ymm0, ymm6, 0x1	loop:
vcmppd ymm14, ymm0, ymm11, 0x1	vmovups zmm0, zmmword ptr [rax+rcx*8]
vandpd ymm16, ymm1, ymm4	inc r9d
vandpd ymm17, ymm6, ymm9	vmovups zmm2, zmmword ptr [rax+rcx*8+0x40]
vmulpd ymm3, ymm16, ymm2	vmovups zmm4, zmmword ptr [rax+rcx*8+0x80]
vmulpd ymm8, ymm17, ymm7	vmovups zmm6, zmmword ptr [rax+rcx*8+0xc0]
vmovupd ymm1, ymmword ptr [rax+rcx*8+0x60]	vmovups zmm1, zmmword ptr [r11+rcx*8]
vmovupd ymm6, ymmword ptr [rax+rcx*8+0x80]	vmovups zmm3, zmmword ptr [r11+rcx*8+0x40]
vblendvpd ymm5, ymm2, ymm3, ymm4	vmovups zmm5, zmmword ptr [r11+rcx*8+0x80]
vblendvpd ymm10, ymm7, ymm8, ymm9	vmovups zmm7, zmmword ptr [r11+rcx*8+0xc0]
vmovupd ymm2, ymmword ptr [r11+rcx*8+0x60]	vcmppd k1, zmm8, zmm0, 0x1
vmovupd ymm7, ymmword ptr [r11+rcx*8+0x80]	vcmppd k2, zmm8, zmm2, 0x1
vmovupd ymmword ptr [rsi], ymm5	vcmppd k3, zmm8, zmm4, 0x1
vmovupd ymmword ptr [rsi+0x20], ymm10	vcmppd k4, zmm8, zmm6, 0x1
vcmppd ymm4, ymm0, ymm1, 0x1	vmulpd zmm1{k1}, zmm0, zmm1
vcmppd ymm9, ymm0, ymm6, 0x1	vmulpd zmm3{k2}, zmm2, zmm3
vandpd ymm18, ymm11, ymm14	vmulpd zmm5{k3}, zmm4, zmm5
vandpd ymm19, ymm1, ymm4	vmulpd zmm7{k4}, zmm6, zmm7
vandpd ymm20, ymm6, ymm9	vmovups zmmword ptr [rsi], zmm1
vmulpd ymm13, ymm18, ymm12	vmovups zmmword ptr [rsi+0x40], zmm3
vmulpd ymm3, ymm19, ymm2	vmovups zmmword ptr [rsi+0x80], zmm5
vmulpd ymm8, ymm20, ymm7	vmovups zmmword ptr [rsi+0xc0], zmm7
vmovupd ymm11, ymmword ptr [rax+rcx*8+0xa0]	add rcx, 0x20
vmovupd ymm1, ymmword ptr [rax+rcx*8+0xc0]	add rsi, 0x100
vmovupd ymm6, ymmword ptr [rax+rcx*8+0xe0]	cmp r9d, r8d
vblendvpd ymm15, ymm12, ymm13, ymm14	jb loop
vblendvpd ymm5, ymm2, ymm3, ymm4	
vblendvpd ymm10, ymm7, ymm8, ymm9	
vmovupd ymm12, ymmword ptr [r11+rcx*8+0xa0]	
vmovupd ymm2, ymmword ptr [r11+rcx*8+0xc0]	
vmovupd ymm7, ymmword ptr [r11+rcx*8+0xe0]	
vmovupd ymmword ptr [rsi+0x40], ymm15	

Example 17-4. Masking with Assembly (Contd.)

Intel® AVX2 Assembly Code	Intel® AVX-512 Assembly Code
vmovupd ymmword ptr [rsi+0x60], ymm5 vmovupd ymmword ptr [rsi+0x80], ymm10 vcmppd ymm14, ymm0, ymm11, 0x1 vcmppd ymm4, ymm0, ymm1, 0x1 vcmppd ymm9, ymm0, ymm6, 0x1 vandpd ymm21, ymm11, ymm14 add rcx, 0x20 vandpd ymm22, ymm1, ymm4 vandpd ymm23, ymm6, ymm9 vmulpd ymm13, ymm21, ymm12 vmulpd ymm3, ymm22, ymm2 vmulpd ymm8, ymm23, ymm7 vblendvpd ymm15, ymm12, ymm13, ymm14 vblendvpd ymm5, ymm2, ymm3, ymm4 vblendvpd ymm10, ymm7, ymm8, ymm9 vmovupd ymmword ptr [rsi+0xa0], ymm15 vmovupd ymmword ptr [rsi+0xc0], ymm5 vmovupd ymmword ptr [rsi+0xe0], ymm10 add rsi, 0x100 cmp r9d, r8d jb loop	
Baseline	Speedup: 2.9x

17.2.2 MASKING COST

Using masking may result in lower performance than the corresponding non-masked code. This may be caused by one of the following situations:

- An additional blend operation on each load.
- Dependency on the destination when using merge masking. This dependency does not exist when using zero masking.
- More restrictive masking forwarding rules (see Forwarding and Memory Masking for more information).

The following example shows how using merge masking creates a dependency on the destination register.

Example 17-5. Masking Example

No Masking	Merge Masking	Zero Masking
mov rbx, iter loop: vmulps zmm0, zmm9, zmm8 vmulps zmm1, zmm9, zmm8 dec rbx jnle loop	mov rbx, iter loop: vmulps zmm0{k1}, zmm9, zmm8 vmulps zmm1{k1}, zmm9, zmm8 dec rbx jnle loop	mov rbx, iter loop: vmulps zmm0{k1}{z}, zmm9, zmm8 vmulps zmm1{k1}{z}, zmm9, zmm8 dec rbx jnle loop
Baseline	Slowdown: 4x	Slowdown: Equal to baseline.

With no masking, the processor executes 2 multiplies per cycle on a 2 FMA server.

With merge masking, the processor executes 2 multiplies every 4 cycles as the multiplies in iteration N depend on the output of the multiplies in iteration N-1.

Zero masking does not have a dependency on the destination register and therefore can execute 2 multiplies per cycle on a 2 FMA server.

Recommendation: Masking has a cost, so use it only when necessary. When possible, use zero masking rather than merge masking.

17.2.3 MASKING VS. BLENDING

This section discusses the advantages and disadvantages of using blending vs. masking for conditional code.

Consider the following code:

```
for ( i=0; i<SIZE; i++ )
{
        if ( a[i] > 0 )
            {
        b[i] *= 2;
            }
        else
            {
        b[i] /= 2;
            }
}
```

Example 17-6 shows two possible compilation alternatives of the code.

- Alternative 1 uses masked code and straight-forward arithmetic processing of data.
- Alternative 2 splits code to two independent unmasked flows that are processed one after another, and then a masked move (blending), just before storing to memory.

Example 17-6. Masking vs. Blending Example 1

Alternative 1	Alternative 2

mov rax, pImage
  mov rbx, pImage1
  mov rcx, pOutImage
  mov rdx, len
  vpxord zmm0, zmm0, zmm0
mainloop:
  vmovdqa32 zmm2, [rax+rdx*4-0x40]
  vmovdqa32 zmm1, [rbx+rdx*4-0x40]
  vpcmpgtd k1, zmm1, zmm0
  knotw k2, k1
	mov rax, pImage
	  mov rbx, pImage1
	  mov rcx, pOutImage
	  mov rdx, len
	  vpxord zmm0, zmm0, zmm0
	mainloop:
	  vmovdqa32 zmm2, [rax+rdx*4-0x40]
	  vmovdqa32 zmm1, [rbx+rdx*4-0x40]
	  vpcmpgtd k1, zmm1, zmm0
	  vmovdqa32 zmm3, zmm2
	  vpslld zmm2, zmm2, 1
	  vpsrld zmm3, zmm3, 1

Example 17-6. Masking vs. Blending Example 1 (Contd.)

Alternative 1	Alternative 2

(1)vpslld zmm2 {k1}, zmm2, 1
(2)vpsrld zmm2 {k2}, zmm2, 1
(3)vmovdqa32 [rcx+rdx*4-0x40], zmm2
  sub rdx, 16
  jne mainloop
	(1)vmovdqa32 zmm3 {k1}, zmm2
	(2)vmovdqa32 [rcx+rdx*4-0x40], zmm3
	  sub rdx, 16
	  jne mainloop

Baseline cycles 1x Baseline instructions 1x	Speedup: 1.23x Instructions: 1.11x

In Alternative 1, there is a dependency between instructions (1) and (2), and (2) and (3). That means that instruction (2) has to wait for the result of the blending of instruction (1), before starting execution, and instruction (3) needs to wait for instruction (2).

In Alternative 2, there is only one such dependency because each branch of conditional code is executed in parallel on all the data, and a mask is used for blending back to one register only before writing data back to the memory.

Blending is faster, but it does not mask exceptions, which may occur on the unmasked data.

Alternative 2 executes 11% more instructions; it provides 23% speedup in overall execution. Alternative 2 uses an extra register (zmm3). This extra register usage may cause extra latency in case of register pressure (freeing register to memory and loading it afterwards).

The following code is another example of masking vs. blending.

```
for (int i = 0;i<len;i++)
{
                                        if (a[i] > b[i]){
                                           a[i] += b[i];
                                                        }
}
```

Example 17-7. Masking vs. Blending Example 2

Alternative 1	Alternative 2

mov rax,a
  mov rbx,b
  mov rdx,size2
loop1:
  vmovdqa32 zmm1,[rax +rdx*4 -0x40]
  vmovdqa32 zmm2,[rbx +rdx*4 -0x40]
(1) vpcmpgtd k1,zmm1,zmm2
(2) vmovdqa32 zmm3{k1}{z},zmm2
(3) vpaddd zmm1,zmm1,zmm3
  vmovdqa32 [rax +rdx*4 -0x40],zmm1
  sub rdx,16
  jne loop1
	mov rax,a
	  mov rbx,b
	  mov rdx,size2
	loop1:
	  vmovdqa32 zmm1,[rax +rdx*4 -0x40]
	  vmovdqa32 zmm2,[rbx +rdx*4 -0x40]
	(1)vpcmpgtd k1,zmm1,zmm2
	(2)vpaddd zmm1{k1},zmm1,zmm2
	  vmovdqa32 [rax +rdx*4 -0x40],zmm1
	  sub rdx,16
	  jne loop1

Baseline cycles 1x Baseline instructions 1x	Speedup: 1.05x Instructions: 0.87x

In Alternative 1, there is a dependency between instructions (1) and (2), and (2) and (3).

In Alternative 2, there are only 2 instructions in the dependency chain: (1) and (2).

17.2.4 NESTED CONDITIONS / MASK AGGREGATION

Intel AVX-512 contains a set of instructions for mask operation, which enable executing all bitwise logical operators on a mask register, facilitating implementation of nested and/or multiply conditions.

In the following example, logical and (&&) is executed using a *kandw* instruction.

```
for(int iX = 0; iX < iBufferWidth; iX++)
{
        if ((*pInImage)>0 && ((*pInImage)&3)==3)
        {
        *pRefImage =  (*pInImage)+5;
        }
        else
        {
        *pRefImage = (*pInImage);
        }

        pRefImage++;
        pInImage++;
}
```

Example 17-8. Multiple Condition Execution

Scalar	Intel® AVX2	Intel® AVX-512
mov rsi, pImage mov rdi, pOutImage mov rbx, len xor rax, rax mainloop: mov r8d, dword ptr [rsi+rax*4] mov r9d, r8d cmp r8d, 0 jle label1 and r9d, 0x3 cmp r9d, 3 jne label1 add r8d, 5 label1: mov dword ptr [rdi+rax*4], r8d add rax, 1 cmp rax, rbx jne mainloop	mov rsi, pImage mov rdi, pOutImage mov rbx, len xor rax, rax vpbroadcastd ymm1, [five] vpbroadcastd ymm7, [three] vpxor ymm3, ymm3, ymm3 mainloop: vmovdqa ymm0, [rsi+rax*4] vmovaps ymm6, ymm0 vpcmpgtd ymm5, ymm0, ymm3 vpand ymm6, ymm6, ymm7 vpcmpeqd ymm6, ymm6, ymm7 vpand ymm5, ymm5, ymm6 vpaddd ymm4, ymm0, ymm1 vblendvps ymm4, ymm0, ymm4, cmp rax, rbx jne mainloop	mov rsi, pImage mov rdi, pOutImage mov rbx, len xor rax, rax vpbroadcastd zmm1, [five] vpbroadcastd zmm5, [three] vpxord zmm3, zmm3, zmm3 mainloop: vmovdqa32 zmm0, [rsi+rax*4] vpcmpgtd k1, zmm0, zmm3 vpandd zmm6, zmm5, zmm0 vpcmpeqd k2, zmm6, zmm5 kandw k1, k2, k1 vpaddd zmm0 {k1}, zmm0, zmm1 vmovdqa32 [rdi+rax*4], zmm0 add rax, 16 cmp rax, rbx jne mainloop

Example 17-8. Multiple Condition Execution (Contd.)

Scalar	Intel® AVX2	Intel® AVX-512
Baseline 1x	Speedup: 5x	Speedup: 11x

17.2.5 MEMORY MASKING MICROARCHITECTURE IMPROVEMENTS

Masking improvements since Broadwell microarchitecture are detailed in Example 17-2

Table 17-2. Cache Comparison Between Skylake Server Microarchitecture and Broadwell Microarchitecture

Item	Broadwell Microarchitecture	Skylake Server Microarchitecture
1	• Address of a **vmaskmov** store is considered as resolved only after the mask is known. • Loads following a masked store may be blocked, depending on the memory disambiguation predictor, until the mask value is known.	• Issue is resolved. • Address of a vmaskmov store can be resolved before mask is known.
2	• **If mask is not all 1 or all 0**: loads depending upon the masked store must wait until the store data is written to the cache. • **If mask is all 1**: the data can be forwarded from the masked store to the dependent loads. • **If mask is all 0:** loads do not depend on the masked store.	• **If mask is not all 1 or all 0**: loads that depend on the masked store must wait until store data is written tocache. • **If mask is all 1**: data can be forwarded from the masked store to the dependent loads. • **If mask is all 0:** loads do not depend on the masked store.
3	• When including an illegal memory address range with masked loads (using the vmaskmov instruction): processor might take a multi-cycle "assist" to determine if any part of the illegal range has a one mask value. • Assist might occur even when mask was "all-zero" and seemed obvious that the load should not be executed.	**For Intel AVX-512 masking**: if mask is all-zeros then memory faults will be ignored and no assist will be issued.

17.2.6 PEELING AND REMAINDER MASKING

Accessing cache line aligned data gives better performance than accessing non-aligned data. In many cases, the address is not known in compile time, or known and not-aligned. In these cases a peeling algorithm may be proposed, to process first elements in masked mode, up to first aligned address, and then process unmasked body and masked remainder. This method increases code size, but improves data processing overall.

The following code is an example of peeling and remainder masking.

```
for (size_t i = 0; i < len; i++)
                                        pOutImage[i] = (pInImage[i] * alfa) + add_value;
```

The table below shows the difference in implementation and execution speed of two versions of the code, both working on unaligned output data array.

Example 17-9. Peeling and Remainder Masking

No peeling, unmasked body, masked remainder	Peeling, unmasked body, masked remainder
	mov rax, pImage // Input
mov rbx, pOutImage // Output	mov rbx, pOutImage // Output
mov rax, pImage // Input	mov rcx, len
mov rcx, len	movss xmm0, addValue
mov edx, addValue	vpbroadcastd zmm0, xmm0
vpbroadcastd zmm0, edx	movss xmm1, alfa
mov edx, alfa	vpbroadcastd zmm3, xmm1
vpbroadcastd zmm3, edx	xor r8, r8
mov rdx, rcx	xor r9, r9
sar rdx, 4 // 16 elements per iteration, RDX - number	vmovups zmm10, [indices]
of full iterations	vpbroadcastd zmm12, ecx
jz remainder // no full iterations	
xor r8, r8	peeling:
vmovups zmm10, [indices]	mov rdx, rbx
	and rdx, 0x3F
mainloop:	jz endofpeeling //nothing to peel
vmovups zmm1, [rax + r8]	neg rdx
vfmadd213ps zmm1, zmm3, zmm0	add rdx, 64 // 64 - X
vmovups [rbx + r8], zmm1	// now rdx contains the number of bytes to the closest
add r8, 0x40	alignment
sub rdx, 1	mov r9, rdx
jne mainloop	sar r9, 2 // now r9 contains number of elements in
	peeling
remainder:	
// produce mask for remainder	vpbroadcastd zmm12, r9d
and rcx, 0xF // number of elements in remainder	vpcmpd k2, zmm10, zmm12, 1 //compare lower to
jz end // no elements in remainder	produce mask for peeling
vpbroadcastd zmm2, ecx	
vpcmpd k2, zmm10, zmm2, 1 //compare lower	vmovups zmm1 {k2}{z}, [rax]
	vfmadd213ps zmm1 {k2}{z}, zmm3, zmm0
vmovups zmm1 {k2}{z}, [rax + r8]	vmovups [rbx] {k2}, zmm1 //unaligned store
vfmadd213ps zmm1 {k2}{z}, zmm3, zmm0	
vmovups [rbx + r8] {k2}, zmm1	endofpeeling:
end:	sub rcx, r9
	mov r8, rcx
	sar r8, 4 //number of full iterations
	jz remainder //no full iterations

Example 17-9. Peeling and Remainder Masking (Contd.)

No peeling, unmasked body, masked remainder	Peeling, unmasked body, masked remainder

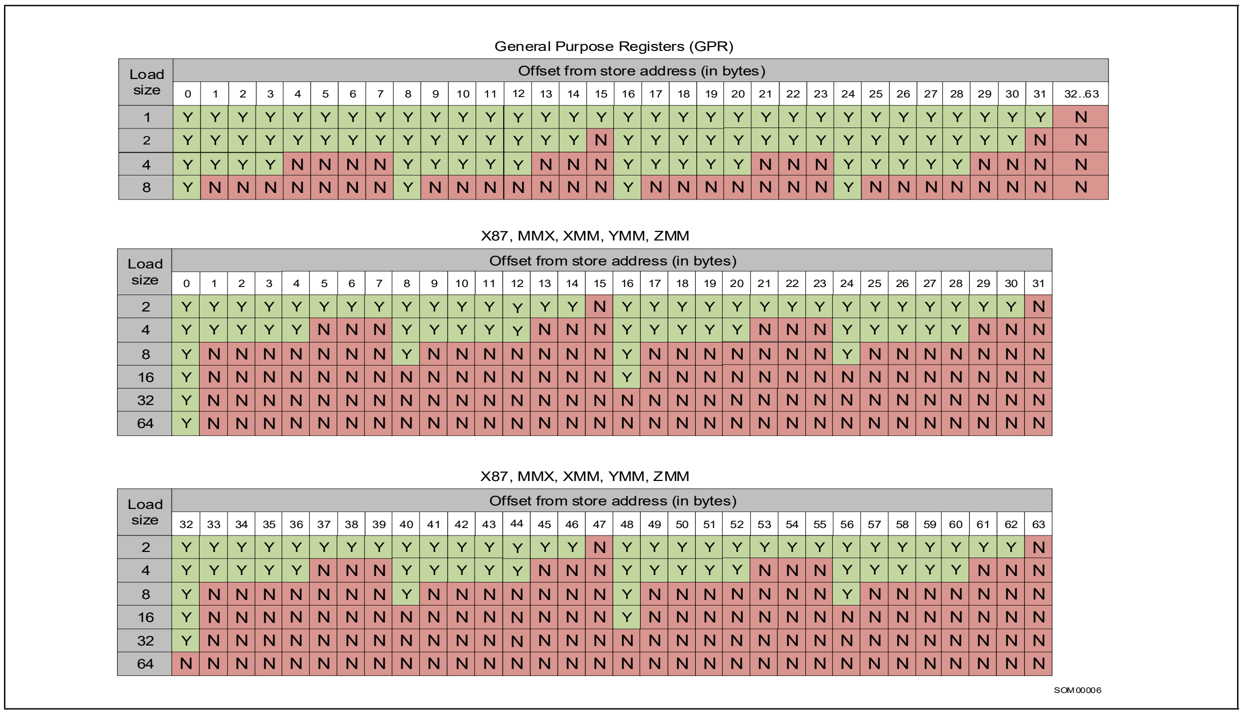
	mainloop:
	  vmovups zmm1, [rax + rdx]
	  vfmadd213ps zmm1, zmm3, zmm0
	  vmovaps [rbx + rdx], zmm1 // aligned store is safe here !!
	  add rdx, 0x40
	  sub r8, 1
	  jne mainloop
	remainder:
	  // produce mask for remainder
	  and rcx, 0xF // number of elements in remainder
	  jz end // no elements in remainder
	  vpbroadcastd zmm2, ecx
	    vpcmpd k2, zmm10, zmm2, 1 //compare lower
	  vmovups zmm1 {k2}{z}, [rax + rdx]
	  vfmadd213ps zmm1 {k2}{z}, zmm3, zmm0
	  vmovaps [rbx + rdx] {k2}, zmm1 //aligned
	end:

Baseline 1x	Speedup: 1.04x

17.3 FORWARDING AND UNMASKED OPERATIONS

When using an unmasked store instruction, and load instruction after it, data forwarding depends on load type, size and address offset from store address, and does not depend on the store address itself (i.e., the store address does not have to be aligned to or fit into cache line, forwarding will occur for non-aligned and even line-split stores).

The figure below describes all possible cases when data forwarding will occur.



General Purpose Registers (GPR)

Load size	Offset from store address (in bytes)																															
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32..63
1	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	N
2	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	N	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	N	N
4	Y	Y	Y	Y	N	N	N	N	Y	Y	Y	Y	Y	N	N	N	Y	Y	Y	Y	Y	N	N	N	Y	Y	Y	Y	Y	N	N	N	N
8	Y	N	N	N	N	N	N	N	Y	N	N	N	N	N	N	N	Y	N	N	N	N	N	N	N	Y	N	N	N	N	N	N	N	N

X87, MMX, XMM, YMM, ZMM

Load size	Offset from store address (in bytes)																														
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	N	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	N
4	Y	Y	Y	Y	Y	N	N	N	Y	Y	Y	Y	Y	N	N	N	Y	Y	Y	Y	Y	N	N	N	Y	Y	Y	Y	Y	N	N	N
8	Y	N	N	N	N	N	N	N	Y	N	N	N	N	N	N	N	Y	N	N	N	N	N	N	N	Y	N	N	N	N	N	N	N
16	Y	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	Y	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
32	Y	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
64	Y	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N

X87, MMX, XMM, YMM, ZMM

Load size	Offset from store address (in bytes)																														
	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
2	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	N	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	N
4	Y	Y	Y	Y	Y	N	N	N	Y	Y	Y	Y	Y	N	N	N	Y	Y	Y	Y	Y	N	N	N	Y	Y	Y	Y	Y	N	N	N
8	Y	N	N	N	N	N	N	N	Y	N	N	N	N	N	N	N	Y	N	N	N	N	N	N	N	Y	N	N	N	N	N	N	N
16	Y	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	Y	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
32	Y	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
64	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N

SOM00006

Figure 17-5. Data Forwarding Cases

There are two important points to be considered when using data forwarding.

1. Data forwarding to GPR is possible only from the lower 256 bits of store instruction. Note this when loading GPR with data that has recently been written.
2. Do not use masks, as forwarding is supported only for certain masks.

17.4 FORWARDING AND MEMORY MASKING

When using masked store and load, consider the following:

- When the mask is not all-ones or all-zeroes, the load operation, following the masked store operation from the same address is blocked, until the data is written to the cache.
- Unlike GPR forwarding rules, vector loads whether or not they are masked, do not forward unless load and store addresses are exactly the same.
  - st_mask = 10101010, ld_mask = 01010101, can forward: no, should block: yes
  - st_mask = 00001111, ld_mask = 00000011, can forward: no, should block: yes
- When the mask is all-ones, blocking does not occur, because the data may be forwarded to the load operation.
  - st_mask = 11111111, ld_mask = don’t care, can forward: yes, should block: no

- When mask is all-zeroes, blocking does not occur, though neither does forwarding.
  - — st_mask = 00000000, ld_mask = don't care, can forward: no, should block: no

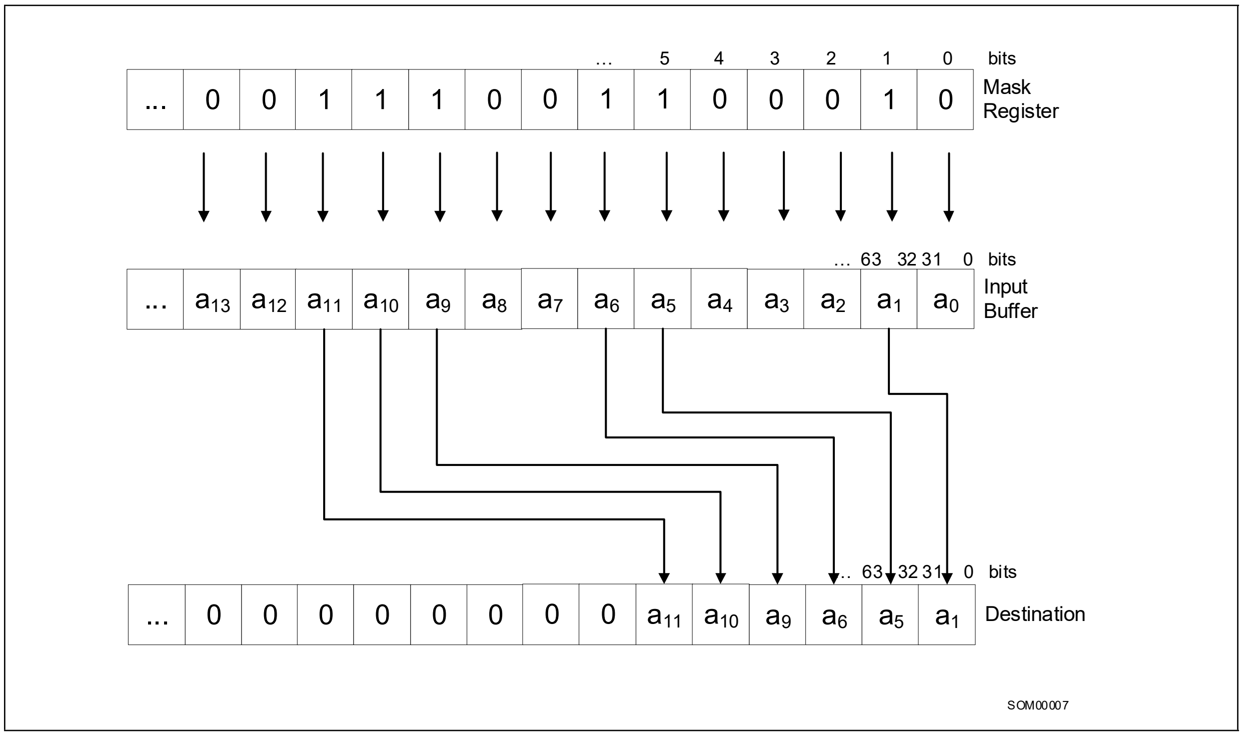
In summary, a masked store should be used carefully, for example, if the remainder size is known at compile time to be 1, and there is a load operation from the same cache line after it (or there is an overlap in addresses + vector lengths), it may be better to use scalar remainder processing, rather than a masked remainder block.

