Software Optimization Guide for the AMD Zen5 Microarchitecture

Publication # **58455** Revision: **1.00**
Issue Date: **August 2024**

Advanced Micro Devices

The information contained herein is for informational purposes only, and is subject to change without notice. While every precaution has been taken in the preparation of this document, it may contain technical inaccuracies, omissions and typographical errors, and AMD is under no obligation to update or otherwise correct this information. Advanced Micro Devices, Inc. makes no representations or warranties with respect to the accuracy or completeness of the contents of this document, and assumes no liability of any kind, including the implied warranties of noninfringement, merchantability or fitness for particular purposes, with respect to the operation or use of AMD hardware, software or other products described herein. No license, including implied or arising by estoppel, to any intellectual property rights is granted by this document. Terms and limitations applicable to the purchase or use of AMD's products are as set forth in a signed agreement between the parties or in AMD's Standard Terms and Conditions of Sale. Any unauthorized copying, alteration, distribution, transmission, performance, display or other use of this material is prohibited.

Trademarks

AMD, the AMD Arrow logo, AMD AllDay, AMD Virtualization, AMD-V, PowerPlay, Vari-Bright, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

Dolby is a trademark of Dolby Laboratories.

HDMI is a trademark of HDMI Licensing, LLC.

HyperTransport is a licensed trademark of the HyperTransport Technology Consortium.

Microsoft, Windows, Windows Vista, and DirectX are registered trademarks of Microsoft Corporation in the US and/or other countries.

MMX is a trademark of Intel Corporation.

OpenCL is a trademark of Apple Inc. used by permission by Khronos.

PCIe is a registered trademark of PCI-Special Interest Group (PCI-SIG).

USB Type-C® and USB-C® are registered trademarks of USB Implementers Forum.

Reverse engineering or disassembly is prohibited.

USE OF THIS PRODUCT IN ANY MANNER THAT COMPLIES WITH THE MPEG ACTUAL OR DE FACTO VIDEO AND/OR AUDIO STANDARDS IS EXPRESSLY PROHIBITED WITHOUT ALL NECESSARY LICENSES UNDER APPLICABLE PATENTS. SUCH LICENSES MAY BE ACQUIRED FROM VARIOUS THIRD PARTIES INCLUDING, BUT NOT LIMITED TO, IN THE MPEG PATENT PORTFOLIO, WHICH LICENSE IS AVAILABLE FROM MPEG LA, L.L.C., 6312 S. FIDDLERS GREEN CIRCLE, SUITE 400E, GREENWOOD VILLAGE, COLORADO 80111.

Contents

List of Figures

List of Tables

Revision History

Date	Revision	Description
August 2024	1.00	Initial Public release: • In the latency workbook, added Row 13 to Notes sheet. • Added Section 2.9.9, “Other Decode Considerations.” • Added Section 2.13.6, “String Store Optimizations.”
May 2024	0.50	Initial NDA release.

Chapter 1 Introduction

This guide provides optimization information and recommendations for AMD Zen5 microarchitecture. In this guide, processor refers to AMD Zen5 processors.

AMD Zen5 microarchitecture is present in Family 1Ah processors with Models 00h-4Fh and Models 60h-6Fh.

1.1 Intended Audience

This book is intended for compiler and assembler designers, as well as C, C++, and assembly language programmers writing performance-sensitive code sequences. This guide assumes that you are familiar with the AMD64 instruction set and the AMD64 architecture (registers and programming modes).

1.2 Related Documents

For complete information on the AMD64 architecture and instruction set, see the multi-volume AMD64 Architecture Programmer’s Manual available from AMD.com. Individual volumes and their order numbers are provided below:

Table 1. Related Documents

Title	Order Number
Volume 1: Application Programming	24592
Volume 2: System Programming	24593
Volume 3: General-Purpose and System Instructions	24594
Volume 4: 128-Bit and 256-Bit Media Instructions	26568
Volume 5: 64-Bit Media and x87 Floating-Point Instructions	26569

The following documents provide a set of guidelines for writing efficient code that have general applicability to the AMD Zen5 microarchitecture:

- Software Optimization Guide for AMD Family 10h and 12h Processors, order# 40546
- Software Optimization Guide for AMD Family 15h Processors, order# 47414
- Software Optimization Guide for AMD Family 17h Processors, order# 55723
- Software Optimization Guide for AMD Family 19h Processors, order# 56665

Refer to the following document for more information about machine-specific registers, debug, and performance profiling tools:

- Processor Programming Reference (PPR) for AMD Family 19h Models 10h-1Fh, order# 55901

1.3 Specialized Terminology

The following specialized terminology is used in this document:

Term	Definition
CPU Complex	A CPU complex is a group of cores that share one L3 cache.
Dispatching	Dispatching refers to the act of transferring macro-ops from the front end of the processor to the out-of-order backend.
Issuing	Issuing refers to the act of picking from the scheduler to send a micro-op into an execution pipeline. Some macro-ops may be issued as multiple micro-ops into different execution pipelines.