17.5 DATA COMPRESS

The data compress operation reads elements from an input buffer on indices specified by mask register 1's bits. The elements which have been read, are then written to the destination buffer. If the number of elements is less than the destination register size, the rest of the space is filled with zeros.

The following figure describes the data compress operation.

```
if (k[i] == 1)
{
        dest[a] = src[i];
                a++;
}
```



…
5
4
3
2
1
0
bits
… 0 0 1 1 1 0 0 1 1 0 0 0 1 0
Mask Register
… 63 32 31 0 bits
… a_{13} a_{12} a_{11} a_{10} a_9 a_8 a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0
Input Buffer
… 63 32 31 0 bits
… 0 0 0 0 0 0 0 0 a_{11} a_{10} a_9 a_6 a_5 a_1
Destination
SOM00007

Figure 17-6. Data Compress Operation

17.5.1 DATA COMPRESS EXAMPLE

The following snippet shows collection of all positive elements from one array to another array.

```
for (int i=0; i<SIZE; i++)
{
            if ( a[i] > 0 )
            b[j++] = a[i];
}
```

Following are four implementations for the compress operation from an array of dword elements.

- **Alternative 1:** uses scalar data access and checks each element separately. If it is greater than 0 it is written to the destination array.
- **Alternative 2:** Intel AVX code that uses a shuffle instruction together with the pre-allocated and pre-initialized table with shuffle keys. The compare instruction provides the entry point number to the shuffle-key table. Then the key is loaded and the original array is shuffled according to the keys. Four elements are processed in each iteration.
- **Alternative 3:** uses the same algorithm as in Alternative 2, but uses Intel AVX2 256-bit registers, and a permutation on the dword instruction instead of using byte shuffle. Eight elements are processed in each iteration.
- **Alternative 4:** an Intel AVX-512 algorithm, which uses the *vpcompress* instruction together with the mask register as a compress key. 16 elements are processed in each iteration

Example 17-10. Comparing Intel® AVX-512 Data Compress with Alternative 1

Alternative 1: Scalar
```
    mov rsi, source
    mov rdi, dest
    mov r9, len

    xor r8, r8
    xor r10, r10
mainloop:
    mov r11d, dword ptr [rsi+r8*4]
    test r11d, r11d
    jle  m1
    mov dword ptr [rdi+r10*4], r11d
    inc r10
m1:
    inc r8
    cmp r8, r9
    jne mainloop
```
Baseline 1x
Speedup: 2.87x

Example 17-11. Comparing Intel® AVX-512 Data Compress with Alternative 2

Alternative 2: Intel® AVX

```
        mov rsi, source
        mov rdi, dest
        mov r14, shuffle_LUT
        mov r15, write_mask
        mov r9, len

        xor r8, r8
        xor r11, r11
        vpxor xmm0, xmm0, xmm0
mainloop:
        vmovdqa xmm1, [rsi+r8*4]
        vpcmpgtd xmm2, xmm1, xmm0
        mov r10, 4
        vmovmskps r13, xmm2
        shl r13, 4
        vmovdqu xmm3, [r14+r13]
        vpshufb xmm2, xmm1, xmm3
        popcnt r13, r13
        sub r10, r13
        vmovdqu xmm3, [r15+r10*4]
        vmaskmovps [rdi+r11*4], xmm3, xmm2
        add r11, r13
        add r8, 4
        cmp r8, r9
        jne mainloop

shuffle_LUT:
  .int 0x80808080, 0x80808080, 0x80808080, 0x80808080
  .int 0x03020100, 0x80808080, 0x80808080, 0x80808080
  .int 0x07060504, 0x80808080, 0x80808080, 0x80808080
  .int 0x03020100, 0x07060504, 0x80808080, 0x80808080
  .int 0x0b0A0908, 0x80808080, 0x80808080, 0x80808080
  .int 0x03020100, 0x0b0A0908, 0x80808080, 0x80808080
  .int 0x07060504, 0x0b0A0908, 0x80808080, 0x80808080
  .int 0x03020100, 0x07060504, 0x0b0A0908, 0x80808080
  .int 0x0F0E0D0C, 0x80808080, 0x80808080, 0x80808080
  .int 0x03020100, 0x0F0E0D0C, 0x80808080, 0x80808080
  .int 0x07060504, 0x0F0E0D0C, 0x80808080, 0x80808080
  .int 0x03020100, 0x07060504, 0x0F0E0D0C, 0x80808080
  .int 0x0b0A0908, 0x0F0E0D0C, 0x80808080, 0x80808080
  .int 0x03020100, 0x0b0A0908, 0x0F0E0D0C, 0x80808080
  .int 0x07060504, 0x0b0A0908, 0x0F0E0D0C, 0x80808080
  .int 0x03020100, 0x07060504, 0x0b0A0908, 0x0F0E0D0C

write_mask:
  .int 0x80000000, 0x80000000, 0x80000000, 0x80000000
  .int 0x00000000, 0x00000000, 0x00000000, 0x00000000
```

Speedup: 2.87x

Example 17-12. Comparing Intel® AVX-512 Data Compress with Alternative 3

Alternative 3: Intel® AVX2

```
        mov rsi, source
        mov rdi, dest
        mov r14, shuffle_LUT
        mov r15, write_mask
        mov r9, len

        xor r8, r8
        xor r11, r11
        vpxor ymm0, ymm0, ymm0
mainloop:
        vmovdqa ymm1, [rsi+r8*4]
        vpcmpgtd ymm2, ymm1, ymm0
        mov r10, 8
        vmovmskps r13, ymm2
        shl r13, 5
        vmovdqu ymm3, [r14+r13]
        vpermd ymm2, ymm3, ymm1
        popcnt r13, r13
        sub r10, r13
        vmovdqu ymm3, [r15+r10*4]
        vmaskmovps [rdi+r11*4], ymm3, ymm2
        add r11, r13
        add r8, 8
        cmp r8, r9
        jne mainloop

// The lookup table is too large to reproduce in the document. It consists of 256 rows of 8 32 bit integers.
//The first 8 and the last 8 rows are shown below.

shuffle_LUT:
.int 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0
.int 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0
.int 0x1, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0
.int 0x0, 0x1, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0
.int 0x2, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0
.int 0x0, 0x2, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0
.int 0x1, 0x2, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0
.int 0x0, 0x1, 0x2, 0x0, 0x0, 0x0, 0x0, 0x0
// Skipping 240 lines
.int 0x3, 0x4, 0x5, 0x6, 0x7, 0x0, 0x0, 0x0
.int 0x0, 0x3, 0x4, 0x5, 0x6, 0x7, 0x0, 0x0
.int 0x1, 0x3, 0x4, 0x5, 0x6, 0x7, 0x0, 0x0
.int 0x0, 0x1, 0x3, 0x4, 0x5, 0x6, 0x7, 0x0
.int 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x0, 0x0
.int 0x0, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x0
.int 0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x0
.int 0x0, 0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7
write_mask:
.int 0x80000000, 0x80000000, 0x80000000, 0x80000000
.int 0x80000000, 0x80000000, 0x80000000, 0x80000000
```

Example 17-12. (Contd.)Comparing Intel® AVX-512 Data Compress with Alternative 3 (Contd.)

Alternative 3: Intel® AVX2
.int 0x00000000, 0x00000000, 0x00000000, 0x00000000 .int 0x00000000, 0x00000000, 0x00000000, 0x00000000
Speedup: 5.27x

Example 17-13. Comparing Intel® AVX-512 Data Compress with Alternative 4

Alternative 4: Intel® AVX-512

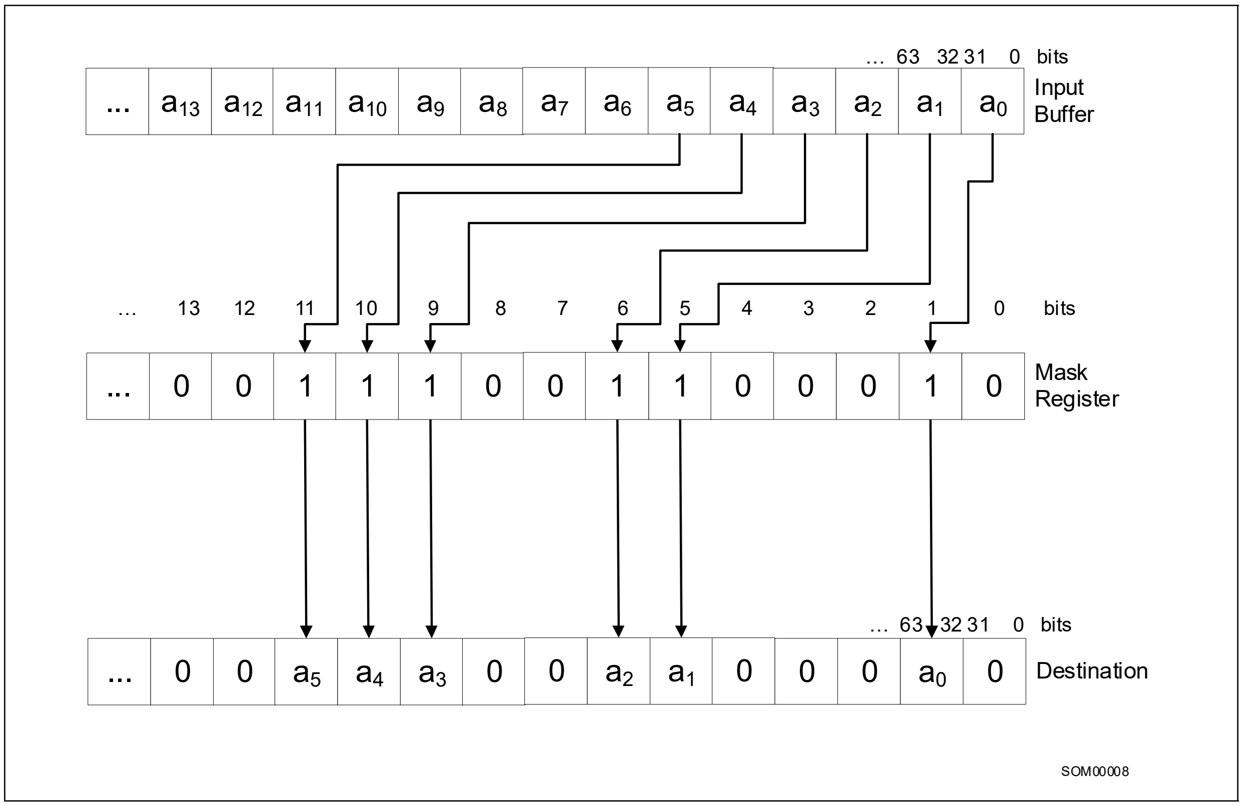
```
    mov rsi, source
    mov rdi, dest
    mov r9, len

    xor r8, r8
    xor r10, r10
    vpxord zmm0, zmm0, zmm0
mainloop:
    vmovdqa32 zmm1, [rsi+r8*4]
    vpcmpgtd k1, zmm1, zmm0
    vpcompressd zmm2 {k1}, zmm1
    vmovdqu32 [rdi+r10*4], zmm2
    kmovd r11d, k1
    popcnt r12, r11
    add r8, 16
    add r10, r12
    cmp r8, r9
    jne mainloop
```

Speedup: 11.9x

17.6 DATA EXPAND

Data expand operations read elements from the source array (register) and put them in the destination register in the places indicated by enabled bits in the mask register. If the number of enabled bits is less than destination register size, the extra values are ignored.

```
if (k[i] == 1)
{
        dest[i] = src[a];
            a++;
}
```



… 63 32 31 0 bits
… a_{13} a_{12} a_{11} a_{10} a_9 a_8 a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0
Input Buffer
… 13 12 11 10 9 8 7 6 5 4 3 2 1 0 bits
… 0 0 1 1 1 0 0 1 1 0 0 0 1 0
Mask Register
… 63 32 31 0 bits
… 0 0 a_5 a_4 a_3 0 0 a_2 a_1 0 0 0 a_0 0
Destination
SOM00008

Figure 17-7. Data Expand Operation

17.6.1 DATA EXPAND EXAMPLE

The following snippet shows an example of using the expand operation. For every positive number in an array, the code sets its consecutive number among positives.

```
for (int i=0; i<SIZE; i++)
{
        if (a[i] > 0)
        dest[i] = a[count++];
        else
        dest[i] = 0;
}
```

Here are three implementations for the expand operation from an array of 16 dword elements.

- **Alternative 1:** uses scalar data access and checks each element separately. If it is greater than 0 then the corresponding element in the destination array is rewritten with the value from source value at index count, and the counter is incremented.
- **Alternative 2:** shows Intel AVX2 code that uses a shuffle instruction together with the preallocated and preinitialized table with shuffle keys.
  - — The compare instruction provides the entry point number to the shuffle-key table.
  - — The key is then loaded and the original array is shuffled according to the keys.

  — Four elements are processed in each iteration.

- **Alternative 3:** shows Intel AVX-512 code, which uses the vpexpandd instruction together with the mask register as an expand key. Sixteen elements are processed in each iteration.

Example 17-14. Comparing Intel® AVX-512 Data Expand Operation with Other Alternatives

Alternative 1: Scalar	Alternative 2: Intel® AVX2	Alternative 3: Intel® AVX-512

mov rsi, input
    mov rdi, output
    mov r9, len
    xor r8, r8
    xor r10, r10
mainloop:
    mov r11d, dword ptr [rsi+r8*4]
    test r11d, r11d
    jle m1
    mov r11d, dword ptr [rsi+r10*4]
    mov dword ptr [rdi+r8*4], r11d
    inc r10
m1:
    inc r8
    cmp r8, r9
    jne mainloop
	mov rsi, input
	    mov rdi, output
	    mov r9, len
	    xor r8, r8
	    xor r10, r10
	    vpxor ymm0, ymm0, ymm0
	    mov r14, shuf2
	mainloop:
	    vmovdqa ymm1, [rsi+r8*4]
	    vpxor ymm4, ymm4, ymm4
	    vpcmpgtd ymm2, ymm1, ymm0
	    vmovdqu ymm1, [rsi+r10*4]
	    vmovmskps r13, ymm2
	    shl r13, 5
	    vmovdqa ymm3, [r14+r13]
	    vpermd ymm4, ymm3, ymm1
	    popcnt r13, r13
	    add r10, r13
	    vmaskmovps [rdi+r8*4], ymm2, ymm4
	    add r8, 8
	    cmp r8, r9
	    jne mainloop
	// The lookup table is too large to
	// reproduce in the document. It
	// consists of 256 rows of 8 32-bit
	// integers. The first 8 and the last 8
	// rows are shown below. The table
	// needs to be 32-byte aligned.
	shuf2:
	  .int 0, 0, 0, 0, 0, 0, 0, 0
	  .int 0, 0, 0, 0, 0, 0, 0, 0
	  .int 0, 0, 0, 0, 0, 0, 0, 0
	  .int 0, 1, 0, 0, 0, 0, 0, 0
	  .int 0, 0, 0, 0, 0, 0, 0, 0
	  .int 0, 0, 1, 0, 0, 0, 0, 0
	  .int 0, 0, 1, 0, 0, 0, 0, 0
	  .int 0, 1, 2, 0, 0, 0, 0, 0
		vpxord zmm0, zmm0, zmm0
		mainloop:
		    vmovdqa32 zmm1, [rsi+r8*4]
		    vpcmpgtd k1, zmm1, zmm0
		    vmovdqu32 zmm1, [rsi+r10*4]
		    vpexpandd zmm2 {k1}{z}, zmm1
		    vmovdqu32 [rdi+r8*4], zmm2
		    add r8, 16
		    kmovd r11d, k1
		    popcnt r12, r11
		    add r10, r12
		    cmp r8, r9
		    jne mainloop

Example 17-14. (Contd.)Comparing Intel® AVX-512 Data Expand Operation with Other Alternatives

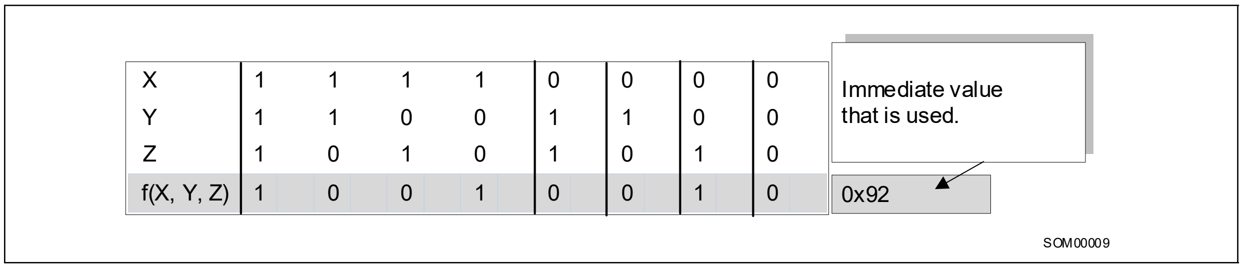
Alternative 1: Scalar	Alternative 2: Intel® AVX2	Alternative 3: Intel® AVX-512
	// Skipping 240 lines .int 0, 0, 0, 0, 1, 2, 3, 4 .int 0, 0, 0, 1, 2, 3, 4, 5 .int 0, 0, 0, 1, 2, 3, 4, 5 .int 0, 1, 0, 2, 3, 4, 5, 6 .int 0, 0, 0, 1, 2, 3, 4, 5 .int 0, 0, 1, 2, 3, 4, 5, 6 .int 0, 0, 1, 2, 3, 4, 5, 6 .int 0, 1, 2, 3, 4, 5, 6, 7	
Baseline 1x	Speedup: 4.23x	Speedup: 8.58x

17.7 TERNARY LOGIC

A ternary logic *vpternlog* operation executes any bitwise logical function between three operands in one instruction. The instruction requires three operands and an immediate value, which is the truth table of this logical expression. The first operand is used as destination, and, therefore, destroyed after the execution.

17.7.1 TERNARY LOGIC EXAMPLE 1

The following example shows a bitwise logic function of three variables. The function in this example is defined by the following truth table.



X	1	1	1	1	0	0	0	0
Y	1	1	0	0	1	1	0	0
Z	1	0	1	0	1	0	1	0
f(X, Y, Z)	1	0	0	1	0	0	1	0

Immediate value that is used.
0x92
SOM00009

Figure 17-8. Ternary Logic Example 1 Truth Table

Using Karnaugh maps on this truth table, we can define the function as:

f(X,Y,Z) = $\boldsymbol{\bar{y}(z \oplus x) \vee xyz}$

The C code for the function above is as follows:

```
for (int i=0; i<SIZE; i++)
{
    Dst[i] = ((~Src2[i]) & (Src1[i] ^ Src3[i])) | (Src1[i] & Src2[i] & Src3[i]);
}
```

The value of the function for each combination of X, Y and Z gives an immediate value that is used in the instruction.

Here are three implementations for this logical function applied to all values in X, Y and Z arrays.

- **Alternative 1:** an Intel AVX2 256-bit vector computation, using bitwise logical functions available in Intel AVX2.

- **Alternative 2:** a 512-bit vector computation, using bitwise logical functions available in Intel AVX-512, without using the *vpternlog* instruction.
- **Alternative 3:** an Intel AVX-512 512-bit vector computation, using the *vpternlog* instruction.

All alternatives in the table are unrolled by factor 2.

Example 17-15. Comparing Ternary Logic to Alternative 1

Alternative 1: Intel® AVX2
```
  mov rax, src1
  mov rbx, src2
  mov rcx, src3
  mov r11, dst
  mov r8, len
  xor r10, r10
mainloop:
  vmovdqu ymm1, ymmword ptr [rax+r10*4]
  vmovdqu ymm3, ymmword ptr [rdx+r10*4]
  vmovdqu ymm2, ymmword ptr [rcx+r10*4]
  vmovdqu ymm10, ymmword ptr [rcx+r10*4+0x20]
  vpand ymm0, ymm1, ymm3

  vpxor ymm4, ymm1, ymm2
  vpand ymm5, ymm0, ymm2
  vpandn ymm6, ymm3, ymm4
vpor ymm7, ymm5, ymm6
  vmovdqu ymmword ptr [r11+r10*4], ymm7
  vmovdqu ymm9, ymmword ptr [rax+r10*4+0x20]
  vmovdqu ymm11, ymmword ptr [rdx+r10*4+0x20]
  vpxor ymm12, ymm9, ymm10
  vpand ymm8, ymm9, ymm11
  vpandn ymm14, ymm11, ymm12
  vpand ymm13, ymm8, ymm10
  vpor ymm15, ymm13, ymm14
  vmovdqu ymmword ptr [r11+r10*4+0x20], ymm15

  add r10, 0x10
  cmp r10, r8
  jb mainloop
```
Baseline 1x

Example 17-16. Comparing Ternary Logic to Alternatives 2 and 3

Alternative 2: Intel® AVX-512 Logic Instructions	Alternative 3: Intel® AVX-512 using *vpternlog* Instruction
mov rdi, src1 mov rsi, src2 mov rdx, src3 mov r11, dst mov r8, len xor r10, r10 mainloop: vmovups zmm2, zmmword ptr [rdi+r10*4] vmovups zmm4, zmmword ptr [rdi+r10*4+0x40] vmovups zmm6, zmmword ptr [rsi+r10*4] vmovups zmm8, zmmword ptr [rsi+r10*4+0x40] vmovups zmm3, zmmword ptr [rdx+r10*4] vmovups zmm5, zmmword ptr [rdx+r10*4+0x40] vpandd zmm0, zmm2, zmm6 vpandd zmm1, zmm4, zmm8 vpxord zmm7, zmm2, zmm3 vpxord zmm9, zmm4, zmm5 vpandd zmm10, zmm0, zmm3 vpandd zmm12, zmm1, zmm5 vpandnd zmm11, zmm6, zmm7 vpandnd zmm13, zmm8, zmm9 vpord zmm14, zmm10, zmm11	mov r9, src1 mov r8, src2 mov r10, src3 mov r11, dst mov rsi, len xor rax rax mainloop: vmovaps zmm1, [r8+rax*4] vmovaps zmm0, [r9+rax*4] vpternlogd zmm0,zmm1,[r10], 0x92 vmovaps [r11], zmm0 vmovaps zmm1, [r8+rax*4+0x40] vmovaps zmm0, [r9+rax*4+0x40] vpternlogd zmm0,zmm1, [r10+0x40], 0x92 vmovaps [r11+0x40], zmm0 addrax, 32 add r10, 0x80 add r11, 0x80 cmp rax, rsi jne mainloop
vpord zmm15, zmm12, zmm13 vmovups zmmword ptr [r11+r10*4], zmm14 vmovups zmmword ptr [r11+r10*4+0x40], zmm15 add r10, 0x20 cmp r10, r9 jb mainloop	
Speedup: 1.94x	Speedup: 2.36x (1.22x vs Intel® AVX-512 with logic instructions)

17.7.2 TERNARY LOGIC EXAMPLE 2

The next example is a sign change operation, frequently used in Fortran. Consider the following code, running on two arrays of floating point numbers.

```
for (int i=0; i<SIZE; i++)
{
                    b[i] = a[i] > 0 ? b[i] : -b[i];
}
```

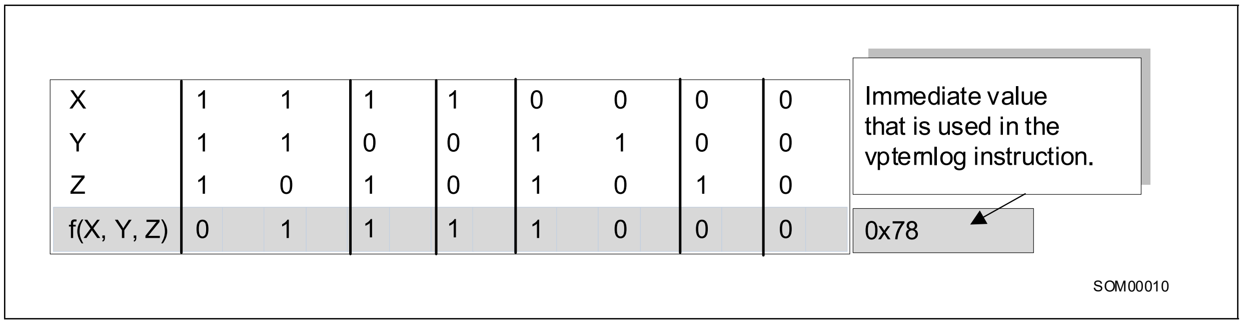
This code is equivalent to:

```
for (int i=0; i<SIZE; i++)
{
    b[i] = ( a[i] & 0x80000000 ) ^ b[i];
}
```

Or, in other words:

$x = (y \wedge z) \oplus x$

This logic expression gives the following truth table.



X	1	1	1	1	0	0	0	0
Y	1	1	0	0	1	1	0	0
Z	1	0	1	0	1	0	1	0
f(X, Y, Z)	0	1	1	1	1	0	0	0

Immediate value that is used in the vpternlog instruction.
0x78
SOM00010

Figure 17-9. Ternary Logic Example 2 Truth Table

Therefore one *vpternlog* instruction can be used instead of using two logic instructions (*vpand* and *vpxor*):

vpternlog x,y,z,0x78

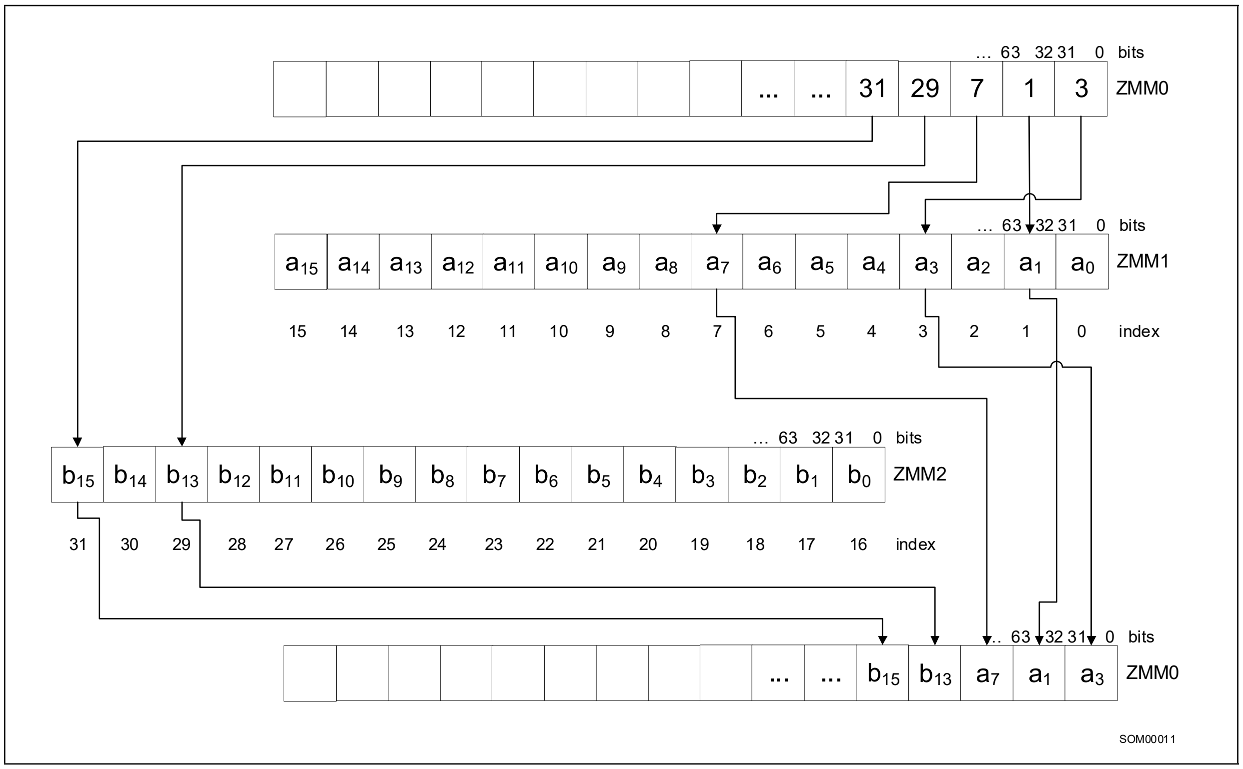
17.8 NEW SHUFFLE INSTRUCTIONS

Intel AVX-512 added 3 new shuffle operations.

- vpermw: a new single source any-to-any word permute.
- permt2[w/d/q/ps/pd]: a new any to any 2 source permute (overriding src register).
- permi2[w/d/q/ps/pd]: a new any to any 2 source permute (overriding control register).

The following figure shows how *vpermi2ps* is used. Notice that in the following example zmm0 is the shuffle control but also the output register (the control register is overridden).

vpermi2ps zmm0, zmm1, zmm2



... 63 32 31 0 bits
... ... 31 29 7 1 3
ZMM0
... 63 32 31 0 bits
a_{15} a_{14} a_{13} a_{12} a_{11} a_{10} a_9 a_8 a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0
ZMM1
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 index
... 63 32 31 0 bits
b_{15} b_{14} b_{13} b_{12} b_{11} b_{10} b_9 b_8 b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0
ZMM2
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 index
... 63 32 31 0 bits
... ... b_{15} b_{13} a_7 a_1 a_3
ZMM0
SOM00011

Figure 17-10. VPERMI2PS Instruction Operation

Note that the index register values must have the same resolution as the instruction and source registers (word when working on words, dword when working on dwords, etc.).

17.8.1 TWO SOURCE PERMUTE EXAMPLE

In this example we will show the use of the two source permute instructions in a matrix transpose operation. The matrix we want to transpose is square 8x8 matrix of word elements.

$$\begin{bmatrix} a_{00} & \cdots & a_{17} \\ \vdots & \ddots & \vdots \\ a_{71} & \cdots & a_{77} \end{bmatrix}^{T} \longrightarrow \begin{bmatrix} a_{00} & \cdots & a_{71} \\ \vdots & \ddots & \vdots \\ a_{17} & \cdots & a_{77} \end{bmatrix}$$

Figure 17-11. Two-Source Permute Instructions in a Matrix Transpose Operation

The corresponding C code is as follows (assuming each matrix occupies a continuous block of 8*8*2 = 128 bytes):

```
for(int iY = 0; iY < 8; iY++)
                                                    {
for(int iX = 0; iX < 8; iX++)
 {
                    trasposedMatrix[iY*8+iX] = originalMatrix[iX*8+iY];
                                                    }
}
```

Here are three implementations for this matrix transpose.

- Alternative 1 is scalar code, which accesses each element of the source matrix and puts it to the corresponding place in the destination matrix. This code does 64 (8x8) iterations per 1 matrix.
- Alternative 2 is Intel AVX2 code, which uses Intel AVX2 permutation and shuffle (unpack) instructions. Only 1 iteration per 8x8 matrix is required.
- Alternative 3 Intel AVX-512 code which uses the Two Source Permutation instructions. Note that this code first loads permutation masks, and then matrix data. The mask used to perform the permutation is stored in the following array:

```
short permMaskBuffer [8*8] = { 0, 8, 16, 24, 32, 40, 48, 56,
                                1, 9, 17, 25, 33, 41, 49, 57,
                               2, 10, 18, 26, 34, 42, 50, 58,
                               3, 11, 19, 27, 35, 43, 51, 59,
                               4, 12, 20, 28, 36, 44, 52, 60,
                               5, 13, 21, 29, 37, 45, 53, 61,
                               6, 14, 22, 30, 38, 46, 54, 62,
                                7, 15, 23, 31, 39, 47, 55, 63 };
```

Each alternative transposes 50 matrixes, 8x8 2-byte elements each.

Example 17-17. Matrix Transpose Alternatives

Alternative 1: Scalar code	Alternative 2: Intel® AVX2 Code	Alternative 3: Intel® AVX-512 Code
mov rsi, pImage mov rdi, pOutImage xor rdx, rdx matrix_loop: xor rax, rax outerloop: xor rbx, rbx innerloop: mov rcx, rax shl rcx, 3 add rcx, rbx mov r8w, word ptr [rsi+rcx*2] mov rcx, rbx shl rcx, 3 add rcx, rax mov word ptr [rdi+rcx*2], r8w add rbx, 1 cmp rbx, 8 jne innerloop add rax, 1 cmp rax, 8 jne outerloop add rdx, 1 add rsi, 64*2 add rdi, 64*2 cmp rdx, 50 jne matrix_loop	mov rsi, pImage mov rdi, pOutImage xor rdx, rdx matrix_loop: vmovdqa xmm0, [rsi] vmovdqa xmm1, [rsi+0x10] vmovdqa xmm2, [rsi+0x20] vmovdqa xmm3, [rsi+0x30] vinserti128 ymm0, ymm0, [rsi+0x40], 0x1 vinserti128 ymm1, ymm1, [rsi+0x50], 0x1 vinserti128 ymm2, ymm2, [rsi+0x60], 0x1 vinserti128 ymm3, ymm3, [rsi+0x70], 0x1 vpunpcklwd ymm4,ymm0,ymm1 vpunpckhwd ymm5,ymm0,ymm1 vpunpcklwd ymm6,ymm2,ymm3 vpunpckhwd ymm7,ymm2,ymm3 vpunpckldq ymm0,ymm4,ymm6 vpunpckhdq ymm1,ymm4,ymm6 vpunpckldq ymm2,ymm5,ymm7 vpunpckhdq ymm3,ymm5,ymm7 vpermq ymm0, ymm0, 0xD8 vpermq ymm1, ymm1, 0xD8 vpermq ymm2, ymm2, 0xD8 vpermq ymm3, ymm3, 0xD8 vmovdqa [rdi], ymm0 vmovdqa [rdi+0x20], ymm1 vmovdqa [rdi+0x40], ymm2 vmovdqa [rdi+0x60], ymm3 add rdx, 1 add rsi, 64*2 add rdi, 64*2 cmp rdx, 50 jne matrix_loop	mov rax, permMaskBuffer vmovdqa32 zmm10, [rax] vmovdqa32 zmm11, [rax+0x40] mov rsi, pImage mov rdi, pOutImage xor rdx, rdx matrix_loop: vmovdqa32 zmm2, [rsi] vmovdqa32 zmm3, [rsi+0x40] vmovdqa32 zmm0, zmm10 vmovdqa32 zmm1, zmm11 vpermi2w zmm0, zmm2, zmm3 vpermi2w zmm1, zmm2, zmm3 vmovdqa32 [rdi], zmm0 vmovdqa32 [rdi+0x40], zmm1 add rdx, 1 add rsi, 64*2 add rdi, 64*2 cmp rdx, 50 jne matrix_loop
Baseline 1x	Speedup: 13.7x	Speedup: 37.3x (2.7x vs Intel® AVX2 code)

17.9 BROADCAST

17.9.1 EMBEDDED BROADCAST

Intel AVX-512 introduces embedded broadcast operations, in which a broadcast operation is implied within the syntax of a non-broadcast instruction. A source from memory can be broadcast, that is, repeated, across all the elements of the effective source operand, up to 16 times for a 32-bit data element, and up to 8 times for a 64-bit data element, without using an additional source register. This is useful when we want to reuse the same scalar operand for all the operations in a vector instruction.

Embedded broadcast is only enabled on instructions with an element size of 32 or 64 bits; however, new FP16 instructions allow embedded broadcast. In the case of older technologies, byte and word element broadcasts do not support embedded broadcast. Use a broadcast instruction, rather than embedded broadcast, to broadcast a byte or word.

Using embedded broadcast can reduce the number of registers used in the code, which may be helpful when register pressure exists.

In addition, when using embedded broadcast the load micro-op is in the same instruction as the operation micro-op, and therefore can benefit from micro fusion.

For example, replace the following code:

```
vbroadcastss zmm3, [rax]
vmulps zmm1, zmm2, zmm3
with:
vmulps zmm1, zmm2, [rax] {1to16}
```

The {1to16} primitive does the following:

1. Loads one float32 (single precision) element from memory.
2. Replicates it 16 times to form a vector of 16 32-bit floating point elements.

Intel AVX-512 instructions with store semantics and pure load instructions do not support broadcast primitives.

17.9.2 BROADCAST EXECUTED ON LOAD PORTS

In Skylake Server microarchitecture, a broadcast instruction with a memory operand of 32 bits or above is executed on the load ports; it is not executed on port 5 as other shuffles are. Alternative 2 in the following example shows how executing the broadcast on the load ports reduces the workload on port 5 and increases performance. Alternative 3 shows how embedded broadcast benefits from both executing the broadcast on the load ports and micro fusion.

Example 17-18. Broadcast Executed on Load Ports Alternatives

Alternative 1: 32-bit Load and Register Broadcast	Alternative 2: Broadcast with a 32-bit Memory Operand	Alternative 3: 32-bit Embedded Broadcast

loop:
  vmovd xmm0, [rax]
  vpbroadcastd zmm0, xmm0
  vpaddd zmm2, zmm1, zmm0
  vpermd zmm2, zmm3, zmm2
  add rax, 0x4
  sub rdx, 0x1
  jnz loop
	loop:
	  vpbroadcastd zmm0, [rax]
	  vpaddd zmm2, zmm1, zmm0
	  vpermd zmm2, zmm3, zmm2
	  add rax, 0x4
	  sub rdx, 0x1
	  jnz loop
		loop:
		  vpaddd zmm2, zmm1, [rax]{1to16}
		  vpermd zmm2, zmm3, zmm2
		  add rax, 0x4
		  sub rdx, 0x1
		  jnz loop

Baseline 1x	Speedup: 1.57x	Speedup: 1.9x

The following example shows that on Skylake Server microarchitecture, 16-bit broadcast is executed on port 5 and therefore does not gain from the memory operand broadcast.

Example 17-19. 16-bit Broadcast Executed on Port 5

Alternative 1: 16-bit Load and Register Broadcast	Alternative 2: Broadcast with a 16-bit Memory Operand

loop:
  vmovd xmm0, [rax]
  vpbroadcastw zmm0, xmm0
  vpaddw zmm2, zmm1, zmm0
  vpermw zmm2, zmm3, zmm2
  add rax, 0x4
  sub rdx, 0x1
  jnz loop
	loop:
	  vpbroadcastw zmm0, [rax]
	  vpaddw zmm2, zmm1, zmm0
	  vpermw zmm2, zmm3, zmm2
	  add rax, 0x2
	  sub rdx, 0x1
	  jnz loop

Baseline 1x	Speedup: equal to baseline

Notice that embedded broadcast is not supported for 16-bit memory operands.

17.10 EMBEDDED ROUNDING

By default, the Rounding Mode is set by bits 13:14 of the MXCSR register.

Intel AVX-512 introduces a new instruction attribute called Static (per instruction) Rounding Mode (RM) or Rounding Mode override. This attribute allows a specific arithmetic rounding mode to be applied, ignoring the value of the RM bits in the MXCSR. In combination with the rounding-mode, Intel AVX-512 also has an SAE (“suppress-all-exceptions”) attribute, to disable reporting any floating-point exception flag in the MXCSR. SAE is always implied when rounding-mode is enabled.

Static Rounding Mode and SAE control can be enabled in the encoding of the instruction by setting the EVEX.b bit to 1 in a register-register vector instruction. In this case, vector length is assumed to be the maximal possible vector length (512-bit in case of Intel AVX-512). The table below summarizes the possible static rounding-mode assignments in Intel AVX-512. Note that some instructions already allow the rounding mode to be statically specified via immediate bits. In such case, the immediate bits take precedence over the embedded rounding mode in the same way as they take precedence over the bits in MXCSR.RM

17.10.1 STATIC ROUNDING MODE

Static rounding mode functions and descriptions are listed below.

Table 17-3. Static Rounding Mode Functions

Function	Description
{rn-sae}	Round to nearest (even) + SAE
{rd-sae}	Round down (toward -infinity) + SAE
{ru-sae}	Round up (toward +infinity) + SAE
{rz-sae}	Round toward zero (Truncate) + SAE

The following code snippet shows a usage example.

Example 17-20. Embedded vs Non-Embedded Rounding

Using Embedded Rounding	Without Embedded Rounding
vaddps zmm7 {k6}, zmm2, zmm4, {ru-sae}	;rax & rcx point to temporary dword values in memory used to load and save (for restoring) MXCSR value vstmxcsr [rax] ;load mxcsr value to memory mov ebx, [rax] ;move to register and ebx, 0xFFFF9FFF ;zero RM bits or ebx, 0x5F80 ;put {ru} to RM bits and suppress all exceptions mov [rcx], ebx ;move new value to the memory vldmxcsr [rcx] ;save to MXCSR vaddps zmm7 {k6}, zmm2, zmm4 ;operation itself vldmxcsr [rax] ;restore previous MXCSR value

This piece of code would perform the single-precision floating point addition of vectors zmm2 and zmm4 with round-towards-plus-infinity, leaving the result in vector zmm7 using k6 as a conditional writemask. Note that MXCSR.RM bits are ignored and unaffected by the outcome of this instruction.

The following are examples of instructions instances where the static rounding-mode is not allowed.

```
; rounding-mode already specified in the instruction immediate
vrndscaleps zmm7 {k6}, zmm2 {rd}, 0x00

; instructions with memory operands
vmulps zmm7 {k6}, zmm2, [rax] {rd}

; instructions with vector length different than maximal vector length (512-bit)
vaddps ymm7 {k6}, ymm2, ymm4 {rd}

; non-floating point instructions
vpaddd zmm7 {k6}, zmm2, zmm4 {rd}
```

17.11 SCATTER INSTRUCTION

This instruction performs a non-continuous store of data (scatter). Given a base address, a set of signed offsets and a data item, the instruction writes each element in the data register to the memory location computed from the base address and corresponding offset. The instruction stores up to 16 elements (8 elements for qword indices) in a doubleword vector or 8 elements in a quadword vector, to the memory locations pointed to by the base address and index vector. Elements are stored only if their corresponding mask bit is one. The figure below describes the following operation.

```
vscatterdpd   [rax + zmm0]{k1} , zmm1
```

In this example, `rax` contains the base address, `zmm0` contains a set of offsets, and `zmm1` contains data to be written.

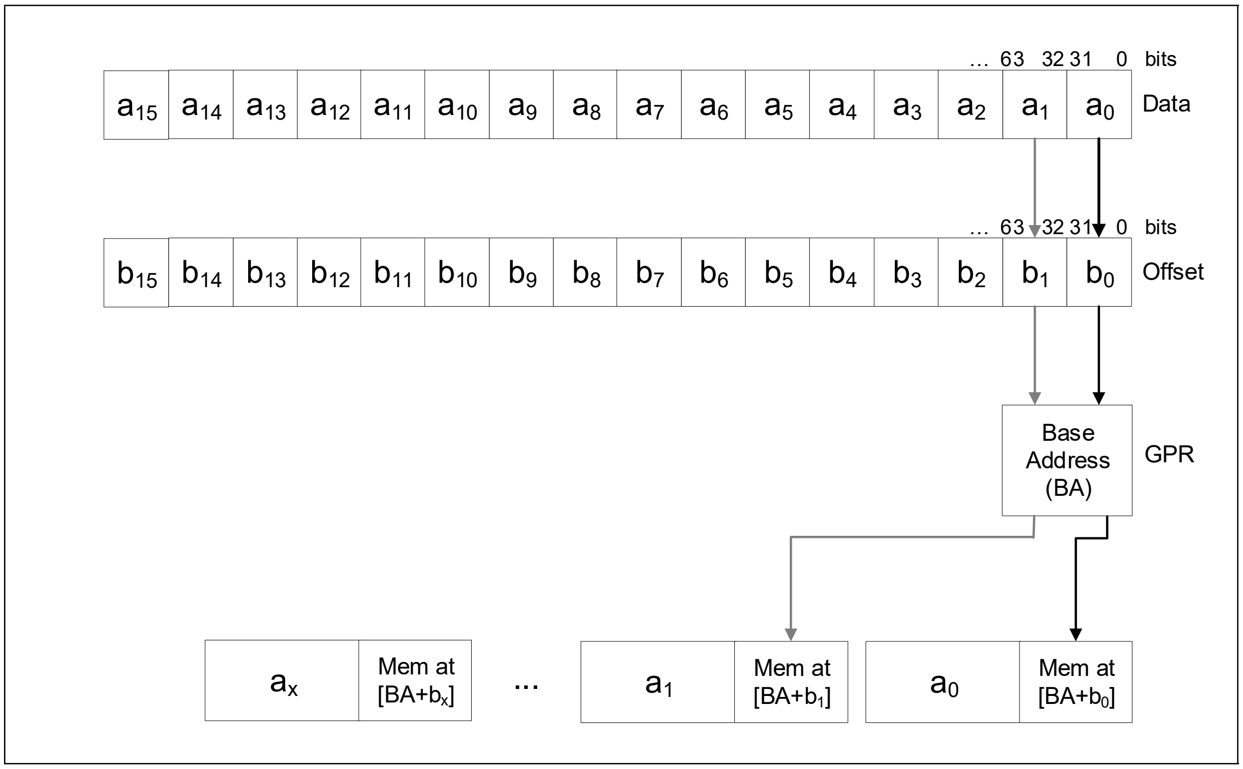


Figure 17-12. VSCATTERDPD Instruction Operation

17.11.1 DATA SCATTER EXAMPLE

Given an array of unique indexes, ranging from 0 to N, we want to sort the array of N values, according to the corresponding index, while converting the values from long long integers (64 bits) to floating point numbers (32 bits).

```
for ( int i=0; i < N; i++ )
{
                    dst[ ind [i] ] = (float)src[i];
}
```

Here are three implementations of the code above.

- **Alternative 1:** pure scalar code.
- **Alternative 2:** a software sequence for scatter.
- **Alternative 3:** a hardware scatter.

NOTE

A hardware Scatter operation issues as many store operations, as the number of elements in the vector. Do not use a scatter operation to store sequential elements, which can be stored with one vmov instruction.

Example 17-21. Scalar Scatter

Scalar

mov rax, pImage   //input
  mov rcx, pOutImage //output
  mov rbx, pIndex   //indexes
  mov rdx, len   //length
 	
  xor r9, r9
mainloop:
  mov r9d, [rbx+rdx-0x4]
  vcvtsi2ss xmm0, xmm0, qword ptr [rax+rdx*2-0x8]
  vmovss [rcx+r9*4], xmm0
  sub rdx, 4
  jnz mainloop

Baseline 1x

Table 17-4. Software Sequence and Hardware Scatter

Software Sequence	Hardware Scatter

shufMaskP:
    .quad·0x0000000200000001
    .quad·0x0000000400000003
    .quad·0x0000000600000005
    .quad·0x0000000800000007
 	
  mov rax, pImage   //input
  mov rcx, pOutImage //output
  mov rbx, pIndex   //indexes
  mov rdx, len   //length
  mov r9, shufMaskP
  vmovaps ymm2, [r9]
mainloop:
  vmovaps zmm1, [rax + rdx*2 - 0x80] //load data
  vcvtuqq2ps ymm0, zmm1 //convert to float
  movsxd r9, [rbx + rdx - 0x40] //load 8th index
  vmovss [rcx + 4*r9], xmm0
  vpermd ymm0, ymm2, ymm0
  movsxd r9, [rbx + rdx - 0x3c] //load 7th index
  vmovss [rcx + 4*r9], xmm0
  vpermd ymm0, ymm2, ymm0
  movsxd r9, [rbx + rdx - 0x38] //load 6th index
  vmovss [rcx + 4*r9], xmm0
  vpermd ymm0, ymm2, ymm0
  movsxd r9, [rbx + rdx - 0x34] //load 5th index
  vmovss [rcx + 4*r9], xmm0
  vpermd ymm0, ymm2, ymm0
	
	mov rax, pImage //input
	  mov rcx, pOutImage //output
	  mov rbx, pIndex //indexes
	  mov rdx, len //length
	mainloop:
	  vmovdqa32 zmm0, [rbx+rdx-0x40]
	  vmovdqa32 zmm1, [rax+rdx*2-0x80]
	  vcvtuqq2ps ymm1, zmm1
	   vmovdqa32 zmm2, [rax+rdx*2-0x40]
	  vcvtuqq2ps ymm2, zmm2
	   vshuff32x4 zmm1, zmm1, zmm2, 0x44
	  kxnorw k1,k1,k1
	  vscatterdps [rcx+4*zmm0] {k1}, zmm1
	  sub rdx, 0x40
	  jnz mainloop

Table 17-4. (Contd.)Software Sequence and Hardware Scatter

Software Sequence	Hardware Scatter
movsxd r9, [rbx + rdx - 0x30] //load 4th index vmovss [rcx + 4*r9], xmm0 vpermd ymm0, ymm2, ymm0 movsxd r9, [rbx + rdx - 0x2c] //load 3rd index vmovss [rcx + 4*r9], xmm0 vpermd ymm0, ymm2, ymm0 movsxd r9, [rbx + rdx - 0x28] //load 2nd index vmovss [rcx + 4*r9], xmm0 vpermd ymm0, ymm2, ymm0 movsxd r9, [rbx + rdx - 0x24] //load 1st index vmovss [rcx + 4*r9], xmm0 vmovaps zmm1, [rax + rdx*2 - 0x40] //load data vcvtuqq2ps ymm0, zmm1 //convert to float movsxd r9, [rbx + rdx - 0x20] //load 8th index vmovss [rcx + 4*r9], xmm0 vpermd ymm0, ymm2, ymm0 movsxd r9, [rbx + rdx - 0x1c] //load 7th index vmovss [rcx + 4*r9], xmm0 vpermd ymm0, ymm2, ymm0 movsxd r9, [rbx + rdx - 0x18] //load 6th index vmovss [rcx + 4*r9], xmm0 vpermd ymm0, ymm2, ymm0 movsxd r9, [rbx + rdx - 0x14] //load 5th index vmovss [rcx + 4*r9], xmm0 vpermd ymm0, ymm2, ymm0 movsxd r9, [rbx + rdx - 0x10] //load 4th index vmovss [rcx + 4*r9], xmm0 vpermd ymm0, ymm2, ymm0 movsxd r9, [rbx + rdx - 0xc] //load 3rd index vmovss [rcx + 4*r9], xmm0 vpermd ymm0, ymm2, ymm0 movsxd r9, [rbx + rdx - 0x8] //load 2nd index vmovss [rcx + 4*r9], xmm0 vpermd ymm0, ymm2, ymm0 movsxd r9, [rbx + rdx - 0x4] //load 1st index vmovss [rcx + 4*r9], xmm0 sub rdx, 0x40 jnz mainloop	
Speedup: 1.48x	Speedup: 1.53x

17.12 STATIC ROUNDING MODES, SUPPRESS-ALL-EXCEPTIONS (SAE)

The Suppress-all-exceptions (SAE) feature was added to Intel AVX-512 floating-point instructions. This feature is helpful when spurious flag settings are undesirable. Although current implementations of vector math functions usually allow spurious flag settings, they can cause problems for applications that run with exceptions enabled. Standard-compliant code does not allow spurious flag settings.