Chapter 2 Zen5 Processor Microarchitecture

An understanding of the term's *architecture*, *microarchitecture*, and *design implementation* is important when discussing processor design.

The *architecture* consists of the instruction set and processor features visible to software programs running on the processor. The architecture determines what software the processor can run. The AMD64 architecture of the AMD Zen5 processor is compatible with the industry-standard x86 instruction set.

Microarchitecture refers to the design features used to reach the target cost, performance, and functionality goals of the processor.

Design implementation refers to a particular combination of physical logic and circuit elements that comprise a processor that meets the microarchitecture specifications.

The processor employs a reduced instruction set execution core with a preprocessor that decodes and decomposes most of the simpler AMD64 instructions into a sequence of one or two macro-ops. More complex instructions are implemented using microcode routines.

Decode is decoupled from execution, and the execution core employs a super-scalar organization in which multiple execution units operate essentially independently. The execution core design allows it to implement a small number of simple instructions that can be executed in a single processor cycle. This design simplifies circuit design, achieving lower power consumption and fast execution at optimized processor clock frequencies.

2.1 Key Microarchitecture Features

The processor implements a specific subset of AMD64 instruction set architecture defined by the APM.

The following major classes of instructions are supported:

- General-purpose instructions, including support for 64-bit operands
- x87 Floating-point instructions
- 64-bit Multi-media (MMX™) instructions
- 128-bit and 256-bit single-instruction / multiple-data (SIMD) instructions.
- AMD Virtualization™ technology (AMD-V™)

The following Streaming SIMD Extensions subsets are supported:

- Streaming SIMD Extensions 1 (SSE1)
- Streaming SIMD Extensions 2 (SSE2)
- Streaming SIMD Extensions 3 (SSE3)
- Supplemental Streaming SIMD Extensions 3 (SSSE3)
- Streaming SIMD Extensions 4a (SSE4a)
- Streaming SIMD Extensions 4.1 (SSE4.1)
- Streaming SIMD Extensions 4.2 (SSE4.2)
- Advanced Vector Extensions (AVX)
  - AVX_VNNI – AVX/VEX variants of Vector Neural Network Instructions
- Advanced Vector Extensions 2 (AVX2)
  - AVX2 variants of VAES / VPCLMULQDQ
- Advanced Encryption Standard (AES) acceleration instructions
- AVX-512 Advanced Vector Extensions, including the following extensions
  - AVX-512F - Foundation
  - AVX-512DQ - Packed Integer Instructions
  - AVX-512_IFMA - Integer fused multiply-add instructions
  - AVX-512CD - Conflict Detection for vectorizing loops
  - AVX-512BW - Byte and word instructions (packed integer instructions)
  - AVX-512VL - Allows most AVX-512 instructions to also operate on XMM and YMM registers
  - AVX-512_VBMI - Vector Byte Manipulation Instructions

  - AVX-512_VNNI - Vector Neural Network Instructions
  - AVX-512_BITALG - Bit Algorithm Instructions
  - AVX-512_VPOPCNTDQ - Vector POPCNT instructions
  - AVX-512_4FMAPS - Fused Multiply Accumulation Packed Single precision
  - AVX-512_BF16 - bfloat16 support (dot-product and convert instructions)
- GFNI - Galois Field New Instructions

The following miscellaneous instruction subsets are supported:

- SHA, RDRAND
- Read and write FS.base and GS.base instructions
- Half-precision floating-point conversion (F16C)
- Carry-less Multiply (CLMUL) instructions
- Move Big-Endian instruction (MOVBE)
- XSAVE / XSAVEOPT
- LZCNT / POPCNT
- WBNOINVD
- RDPRU
- UMIP
- CLWB
- INVLPGB (support varies by model number, refer to the PPR for individual model support)
- TLBSYNC (support varies by model number, refer to the PPR for individual model support)
- INVPCID (support varies by model number, refer to the PPR for individual model support)
- RDPKRU
- WRPKRU

The following Bit Manipulation Instruction subsets are supported:

- BMI1
- BMI2

The processor does not support the following instructions/instruction subsets:

- Four operand Fused Multiply/Add instructions (FMA4)

- XOP instructions
- Trailing bit manipulation (TBM) instructions
- Light-weight profiling (LWP) instructions

The processor also has support for shadow stack protection:

- INCSSP
- RDSSP
- SAVEPREVSSP
- RSTORSSP
- WRSS
- WRUSS
- SETSSBSY
- CLRSSBSY

The processor also has support for Secure Nested Paging (support varies by model number, refer to the PPR for individual model support):