In addition to standard-mandated uses (IEEE, OpenCL), static rounding modes have applications in math libraries that operate under the default rounding mode (which can be dynamically set).

17.13 QWORD INSTRUCTION SUPPORT

Intel AVX-512 extends QWORD support to many instructions introduced in Intel AVX and Intel AVX2. QWORD support was added to the instructions as detailed in the following sections.

17.13.1 QUADWORD SUPPORT IN ARITHMETIC INSTRUCTIONS

Intel AVX-512 adds new quadword extension to *vpmaxsq, vpmaxuq, vpminsq, vpminuq, and vpmullq*.

The following example will store to array c the max value between the sum and the multiply of two 64bit numbers.

```
const int N = miBufferWidth;
const __int64* restrict a = A;
const __int64* restrict b = B;
__int64* restrict c = Cref;

for (int i = 0; i < N; i++){
    __int64 sum = a[i] + b[i];
    __int64 mul = a[i] * b[i];
    c[i] = mul > sum ? mul : sum;
}
```

The code below shows how the new support reduces instruction count from 118 in Intel AVX2 to 30 in Intel AVX-512 and results in a 3.1x speedup.

Example 17-22. QWORD Example, Intel® AVX2 vs. Intel® AVX-512 Intrinsics

Intel® AVX2 Intrinsics	Intel® AVX-512 Intrinsics

for (int i = 0; i < N; i+= 32){
    __m256i aa, bb, aah, bbh, mul, sum;
    #pragma unroll(8)
    for (int j = 0; j < 8; j++){
        aa = _mm256_loadu_si256((const __m256i*)(a+i+4*j));
        bb = _mm256_loadu_si256((const __m256i*)(b+i+4*j));
	for (int i = 0; i < N; i+= 32){
	    __m512i aa, bb, mul, sum;
	    #pragma unroll(4)
	    for (int j = 0; j < 4; j++){
	        aa = _mm512_loadu_si512((const __m512i*)(a+i+8*j));
	        bb = _mm512_loadu_si512((const __m512i*)(b+i+8*j));

Example 17-22. QWORD Example, Intel® AVX2 vs. Intel® AVX-512 Intrinsics (Contd.)

Intel® AVX2 Intrinsics	Intel® AVX-512 Intrinsics

sum = _mm256_add_epi64(aa, bb);
mul = _mm256_mul_epu32(aa, bb);
aah = _mm256_srli_epi64(aa, 32);
bbh = _mm256_srli_epi64(bb, 32);
aah = _mm256_mul_epu32(aah, bb);
bbh = _mm256_mul_epu32(bbh, aa);
aah = _mm256_add_epi32(aah, bbh);
aah = _mm256_slli_epi64(aah, 32);
mul = _mm256_add_epi64(mul, aah);
aah = _mm256_cmpgt_epi64(mul, sum);
aa = _mm256_castpd_si256 (_mm256_blendv_pd(_mm256_castsi256_pd (sum), _mm256_castsi256_pd(mul), _mm256_castsi256_pd(aah)));
_mm256_storeu_si256((__m256i*)(c+4*j), aa);
}
c += 32;
}
	sum = _mm512_add_epi64(aa, bb);
	mul = _mm512_mullo_epi64(aa, bb);
	aa = _mm512_max_epi64(sum, mul);
	_mm512_storeu_si512((__m512i*)(c+8*j), aa);
	
	}
	
	c += 32;
	}

Baseline 1x	Speedup: 3.1x

Example 17-23. QWORD Example, Intel® AVX2 vs. Intel® AVX-512 Assembly

Intel® AVX2 Assembly	Intel® AVX-512 Assembly
loop: vmovdqu32 ymm28, ymmword ptr [rax+rcx*8+0x20] inc r9d vmovdqu32 ymm26, ymmword ptr [r11+rcx*8+0x20] vmovdqu32 ymm17, ymmword ptr [r11+rcx*8] vmovdqu32 ymm19, ymmword ptr [rax+rcx*8] vmovdqu ymm13, ymmword ptr [rax+rcx*8+0x40] vmovdqu ymm11, ymmword ptr [r11+rcx*8+0x40] vpsrlq ymm25, ymm28, 0x20 vpsrlq ymm27, ymm26, 0x20 vpsrlq ymm16, ymm19, 0x20 vpsrlq ymm18, ymm17, 0x20 vpaddq ymm6, ymm28, ymm26 vpsrlq ymm10, ymm13, 0x20 vpsrlq ymm12, ymm11, 0x20 vpaddq ymm0, ymm19, ymm17 vpmuludq ymm29, ymm25, ymm26 vpmuludq ymm30, ymm27, ymm28 vpaddd ymm31, ymm29, ymm30 vmovdqu32 ymm29, ymmword ptr [r11+rcx*8+0x80] vpsllq ymm5, ymm31, 0x20 vmovdqu32 ymm31, ymmword ptr [rax+rcx*8+0x80] vpsrlq ymm30, ymm29, 0x20 vpmuludq ymm20, ymm16, ymm17	loop: vmovups zmm0, zmmword ptr [rax+rcx*8] inc r9d vmovups zmm5, zmmword ptr [rax+rcx*8+0x40] vmovups zmm10, zmmword ptr [rax+rcx*8+0x80] vmovups zmm15, zmmword ptr [rax+rcx*8+0xc0] vmovups zmm1, zmmword ptr [r11+rcx*8] vmovups zmm6, zmmword ptr [r11+rcx*8+0x40] vmovups zmm11, zmmword ptr [r11+rcx*8+0x80] vmovups zmm16, zmmword ptr [r11+rcx*8+0xc0] vpaddq zmm2, zmm0, zmm1 vpmullq zmm3, zmm0, zmm1 vpaddq zmm7, zmm5, zmm6 vpmullq zmm8, zmm5, zmm6 vpaddq zmm12, zmm10, zmm11 vpmullq zmm13, zmm10, zmm11 vpaddq zmm17, zmm15, zmm16 vpmullq zmm18, zmm15, zmm16 vpmaxsq zmm4, zmm2, zmm3 vpmaxsq zmm9, zmm7, zmm8 vpmaxsq zmm14, zmm12, zmm13 vpmaxsq zmm19, zmm17, zmm18 vmovups zmmword ptr [rsi], zmm4

Example 17-23. QWORD Example, Intel® AVX2 vs. Intel® AVX-512 Assembly (Contd.)

Intel® AVX2 Assembly	Intel® AVX-512 Assembly
vpmuludq ymm21, ymm18, ymm19 vpmuludq ymm4, ymm28, ymm26 vpaddd ymm22, ymm20, ymm21 vpaddq ymm7, ymm4, ymm5 vpsrlq ymm28, ymm31, 0x20 vmovdqu32 ymm20, ymmword ptr [r11+rcx*8+0x60] vpsllq ymm24, ymm22, 0x20 vmovdqu32 ymm22, ymmword ptr [rax+rcx*8+0x60] vpsrlq ymm21, ymm20, 0x20 vpaddq ymm4, ymm22, ymm20 vpcmpgtq ymm8, ymm7, ymm6 vblendvpd ymm9, ymm6, ymm7, ymm8 vmovups ymmword ptr [rsi+0x20], ymm9 vpmuludq ymm14, ymm10, ymm11 vpmuludq ymm15, ymm12, ymm13 vpmuludq ymm8, ymm28, ymm29 vpmuludq ymm9, ymm30, ymm31 vpmuludq ymm23, ymm19, ymm17 vpaddd ymm16, ymm14, ymm15 vpsrlq ymm19, ymm22, 0x20 vpaddd ymm10, ymm8, ymm9 vpaddq ymm1, ymm23, ymm24 vpsllq ymm18, ymm16, 0x20 vmovdqu32 ymm28, ymmword ptr [rax+rcx*8+0xc0] vpsllq ymm12, ymm10, 0x20 vpmuludq ymm23, ymm19, ymm20 vpmuludq ymm24, ymm21, ymm22 vpaddd ymm25, ymm23, ymm24 vmovdqu32 ymm19, ymmword ptr [rax+rcx*8+0xa0] vpsllq ymm27, ymm25, 0x20 vpsrlq ymm25, ymm28, 0x20 vpsrlq ymm16, ymm19, 0x20 vpcmpgtq ymm2, ymm1, ymm0 vblendvpd ymm3, ymm0, ymm1, ymm2 vpaddq ymm0, ymm13, ymm11 vmovups ymmword ptr [rsi], ymm3 vpmuludq ymm17, ymm13, ymm11 vpmuludq ymm11, ymm31, ymm29 vpaddq ymm1, ymm17, ymm18 vpaddq ymm13, ymm31, ymm29 vpaddq ymm14, ymm11, ymm12 vmovdqu32 ymm17, ymmword ptr [r11+rcx*8+0xa0] vmovdqu ymm12, ymmword ptr [r11+rcx*8+0xe0] vpsrlq ymm18, ymm17, 0x20 vpcmpgtq ymm2, ymm1, ymm0 vpmuludq ymm26, ymm22, ymm20 vpcmpgtq ymm15, ymm14, ymm13 vblendvpd ymm3, ymm0, ymm1, ymm2 vblendvpd ymm0, ymm13, ymm14, ymm15 vmovdqu ymm14, ymmword ptr [rax+rcx*8+0xe0]	vmovups zmmword ptr [rsi+0x40], zmm9 vmovups zmmword ptr [rsi+0x80], zmm14 vmovups zmmword ptr [rsi+0xc0], zmm19 add rcx, 0x20 add rsi, 0x100 cmp r9d, r8d jb loop

Example 17-23. QWORD Example, Intel® AVX2 vs. Intel® AVX-512 Assembly (Contd.)

Intel® AVX2 Assembly	Intel® AVX-512 Assembly
vmovups ymmword ptr [rsi+0x40], ymm3 vmovups ymmword ptr [rsi+0x80], ymm0 vpaddq ymm5, ymm26, ymm27 vpsrlq ymm11, ymm14, 0x20 vpsrlq ymm13, ymm12, 0x20 vpaddq ymm1, ymm19, ymm17 vpaddq ymm0, ymm14, ymm12 vmovdqu32 ymm26, ymmword ptr [r11+rcx*8+0xc0] vpmuludq ymm20, ymm16, ymm17 add rcx, 0x20 vpmuludq ymm21, ymm18, ymm19 vpaddd ymm22, ymm20, ymm21 vpsrlq ymm27, ymm26, 0x20 vpsllq ymm24, ymm22, 0x20 vpmuludq ymm29, ymm25, ymm26 vpmuludq ymm30, ymm27, ymm28 vpmuludq ymm15, ymm11, ymm12 vpmuludq ymm16, ymm13, ymm14 vpmuludq ymm23, ymm19, ymm17 vpaddd ymm31, ymm29, ymm30 vpaddd ymm17, ymm15, ymm16 vpaddq ymm2, ymm23, ymm24 vpsllq ymm19, ymm17, 0x20 vpcmpgtq ymm6, ymm5, ymm4 vblendvpd ymm7, ymm4, ymm5, ymm6 vpsllq ymm6, ymm31, 0x20 vmovups ymmword ptr [rsi+0x60], ymm7 vpaddq ymm7, ymm28, ymm26 vpcmpgtq ymm3, ymm2, ymm1 vpmuludq ymm5, ymm28, ymm26 vpmuludq ymm18, ymm14, ymm12 vblendvpd ymm4, ymm1, ymm2, ymm3 vpaddq ymm8, ymm5, ymm6 vpaddq ymm1, ymm18, ymm19 vmovups ymmword ptr [rsi+0xa0], ymm4 vpcmpgtq ymm9, ymm8, ymm7 vpcmpgtq ymm2, ymm1, ymm0 vblendvpd ymm10, ymm7, ymm8, ymm9 vblendvpd ymm3, ymm0, ymm1, ymm2 vmovups ymmword ptr [rsi+0xc0], ymm10 vmovups ymmword ptr [rsi+0xe0], ymm3 add rsi, 0x100 cmp r9d, r8d jb loop	
Baseline 1x	Speedup: 3.1x

17.13.2 QUADWORD SUPPORT IN CONVERT INSTRUCTIONS

The following tables demonstrate the new quadword extension in convert instructions.

Table 17-5. Vector Quadword Extensions

From / To	Vector SP	Vector DP	Vector int64	Vector uint64
Vector SP	-		vcvtps2qq	vcvtps2uqq
Vector DP		-	vcvtpd2qq	vcvtpd2qq
Vector int64	vcvtqq2ps	vcvtqq2pd	-	
Vector uint64	vcvtqq2ps	vcvtuqq2pd		-

Table 17-6. Scalar Quadword Extensions

From / To	Scalar SP	Scalar DP	Scalar int64	Scalar uint64
Scalar SP	-	-	vcvtss2si	vcvtss2usi
Scalar DP	-	-	vcvtsd2si	vcvtsd2usi
Scalar int64	vcvtsi2sd	vcvtsi2sd	-	-
Scalar uint64	vcvtusi2sd	vcvtusi2sd	-	-

17.13.3 QUADWORD SUPPORT FOR CONVERT WITH TRUNCATION INSTRUCTIONS

The following tables demonstrate the new quadword extension in convert with truncate instructions.

Table 17-7. Vector Quadword Extensions

From / To	Vector int64	Vector uint64
Vector SP	vcvttps2qq	vcvttps2uqq
Vector DP	vcvttpd2qq	vcvttpd2qq

Table 17-8. Scalar Quadword Extensions

From / To	Scalar int64	Scalar uint64
Scalar SP	vcvttss2si	vcvttss2usi
Scalar DP	vcvttsd2si	vcvttsd2usi

17.14 VECTOR LENGTH ORTHOGONALITY

All Intel AVX-512 instructions, in processors that support Vector Length Extensions (VL), can operate at three vector lengths: 128-bit, 256-bit and 512-bit. All of these vector lengths are supported by all Intel AVX-512 instructions, except instructions with Embedded Rounding.

In the instruction encoding, the same two bits are used for encoding vector length and embedded rounding control, therefore when embedded rounding is used, the vector length is automatically assumed to be 512 bits (maximum vector length in Intel AVX-512).

See also Section 17.10.

17.15 INTEL® AVX-512 INSTRUCTIONS FOR TRANSCENDENTAL SUPPORT

This section lists and describes the new instructions introduced by Intel AVX-512 for transcendental support.

17.15.1 VRCP14, VRSQRT14 - SOFTWARE SEQUENCES FOR 1/X, X/Y, SQRT(X)

Syntax:

VRCP14PD/PS dest, src

VRSQRT14PD/PS dest, src

17.15.1.1 Application Examples

There are software sequences for Reciprocal, Division, Square Root, and Inverse Square Root instructions.

Software sequences for 1/x, x/y, sqrt(x) are beneficial for throughput (not so much for latency, unless the accuracy is quite low). They are typically implemented via Newton-Raphson approximations, or polynomial approximations.

One advantage of VRCP14 and VRSQRT14 is the improved accuracy, compared with the legacy RCPPS, RSQRTPS. This helps shorten the computation, in particular for double precision (which requires two instead of three Newton-Raphson iterations for a 50-52 bit approximation).

Another advantage of these instructions is that they have double-precision versions (while the legacy RCP/RSQRT instructions did not). This further boosts double-precision performance. On Skylake Server microarchitecture, double precision reciprocal and square root software sequences have significantly better throughput than the VDIV and VSQRT instructions in 512-bit vector mode Double Precision Transcendental Argument Reductions (e.g., log, cbrt).

In functions such as log() or the cube root (cbrt), a rounded VRCP14PD result can be used in place of an expensive reciprocal table lookup. The same technique could be used before via RCPPS, but was less efficient for double-precision.

See Section 17.15.3 for a log() argument reduction example.

17.15.2 VGETMANT VGETEXP - VECTOR GET MANTISSA AND VECTOR GET EXPONENT

Syntax:

VGETMANTPD/PS dest_mant, src, imm

VGETEXPPD/PS dest_exp, src

17.15.2.1 Application Examples

Logarithm Function

log2(x) = VGETEXP(x) + log2(VGETMANT(x,8))

log(x) = VGETEXP(x)*log(2.0) + log(VGETMANT(x,8))

As seen above, the computation is reduced to computing log(VGETMANT(x,8)), where VGETMANT(x,8) is guaranteed to be in [1,2) for all valid function inputs, and NaN for invalid inputs (x<0).

A variety of algorithms can be applied to compute the logarithm of the mantissa. The selection of a particular algorithm may depend on the desired accuracy, on optimization goals (latency or throughput optimized), or on specifics of the microarchitecture. Some algorithms may use other normalization options for the mantissa: [0.5, 1) or [0.75, 1.5); however, the basic identity underlying the computation is shown above.

See Section 17.15.5 for details on X^{alpha} (constant alpha) and division.

17.15.3 VRNDSCALE - VECTOR ROUND SCALE

Syntax:

VRNDSCALEPD/PS dest, src, imm

17.15.3.1 Application Examples

Lookup tables are frequently used in transcendental function implementations. The table index is most often based on a few leading bits of the input. VRNDSCALE can be used as part of the argument reduction process, to form the floating-point input value corresponding to the table index. The following example implements the argument reduction for log(x), where $1 \leq x < 2$:

```
y = RCP14(x);           // y is in (0.5, 1]

y0=RNDSCALE(y, k*16);    // y0 has k mantissa bits (leading 1

                                                                   // included)

R = x?y0 - 1;           // |R|<2-14+2-k.
```

Therefore log(x) = -log(y0) + log(1+R).

log(1+R)can be computed via a polynomial, and log(y0) can be retrieved from a lookup table of 2k-1+1 elements, or 2k-1 elements, at the expense of an additional check.

17.15.4 VREDUCE - VECTOR REDUCE

Syntax:

VREDUCEPD/PS dest, src, imm

17.15.4.1 Application Examples

The most significant benefit of VREDUCE is latency reduction in common transcendental operations such as exp2 and pow (which includes an exp2 operation). Uses in other transcendental functions such as atan() are also possible.

17.15.5 VSCALEF - VECTOR SCALE

Syntax:

VSCALEFPD/PS dest, src1, src2

17.15.5.1 Application Examples

exp2 (2x)

exp2(x) = VSCALEF(2VREDUCE(x, RD_mode), x)

R(x) = VREDUCE(x, RD_mode) = x - floor(x) is in [0, 1). 2R(x) is computed by other means, such as polynomial approximation, or table lookup with polynomial approximation. VSCALEF correctly handles overflow and underflow. It is also defined to handle exp() special cases correctly (such as when the input is an Infinity), so there is no need for special paths in a vector implementation. In the absence of VSCALEF, inputs that are very large in magnitude require a separate path.

Since explicit exponent manipulation is no longer needed, VSCALEF also helps improve throughput.

Exp(x)

Exp(x) = VSCALEF(2R(x), x*(1/log(2.0)),

where,

R(x) = x - log(2.0)*floor(x*(1/log(2.0));

R(x) is accurately computed by using a sufficiently long log(2.0) approximation (longer than the native floating-point format).

As with exp2(), the advantages of using VSCALEF are better throughput and elimination of secondary branches.

x^{alpha} (constant alpha)

For example, alpha=1/3 (the cube root function, cbrt).

The basic reduction for this computation is:

x^{alpha} = VSCALEF((VGETMANT(x, imm))alpha?2VREDUCE (VGETEXP(x)*alpha, RD_mode), VGETEXP(x)*alpha)

selecting the immediate (imm) is based on the value of the alpha constant.

Division:

a/b = VSCALEF(VGETMANT(a,0)/VGETMANT(b,0), VGETEXP(a)-VGETEXP(b))

This reduction allows for a branch-free implementation of divide, that covers overflow, underflow, and special inputs (zeroes, Infinities, or denormals).

|VGETMANT(x,0)| is in [1,2) for all non-NaN inputs.

VGETMANT(a,0)/VGETMANT(b,0) can be computed to the desired accuracy.

The suppress-all-exceptions (SAE) feature available in Intel AVX-512 can help ensure spurious flag settings do not occur. Flags can be set correctly as part of the computation (except for divide-by-zero, which requires an additional step).

For high accuracy or IEEE compliance, the hardware instruction typically provides better performance, especially in terms of latency.

17.15.6 VFPCLASS - VECTOR FLOATING POINT CLASS

Syntax:

VFPCLASSPD/PS dest_mask, src, imm

17.15.6.1 Application Examples

The VFPCLASS instruction is used to detect special cases so they can be directed to a special path, or alternatively, handled with masked operations in the main path. See two examples below.

Reciprocal Sequence, Square Root Sequence:

- The reduced argument for the 1/x computation is e=1-x*RCP14(x).
  - This expression evaluates to NaN when x is ±0 or ±Inf, as RCP14 returns the correct result for these special cases.
  - VFPCLASS enables you to set mask=1 for x=±0 or ±Inf, and mask=0 for all other x.
    - **This mask can then be used to select between the RCP14 output (result for special cases), or the result of a reciprocal refinement computation starting with RCP14 (for typical inputs).**
- In a similar manner, a square root computation based on RSQRT14 can use the VFPCLASS instruction to create a mask for =±0 or x=+Inf.

Pow(x,y) function:

- The main path of pow(x,y)=2y*log2(x) does not operate on x?0, x=Inf/NaN, or y=Inf/NaN.
- One VFPCLASS op can be used to set special_x_mask=1 for x?0 or x=Inf/NaN.
- A second VFPCLASS op would be used to set special_y_mask=1 for y=Inf/NaN.
- A branch to a secondary path is taken if either mask is set.

17.15.7 VPERM, VPERMI2, VPERMT2 - SMALL TABLE LOOKUP IMPLEMENTATION

17.15.7.1 Application Examples

Math library functions are frequently implemented using table lookups. In vector mode, large table lookups would use vector gather. Small table lookups can be implemented via the VPERM* instructions, which are significantly faster.

Examples of common transcendental functions that achieved very significant speedup using VPERM* for table lookups: exp(), log(), pow() - both single and double precision.

17.16 CONFLICT DETECTION

The Intel AVX-512 Conflict Detection instructions are instructions that, together with Intel AVX-512 Foundation instructions, enable efficient vectorization of loops with possible vector dependencies (i.e., conflicts) through memory. VPCONFLICT performs horizontal comparisons of elements within a single vector register. VPCONFLICT compares each element of a vector register with all previous elements in that register, and outputs the results of all of the comparisons. These horizontal comparisons can be used for other purposes.

Other conflict detection instructions allow for efficient manipulation of the comparison results. The VPLZCNT instruction lets us generate controls for in-register permute operations used to combine vector elements with matching values.

17.16.1 VECTORIZATION WITH CONFLICT DETECTION

The Intel AVX-512CD instructions allow efficient vectorization of loops with reads and writes through an array of pointers

For example:

***ptr[i] += val[i]) or an indirectly addressed array (e.g., A[B[i]] += val[i]).**

Consider the following histogram computation:

```
for(int i = 0; i < num_inputs; i++)
                                                                         {
                                              histogram[input[i] & (num_bins - 1)]++;
                                                                         }
```

If input[0] = input[1] = 3, we will get an incorrect answer if we use SIMD instructions to read histogram[input[0]] and histogram[input[1]] into a register (with a gather), increment them, and then write them back (with a scatter). After this sequence, the value in histogram[3] will be 1, when it should be 2.

The problem occurs because we have duplicate indices; this creates a dependence between the write to the histogram in iteration 0 and the read from the histogram in iteration 1 - the read should get the value of the previous write.

To detect this scenario, look for duplicate indices (or pointer values), using the VPCONFLICT instruction. This instruction compares each element of a vector register with all previous elements in that register.

Example:

```
vpconflictd zmm0, zmm1
```

The figure below is an example that shows the execution of a VPCONFLICTD instruction. The input, ZMM1, contains 16 integers, shown in the light grey boxes. ZMM1 is at the top of the figure, and also visually transposed along the left-hand side of the figure. The white boxes show the equality comparisons that the hardware performs between different elements of ZMM1, and the outcome of each comparison (0 = not equal, 1 = equal). Each comparison output is a single bit in the output of the instruction. Comparisons that are not performed (i.e., the dark grey boxes) produce a single '0' bit in the output. Finally, the output register, ZMM0, is shown at the bottom of the figure. Each element is shown as a decimal representation of the bits above it.

Use VPCONFLICT in different ways to help vectorize loops.

The simplest option is to check for any duplicate indices in a given SIMD register. If there are none, SIMD instructions can be used to compute all elements simultaneously. If conflicts are present, execute a scalar loop for that group of elements.

Branching to a scalar version of the loop on any duplicate indices can work well if duplicates are extremely rare. However, if the chance of getting even one duplicate in a given iteration of the vectorized loop is large enough, then it is better to use SIMD as much as possible, to exploit as much parallelism as possible.

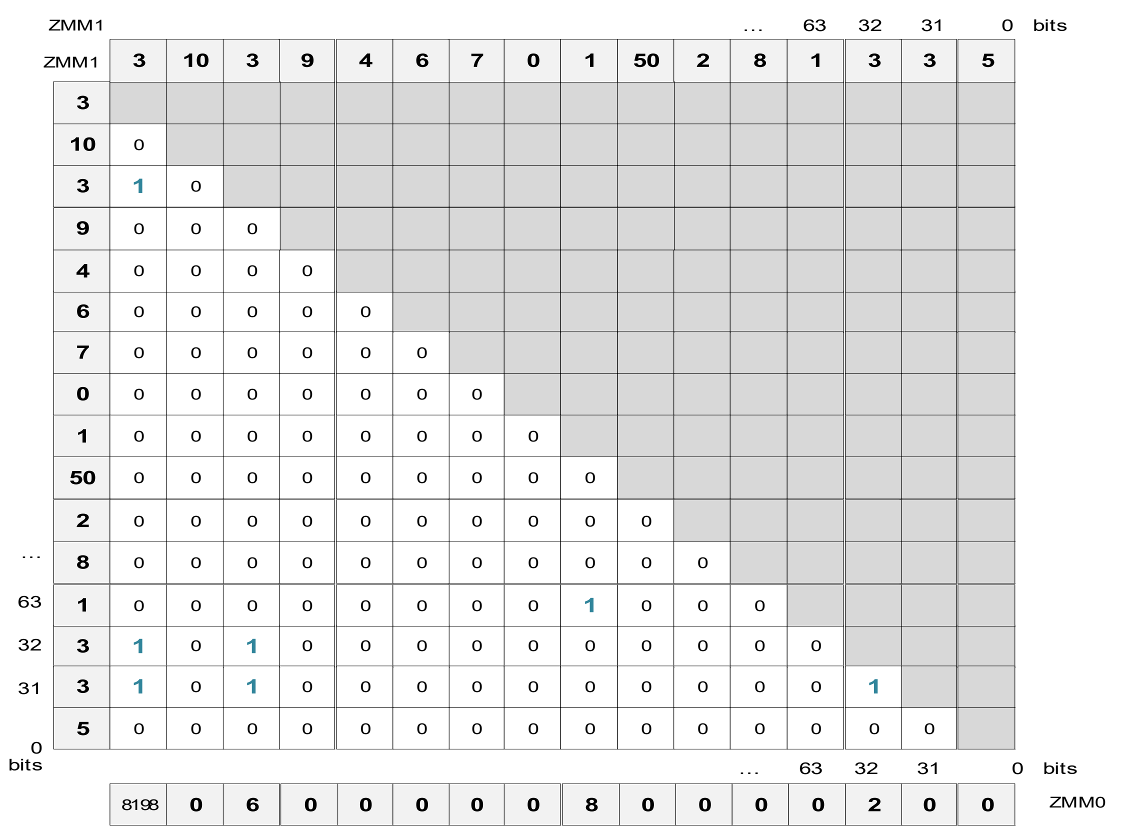


Figure 17-13. VPCONFLICTD Instruction Execution

For loops performing updates to memory locations, such as in the histogram example, minimize store-load forwarding by merging the updates to each distinct index while the data is in registers, and only perform a single write to each memory location. Further, the merge can be performed in a parallel fashion.

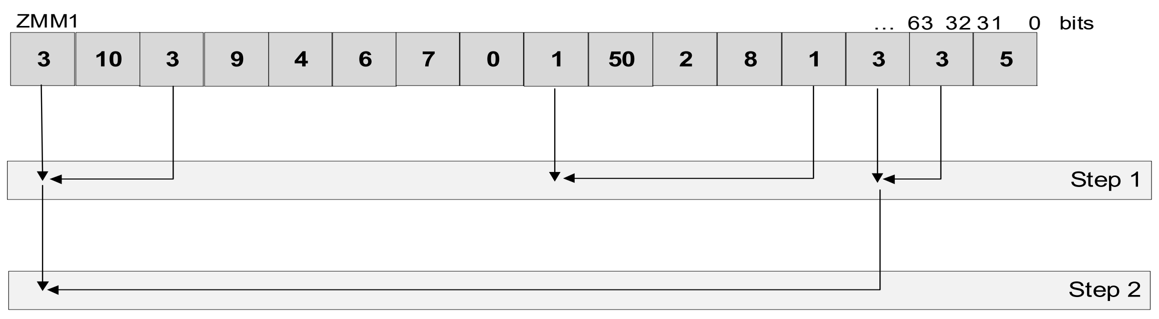


Figure 17-14. VPCONFLICTD Merging Process

The figure above shows the merging process for the example set of indices. While the figure shows only the indices, it actually merges the values. Most of the indices are unique, and thus require no merging. Step 1 combines three pairs of indices: two pairs of '3's and one pair of '1's. Step 2 combines the intermediate results for the '3's from step 1, so that there is now a single value for each distinct index. Notice that in only two steps, the four elements with an index value of 3 are merged, because we performed a tree reduction; we merged pairs of results or intermediate results at each step.

The merging (combining or reduction) process shown above is done with a set of permute operations. The initial permute control is generated with a VPLZCNT+VPSUB sequence. VPLZCNT provides the number of leading zeros for each vector element (i.e., contiguous zeros in the most significant bit positions). Subtracting the results of VPLZCNT from the number of bits in each vector element, minus one, provides the bit position of the most significant '1' bit in the result of the VPCONFLICT instruction, or results in a '-1' for an element if it has no conflicts. In the example above this sequence results in the following permute control.

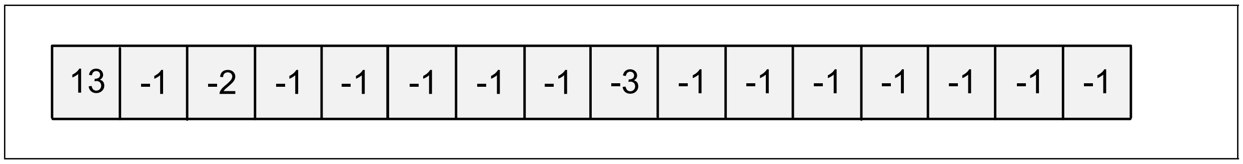


Figure 17-15. VPCONFLICTD Permute Control

The permute loop for merging matching indices and generating the next set of permute indices repeats until all values in the permute control become equal to '-1'.

The assembly code below shows both the scalar version of a histogram loop, and the vectorized version with a tree reduction. Speedups are modest because the loop contains little computation; the SIMD benefit comes almost entirely from vectorizing just the logical AND operation and the increment. SIMD speedups can be much higher for loops containing more vectorizable computation.

Example 17-24. Scatter Implementation Alternatives

Scalar Code (Unrolled Two Times)

```
  mov r9d, bins_minus_1
  mov ebx, num_inputs
  mov r10, pInput
  mov r15, pHistogram
  xor rax, rax
histogram_loop:
  lea ecx, [rax + rax]
  inc eax
  movsxd rcx, ecx
  mov esi, [r10+rcx*4]
  and esi, r9d
  mov r8d, [r10+rcx*4+4]
  movsxd rsi, esi
  and r8d, r9d
  movsxd r8, r8d
  inc dword ptr [r15+rsi*4]
  inc dword ptr [r15+r8*4]
  cmp eax, ebx
  jb histogram_loop
```

Intel® AVX-512 Code

```
  vmovaps zmm4, all_1 // {1, 1, …, 1}
  vmovaps zmm5, all_negative_1
  vmovaps zmm6, all_31
  vmovaps zmm7, all_bins_minus_1
  mov ebx, num_inputs
  mov r10, pInput
  mov r15, pHistogram
  xor rcx, rcx
histogram_loop:
  vpandd zmm3, zmm7, [r10+rcx*4]
  vpconflictd zmm0, zmm3
  kxnorw k1, k1, k1
  vmovaps zmm2, zmm4
  vpxord zmm1, zmm1, zmm1
  vpgatherdd zmm1{k1}, [r15+zmm3*4]
  vptestmd  k1, zmm0, zmm0
  kortestw  k1, k1
  je update

  vplzcntd zmm0, zmm0
  vpsubd zmm0, zmm6, zmm0

conflict_loop:
  vpermd zmm8{k1}{z}, zmm0, zmm2
  vpermd zmm0{k1}, zmm0, zmm0
  vpaddd zmm2{k1}, zmm2, zmm8
  vpcmpned k1, zmm5, zmm0
  kortestw  k1, k1
  jne conflict_loop

update:
  vpaddd zmm0, zmm2, zmm1
  kxnorw k1, k1, k1
  add rcx, 16
  vpscatterdd [r15+zmm3*4]{k1}, zmm0
  cmp ecx, ebx
  jb histogram_loop
```

Scalar Code (Unrolled Two Times)	Intel® AVX-512 Code
Scalar, Baseline, 1x	Speedup: 1.11x (random inputs); 1.34x (input values identical)

Notice that the end result of the conflict loop (i.e., the resulting vector after all merging is done, ZMM2 in the above sequence) holds the complete set of partial sums. That is, for each element, the result contains the value of that element merged with all earlier elements with the same index value. Using the earlier example values, ZMM2 contains the result shown in Figure 17-16.

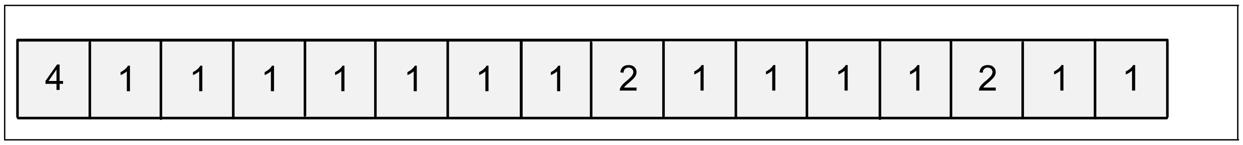


Figure 17-16. VPCONFLICTD ZMM2 Result

While the above sequence does not take advantage of this, other use cases might.

17.16.2 SPARSE DOT PRODUCT WITH VPCONFLICT

A sparse vector may be stored as a pair of arrays: one containing non-zero values, and one containing the original locations of those values in the vector. Note that the indices are sorted in increasing order.

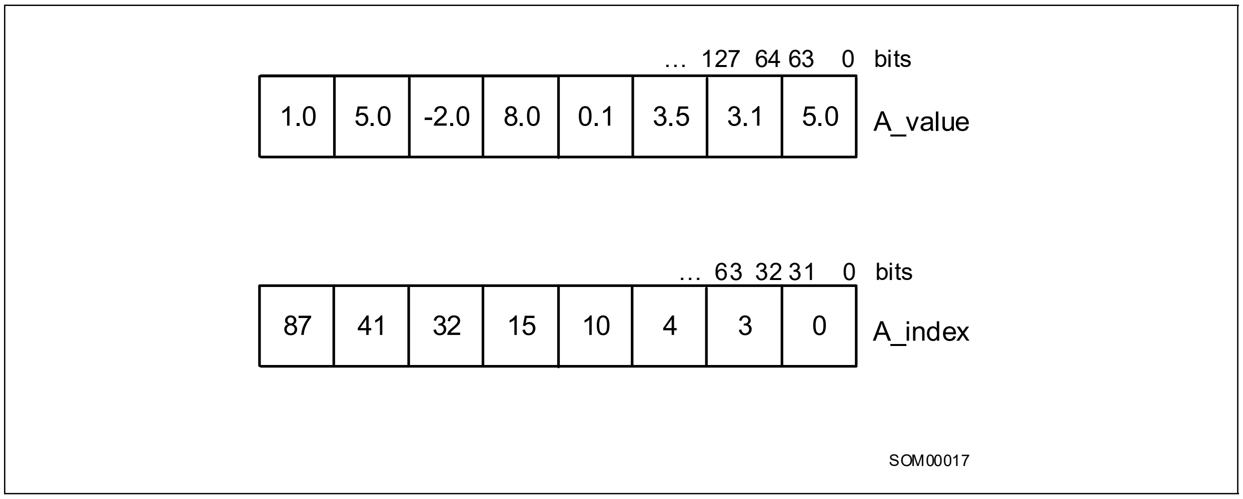


Figure 17-17. Sparse Vector Example

To perform a dot product of two sparse vectors efficiently, we need to find elements with matching indices; those are the only ones on which we should perform the multiply and accumulation. The scalar method for doing this is to start at the beginning of the two index arrays, compare those indices, and if there is a match, do the multiply and accumulate, then advance the indices of both vectors. If there is no match, we advance the index of the lagging vector.

```
A_offset = 0; B_offset = 0; sum = 0;
while ((A_offset < A_length) && (B_offset < B_length))
{
    if (A_index[A_offset] == B_index[B_offset]) // match
    {
        sum += A_value[A_offset] * B_value[B_offset];
        A_offset++;
        B_offset++;
    }
```

```
        else if (A_index[A_offset] < B_index[B_offset])
                {
                A_offset++;
                }
                else
                {
                B_offset++;
                }
        }
```

The Intel AVX-512CD instructions provide an efficient way to vectorize this loop. Instead of comparing one index from each vector at a time, we can compare eight of them. First we combine eight indices from each vector into a single vector register. Then, the VPCONFLICT instruction compares the indices. We use the output to create a mask of elements in vector A that have a match, and also to create permute controls to move the corresponding values of B to the same location, so that we can use a vector FMA instruction.

Example 17-25 shows the assembly code for both the scalar and vector versions of a single comparison and FMA. For brevity, the offset updates and looping are omitted.

Example 17-25. Scalar vs. Vector Update Using AVX-512CD

Scalar Code	Intel® AVX-512 Code

mov rdx, A_index
mov rcx, A_offset
mov rax, A_value
mov r12, B_index
mov r13, B_offset
mov rbx, B_value
 	
mov r10d, [rdx+rcx*4]
mov r11d, [r12+r13*4]
cmp r10d, r11d
jne skip_fma
 	
// do the fma on a match
movsd xmm5, [rbx+r13*8]
mulsd xmm5, [rax+rcx*8]
addsd xmm4, xmm5
skip_fma:
	mov rdx, A_index
	mov rcx, A_offset
	mov rax, A_value
	mov r12, B_index
	mov r13, B_offset
	mov rbx, B_value
	mov r14, all_31s // array of {31, 31, ...}
	vmovaps zmm2, [r14]
	mov r15, upconvert_control // array of {0, 7, 0, 6, 0, 5, 0, 4, 0, 3, 0, 2, 0, 1, 0, 0}
	vmovaps zmm1, [r15]
	vpternlogd zmm0, zmm0, zmm0, 255
	movl esi, 21845
	kmovw k1, esi // odd bits set
	/ read 8 indices for A
	vmovdqu ymm5, [rdx+rcx*4]
	// read 8 indices for B, and put
	// them in the high part of zmm6
	vinserti64x4 zmm6, zmm5, [r12+r13*4], 1
	vpconflictd zmm7, zmm6
	// extract A vs. B comparisons
	//vextracti64x4 ymm8, zmm7, 1
	// convert comparison results to

Example 17-25. Scalar vs. Vector Update Using AVX-512CD (Contd.)

Scalar Code	Intel® AVX-512 Code

	// permute control
	  vplzcntd zmm9, zmm8
	  vptestmd k2, zmm8, zmm0
	  vpsubd zmm10, zmm2, zmm9
	  // upconvert permute controls from
	  // 32b to 64b, since data is 64b
	  vpermd zmm11{k1}, zmm1, zmm10
	  // Move A values to corresponding
	  // B values, and do FMA
	  vpermpd zmm12{k2}{z}, zmm11, [rax+rcx*8]
	  vfmadd231pd zmm4, zmm12, [rbx+r13*8]

Baseline, 1x	Speedup, 4.4x

17.17 INTEL® AVX-512 VECTOR BYTE MANIPULATION INSTRUCTIONS (VBMI)

Intel® AVX-512 VBMI instructions are a set of 512-bit instructions that are designed to speed up bit manipulation operations. The following sections describe the new instructions and show simple usage examples.

See Chapter 15, “Programming with Intel® AVX-512” in the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1 for complete instruction definitions.

Processors that provide VBMI1 and VBMI2 are enumerated by the CPUID feature flags CPUID:(EAX=07H, ECX=0):ECX[bit 01] = 1 and CPUID:(EAX=07H, ECX=0):ECX[bit 06] = 1, respectively.

17.17.1 PERMUTE PACKET BYTES ELEMENTS ACROSS LANES (VPERMB)

The VPERMB instruction is a single source, any-to-any byte permute instruction. The following figure shows a VPERMB instruction operation example.

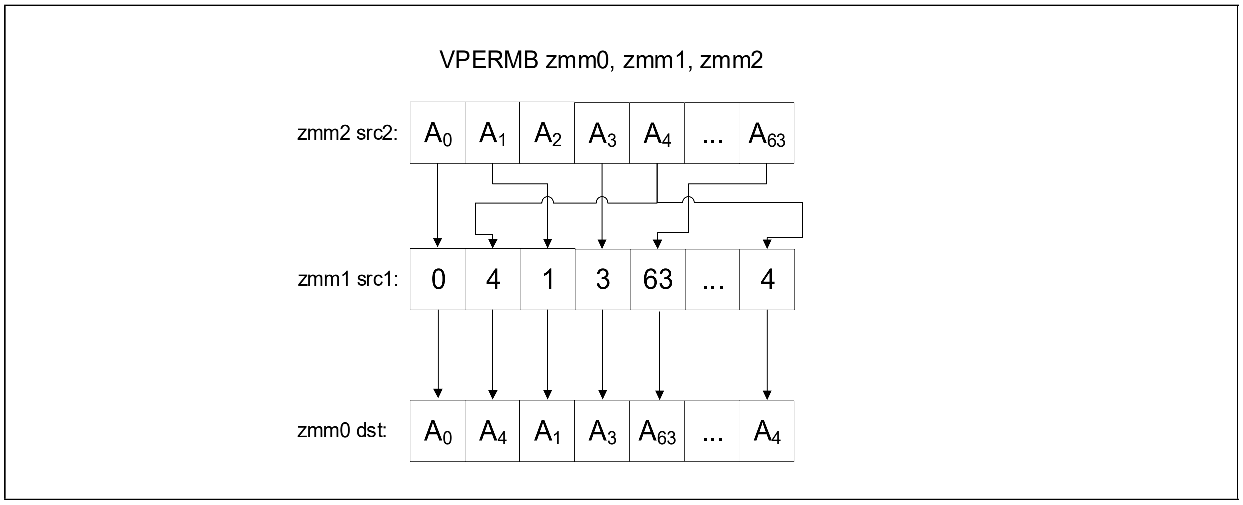

VPERMB zmm0, zmm1, zmm2
zmm2 src2:
A_0 A_1 A_2 A_3 A_4 ... A_{63}
zmm1 src1:
0 4 1 3 63 ... 4
zmm0 dst:
A_0 A_4 A_1 A_3 A_{63} ... A_4

Figure 17-18. VPERMB Instruction Operation

VPERMB Operation:

```
// vpermb zmm Dst {k1}, zmm Src1, zmm Src2
bool zero_masking=false;
unsigned char *Dst, *Src1, *Src2;

for(int i=0;i<64;i++){
        if(k1[i]){
            Dst[i]= Src2[Src1[i]];
        }else{
            Dst[i]= zero_masking? 0 : Dst[i];
        }
}
```

The following example shows a 64-byte lookup table implementation.

Scalar code:

```
void lookup(unsigned char* in_bytes, unsigned char* out_bytes, unsigned char* dictionary_bytes, int
numOfElements){
        for(int i = 0; i < numOfElements; i++) {
                out_bytes[i] = dictionary_bytes[in_bytes[i] & 63];
          }
}
```

Example 17-26. Improvement with VPERMB Implementation

Alternative 1: Vector Implementation Without VBMI	Alternative 2: VPERMB Implementation
mov rsi, dictionary_bytes mov r11, in_bytes mov rax, out_bytes mov r9d, numOfElements xor r8, r8 vpmovzxbw zmm3, [rsi] vpmovzxbw zmm4, [rsi+32] loop: vpmovzxbw zmm1, [r11+r8*1] vpmovzxbw zmm2, [r11+r8*1+32] vpermi2w zmm1, zmm3, zmm4 vpermi2w zmm2, zmm3, zmm4 vpmovwb [rax+r8*1], zmm1 vpmovwb [rax+r8*1+32], zmm2 add r8, 64 cmp r8, r9 jl loop	mov rsi, dictionary_bytes mov r11, in_bytes mov rax, out_bytes mov r9d, numOfElements xor r8, r8 vmovdqu32 zmm2, [rsi] loop: vmovdqu32 zmm1, [r11+r8*1] vpermb zmm1, zmm1, zmm2 vmovdqu32 [rax+r8*1], zmm1 add r8, 64 cmp r8, r9 jl loop
Base Measurement: 1x	Speedup: 6.5x

17.17.2 TWO-SOURCE BYTE PERMUTE ACROSS LANES (VPERMI2B, VPERMT2B)

The VPERMI2B and VPERMT2B instructions are two-source byte, permute instructions. The destination is also an operation source; in VPERMI2B the destination is the operation index, and in VPERMT2B the destination is one of the data sources.

The Figure 17-19 shows a VPERMI2B instruction operation example.

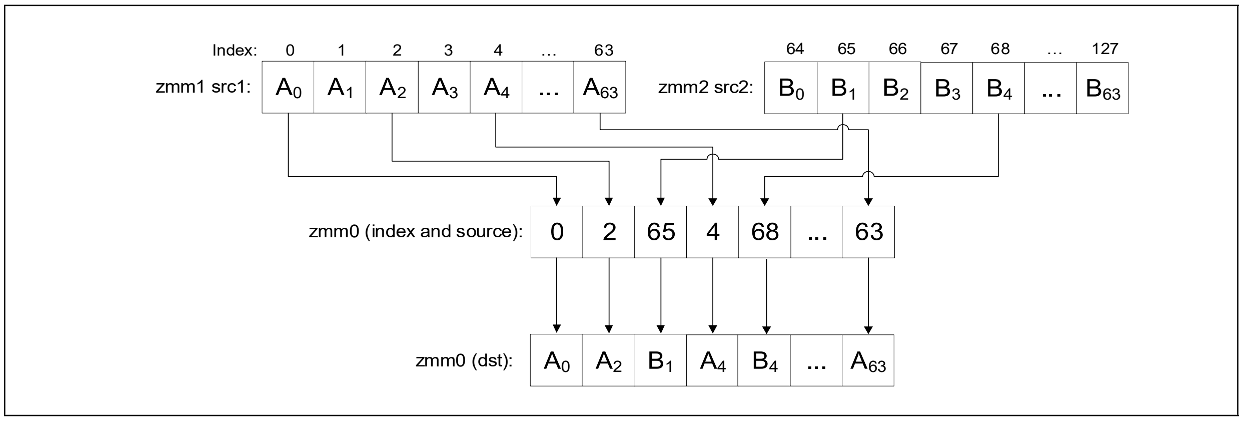

Index: 0 1 2 3 4 … 63
zmm1 src1: A_0 A_1 A_2 A_3 A_4 ... A_{63}
64 65 66 67 68 … 127
zmm2 src2: B_0 B_1 B_2 B_3 B_4 ... B_{63}
zmm0 (index and source): 0 2 65 4 68 ... 63
zmm0 (dst): A_0 A_2 B_1 A_4 B_4 ... A_{63}

Figure 17-19. VPERMI2B Instruction Operation

VPERMI2B Operation:

```
/// vpermi2b Dst{k1}, Src1, Src2
bool zero_masking=false;
unsigned char *Dst, *Src1, *Src2;
for(int i=0;i<64;i++){
        if(k1[i]){
            Dst[i]= Dst [i]>63 ? Src1[Dst [i] & 63] : Src2[Dst [i] & 63]  ;
        }else{
            Dst[i]= zero_masking? 0 : Dst[i];
        }
}
```

The following figure shows a VPERMT2B instruction operation example.

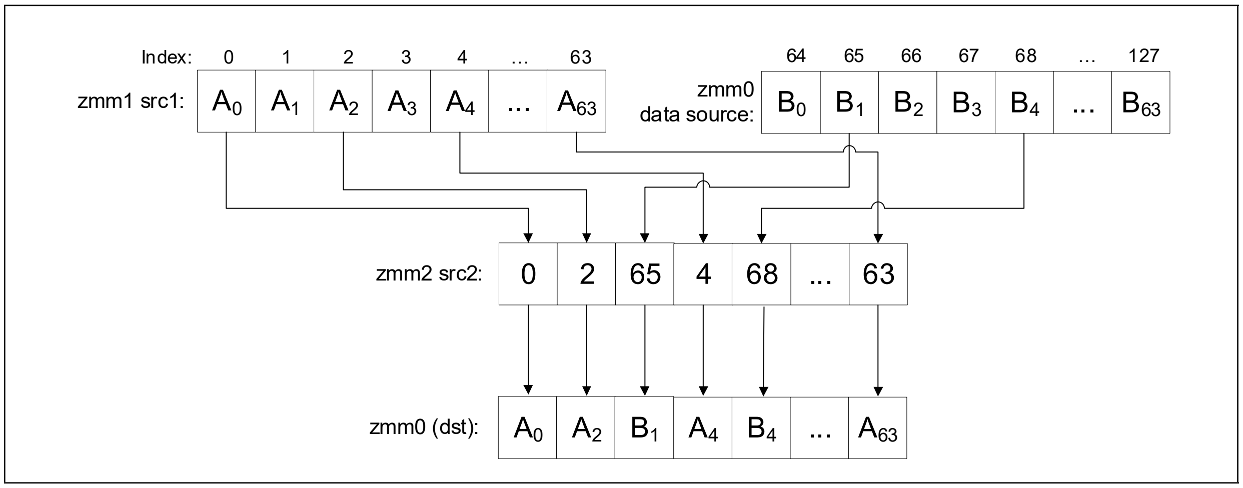


Figure 17-20. VPERMT2B Instruction Operation

VPERMT2B Operation:

```
// vpermt2b Dst{k1}, Src1, Src2
bool zero_masking=false;
unsigned char *Dst, *Src1, * Src2;
data2= copy(Dst);
for(int i=0;i<64;i++){
    if(k1[i]){
        Dst[i]= Src2[i]>63 ? Src1[Src2 [i] & 63] : Dst[Src2[i] & 63]  ;
    }else{
        Dst[i]= zero_masking? 0 : Dst[i];
    }
}
```

The following example shows a 128-byte lookup table implementation.