- RMPUPDATE
- PVALIDATE
- PSMASH
- RMPADJUST
- RMPQUERY
- RMPREAD

The processor includes many features designed to improve software performance. These include the following key features:

- Simultaneous multi-threading
- Unified 1 Mbyte L2 cache per core.
- Up to 96-Mbyte shared, victim L3, depending on configuration.
- Integrated memory controller
- 32-Kbyte L1 instruction cache (IC) per core
- 48-Kbyte L1 data cache (DC) per core
- 6.75 K Op Cache (OC)

- Prefetchers for L2 cache, L1 data cache, and L1 instruction cache
- Advanced dynamic branch prediction
- 32-byte instruction cache fetch
- 4-way x86 instruction decoding with sideband stack optimizer
- Dynamic out-of-order scheduling and speculative execution
- Six-way integer execution; Four-way address generation (loads or stores)
- Four-way 512-bit wide floating-point and packed integer execution plus dedicated floating-point store data movement unit and floating-point to integer data movement unit
- Integer hardware divider
- L1 and L2 Instruction TLB and L1 and L2 Data TLB
- Six fully symmetric core performance counters per thread

2.2 Cache Line, Fetch, and Data Type Widths

The following tables diagram the cache line size, the widths of various data pipes, and registers:

Table 2. Cache Line Size, Fetch, and Decode Widths in Bytes

<table>
<tr><th colspan="13">Fetch Width</th></tr>
<tr><td rowspan="2">OC Fetch</td><td colspan="8">128</td><td colspan="4"></td></tr>
<tr><td colspan="4"></td><td colspan="8">128</td></tr>
<tr><td>Cacheline</td><td colspan="4">64</td><td colspan="4">64</td><td colspan="4">64</td></tr>
<tr><td>IC Fetch</td><td colspan="2">32</td><td colspan="2">32</td><td colspan="2">32</td><td colspan="2">32</td><td colspan="2">32</td><td colspan="2">32</td></tr>
<tr><td>Decode</td><td>16</td><td>16</td><td>16</td><td>16</td><td>16</td><td>16</td><td>16</td><td>16</td><td>16</td><td>16</td><td>16</td><td>16</td></tr>
</table>

Table 3. Data Pipe Widths in Bytes

Data	
Cacheline	64
Data Pipe	64
Store Commit	64

Table 4. Data Type Widths in Bytes

<table>
<tr><th colspan="17">Data Types</th></tr>
<tr><td>ZMMWORD</td><td colspan="16">64</td></tr>
<tr><td>YMMWORD</td><td colspan="8">32</td><td colspan="8">32</td></tr>
<tr><td>XMMWORD</td><td colspan="4">16</td><td colspan="4">16</td><td colspan="4">16</td><td colspan="4">16</td></tr>
<tr><td>QWORD</td><td colspan="2">8</td><td colspan="2">8</td><td colspan="2">8</td><td colspan="2">8</td><td colspan="2">8</td><td colspan="2">8</td><td colspan="2">8</td><td colspan="2">8</td></tr>
<tr><td>DWORD</td><td>4</td><td>4</td><td>4</td><td>4</td><td>4</td><td>4</td><td>4</td><td>4</td><td>4</td><td>4</td><td>4</td><td>4</td><td>4</td><td>4</td><td>4</td><td>4</td></tr>
</table>

2.3 Instruction Decomposition

The processor implements AMD64 instruction set by means of *macro-ops* (the primary units of work managed by the processor) and *micro-ops* (the primitive operations executed in the processor's execution units). These operations are designed to include direct support for AMD64 instructions and adhere to the high-performance principles of fixed-length encoding, regularized instruction fields, and a large register set. This enhanced microarchitecture enables higher processor core performance and promotes straightforward extensibility for future designs.

Instructions are marked as fast path single (one macro-op), fast path double (two macro-ops), or microcode (greater than two macro-ops). Macro ops can normally contain up to two micro-ops. The table below lists some examples showing how instructions are mapped to macro-ops and how these macro-ops are mapped into one or more micro-ops.

Table 5. Typical Instruction Mappings

Instruction	Macro-ops	Micro-ops	Comments
MOV reg,[mem]	1	1: load	Fastpath single
MOV [mem],reg	1	2: store data, store	Fastpath single
MOV [mem],imm	1	2: store data, store	Fastpath single
REP MOVS [mem],[mem]	Many	Many	Microcode
ADD reg,reg	1	1: add	Fastpath single
ADD reg,[mem]	1	2: load, add	Fastpath single
ADD [mem],reg	1	2: load/store, add	Fastpath single
MOVAPD [mem],xmm	1	2: store, FP-store-data 128b	Fastpath single
VMOVAPD [mem],ymm	1	2: store, FP-store-data 256b	Fastpath single
VMOVAPD [mem],zmm	2	2: store 512b, FP-store- data 512b	Fastpath double
ADDPD xmm,xmm	1	1: addpd 128b	Fastpath single
ADDPD xmm,[mem]	1	2: load, addpd 128b	Fastpath single
VADDPD ymm,ymm	1	1: addpd 256b	Fastpath single
VADDPD ymm,[mem]	1	2: load, addpd256b	Fastpath single
VADDPD zmm, zmm	1	1: addpd 512b	Fastpath single
VADDPC zmm, [mem]	1	2: load 512b, addpd 512b	Fastpath single