C Code:

```
void lookup(unsigned char* in_bytes, unsigned char* out_bytes, unsigned char* dictionary_bytes, int
numOfElements){
        for(int i = 0; i < numOfElements; i++) {
            out_bytes[i] = dictionary_bytes[in_bytes[i] & 127];
        }
}
```

Example 17-27. Improvement with VPERMI2B Implementation

Alternative 1: Vector Implementation Without VBMI	Alternative 2: VPERMI2B Implementation

//get data sent to function
mov rsi, dictionary_bytes
mov r11, in_bytes
mov rax, out_bytes
mov r9d, numOfElements
xor r8, r8
//Reorganize dictionary
vpmovzxbw zmm10, [rsi]
vpmovzxbw zmm15, [rsi+64]
vpsllw zmm15, zmm15, 8
vpord zmm10, zmm15, zmm10
vpmovzxbw zmm11, [rsi+32]
vpmovzxbw zmm15, [rsi+96]
vpsllw zmm15, zmm15, 8
vpord zmm11, zmm15, zmm11
//initialize constants
mov r10, 0x00400040
vpbroadcastw zmm12, r10d
mov r10, 0
vpbroadcastd zmm13, r10d
mov r10, 0x00ff00ff
vpbroadcastd zmm14, r10d
//start iterations
loop:
vpmovzxbw zmm1, [r11+r8*1]
vpandd zmm2, zmm1, zmm12
vpcmpw k1, zmm2, zmm13, 4
vpermi2w zmm1, zmm10, zmm11
vpsrlw zmm1{k1}, zmm1, 8
vpandd zmm1, zmm1, zmm14
vpmovwb [rax+r8*1], zmm1
add r8, 32
cmp r8, r9
jl loop
	mov rsi, dictionary_bytes
	mov r11, in_bytes
	mov rax, out_bytes
	mov r9d, numOfElements
	xor r8, r8
	vmovdqu32 zmm2, [rsi]
	vmovdqu32 zmm3, [rsi+64]
	loop:
	vmovdqu32 zmm1, [r11+r8*1]
	vpermi2b zmm1, zmm2, zmm3
	vmovdqu32 [rax+r8*1], zmm1
	add r8, 64
	cmp r8, r9
	jl loop

Base Measurement: 1x	Speedup: 5.3x

17.17.3 SELECT PACKED UNALIGNED BYTES FROM QUADWORD SOURCES (VPMULTISHIFTQB)

The VPMULTISHIFTQB instruction selects eight unaligned bytes from each input qword element of the second source operand and writes eight assembled bytes for each qword element in the destination operand.

The following figure shows a VPMULTISHIFTQB instruction operation example.

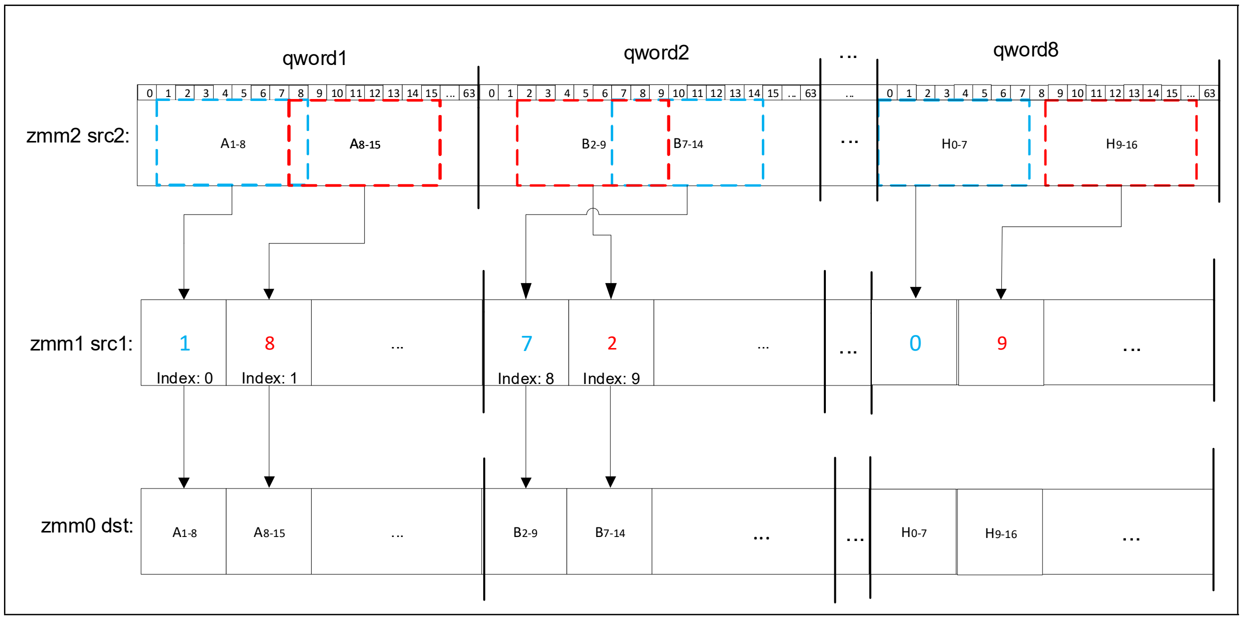


Figure 17-21. VPMULTISHIFTQB Instruction Operation

VPMULTISHIFTQB Operation:

```
// vpmultishiftqb Dst{k1},Src1,Src2
bool zero_masking=false;
unsigned char *Dst, * Src1;
unsigned __int64 *Src2;
bit * k1;
for(int i=0;i<8;i++){
        for(int j=0;j<8;j++){
            if(k1[i*8 +j]){
                Dst[i*8 +j]= (src2[i]>> Src1[i*8 +j]) &0xFF  ;
            }else{
                Dst[i*8 +j]= zero_masking? 0 : Dst[i*8 +j];
            }
        }
}
```

The following example converts a 5-bit unsigned integer array to a 1-byte unsigned integer array.

C code:

```
void decompress (unsigned char* compressedData, unsigned char* decompressedData, int
numOfElements){
        for(int i = 0; i < numOfElements; i += 8){
             unsigned __int64 * data = (unsigned __int64 * )compressedData;
             decompressedData[i+0] = * data & 0x1f;
             decompressedData[i+1] = (*data >> 5 ) & 0x1f;
             decompressedData[i+2] = (*data >> 10 ) & 0x1f;
             decompressedData[i+3] = (*data >> 15 ) & 0x1f;
             decompressedData[i+4] = (*data >> 20 ) & 0x1f;
             decompressedData[i+5] = (*data >> 25 ) & 0x1f;
             decompressedData[i+6] = (*data >> 30 ) & 0x1f;
             decompressedData[i+7] = (*data >> 35 ) & 0x1f;
             compressedData += 5;
        }
}
```

Example 17-28. Improvement with VPMULTISHIFTQB Implementation

Alternative 1: Vector Implementation Without VBMI	Alternative 2: VPMULTISHIFTQB Implementation

mov rdx, compressedData
mov r9, decompressedData
mov eax, numOfElements
shr eax,3
xor rsi, rsi
loop:
mov rcx, qword ptr [rdx]
mov r10, rcx
and r10, 0x1f
mov r11, rcx
mov byte ptr [r9+rsi*8], r10b
mov r10, rcx
shr r10, 0xa
add rdx, 0x5
and r10, 0x1f
mov byte ptr [r9+rsi*8+0x2], r10b
mov r10, rcx
shr r10, 0xf
and r10, 0x1f
mov byte ptr [r9+rsi*8+0x3], r10b
mov r10, rcx
shr r10, 0x14
and r10, 0x1f
mov byte ptr [r9+rsi*8+0x4], r10b
mov r10, rcx
shr r10, 0x19
and r10, 0x1f
mov byte ptr [r9+rsi*8+0x5], r10b
mov r10, rcx
shr r11, 0x5
shr r10, 0x1e
and r11, 0x1f
 	
shr rcx, 0x23
and r10, 0x1f
and rcx, 0x1f
mov byte ptr [r9+rsi*8+0x1], r11b
mov byte ptr [r9+rsi*8+0x6], r10b
mov byte ptr [r9+rsi*8+0x7], cl
inc rsi
cmp rsi, rax
jb loop
	//constants :
	__declspec (align(64)) const unsigned __int8
	permute_ctrl[64] = {
	  0, 1, 2, 3, 4, 0, 0, 0
	  5, 6, 7, 8, 9, 0, 0, 0
	  10, 11, 12, 13, 14, 0, 0, 0
	  15, 16, 17, 18, 19, 0, 0, 0
	  20, 21, 22, 23, 24, 0, 0, 0
	  25, 26, 27, 28, 29, 0, 0, 0
	  30, 31, 32, 33, 34, 0, 0, 0
	  35, 36, 37, 38, 39, 0, 0, 0
	};
	__declspec (align(64)) const unsigned __int8
	multishift_ctrl[64] = {
	  0, 5, 10, 15, 20, 25, 30, 35
	  0, 5, 10, 15, 20, 25, 30, 35
	  0, 5, 10, 15, 20, 25, 30, 35
	  0, 5, 10, 15, 20, 25, 30, 35
	  0, 5, 10, 15, 20, 25, 30, 35
	  0, 5, 10, 15, 20, 25, 30, 35
	  0, 5, 10, 15, 20, 25, 30, 35
	  0, 5, 10, 15, 20, 25, 30, 35
	};
	//asm:
	mov rsi, compressedData
	mov rdi, decompressedData
	mov r8d, numOfElements
	lea r8, [rdi+r8]
	mov r9, 0x1F1F1F1F
	vpbroadcastd zmm12, r9d
	vmovdqu32 zmm10, permute_ctrl
	vmovdqu32 zmm11, multishift_ctrl
	
	loop:
	vmovdqu32 zmm1, [rsi]
	vpermb zmm2, zmm10, zmm1
	vpmultishiftqb zmm2, zmm11, zmm2
	vpandq zmm2, zmm12, zmm2
	vmovdqu32 [rdi], zmm2
	add rdi, 64
	add rsi, 40
	cmp rdi, r8
	jl loop

Base Measurement: 1x	Speedup: 26x

17.18 FMA LATENCY

When executing in 512-bit register port scheme, Port 0 FMA has a latency of four cycles, and Port 5 FMA has a latency of six cycles. Bypass can have a -2 (fast bypass) to +1 cycle delay. Therefore, instructions that execute on the Skylake microarchitecture FMA have a latency of four to seven cycles.

The instructions are divided into the following two groups.

- Group A Instructions: vadd*; vfmadd*; vfnmsub*; vfnmadd*; vfnmsub*; vmax*; vmin*; vmul*; vscalef*; vsub*; vcvt*; vgetexp*; vfixupimm*; vrange*; vgetmant*; vreduce*; vcmp*, vcomi*, vdpp*, vhadd*, vhsub*, vrndscale*, vround*
- Group B Instructions: vpmaddubsw; vpmaddwd; vpmuldq; vpmulhrsw; vpmulhuw; vpmulhw; vpmullw; vpmuludq

The FMA unit supports fast bypass when all instruction sources come from the FMA unit. In this case Group A has a latency of four cycles for both ports 0 and 5, and Group B has a latency of five cycles for both ports 0 and 5.

The figure below explains fast bypass when all sources come from the FMA unit.

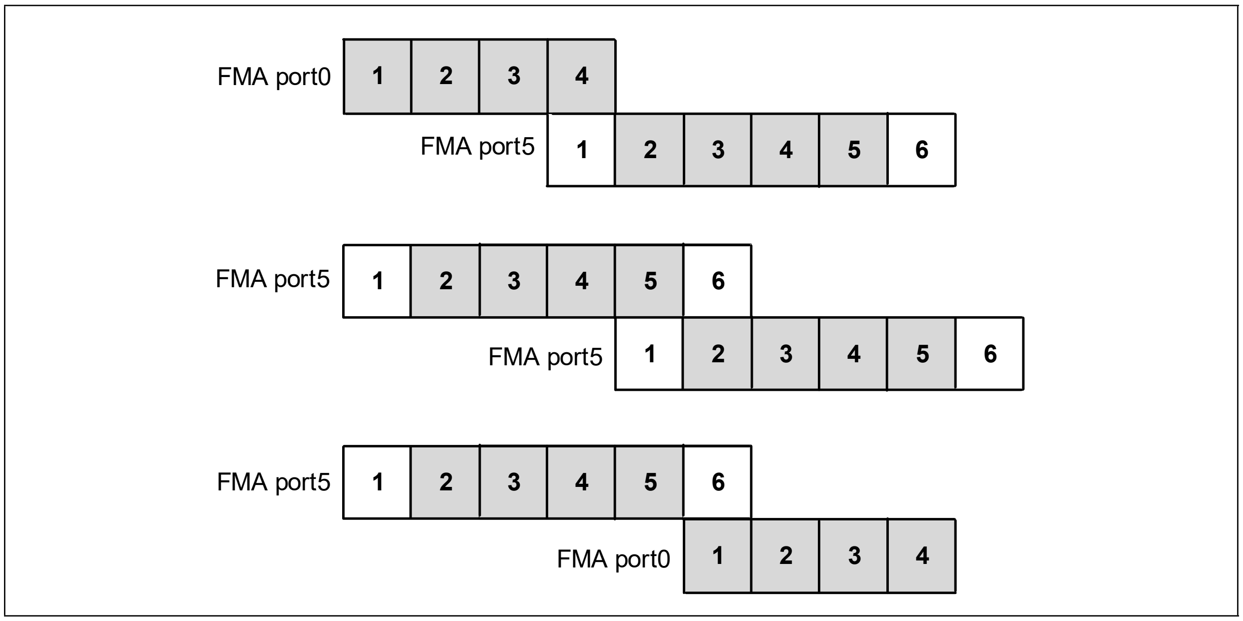


Figure 17-22. Fast Bypass When All Sources Come from FMA Unit

The gray boxes represent compute cycles. The white boxes represent data transfer for the port5 FMA unit.

If fast bypass is not used, that is, when not all sources come from the FMA unit, group A instructions have a latency of four cycles on Port0 and six cycles on port5, while group B instructions have an additional cycle and hence have a latency of five cycles on Port0 and seven cycles on port5.

The following table summarizes the FMA unit latency for the various options.

Table 17-9. FMA Unit Latency

	Fast Bypass (FMA Data Reuse)		No Fast Bypass (No FMA Data Reuse)	
Instruction Group	**Port 0**	**Port 5**	**Port 0**	**Port 5**
Group A	4	4	4	6
Group B	5	5	5	7

17.19 MIXING INTEL® AVX OR INTEL® AVX-512 EXTENSIONS WITH INTEL® STREAMING SIMD EXTENSIONS (INTEL® SSE) CODE

There are two main instruction groups that affect the processor states:

- Group A: Instruction types that either set bits 128-511 of vector registers 0-15 to zero, or do not modify them at all.
  - Intel SSE instructions.
  - 128-bit Intel AVX instructions, 128-bit Intel AVX-512 instructions.
  - 256-bit (ymm16-ymm31) Intel AVX-512 instructions.
  - 512-bit (zmm16-zmm31) Intel AVX-512 instructions.
  - AVX-512 instructions that write to mask registers k0-k7.
  - GPR instructions.
- Group B: Instructions types that modify bits 128-511 of vector registers 0-15.
  - 256-bit (ymm0-ymm15) Intel AVX instructions, Intel AVX-512 instructions.
  - 512-bit (zmm0-zmm15) Intel AVX-512 instructions.

The following figure illustrates Skylake Server microarchitecture's model for mixing Intel AVX instructions or Intel AVX-512 instructions with Intel SSE instructions.

The implementation is similar to Skylake client microarchitecture, where every Intel SSE instruction executed in Dirty Upper State (2) needs to preserve bits 128-511 of the destination register, and therefore the operation has an additional dependency on the destination register and a blend operation with bits 128-511.

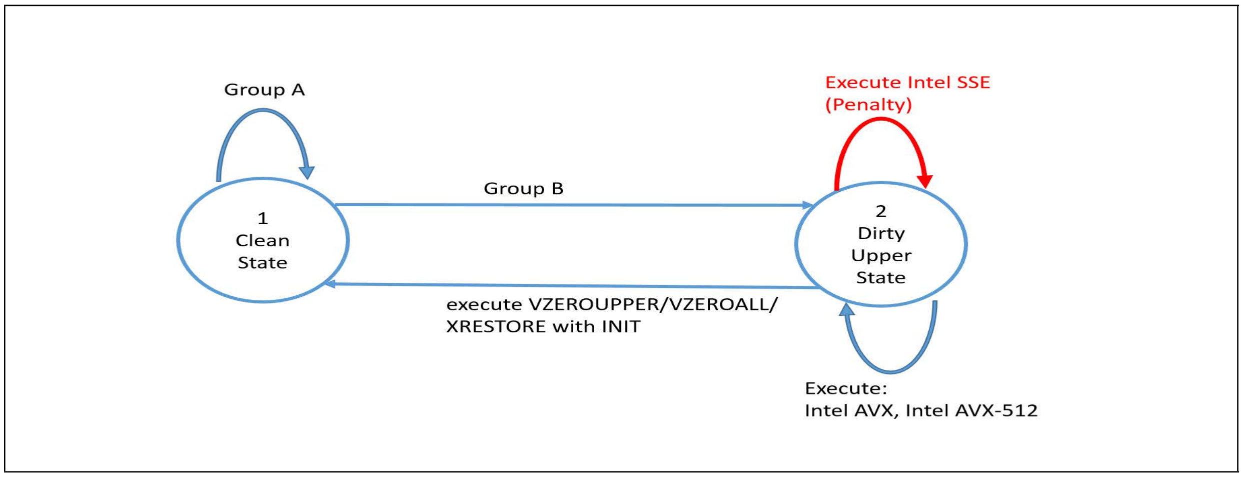

Group A
Execute Intel SSE (Penalty)
Group B
1 Clean State
2 Dirty Upper State
execute VZEROUPPER/VZEROALL/ XRESTORE with INIT
Execute: Intel AVX, Intel AVX-512

Figure 17-23. Mixing Intel AVX Instructions or Intel AVX-512 Instructions with Intel SSE Instructions

Recommendations:

- When mixing group B instructions with Intel SSE instructions, or suspecting that such a mixture might occur, use the VZEROUPPER instruction whenever a transition is expected.
- Add VZEROUPPER after group B instructions were executed and before any function call that might lead to an Intel SSE instruction execution.
- Add VZEROUPPER at the end of any function that uses group B instructions.
- Add VZEROUPPER before thread creation if not already in a clean state so that the thread does not inherit a Dirty Upper State.

17.20 MIXING ZMM VECTOR CODE WITH XMM/YMM

Skylake microarchitecture has two port schemes, one for using 256-bit or less registers, and another for using 512-bit registers.

When using registers up to or including 256 bits, FMA operations dispatch to ports 0 and 1 and SIMD operations dispatch to ports 0, 1 and 5. When using 512-bit register operations, both FMA and SIMD operations dispatch to ports 0 and 5.

The maximum register width in the reservation station (RS) determines the 256 or 512 port scheme.

Notice that when using AVX-512 encoded instructions with YMM registers, the instructions are considered to be 256-bit wide.

The result of the 512-bit port scheme is that XMM or YMM code dispatches to two ports (0 and 5) instead of three ports (0, 1, and 5) and may have lower throughput and longer latency compared to the 256-bit port scheme.

Example 17-29. 256-bit Code vs. 256-bit Code Mixed with 512-bit Code

256-bit Code Only	256-bit Code Mixed with 512-bit Code
Loop: vpbroadcastd ymm0, dword ptr [rsp] vfmadd213ps ymm7, ymm7, ymm7 vfmadd213ps ymm8, ymm8, ymm8 vfmadd213ps ymm9, ymm9, ymm9 vfmadd213ps ymm10, ymm10, ymm10 vfmadd213ps ymm11, ymm11, ymm11 vfmadd213ps ymm12, ymm12, ymm12 vfmadd213ps ymm13, ymm13, ymm13 vfmadd213ps ymm14, ymm14, ymm14 vfmadd213ps ymm15, ymm15, ymm15 vfmadd213ps ymm16, ymm16, ymm16 vfmadd213ps ymm17, ymm17, ymm17 vfmadd213ps ymm18, ymm18, ymm18 vpermd ymm1, ymm1, ymm1 vpermd ymm2, ymm2, ymm2 vpermd ymm3, ymm3, ymm3 vpermd ymm4, ymm4, ymm4 vpermd ymm5, ymm5, ymm5 vpermd ymm6, ymm6, ymm6 dec rdx jnle Loop	Loop: vpbroadcastd zmm0, dword ptr [rsp] vfmadd213ps ymm7, ymm7, ymm7 vfmadd213ps ymm8, ymm8, ymm8 vfmadd213ps ymm9, ymm9, ymm9 vfmadd213ps ymm10, ymm10, ymm10 vfmadd213ps ymm11, ymm11, ymm11 vfmadd213ps ymm12, ymm12, ymm12 vfmadd213ps ymm13, ymm13, ymm13 vfmadd213ps ymm14, ymm14, ymm14 vfmadd213ps ymm15, ymm15, ymm15 vfmadd213ps ymm16, ymm16, ymm16 vfmadd213ps ymm17, ymm17, ymm17 vfmadd213ps ymm18, ymm18, ymm18 vpermd ymm1, ymm1, ymm1 vpermd ymm2, ymm2, ymm2 vpermd ymm3, ymm3, ymm3 vpermd ymm4, ymm4, ymm4 vpermd ymm5, ymm5, ymm5 vpermd ymm6, ymm6, ymm6 dec rdx jnle Loop
Baseline 1x	Slowdown: 1.3x

In the 256-bit code only example, the FMAs are dispatched to ports 0 and 1, and *permd* is dispatched to port 5 as the broadcast instruction is 256 bits wide. In the 256-bit and 512-bit mixed code example, the broadcast is 512 bits wide; therefore, the processor uses the 512-bit port scheme where the FMAs dispatch to ports 0 and 5 and *permd* to port 5, thus increasing the pressure on port 5.

17.21 SERVERS WITH A SINGLE FMA UNIT

Some processors based on Skylake microarchitecture have two Intel AVX-512 FMA units, on ports 0 and 5, while other processors based on Skylake microarchitecture have a single Intel AVX-512 FMA unit, which is located on port 0.

Code that is optimized to run on a processor with two FMA units might not be optimal when run on a processor with one FMA unit.

The following example code shows how to detect whether a system has one or two Intel AVX-512 FMA units. It includes the following:

- An Intel AVX-512 warmup.
- A function that executes only FMA instructions.
- A function that executes both FMA and shuffle instructions.
- Code that, based on the results of these two tests, identifies whether the processor has one or two FMA units.

Notice that each test is executed three times to improve test accuracy.

To reduce the program overhead, it is highly recommended not to execute this test in every function call, but as part of installation, or once at startup.

The differentiation between the two processors is based on the ratio between the two throughput tests. Processors with two FMA units are able to run the FMA-only test twice as fast as the FMA and shuffle test. However, a processor with one FMA unit will run both tests at the same speed.