2.4 Superscalar Organization

The processor is an out-of-order, two thread superscalar AMD64 processor. The processor uses decoupled execution units to process instructions through fetch/branch-predict, decode, schedule/execute, and retirement pipelines.

The processor uses an independent, banked integer scheduler queue, servicing six ALU pipelines; an independent, banked address generation queue, servicing four AGU pipelines; two decoupled independent floating point schedulers each servicing two FP pipelines; and one independent floating point scheduler servicing store data or FP-to-integer pipelines. These schedulers can simultaneously issue up to sixteen micro-ops to the six ALU pipes, four Address Generation Unit (AGU) pipes, and six FPU pipes.

2.5 Processor Block Diagram

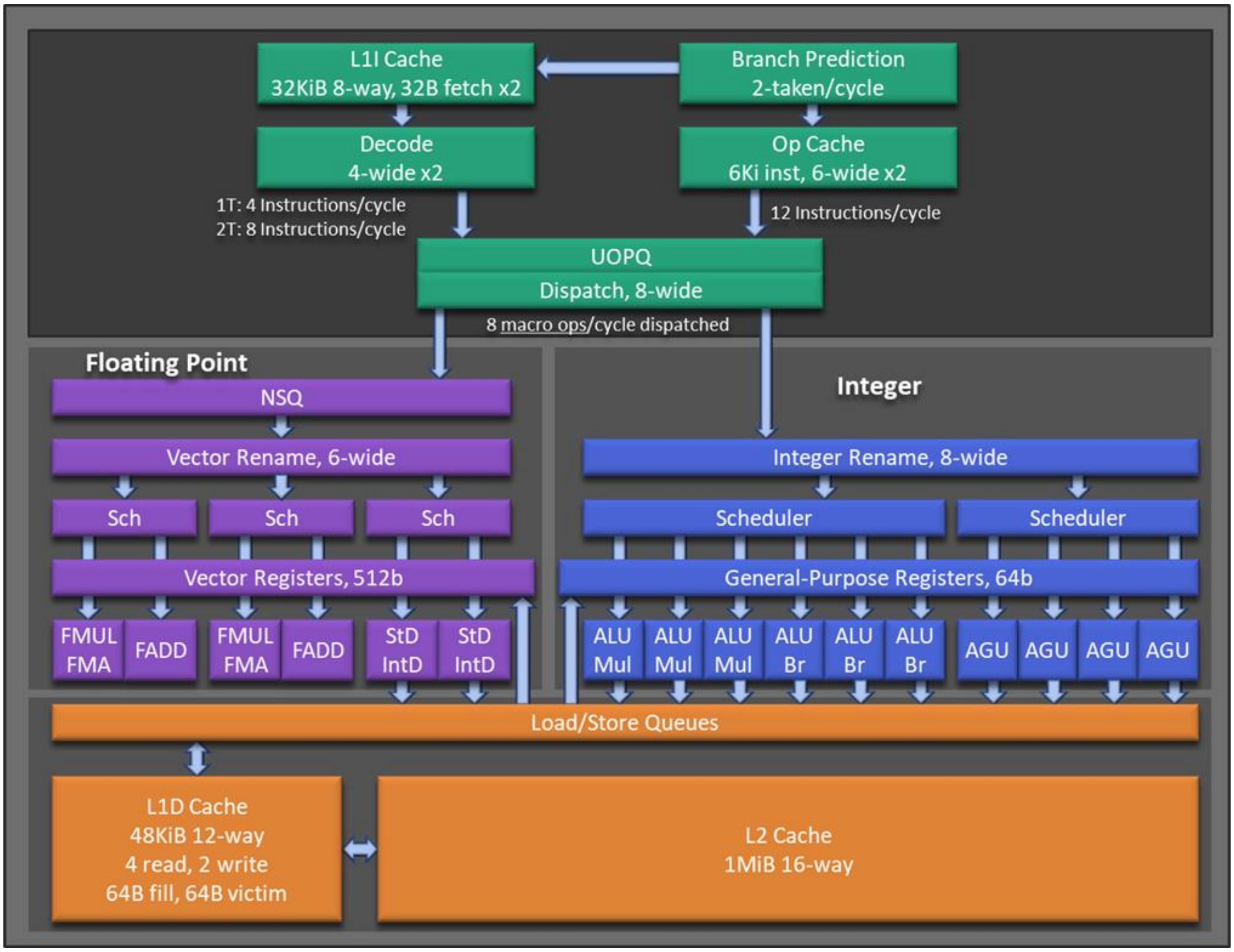

Figure 1. AMD Zen5 CPU Core Block Diagram

2.6 Processor Cache Operation

The AMD Zen5 microarchitecture uses five caches at three hierarchy levels to accelerate instruction execution and data processing:

- Dedicated L1 instruction cache
- Dedicated L1 data cache
- Dedicated L1 Op Cache
- Unified (instruction and data) L2 cache per core
- Up to 96-Mbyte L3 cache (depending on configuration)

2.6.1 L1 Instruction Cache

The AMD Zen5 microarchitecture contains a 32-Kbyte, 8-way set associative L1 instruction cache. Cache line size is 64 bytes; 64 bytes (up to two 32-byte aligned blocks from within a cache line) can be fetched in a cycle to feed two decode pipelines. Functions associated with the L1 instruction cache are fetching cache lines from the L2 cache, providing instruction bytes to the decoder, and prefetching instructions. Requests that miss in the L1 instruction cache are fetched from the L2 cache or, if not resident in the L2 cache, from the L3 cache, if present. Requests that miss in all levels of cache are fetched from system memory.

On misses, the L1 instruction cache generates fill requests for the naturally aligned 64-byte cache line that includes the miss address. These cache lines are prefetched from addresses generated by the Branch Predict unit. For never-before-seen cache lines, the branch predictor typically predicts sequential accesses, thus acting like a line prefetcher for avoiding downstream cache miss stalls.

Because code typically exhibits spatial locality, prefetching is an effective technique for avoiding cache miss stalls. Cache line replacement is based on a least recently used replacement algorithm. The L1 instruction cache is protected from error using parity.

2.6.2 L1 Data Cache

The processor contains a 48-Kbyte, 12-way set associative L1 data cache. This is a write-back cache that supports up to four memory operations per cycle. All four may be loads, but at most two may be 512-bit, 256-bit, or 128-bit loads. Up to two may be stores of sizes 8b to 256b, but at most one may be a 512-bit store. In addition, the L1 cache is protected from bit errors using ECC. There is a hardware prefetcher that brings data into the L1 data cache to avoid misses. The L1 data cache has a 4- or 5- cycle integer load-to-use latency, and a 7- or 8-cycle FPU load-to-use latency. See Section 2.12, "Load-Store Unit" on page 39, for more information on load-to-use latency. See Section 2.6.2.2 on page 18 for more information on the handling of 512-bit load and store operations.

The data cache natural alignment boundary is 64 bytes for loads. A misaligned load operation suffers, at minimum, a one cycle penalty in the load-store pipeline if it spans a 64-byte boundary. The store alignment boundary is also 64 bytes for accessing the TLB and tag as well as for writing data to the cache. Throughput for misaligned loads and stores is half that of aligned loads and stores since a misaligned load or store requires two cycles to access the data cache (versus a single cycle for aligned loads and stores).

Notes:

Operations that are misaligned across a 4Kbyte boundary will in some cases incur additional significant penalties.

For aligned memory accesses, the aligned and unaligned load and store instructions (for example, MOVUPS/MOVAPS) provide identical performance.

Natural alignment for a 256-bit vector is 32 bytes and aligning them to a 32-byte boundary provides a performance advantage.

2.6.2.1 Bank Conflicts

The L1 DC provides multiple access ports using a banked structure. The read ports are shared by four load pipes and victim reads. Store commits utilize separate write ports. Address bits 5:3 and the size of an access along with its DC way determine which DC banks are needed for that access. DC way is determined using the linear-address-based utag/way-predictor (see section below). Port or bank conflicts can result in performance degradation due to the need to re-flow one of the conflicting loads, which will appear as a longer-latency load.

When port or bank conflicts are suspected, they can be reduced by the following methods:

- Use aligned memory accesses. Misaligned memory accesses may use more banks or ports.
- Consolidate smaller (i.e. byte) consecutive loads or stores into larger (i.e. doubleword or quadword) loads or stores when possible.
- Stores that are misaligned or less than doubleword size use banks and ports least efficiently.
- Aligned loads to consecutive addresses will never have bank conflicts with each other. In a loop that streams through multiple regions, consider unrolling the loop and placing up to four loads (two if vector/floating point) from each stream together to reduce bank conflicts. Due to out-of-order execution, placing loads together does not guarantee that they will be executed together.