Example 17-30. Identifying One or Two FMA Units in a Processor Based on Skylake Microarchitecture

```
#include <string.h>
#include <stdlib.h>
#include <immintrin.h>
#include <stdio.h>
#include <stdint.h>

static uint64_t rdtsc(void) {
  unsigned int ax, dx;

  __asm__ __volatile__ ("rdtsc" : "=a"(ax), "=d"(dx));

  return ((((uint64_t)dx) << 32) | ax);
}

uint64_t fma_shuffle_tpt(uint64_t loop_cnt){
   uint64_t loops =  loop_cnt;
   __declspec(align(64)) double one_vec[8] = {1, 1, 1, 1,1, 1, 1, 1};
   __declspec(align(64)) int shuf_vec[16] = {0, 1, 2, 3,4, 5, 6, 7,8, 9, 10, 11,12, 13, 14, 15};
    __asm
       {
       vmovups zmm0, [one_vec]
       vmovups zmm1, [one_vec]
       vmovups zmm2, [one_vec]
       vmovups zmm3, [one_vec]
       vmovups zmm4, [one_vec]
       vmovups zmm5, [one_vec]
       vmovups zmm6, [one_vec]
       vmovups zmm7, [one_vec]
       vmovups zmm8, [one_vec]
       vmovups zmm9, [one_vec]
vmovups zmm10, [one_vec]
       vmovups zmm11, [one_vec]
       vmovups zmm12, [shuf_vec]
       vmovups zmm13, [shuf_vec]
       vmovups zmm14, [shuf_vec]
       vmovups zmm15, [shuf_vec]
       vmovups zmm16, [shuf_vec]
       vmovups zmm17, [shuf_vec]
       vmovups zmm18, [shuf_vec]
       vmovups zmm19, [shuf_vec]
       vmovups zmm20, [shuf_vec]
       vmovups zmm21, [shuf_vec]
```

Example 17-30. Identifying One or Two FMA Units in a Processor Based on Skylake Microarchitecture

```
        vmovups zmm22, [shuf_vec]
        vmovups zmm23, [shuf_vec]
        vmovups zmm30, [shuf_vec]
        mov rdx, loops
loop1:
        vfmadd231pd zmm0, zmm0, zmm0
        vfmadd231pd zmm1, zmm1, zmm1
        vfmadd231pd zmm2, zmm2, zmm2
        vfmadd231pd zmm3, zmm3, zmm3
        vfmadd231pd zmm4, zmm4, zmm4
        vfmadd231pd zmm5, zmm5, zmm5
        vfmadd231pd zmm6, zmm6, zmm6
        vfmadd231pd zmm7, zmm7, zmm7
        vfmadd231pd zmm8, zmm8, zmm8
        vfmadd231pd zmm9, zmm9, zmm9
        vfmadd231pd zmm10, zmm10, zmm10
        vfmadd231pd zmm11, zmm11, zmm11
        vpermd zmm12, zmm30, zmm30
        vpermd zmm13, zmm30, zmm30
        vpermd zmm14, zmm30, zmm30
        vpermd zmm15, zmm30, zmm30
        vpermd zmm16, zmm30, zmm30
        vpermd zmm17, zmm30, zmm30
        vpermd zmm18, zmm30, zmm30
        vpermd zmm19, zmm30, zmm30
        vpermd zmm20, zmm30, zmm30
        vpermd zmm21, zmm30, zmm30
        vpermd zmm22, zmm30, zmm30
        vpermd zmm23, zmm30, zmm30
        dec rdx
        jg loop1
        }
}
uint64_t fma_only_tpt(int loop_cnt){
   uint64_t loops =  loop_cnt;
   __declspec(align(64)) double one_vec[8] = {1, 1, 1, 1,1, 1, 1, 1};
        __asm
        {
        vmovups zmm0, [one_vec]
        vmovups zmm1, [one_vec]
        vmovups zmm2, [one_vec]
        vmovups zmm3, [one_vec]
        vmovups zmm4, [one_vec]
        vmovups zmm5, [one_vec]
        vmovups zmm6, [one_vec]
        vmovups zmm7, [one_vec]
        vmovups zmm8, [one_vec]
        vmovups zmm9, [one_vec]
        vmovups zmm10, [one_vec]
        vmovups zmm11, [one_vec]
        mov rdx, loops
```

Example 17-30. Identifying One or Two FMA Units in a Processor Based on Skylake Microarchitecture

```
  loop1:
      vfmadd231pd zmm0, zmm0, zmm0
      vfmadd231pd zmm1, zmm1, zmm1
      vfmadd231pd zmm2, zmm2, zmm2
      vfmadd231pd zmm3, zmm3, zmm3
vfmadd231pd zmm4, zmm4, zmm4
      vfmadd231pd zmm5, zmm5, zmm5
      vfmadd231pd zmm6, zmm6, zmm6
      vfmadd231pd zmm7, zmm7, zmm7
      vfmadd231pd zmm8, zmm8, zmm8
      vfmadd231pd zmm9, zmm9, zmm9
      vfmadd231pd zmm10, zmm10, zmm10
      vfmadd231pd zmm11, zmm11, zmm11
      dec rdx
      jg loop1
      }
}

int main()
{
  int i;
  uint64_t fma_shuf_tpt_test[3];
  uint64_t fma_shuf_tpt_test_min;
  uint64_t fma_only_tpt_test[3];
  uint64_t fma_only_tpt_test_min;
  uint64_t start = 0;
  uint64_t number_of_fma_units_per_core = 2;

  /*********************************************************/
  /* Step 1: Warmup */
  /*********************************************************/
  fma_only_tpt(100000);

  /*********************************************************/
  /* Step 2: Execute FMA and Shuffle TPT Test */
  /*********************************************************/

  for(i = 0; i < 3; i++){
      start = rdtsc();
      fma_shuffle_tpt(1000);
      fma_shuf_tpt_test[i] = rdtsc() - start;
  }

  /*********************************************************/
  /* Step 3: Execute FMA only TPT Test  */
  /*********************************************************/
  for(i = 0; i < 3; i++){
      start = rdtsc();
      fma_only_tpt(1000);
      fma_only_tpt_test[i] = rdtsc() - start;
  }
```

Example 17-30. Identifying One or Two FMA Units in a Processor Based on Skylake Microarchitecture

```
    /*******************************************************/
    /* Step 4: Decide if 1 FMA server or 2 FMA server */
    /*******************************************************/
    fma_shuf_tpt_test_min = fma_shuf_tpt_test[0];
    fma_only_tpt_test_min = fma_only_tpt_test[0];
    for(i = 1; i < 3; i++){
        if ((int)fma_shuf_tpt_test[i] < (int)fma_shuf_tpt_test_min) fma_shuf_tpt_test_min = fma_shuf_tpt_test[i];
        if ((int)fma_only_tpt_test[i] < (int)fma_only_tpt_test_min) fma_only_tpt_test_min = fma_only_tpt_test[i];
    }

    if(((double)fma_shuf_tpt_test_min/(double)fma_only_tpt_test_min) < 1.5){
        number_of_fma_units_per_core = 1;
    }

    printf("%d FMA server\n", number_of_fma_units_per_core);
    return 0;
}
```

17.22 GATHER/SCATTER TO SHUFFLE (G2S/STS)

17.22.1 GATHER TO SHUFFLE IN STRIDED LOADS

In cases where there is data locality between gathered elements in memory, performance can be improved by replacing the gather instruction with a software sequence.

This section discusses the very common strided load pattern. Strided loads are sets of loads where the offset in memory between two consecutive loads is constant.

The following examples show three different code variations performing an Array of Structures (AOS) to Structure of Arrays (SOA) transformation. The code separates the real and imaginary elements in a complex array into two separate arrays.

Consider the following C code:

```
for(int i=0;i<len;i++){
        Real_buffer[i] = Complex_buffer[i].real;
        Imaginary_buffer[i] = Complex_buffer[i].imag;
}
```

Example 17-31. Gather to Shuffle in Strided Loads Example

Alternative 1: Intel® AVX-512 vpgatherdd	Alternative 2: G2S Using Intel® AVX-512 vpermi2d
loop: vpcmpeqb k1, xmm0, xmm0 vpcmpeqb k2, xmm0, xmm0 movsxd rdx, edx movsxd rdi, esi inc esi	vmovups zmm4, [rdx+r9*8] vmovups zmm0, [rdx+r9*8+0x40] vmovups zmm5, [rdx+r9*8+0x80] vmovups zmm1, [rdx+r9*8+0xc0] vmovaps zmm2, zmm7 vmovaps zmm3, zmm7

Example 17-31. Gather to Shuffle in Strided Loads Example (Contd.)

Alternative 1: Intel® AVX-512 vpgatherdd	Alternative 2: G2S Using Intel® AVX-512 vpermi2d
shl rdi, 0x7 vpxord zmm2, zmm2, zmm2 lea rax, [r8+rdx*8] add edx, 0x20 vpgatherdd zmm2{k1}, [rax+zmm1*4] vpxord zmm3, zmm3, zmm3 vpxord zmm4, zmm4, zmm4 vpxord zmm5, zmm5, zmm5 vpgatherdd zmm3{k2}, [rax+zmm0*4] vpcmpeqb k3, xmm0, xmm0 vpcmpeqb k4, xmm0, xmm0 vmovups [r9+rdi*1], zmm2 vmovups [rcx+rdi*1], zmm3 vpgatherdd zmm4{k3}, [rax+zmm1*4+0x80] vpgatherdd zmm5{k4}, [rax+zmm0*4+0x80] vmovups [r9+rdi*1+0x40], zmm4 vmovups [rcx+rdi*1+0x40], zmm5 cmp esi, r14d jb loop	vpermi2d zmm2, zmm4, zmm0 vpermt2d zmm4, zmm6, zmm0 vpermi2d zmm3, zmm5, zmm1 vpermt2d zmm5, zmm6, zmm1 vmovdqu32 [rcx+r9*4], zmm2vmovdqu32 [rcx+r9*4+0x40], zmm3 vmovdqu32 [r8+r9*4], zmm4 vmovdqu32 [r8+r9*4+0x40], zmm5add r9, 0x20 cmp r9, r10 jb loop
Baseline 1x	Speedup: 4.8x

The following constants were loaded into zmm registers and used as gather and permute indices:

Zmm0 (Alternative 1), zmm6 (Alternative 2)

__declspec (align(64)) const __int32 gather_imag_index[16] = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31};

Zmm1 (Alternative 1), zmm7 (Alternative 2)

__declspec (align(64)) const __int32 gather_real_index[16] = {0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30};

Recommendation: For best performance, replace strided loads where the stride is short, with a sequence of loads and permutes.

17.22.2 SCATTER TO SHUFFLE IN STRIDED STORES

The following is an Scatter to Shuffle example that replaces scatter with permute and store instructions

Consider the following C code:

```
for(int i=0;i<len;i++){
        Complex_buffer[i].real = Real_buffer[i];
        Complex_buffer[i].imag = Imaginary_buffer[i]; }
```

Example 17-32. Gather to Shuffle in Strided Stores Example

Alternative 1: Intel® AVX-512 vscatterdps	Alternative 2: S2S using Intel® AVX-512 vpermi2d
loop: vpcmpeqb k1, xmm0, xmm0 lea r11, [r8+rcx*4] vpcmpeqb k2, xmm0, xmm0 vmovups zmm2, [rax+rsi*4] vmovups zmm3, [r9+rsi*4] vscatterdps [r11+zmm1*4]{k1}, zmm2 vscatterdps [r11+zmm0*4]{k2}, zmm3 add rsi, 0x10 add rcx, 0x20 cmp rsi, r10 jl loop	loop: vmovups zmm4, [rax+r8*4] vmovups zmm2, [r10+r8*4] vmovaps zmm3, zmm1 add r8, 0x10 vpermi2d zmm3, zmm4, zmm2 vpermt2d zmm4, zmm0, zmm2 vmovups [r9+rsi*4], zmm3 vmovups [r9+rsi*4+0x40], zmm4 add rsi, 0x20 cmp r8, r11 jl loop
Baseline 1x	Speedup: 4.4x

The following constants were used as scatter indices:

Zmm1:

__declspec (align(64)) const __int32 scatter_real_index[16] = {0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30};

Zmm0:

__declspec (align(64)) const __int32 scatter_imag_index[16] = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31};

The following constants were used as permute indices:

Zmm1:

__declspec (align(64)) const __int32 first_half[16] = {0, 16, 1, 17, 2, 18, 3, 19, 4, 20, 5, 21, 6, 22, 7, 23};

Zmm0:

__declspec (align(64)) const __int32 second_half[16] = {8, 24, 9, 25, 10, 26, 11, 27, 12, 28, 13, 29, 14, 30, 15, 31};

17.22.3 GATHER TO SHUFFLE IN ADJACENT LOADS

In cases where the gathered elements are grouped into adjacent sequences, the gather instruction can be replaced by a software sequence to improve performance.

The following example shows how to load vectors when elements are adjacent.

Notice that in this case the order of the elements in the arrays is set according to an index buffer and therefore the software optimization discussed in Section 17.22.1 is not applicable in this case.

Consider the following C code:

```
typedef struct{
    double var[4];
} ElemStruct;

const int* indices = Indices;
const ElemStruct *in = (const ElemStruct*) InputBuffer;
double* restrict out = OutputBuffer;

for (int i = 0; i < width; i++){
            for (int j = 0; j < 4; j++){
                out[i*4+j] = in[indices[i]].var[j];
        }
}
```

Example 17-33. Gather to Shuffle in Adjacent Loads Example

Alternative 1: vgatherdpd Implementation	Alternative 2: Load and Masked broadcast
loop: vpbroadcastd ymm3, [r9+rsi*4] mov r15d, esi vpbroadcastd xmm2, [r9+rsi*4+0x4] add rsi, 0x2 vpbroadcastd ymm3{k1}, xmm2 vpmulld ymm4, ymm3, ymm1 vpaddd ymm5, ymm4, ymm0 vpcmpeqb k2, xmm0, xmm0 shl r15d, 0x2 movsxd r15, r15d vpxord zmm6, zmm6, zmm6 vgatherdpd zmm6{k2}, [r10+ymm5*1] vmovups [r11+r15*8], zmm6 cmp rsi, rdi jl loop	loop: movsxd r11, [r10+rcx*4] shl r11, 0x5 vmovupd ymm0, [r9+r11*1] movsxd r11, [r10+rcx*4+0x4] shl r11, 0x5 vbroadcastf64x4 zmm0{k1}, [r9+r11*1] mov r11d, ecx shl r11d, 0x2 add rcx, 0x2 movsxd r11, r11d vmovups [r8+r11*8], zmm0 cmp rcx, rsi jl loop
Baseline 1x	Speedup: 2.2x

The following constants were used in the vgatherdpd implementation:

```
ymm0:
        __declspec (align(64)) const __int32 index_inc[8] = {0, 8, 16, 24, 0, 8, 16, 24};
ymm1:
        __declspec (align(64)) const __int32 index_scale[8] = {32, 32, 32, 32, 32, 32, 32, 32};

K1 register value is 0xF0.
```

17.23 DATA ALIGNMENT

This section explains the benefit of aligning data when using the Intel AVX-512 instructions and proposes some methods to improve performance when such alignment is not possible. Most examples in this section are variations of the SAXPY kernel. SAXPY is the Scalar Alpha * X + Y algorithm.

The C code below is a C implementation of SAXPY.

```
for (int i = 0; i < n; i++)
{
c[i] = alpha * a[i] + b[i];
}
```

17.23.1 ALIGN DATA TO 64 BYTES

Aligning data to vector length is recommended. For best results, when using Intel AVX-512 instructions, align data to 64 bytes.

When doing a 64-byte Intel AVX-512 unaligned load/store, every load/store is a cache-line split, since the cache-line is 64 bytes. This is double the cache line split rate of Intel AVX2 code that uses 32-byte registers. A high cache-line split rate in memory-intensive code can cause poor performance.

The following table shows how the performance of the memory intensive SAXPY code is affected by misaligning input and output buffers. The data in the table is based on the following code.

Example 17-34. Data Alignment

```
__asm {
        mov rax, src1
        mov rbx, src2
        mov rcx, dst
        mov rdx, len
        xor rdi, rdi
        vbroadcastss zmm0, alpha
mainloop:
        vmovups zmm1, [rax]
        vfmadd213ps zmm1, zmm0, [rbx]
        vmovups [rcx], zmm1

        vmovups zmm1, [rax+0x40]
        vfmadd213ps zmm1, zmm0, [rbx+0x40]
        vmovups [rcx+0x40], zmm1

        vmovups zmm1, [rax+0x80]
        vfmadd213ps zmm1, zmm0, [rbx+0x80]
        vmovups [rcx+0x80], zmm1

        vmovups zmm1, [rax+0xC0]
        vfmadd213ps zmm1, zmm0, [rbx+0xC0]
        vmovups [rcx+0xC0], zmm1

        add rax, 256
        add rbx, 256
        add rcx, 256
        add rdi, 64
        cmp rdi, rdx
        jl mainloop
}
```

The Table 17-10 summarizes the data alignment effects on SAXPY performance with speedup values for the various options.

Table 17-10. Data Alignment Effects on SAXPY Performance vs. Speedup Value

Data Alignment Effects on SAXPY Performance	Speedup
Alternative 1: Both sources and the destination are 64-byte aligned.	Baseline, 1.0
Alternative 2: Both sources are 64-byte aligned, destination has a 4 byte offset from the alignment.	0.66x
Alternative 3: Both sources and the destinations have 4 bytes offset from the alignment.	0.59x
Alternative 4: One source has a 4 byte offset from the alignment, the other source and the destination are 64-byte aligned.	0.77x

17.24 DYNAMIC MEMORY ALLOCATION AND MEMORY ALIGNMENT

Consider the following structure:

```
float3_SOA {
 __declspec(align(64)) float x[16];
 __declspec(align(64)) float y[16];
};
```

The memory allocated for the structure is aligned to 64 bytes if you use this structure as follows:

```
float3_SOA f;
```

However, if you use dynamic memory allocation as follows, the declspec directive is ignored and the 64-byte memory alignment is not guaranteed:

```
float3_SOA* stPtr = new float3_SOA();
```

In this case, you should use dynamic aligned memory allocation and/or redefine operator *new*.

Recommendation: Align data to 64 bytes, when possible, using the following guidelines.

- Use dynamic data alignment using the _mm_malloc intrinsic instruction with the Intel® Compiler, or _aligned_malloc of the Microsoft* Compiler.
  - For example:

```
//dynamically allocating 64byte aligned buffer with 2048 float elements.
InputBuffer = (float*) _mm_malloc (2048*sizeof(float), 64);
```

- Use static data alignment using __declspec(align(64)).
  - For example:

```
//Statically allocating 64byte aligned buffer with 2048 float elements.
__declspec(align(64)) float InputBuffer[2048];
```

17.25 DIVISION AND SQUARE ROOT OPERATIONS

It is possible to speed up single-precision divide and square root calculations using the VRSQRT14PS/VRSQRT14PD and VRCP14PS/VRCP14PD instructions. These instructions yield an approximation (with 14 bits accuracy) of the Reciprocal Square Roots / Reciprocal Divide of their input.

The Intel AVX-512 implementation of these instructions is pipelined and has:

- For 256-bit vectors: latency of four cycles with a throughput of one instruction every cycle.
- For 512-bit vectors: latency of six cycles with a throughput of one instruction every two cycles.

Skylake microarchitecture introduces the packed-double (PD) variants of reciprocal square-root and reciprocal divide: VRSQRT14PD and VRCP14PD (respectively).

The VRSQRT14PS/VRSQRT14PD and VRCP14PS/VRCP14PD instructions can be used with a single Newton-Raphson iteration or other polynomial approximation to achieve almost the same precision as the VDIVPS and VSQRTPS instructions (see the Intel® 64 and IA-32 Architectures Software Developer's Manual for more information on these instructions), and may yield a much higher throughput.

If the full precision (IEEE) must be maintained, a low latency and high throughput can be achieved due to the significant performance improvement of the Skylake microarchitecture to DIVPS and SQRTPS, comparing to their performance on previous microarchitectures. This is illustrated in Figure 17-13.

NOTE

In some cases, when the divide or square root operations are part of a larger algorithm that hides some of the latency of these operations, the approximation with Newton-Raphson can slow down execution, because more micro-ops, coming from the additional instructions, fill the pipe.

The following sections show the operations with recommended calculation methods depending on the desired accuracy level.

NOTE

There are two definitions for approximation error of a value and it's approximation ν_{approx}:

Absolute error = $|\nu - \nu_{approx}|$

Relative error = $|\nu - \nu_{approx}| / |\nu|$

In this chapter, the “number of bits” error is relative, and not the error of absolute values.

The value ν to which we compare our approximation should be as accurate as possible, better double accuracy.

17.25.1 DIVIDE AND SQUARE ROOT APPROXIMATION METHODS

Table 17-11. Skylake Microarchitecture Recommendations for DIV/SQRT Based Operations (Single Precision)

Operation	Accuracy	Recommended Method
Divide	24 bits (IEEE)	DIVPS
	23 bits	RCP14PS + MULPS + 1 Newton-Raphson iteration
	14 bits	RCP14PS + MULPS
Reciprocal Square Root	22 bits	SQRTPS + DIVPS
	23 bits	RSQRT14PS + 1 Newton-Raphson iteration
	14 bits	RSQRT14PS
Square Root	24 bits (IEEE)	SQRTPS
	23 bits	RSQRT14PS + MULPS + 1 Newton-Raphson iteration
	14 bits	RSQRT14PS + MULPS

Table 17-12. Skylake Microarchitecture Recommendations for DIV/SQRT Based Operations (Double Precision)

Operation	Accuracy	Recommended Method
Divide	53 bits (IEEE)	DIVPD
	52 bits	RCP14PD + MULPD + 2 Newton-Raphson iterations
	26 bits	RCP14PD + MULPD + 1 Newton-Raphson iterations
	14 bits	RCP14PD + MULPD
Reciprocal Square Root	53 bits (IEEE)	SQRTPD + DIVPD
	52 bits	RSQRT14PD+2 N-R + error correction or SQRTPD + DIVPD
	50 bits	RSQRT14PD + Polynomial approximation
	26 bits	RSQRT14PD+1 N-R
	14 bits	RSQRT14PD
Square Root	51 bits (IEEE)	SQRTPD
	52 bits	RSQRT14PD + MULPD + Polynomial approximation
	26 bits	RSQRT14PD + MULPD + 1 N-R
	14 bits	RSQRT14PD + MULPD

17.25.2 DIVIDE AND SQUARE ROOT PERFORMANCE

Performance of vector divide and square root operations on Broadwell and Skylake microarchitectures is shown below.

Table 17-13. 256-bit Intel AVX2 Divide and Square Root Instruction Performance

Broadwell Microarchitecture	**DIVPS**	**SQRTPS**	**DIVPD**	**SQRTPD**
Latency	17	21	23	35
Throughput	10	14	16	28
Skylake Microarchitecture	DIVPS	SQRTPS	DIVPD	SQRTPD
Latency	11	12	14	18
Throughput	5	6	8	12

Table 17-14. 512-bit Intel AVX-512 Divide and Square Root Instruction Performance

Skylake Microarchitecture	**DIVPS**	**SQRTPS**	**DIVPD**	**SQRTPD**
Latency	17	19	23	31
Throughput	10	12	16	24

17.25.3 APPROXIMATION LATENCIES

This section shows the latency and throughput for the approximation methods, and DIV and SQRT instructions. The tables below show that in most cases the throughput gain of the approximation methods is (at least) double that of their IEEE counterparts, in simple loops that compute division or square root.

The throughput benefits of approximation sequences are diminished when the loop iterations contain a lot of other computation (besides divide or square root).

As a rule of thumb, approximations of near-IEEE accuracy are recommended when the loop iteration contains no more than eight to ten additional single precision operations, or no more than twelve to fifteen additional double precision operations. The tables below show that these accurate approximations are beneficial for throughput optimizations only. The less accurate approximations can help with latency, as well as throughput.

It should also be mentioned that Newton-Raphson approximations do not handle the following special cases correctly: denormal inputs, zeros, or Infinities. Some sequences also lose accuracy for nearly denormal inputs, due to underflow in intermediate steps. While zero and Infinity inputs are relatively easy to fix with a few additional operations (as done in some of the sequences below), denormal divisors cannot be addressed without significant performance impact. The approximation sequences work best for “middle-of-the-range” inputs that are not close to overflow or underflow thresholds.

The table below shows the latency and throughput of single precision Intel AVX-512 divide and square root instructions, compared to the approximation methods on Skylake microarchitecture.

Table 17-15. Latency/Throughput of Different Methods of Computing Divide and Square Root on Skylake Microarchitecture for Different Vector Widths, on Single Precision

Operation	Method	Accuracy	256-bit Intel® AVX-512 Instructions		512-bit Intel® AVX-512 Instructions	
			Throughput	Latency	Throughput	Latency
Divide (a/b)	DIVPS	24 bits (IEEE)	5	11	10	17
	RCP14PS + MULPS + 1 Newton-Raphson Iteration	23 bits	2	16	3	20
	RCP14PS + MULPS	14 bits	1	8	2	10-12
Square root	SQRTPS	24 bits (IEEE)	6	12	12	19
	RSQRT14PS + MULPS + 1 Newton-Raphson Iteration	23 bits	3	16	5	20
	RSQRT14PS + MULPS	14 bits	2	9	3	12
Reciprocal square root	SQRTPS + DIVPS	22 bits	11	23	22	36
	RSQRT14PS + 1 Newton-Raphson Iteration	23 bits	3.67	20	4.89	25
	RSQRT14PS	14 bits	1	4	2	6

Table 17-16. Latency/Throughput of Different Methods of Computing Divide and Square Root on Skylake Microarchitecture for Different Vector Widths, on Double Precision

Operation	Method	Accuracy	256-bit Intel® AVX-512 Instructions		512-bit Intel® AVX-512 Instructions	
			Throughput	Latency	Throughput	Latency
Divide (a/b)	DIVPD	53 bits (IEEE)	8	14	16	23
	RCP14PD + MULPD + 2 Newton-Raphson Iteration	22 bits	3.2	27	4.7	28.4
	RCP14PD + MULPD + 1 Newton-Raphson Iteration	26 bits	2	16	3	20
	RCP14PD + MULPD	14 bits	1	8	2	10-12

Table 17-16. Latency/Throughput of Different Methods of Computing Divide and Square Root on Skylake Microarchitecture for Different Vector Widths, on Double Precision (Contd.)

Operation	Method	Accuracy	256-bit Intel® AVX-512 Instructions		512-bit Intel® AVX-512 Instructions	
			Throughput	Latency	Throughput	Latency
Square root	SQRTPD	53 bits (IEEE)	12	18	24	31
	RSQRT14PD + MULPD + Polynomial Approximation	22 bits	4.82	24.54[1]	6.4	28.48[1]
	RSQRT14PD + MULPD + 1 N-R	26 bits	3.76	17	5	20
	RSQRT14PD + MULPD	14 bits	2	9	3	12
Reciprocal square root	SQRTPD + DIVPD	51 bits	20	32	40	53
	RSQRT14PD + 2-NR + ErrorCcorrection	52 bits	5	29.38	6.53	34
	RSQRT14PD+2 N-R	50 bits	3.79	25.73	5.51	30
	RSQRT14PD+1 N-R	26 bits	2.7	18	4.5	21.67
	RSQRT14PD	14 bits	1	4	2	6

NOTES:

1. These numbers are not rounded because their code sequence contains several FMA (Fused-multiply-add) instructions, which have a varying latency of 4/6. Therefore the latency for these sequences is not necessarily fixed.