2.6.2.2 512-Bit Load and Store Operations

The processor supports 512-bit load and store operations that behave as follows:

- 512-bit load operations utilize two 256-bit load pipelines as a pair, for a single cycle. When a 512-bit load executes, an integer load cannot use the paired pipe.
- 512-bit store operations execute as a single store address micro-op and single FP store data micro-op, but require use of two dispatch slots and two store queue entries.

2.6.2.3 Linear Address UTAG/Way-Predictor

The L1 data cache tags contain a linear-address-based microtag (utag) that tags each cache line with the linear address that was used to access the cache line initially. Loads use this utag to determine which way of the cache to read using their linear address, which is available before the load's physical address has been determined via the TLB. The utag is a hash of the load's linear address. This linear address-based lookup enables a very accurate prediction of the way the cache line is located in prior to a read of the cache data. This allows a load to read just a single cache way instead of all 12, saving power and reducing bank conflicts.

It is possible for the utag to be wrong in both directions: it can predict hit when the access will miss, and it can predict miss when the access could have hit. In either case, a fill request to the L2 cache is initiated and the utag is updated when L2 responds to the fill request.

Linear aliasing occurs when two different linear addresses are mapped to the same physical address. This can cause performance penalties for loads and stores to the aliased cache lines. A load to an address that is valid in the L1 DC but under a different linear alias will see an L1 DC miss, which requires an L2 cache request to be made. The latency will generally be no larger than that of an L2 cache hit. However, if multiple aliased loads or stores are in-flight simultaneously, they each may experience L1 DC misses as they update the utag with a particular linear address and remove another linear address from being able to access the cache line.

It is also possible for two different linear addresses that are NOT aliased to the same physical address to conflict in the utag, if they have the same linear hash. At a given L1 DC index (bits [11:6] of the linear address), only one cache line with a given linear hash is accessible at any time; any cache lines with matching linear hashes are marked invalid in the utag and are not accessible.

2.6.3 L2 Cache

The AMD Zen5 microarchitecture implements a unified 16-way set associative write-back L2 cache per core. This on-die L2 cache is inclusive of the L1 caches in the core. The L2 cache size is 1 Mbyte with a variable load-to-use latency of no less than 14 cycles. The L2 to L1 data path is 64 bytes wide.

2.6.4 L3 Cache

The AMD Zen5 microarchitecture implements an up to 96-MB L3 cache (depending on SOC configuration) that is 16-way set associative and shared by the cores inside a CPU complex. The L3 is a write-back cache populated by L2 victims. On L3 hits, lines are invalidated from the L3 if the hit was from a store, or the line was read by just one core. Lines remain in the L3 for L3 code fetch hits or if the line has been read by multiple cores. The L3 maintains shadow tags for each L2 cache in the complex. If a core misses in its local L2 and in the L3, the shadow tags are consulted. If the shadow tags indicate that the data resides in another L2 within the complex, a cache-to-cache transfer is initiated within the complex. The L3 has an average load-to-use latency of 46 cycles. The non-temporal cache fill hint, indicated with PREFETCHNTA, reduces cache pollution for data that will only be used once. Lines filled into the L2 cache with PREFETCHNTA are marked for quicker eviction from the L2, and when evicted from the L2 are not inserted into the L3.

2.7 Memory Address Translation

A translation lookaside buffer (TLB) holds the most-recently-used page mapping information. It assists and accelerates the translation of virtual addresses to physical addresses. A hardware table walker loads page table information into the TLBs.

The AMD Zen5 microarchitecture utilizes a two-level TLB structure.

2.7.1 L1 Translation Lookaside Buffers

The processor contains a fully associative L1 instruction TLB (ITLB) with 64 entries that can hold 4-Kbyte, 2-Mbyte, or 1-Gbyte page entries.

The fully associative L1 data TLB (DTLB) provides 96 entries that hold 4-Kbyte, 16-Kbyte, 2-Mbyte, or 1-Gbyte page entries.

If a 16-Kbyte aligned block of four consecutive 4-Kbyte pages are also consecutive and 16-Kbyte aligned in physical address space and have identical page attributes, the processor may opportunistically store them in a single TLB entry resulting in increased effective capacity for both L1 and L2 DTLB and ITLB. This is only done when the processor is operating in long mode.

2.7.2 L2 Translation Lookaside Buffers

The processor provides an 8-way set associative L2 ITLB with 2048 entries capable of holding 4-Kbyte pages, and 2-Mbyte pages. 1-Gbyte pages are only cached in the L1 ITLB and are not held in the L2 ITLB.

The L2 DTLB provides a unified 16-way set-associative L2 DTLB with 4096 entries capable of holding 4-Kbyte pages, 16-Kbyte, 2-Mbyte pages, and an additional 4-way set-associative 1G page L2 DTLB with 1024 entries.

2.7.3 Hardware Page Table Walkers

The AMD Zen5 microarchitecture has six hardware page table walkers to handle L2 TLB misses. Misses can start speculatively from either the instruction or the data side. A dedicated Page Walker Cache (PWC) structure holds PDPEs and PDEs, which are used to speed up tablewalks by skipping two or three levels of page table reads, respectively. In addition to the PDPE and PDE storage in the PWC, the table walker includes a 64-entry Page Directory Cache (PDC) which holds page-map-level-4 entries (PML4Es) and page-map-level-5 entries (PML5Es) to speed up table walks. The PDC entries and the PDPE and PDE entries in the PWC are usable by all tablewalk requests, including instruction-side table walks.

The table walker natively supports the architecturally defined 4-Kbyte, 2-Mbyte, and 1-Gbyte pages. In legacy mode, 4-Mbyte entries are also supported by returning a smashed 2-Mbyte TLB entry. Page smashing is a feature where the processor creates a TLB entry whose page size is smaller than the page size specified by the page tables for that linear address. In addition to supporting 4 Mbyte page sizes, Zen5 also uses page smashing if it encounters a larger page in the guest page tables which is backed by a smaller page in the host page tables. In this case, it smashes translations of the larger guest page size into the smaller page size found in the host page tables.

In the L1 TLBs, INVLPG, INVPCID, INVLPGB, and INVLPGA instructions cause a flush of all smashed entries corresponding to the same 1-Gbyte guest linear address page. In the L2TLBs, INVLPG, INVPCID, INVLPGB, and INVLPGA cause a flush of all smashed entries in the same context.

2.8 Optimizing Branching

Branching can reduce throughput when instruction execution must wait on the completion of the instructions that determine whether the branch is taken. The AMD Zen5 processor includes branch prediction logic that predicts the outcome of branches.

This prediction is used to speculatively fetch, decode, and execute instructions on the predicted path. When the prediction is correct, waiting is avoided, and the instruction throughput is increased. The branch misprediction penalty is in the range from 12 to 18 cycles, depending on the type of mispredicted branch and whether the instructions are being fed from the Op Cache. The common case penalty is 15 cycles.

2.8.1 Branch Prediction

To predict and accelerate branches, the processor employs:

- next-address logic
- branch target buffer
- return address stack (RAS)
- indirect target predictor
- advanced conditional branch direction predictor
- fetch window tracking structure

The following sections discuss these features.

2.8.1.1 Next-Address Logic

The next-address logic determines addresses for instruction fetch. When no branches are identified in the current fetch block, the next-address logic calculates the starting address of the next sequential naturally aligned 64-byte fetch block. This calculation is performed every cycle to support the bandwidth of the Op Cache. Branching to the end of a 64-byte cacheline results in a shortened fetch block and a loss of fetch bandwidth. When branches are identified, the next-address logic is redirected by the branch target and branch direction prediction hardware to generate a non-sequential fetch block address. The processor facilities that are designed to predict branch targets are detailed in the following sections.

2.8.1.2 Branch Target Buffer

The branch target buffer (BTB) is a two-level structure accessed using a previous fetch block's fetch address. Each BTB entry includes information for branches and their targets. Each BTB entry can hold up to two branches, and two pair cases are supported:

- A conditional branch followed by another branch with both branches having their last byte in the same 64 byte aligned cacheline.
- A direct branch (excluding CALLs) followed by a branch ending within the 64-byte aligned cacheline containing the target of the first branch.

Predicting with BTB pairs allows two fetches to be predicted in one prediction cycle. Each BTB entry has a limited number of bits for target addresses that are shared between up to two branches. Branches with branch targets that differ from their fetch address in a large number of bits may limit a BTB entry to storing only one branch.

Each level of BTB holds additional entries and prediction from higher level BTBs have longer latencies. When possible, keep the critical working set of branches in the code as small as possible. (In some cases, replacing known hard-to-predict branches with conditional mov instructions can improve performance.) The L1 BTB has 16K entries and predicts with zero prediction bubbles for direct calls, same-target indirect branches, conditional, and unconditional direct branches, and a two-cycle bubble for returns and multiple-target indirect branches. The L2 BTB has 8k entries and creates eight prediction bubbles if a prediction misses in L1BTB and hits in L2BTB.

2.8.1.3 Return Address Stack

The processor implements a 52-entry return address stack (RAS) per thread to predict return addresses from a near call. As calls are fetched, the address of the following instruction is pushed onto the return address stack. Typically, the return address is correctly predicted by the address popped off the top of the return address stack. However, mispredictions sometimes arise during speculative execution that can cause incorrect pushes and/or pops to the return address stack. The processor implements mechanisms that correctly recover the return address stack in most cases. If the return address stack cannot be recovered, it is invalidated and restored to a consistent state.

The following sections discuss common coding practices used to optimize subroutine calls and returns.