17.25.4 CODE SNIPPETS

Example 17-35. Vectorized 32-bit Float Division, 24 Bits

Single Precision, Divide, 24 Bits (IEEE)

```
float a = 10;
float b = 5;

__asm {
  vbroadcastss zmm0, a// fill zmm0 with 16 elements of a
  vbroadcastss zmm1, b// fill zmm1 with 16 elements of b
  vdivps zmm2, zmm0, zmm1// zmm2 = 16 elements of a/b
}
```

Example 17-36. Vectorized 32-bit Float Division, 23 and 14 Bits

Single Precision, Divide, 23 Bits

```
/* Input:
      zmm0 = vector of a’s
      zmm1 = vector of b’s
   Output:
      zmm3 = vector of a/b
*/

__asm {
  vrcp14ps zmm2, zmm1
  vmulps zmm3, zmm0, zmm2
  vmovaps zmm4, zmm0
  vfnmadd231ps zmm4, zmm3, zmm1
  vfmadd231ps zmm3, zmm4, zmm2
}
```

Single Precision, Divide, 14 Bits

```
/* Input:
      zmm0 = vector of a’s
      zmm1 = vector of b’s
   Output:
      zmm2 = vector of a/b
*/

__asm {
  vrcp14ps zmm2, zmm1
  vmulps zmm2, zmm0, zmm2
}
```

Example 17-37. Reciprocal Square Root, 22 Bits

Single Precision, Reciprocal Square Root, 22 Bits

```
/* Input:
      zmm0 = vector of a’s
      zmm1 = vector of 1’s
   Output:
      zmm2 = vector of 1/sqrt (a)
*/

float one = 1.0;

__asm {
  vbroadcastss zmm1, one// zmm1 = vector of 16 1’s
  vsqrtps zmm2, zmm0
  vdivps zmm2, zmm1, zmm2
}
```

Example 17-38. Reciprocal Square Root, 23 and 14 Bits

Single Precision, Reciprocal Square Root, 23 Bits	Single Precision, Reciprocal Square Root, 14 Bits

/* Input:
      zmm0 = vector of a’s
   Output:
      zmm2 = vector of 1/sqrt (a)
*/
 	
float half = 0.5;
 	
__asm {
   vbroadcastss zmm1, half// zmm1 = vector of 16 0.5’s
   vrsqrt14ps zmm2, zmm0
   vmulps zmm3, zmm0, zmm2
vmulps zmm4, zmm1, zmm2
   vfnmadd231ps zmm1, zmm3, zmm4
   vfmsub231ps zmm3, zmm0, zmm2
 	
   vfnmadd231ps zmm1, zmm4, zmm3
   vfmadd231ps zmm2, zmm2, zmm1
   }
	/* Input:
	      zmm0 = vector of a’s
	   Output:
	      zmm2 = vector of 1/sqrt (a)
	*/
	
	__asm {
	   vrsqrt14ps zmm2, zmm0
	}

Example 17-39. Square Root , 24 Bits

Single Precision, Square Root, 24 Bits (IEEE)

/* Input:
      zmm0 = vector of a’s
   Output:
      zmm2 = vector of sqrt (a)
*/
 	
__asm {
   vsqrtps zmm2, zmm0
}

Example 17-40. Square Root , 23 and 14 Bits

Single Precision, Square Root, 23 Bits	Single Precision, Square Root, 14 Bits

/* Input:
      zmm0 = vector of a’s
   Output:
      zmm0 = vector of sqrt (a)
*/
 	
float half = 0.5;
	/* Input:
	      zmm0 = vector of a’s
	   Output:
	      zmm0 = vector of sqrt (a)
	*/

Example 17-40. (Contd.)Square Root (Contd.), 23 and 14 Bits

Single Precision, Square Root, 23 Bits	Single Precision, Square Root, 14 Bits

__asm {
  vbroadcastss zmm3, half
  vrsqrt14ps zmm1, zmm0
  vfpclassps k2, zmm0, 0xe
  vmulps zmm2, zmm0, zmm1, {rn-sae}
  vmulps zmm1, zmm1, zmm3
  knotw k3, k2
  vfnmadd231ps zmm0{k3}, zmm2, zmm2
  vfmadd213ps zmm0{k3}, zmm1, zmm2
}
	__asm {
	  vrsqrt14ps zmm1, zmm0
	  vfpclassps k2, zmm0, 0xe
	  knotw k3, k2
	  vmulps zmm0{k3}, zmm0, zmm1
	}

Example 17-41. Dividing Packed Doubles, 53 and 52 Bits

Double Precision, Divide, 53 Bits (IEEE)	Double Precision, Divide, 52 Bits

/* Input:
    zmm0 = vector of a's
    zmm1 = vector of b's
  Output:
    zmm2 = vector of a/b
*/
 	
__asm {
  vdivpd zmm2, zmm0, zmm1
}
	/* Input:
	    zmm15 = vector of a's
	    zmm0 = vector of b's
	  Output:
	    zmm0 = vector of a/b
	*/
	
	double One = 1.0;
	
	__asm {
	  vrcp14pd zmm1, zmm0
	  vmovapd zmm4, zmm0
	  vbroadcastsd zmm2, one
	  vfnmadd213pd zmm0, zmm1, zmm2, {rn-sae}
	  vfpclasspd k2, zmm1, 0x1e
	  vfmadd213pd zmm0, zmm1, zmm1, {rn-sae}}
	  knotw k3, k2
	  vfnmadd213pd zmm4, zmm0, zmm2, {rn-sae}
	  vblendmpd zmm0 {k2}, zmm0, zmm1
	  vfmadd213pd zmm0 {k3}, zmm4, zmm0, {rn-sae}
	  vmulpd zmm0, zmm0, zmm15
	}

Example 17-42. Dividing Packed Doubles, 26 and 14 Bits

Double Precision, Divide, 26 Bits	Double Precision, Divide, 14 Bits

/* Input:
    zmm0 = vector of a's
    zmm1 = vector of b's
  Output:
    zmm3 = vector of a/b
*/
	/* Input:
	    zmm0 = vector of a's
	    zmm1 = vector of b's
	  Output:
	    zmm2 = vector of a/b
	*/

Example 17-42. Dividing Packed Doubles, 26 and 14 Bits (Contd.)

Double Precision, Divide, 26 Bits	Double Precision, Divide, 14 Bits

`__asm {`
`  vrcp14pd zmm2, zmm1`
`  vmulpd zmm3, zmm0, zmm2`
`  vmovapd zmm4, zmm0`
`  vfnmadd231pd zmm4, zmm3, zmm1`
`  vfmadd231pd zmm3, zmm4, zmm2`
`}`
	`__asm {`
	`  vrcp14pd zmm2, zmm1`
	`  vmulpd zmm2, zmm0, zmm2`
	`}`

Example 17-43. Reciprocal Square Root of Doubles, 51 Bits

Double Precision, Reciprocal Square Root, 51 Bits

```
/* Input:
      zmm0 = vector of a’s
      zmm1 = vector of 1’s
  Output:
      zmm0 = vector of 1/sqrt (a)
*/
__asm {
  vsqrtpd zmm0, zmm0
  vdivpd zmm0, zmm1, zmm0
}
```

Example 17-44. Reciprocal Square Root of Doubles, 52 and 50 Bits

Double Precision, Reciprocal Square Root, 52 Bits	Double Precision, Reciprocal Square Root, 50 Bits
/* Input:	/* Input:
zmm4 = vector of a's	zmm3 = vector of a's
Output:	Output:
zmm0 = vector of 1/sqrt (a)	zmm4 = vector of 1/sqrt (a)
*/	*/
// duplicates x eight times	// duplicates x eight times
#define DUP8_DECL(x) x, x, x, x, x, x, x, x	#define DUP8_DECL(x) x, x, x, x, x, x, x, x
// used for aligning data structures to n bytes	// used for aligning data structures to n bytes
#define ALIGNTO(n) __declspec(align(n))	#define ALIGNTO(n) __declspec(align(n))
ALIGNTO(64) __int64 one[ ] =	ALIGNTO(64) __int64 one[ ] =
{DUP8_DECL(0x3FF0000000000000)};	{DUP8_DECL(0x3FF0000000000000)};
ALIGNTO(64) __int64 dc1[ ] =	ALIGNTO(64) __int64 dc1[ ] =
{DUP8_DECL(0x3FE0000000000000)};	{DUP8_DECL(0x3FE0000000000000)};
ALIGNTO(64) __int64 dc2[ ] =	ALIGNTO(64) __int64 dc2[ ] =
{DUP8_DECL(0x3FD8000004600001)};	{DUP8_DECL(0x3FD8000004600001)};
ALIGNTO(64) __int64 dc3[ ] =	ALIGNTO(64) __int64 dc3[ ] =
{DUP8_DECL(0x3FD4000005E80001)};	{DUP8_DECL(0x3FD4000005E80001)};
__asm {	__asm {
vbroadcastsd zmm4, big_num	vmovapd zmm5, one
vmovapd zmm0, one	vmovapd zmm6, dc1
vmovapd zmm5, dc1	vmovapd zmm8, dc3
vmovapd zmm6, dc2	vmovapd zmm7, dc2
vmovapd zmm7, dc3	
vrsqrt14pd zmm3, zmm4	
vfpclasspd k1, zmm4, 0x5e	vrsqrt14pd zmm2, zmm3
vmulpd zmm1, zmm3, zmm4, {rn-sae}	vfpclasspd k1, zmm3, 0x5e
vfnmadd231pd zmm0, zmm3, zmm1	vmulpd zmm0, zmm2, zmm3, {rn-sae}
vfmsub231pd zmm1, zmm3, zmm4, {rn-sae}	vfnmadd231pd zmm0, zmm2, zmm5
vfnmadd213pd zmm1, zmm3, zmm0	vmulpd zmm1, zmm2, zmm0
vmovups zmm0, zmm7	vmovapd zmm4, zmm8
vmulpd zmm2, zmm3, zmm1	vfmadd213pd zmm4, zmm0, zmm7
vfmadd213pd zmm0, zmm1, zmm6	vfmadd213pd zmm4, zmm0, zmm6
vfmadd213pd zmm0, zmm1, zmm5	vfmadd213pd zmm4, zmm1, zmm2
vfmadd213pd zmm0, zmm2, zmm3	vorpd zmm4{k1}, zmm2, zmm2
vorpd zmm0{k1}, zmm3, zmm3	}
}	

Example 17-45. Reciprocal Square Root of Doubles, 26 and 14 Bits

Double Precision, Reciprocal Square Root, 26 Bits	Double Precision, Reciprocal Square Root, 14 Bits

/* Input:
      zmm0 = vector of a’s
   Output:
      zmm1 = vector of 1/sqrt (a)
*/
 	
double half = 0.5;
 	
__asm {
   vrsqrt14pd zmm1, zmm0
   vmulpd zmm0, zmm0, zmm1
   vbroadcastsd zmm3, half
   vmulpd zmm2, zmm1, zmm3
   vfnmadd213pd zmm2, zmm0, zmm3
   vfmadd213pd zmm1, zmm2, zmm1
}
	/* Input:
	      zmm0 = vector of a’s
	   Output:
	      zmm2 = vector of 1/sqrt (a)
	*/
	
	__asm {
	   vrsqrt14pd zmm2, zmm0
	}

Example 17-46. Square Root of Packed Doubles, 53 and 52 Bits

Double Precision, Square Root, 53 Bits (IEEE)	Double Precision, Square Root, 52 Bits

/* Input:
      zmm0 = vector of a’s
   Output:
      zmm2 = vector of sqrt (a)
*/
 	
__asm {
   vsqrtpd zmm2, zmm0
}
	/* Input:
	      zmm0 = vector of a’s
	   Output:
	      zmm0 = vector of sqrt (a)
	*/
	
	double half = 0.5;
	
	__asm {
	   vbroadcastsd zmm4, half
	   vrsqrt14pd zmm1, zmm0
	   vfpclasspd k2, zmm0, 0xe
	   vmulpd zmm2, zmm0, zmm1, {rn-sae}
	   vmulpd zmm1, zmm1, zmm4
	   knotw k3, k2
	   vmovapd zmm3, zmm4
	   vfnmadd231pd zmm3, zmm1, zmm2, {rn-sae}
	   vfmadd213pd zmm2, zmm3, zmm2, {rn-sae}
	   vfmadd213pd zmm1, zmm3, zmm1, {rn-sae}
	   vfnmadd231pd zmm0 {k3}, zmm2, zmm2, {rn-sae}
	   vfmadd213pd zmm0 {k3}, zmm1, zmm2
	}

Example 17-47. Square Root of Packed Doubles, 26 and 14 Bits

Double Precision, Square Root, 26 Bits	Double Precision, Square Root, 14 Bits

/* Input:
        zmm0 = vector of a's
    Output:
        zmm0 = vector of sqrt (a)
*/
 	
// duplicates x eight times
#define DUP8_DECL(x) x, x, x, x, x, x, x, x
 	
// used for aligning data structures to n bytes
#define ALIGNTO(n) __declspec(align(n))
 	
ALIGNTO(64) __int64 OneHalf[ ] =
{DUP8_DECL(0X3FE0000000000000)};
__asm {
    vrsqrt14pd zmm1, zmm0
    vfpclasspd k2, zmm0, 0xe
    knotw k3, k2
    vmulpd zmm0 {k3}, zmm0, zmm1
    vmulpd zmm1, zmm1, ZMMWORD PTR [OneHalf]
    vfnmadd213pd zmm1, zmm0, ZMMWORD PTR
	/* Input:
	        zmm0 = vector of a's
	    Output:
	        zmm0 = vector of sqrt (a)
	*/
	
	__asm {
	    vrsqrt14pd zmm1, zmm0
	    vfpclasspd k2, zmm0, 0xe
	    knotw k3, k2
	    vmulpd zmm0 {k3}, zmm0, zmm1
	}

[OneHalf]
    vfmadd213pd zmm0 {k3}, zmm1, zmm0
}	

17.26 CLDEMOTE

Using the CLDEMOTE instruction, a processor puts a cache line into the last shared level of the cache hierarchy so that other CPU cores 'find' the same cache line in the last shared level and expensive cross-core snoop is avoided. The most significant advantage of CLDEMOTE is that multiple consumers can access the shared cache line amortizing each snoop request portion.

17.26.1 PRODUCER-CONSUMER COMMUNICATION IN SOFTWARE

In a multiprocessor environment, data sharing between the producers and consumers is an undisputed event. A cache hierarchy solves the major problem of accessing the line from the main memory resulting in faster data transfers. Typical cache hierarchy contains:

- Private L1 data and L1 instruction cache.
- A shared L2 cache for sibling hardware thread.
- A common L3 cache for all the CPU cores.

When a producer consumes data from the I/O or produces it, it is brought into the producer's L1 cache. Consumers read the data by initiating read requests, translating it into cross-core snoops, request, and response events. Consumers report L3 cache miss events and producer cores responding to the consumer core's snoop request. Multiplexing these cross-cores requests and responses when dealing with multiple consumers is detrimental.

17.27 TIPS ON COMPILER USAGE

This section explains some of the important compiler options that can be used with the Intel compiler to derive the best performance on a Skylake server.

For complete information on the compiler options and tuning tips, see the Intel Resource & Documentation Center.

Many options have names that are the same on Linux* and Windows*, except that the Windows* form starts with an initial Q. Within text, such option names are shown as [Q]option-name.

The default optimization level is O2 (unless -g is specified, in which case the default is O0). Level O2 enables many compiler optimizations including vectorization. Optimization level O3 is recommended for loop-intensive and HPC applications, as it enables more aggressive loop and memory-access optimizations, such as loop fusion and loop blocking to allow more efficient use of the caches.

- For the best performance of Skylake server microarchitecture, applications should be compiled with the processor-specific option [Q]xCORE-AVX512.
  - Notably, an executable compiled with these options will not run on non-Intel processors or on Intel processors that support only lower instruction sets.
- For users who want to generate a common binary that can be executed on Skylake server microarchitecture and the Intel® Xeon Phi™ processors based on Knights Landing microarchitecture, use the option [Q]xCOMMON-AVX512.
  - This option has a performance cost on both Skylake server microarchitecture and Intel® Xeon Phi™ processors compared with executables generated with the target-specific options:
    - **[Q]xCORE-AVX512 on Skylake server**
    - **[Q]xMIC-AVX512 on Intel® Xeon Phi™ processors.**

Additionally, users can tune the zmm code generation done by the compiler for Skylake server microarchitecture using the additional option **-qopt-zmm-usage=low|high** (/Qopt-zmm-usage:low|high on Windows).

- The argument value of low provides a smooth transition experience from Intel AVX2 ISA to Intel AVX-512 ISA on a Skylake server microarchitecture target, such as for enterprise applications.
  - Tuning for ZMM instruction use via explicit vector syntax such as #pragma omp simd simdlen() is recommended.
- The argument value of high is recommended for applications, such as HPC codes, that are bounded by vector computation to achieve more compute per instruction through use of the wider vector operations:
  - The default value is low for Skylake server microarchitecture-family compilation targets, such as [Q]xCORE-AVX512.
  - High for CORE/MIC AVX512 combined compilation targets such as [Q]xCOMMON-AVX512.

It is also possible to generate a fat binary that supports multiple instruction sets by using the [Q]axtarget option.

For example, if the application is compiled with [Q]axCORE-AVX512,CORE-AVX2:

- The compiler might generate specialized code for the Skylake server microarchitecture and AVX2 targets, while also generating a default code path that will run on any Intel or compatible, non-Intel processor that supports at least Intel® Streaming SIMD Extensions 2 (Intel® SSE2).
  - At runtime, the application automatically detects whether it is running on an Intel processor.
    - **If so, it selects the most appropriate code path for Intel processors;**
    - **if not, the default code path is selected.**
  - Irrespective of the options used, the compiler might insert calls into specialized library routines, such as optimized versions of memset/memcpy, that will dispatch to the appropriate codepath at runtime based on processor detection.

The option **-qopt-report[n]** (/Qopt-report[:n] on Windows) generates a report on the optimizations performed by the compiler, by default it is written to a file with a .optrpt file extension. n specifies the level of detail, from 0 (no report) to 5 (maximum detail).

The option **-qopt-report-phase** (/Qopt-report-phase on Windows) controls report generation from various compiler phases, but it is recommended to use the default setting where the report is generated for all compiler phases.

- The report is a useful tool to gain insight into the performance optimizations performed, or not performed, by the compiler, and also to understand the interactions between multiple optimizations such as inlining, OpenMP* parallelization, loop optimizations (such as loop distribution or loop unrolling) and vectorization.
- The report is based on static compiler analysis. Hence the reports are most useful when correlated with dynamic performance analysis tools, such as Intel® VTune™ Amplifier or Vectorization Advisor (part of Intel® Advisor XE), that do hotspot analysis and provide other dynamic information.
- Once this information is available, the optimization information can be studied for hotspots (functions/loopnests) in compiler reports.
  - — The compiler can generate multiple versions of loop-nests, so it is useful to correlate the analysis with the version actually executed at runtime.
    - **The phase ordering of the compiler loop optimizations is intended to enable optimal vectorization. Often, understanding the loop optimization parameters helps to further tune performance.**
    - **Finer control of these loop optimizations is often available via pragmas, directives, and options.**

If the application contains OpenMP pragmas or directives, it can be compiled with -qopenmp (/Qopenmp on Windows) to enable full OpenMP based multi-threading and vectorization. Alternatively, the SIMD vectorization features of OpenMP alone can be enabled by using the option -qopenmp-simd (/Qopenmp-simd on Windows).

For doing studies where compiler-based vectorization has to be turned off completely, use the options

-no-vec -no-simd -qno-openmp-simd (/Qvec- /Qsimd- /Qopenmp-simd- on Windows).

Data alignment plays an important role in improving the efficiency of vectorization. This usually involves two distinct steps from the user or application:

- Align the data.
  - — When compiling a Fortran program, it is possible to use the option -align array64byte (/align:array64byte on Windows) to align the start of most arrays at a memory address that is divisible by 64.
  - — For C/C++ programs, data allocation can be done using routines such as _mm_malloc(…, 64) to align the return-value pointer at 64 bytes. For more information on data alignment, see https://software.intel.com/en-us/articles/data-alignment-to-assist-vectorization.
- Convey the alignment information to the compiler using appropriate clauses, pragmas, and directives.

Compiler-based software data prefetching can be enabled with the options -O3 -xcore-avx512 -qopt-prefetch[=n] (-O3 /QxCORE-AVX512 /Qopt-prefetch[=n] on Windows), for n=0 (no prefetching) to 5 (maximal prefetching). Using a value of n=5 enables aggressive compiler prefetching, disregarding any hardware prefetching, for strided loads/stores and indexed loads/stores which appear inside loops. Using a value of n=2 reduces the amount of compiler prefetching and restricts it only to direct memory accesses where the compiler heuristics determine that the hardware prefetcher may not be able to handle well. It is recommended to try values of n=2 to 5 to determine the best prefetching strategy for a particular application. It is also possible to use the -qopt-prefetch-distance=n1[,n2] (/Qopt-prefetch-distance=n1[,n2] on Windows) option to fine-tune application performance.

- Useful values to try for n1: 0,4,8,16,32,64.
- Useful values to try for n2: 0,1,2,4,8.

Loop-nests that have a relatively low trip-count value at runtime in hotspots can sometimes lead to sub-optimal AVX-512 performance unless the trip-count is conveyed to the compiler. In many such cases, the compiler will be able to generate better code and deliver better performance if values of loop trip-counts, loop-strides, and array extents (such as for Fortran multi-dimensional arrays) are all known to the compiler. If that is not possible, it may be useful to add appropriate loop_count pragmas to such loops.

Interprocedural optimization (IPO) is enabled using the option -ipo (/Qipo on Windows). This option can be enabled on all the source-files of the application or it can be applied selectively to the source files containing the application hot-spots. IPO permits inlining and other inter-procedural optimizations to happen across these multiple source files. In some cases, this option can significantly increase compile time and code size. Using the option -inline-factor=n (/Qinline-factor:n on Windows) controls the amount of inlining done by the compiler. The default value of n is 100, indicating 100%, or a scale factor of 1. For example, if a value of 200 is specified, all inlining options that define upper limits are multiplied by a factor of 2, thus enabling more inlining than the default.

Profile-guided optimizations (PGO) are enabled using the options -prof-gen and -prof-use (/Qprof-gen and /Qprof-use on Windows). Typically, using PGO increases the effectiveness of using IPO.

The option -fp-model name (/fp:name on Windows) controls tradeoffs between performance, accuracy and reproducibility of floating-point results at a high level. The default value for name is fast=1. Changing it to fast=2 enables more aggressive optimizations at a slight cost in accuracy or reproducibility. Using the value precise for name disallows optimizations that might produce slight variations in floating-point results. When name is double, extended or source, intermediate results are computed in the corresponding precision. In most situations where enhanced floating-point consistency and reproducibility are needed -fp-model precise -fp-model source (/fp:precise /fp:source on Windows) are recommended.

The option -fimf-precision=name (/Qimf-precision=name on Windows) is used to set the accuracy for math library functions. The default is OFF, which means that the compiler uses its own default heuristics. Possible values of name are high, medium, and low. Reduced precision might lead to increased performance and vice versa, particularly for vectorized code. The options -[no-]prec-div and -[no-]prec-sqrt improve[reduce] precision of floating-point divides and square root computations. This may slightly degrade [improve] performance. For more details on floating-point options, see Consistency of Floating-Point Results using the Intel® Compiler (2018) .

The option -[no-]ansi-alias (/Qansi-alias[-] on Windows) enables [disables] ANSI and ISO C Standard aliasing rules. By default, this option is enabled on Linux, but disabled on Windows. On Windows, especially for C++ programs, adding /Qansi-alias to the compilation options enable the compiler to perform additional optimizations, particularly taking advantage of the type-based disambiguation rules of the ANSI Standard, which says for example, that pointer and float variables do not overlap.

If the optimization report specifies that compiler optimizations may have been disabled to reduce compile-time, use the option -qoverride-limits to override such disabling in the compiler and ensure optimization is applied. This can sometimes be important for applications, especially ones with functions that have big bodies. Note that using this additional option may increase compile time and compiler memory usage significantly in some cases.

The list below shows a sampling of loop-level controls available for fine-tuning optimizations - including a way to turn off a particular transformation reported by the compiler.

- #pragma simd reduction(+:sum)
  - The loop is transformed as is, no other loop-optimizations will change the simd-loop.
- #pragma loop_count min(220) avg (300) max (380)
  - Fortran syntax: !dir$ loop count(16)
- #pragma vector aligned nontemporal
- #pragma novector // to suppress vectorization
- #pragma unroll(4)
- #pragma unroll(0) // to suppress loop unrolling

- #pragma unroll_and_jam(2) // before an outer loop
- #pragma nofusion
- #pragma distribute_point
  - If placed as the first statement right after the for-loop, distribution will be suppressed for that loop.
  - Fortran syntax: !dir$ distribute point
- #pragma prefetch *:<hint>:<distance>
  - Apply uniform prefetch distance for all arrays in a loop.
- #pragma prefetch <var>:<hint>:<distance>
  - Fine-grained control for each array
- #pragma noprefetch [<var>]
  - Turns off prefetching [for a particular array]
- #pragma forceinline (recursive)

  If placed before a call, this is a hint to the compiler to recursively inline the entire call-chain.

Optimization Notice