2.8.1.3.1 CALL 0h

When the `CALL` instruction is used with a displacement of zero, it is recognized and treated specially; the RAS remains consistent even if there is not a corresponding RET instruction.

Instead of using CALL 0h, 64-bit software can load the RIP into a register using the LEA instruction with RIP-relative addressing, as in the following example:

```
   LEA RAX, [RIP+0] ;RAX contains the value of RIP
```

2.8.1.3.2 REP RET

For prior processor families, such as Family 10h and 12h, a three-byte return-immediate RET instruction had been recommended as an optimization to improve performance over a single-byte near-return. For the AMD Zen5 microarchitecture, this is no longer recommended, and a single-byte near-return (opcode C3h) can be used with no negative performance impact. This will result in smaller code size over the three-byte method. For the former recommendation's rational see section, see section 6.2 in the Software Optimization Guide for AMD Family 10h and 12h Processors.

2.8.1.3.3 Function Inlining

Returns incur a two cycle bubble for an L1 BTB prediction. Therefore, function calls within hot loops should be inlined for better performance if there are few callers to the function or if the function is small (See section 8.3 of Software Optimization Guide for AMD Family 15h Processor).

2.8.1.4 Indirect Target Predictor

The processor implements a 3072-entry indirect target array used to predict the target of some non-RET indirect branches. If a branch has had multiple different targets, the indirect target predictor chooses among them using global history.

Indirect branches that have always had the same target are predicted using the static target from the branch's BTB entry. For this reason, code should attempt to reduce the number of different targets per indirect branch.

2.8.1.5 Advanced Conditional Branch Direction Predictor

The conditional branch predictor is used for predicting the direction of conditional near branches. Only branches that have been previously discovered to have both taken and non-taken behavior will use the conditional predictor. The conditional branch predictor uses a global history scheme that keeps track of previously executed branches. Global history is not updated for never-taken branches. For this reason, dynamic branches which are biased towards not taken are preferred.

Branch behavior that depends on deep history or that does not correlate well with global history is more likely mispredicted.

Conditional branches that have not yet been discovered to be taken are not marked in the BTBs. These branches are implicitly predicted not taken. Conditional branches are predicted as always-taken after they are first discovered to be taken. Conditional branches that are in the always-taken state are subsequently changed to the dynamic state if they are subsequently discovered to be not-taken, at which point they are eligible for prediction with the dynamic conditional predictor.

2.8.1.6 Fetch Window Tracking Structure

Fetch windows are tracked in a 128-entry (64 entries in SMT mode) FIFO from fetch until retirement. Each entry holds branch prediction information for up to a full 64-byte cache line. If a single BTB entry is not sufficient to allow prediction to the end of the cache line, the fetch window tracking structure uses additional entries for this particular cache line. If no branches are identified in a cache line, the fetch window tracking structure will use a single entry to track the entire cache line.

If the fetch window tracking structure becomes full, instruction fetch stalls until instructions retire from the retire control unit or a branch misprediction flushes some entries. Both mispredicting and retiring branches use information from this structure to update the prediction structures as needed.

2.8.2 Boundary Crossing Branches

Branches whose target crosses a sixteen-megabyte aligned boundary are unable to share BTB entries with other branches. Excessive occurrences of this scenario can reduce effective BTB capacity and bandwidth if the BTB entry could have otherwise been shared.

2.8.3 Loop Alignment

For the processor, loop alignment is not usually a significant issue. However, for hot loops, some further knowledge of trade-offs can be helpful. Because the processor can read an aligned 64-byte fetch block every cycle, it is suggested to either align the start of the loop to the beginning of a 64-byte cache line or align the end of the loop to the last byte of a cache line. Aligning the end of the loop to the end of a cache line is slightly more optimal for performance.

For very hot loops, it may be useful to further consider branch placement. The branch predictor can process the first two branches after the cache line entry point with a single BTB entry. For best performance, keep the number of predicted branches in the same cache line following a branch target at two or below. Since BTB entries can hold up to two branches, predicting a third branch requires an additional BTB entry and additional cycles of prediction latency.

This should not be confused with branches per cache line. For example, it is still optimal to have three or four branches per cache line if the second branch is unconditional or if the first or second branch is taken so frequently that the third and fourth branches are seldom executed.

2.8.3.1 Encoding Padding for Loop Alignment

Aligning loops is typically accomplished by adding `NOP` instructions ahead of the loop. This section provides guidance on the proper way to encode `NOP` padding to minimize its cost. Generally, it is beneficial to code fewer and longer `NOP` instructions rather than many short NOP instructions, because while NOP instructions do not consume execution unit resources, they still must be forwarded from the Decoder and tracked by the Retire Control Unit.

The table below lists encodings for NOP instructions of lengths from 1 to 15. Beyond length 8, longer NOP instructions are encoded by adding one or more operand size override prefixes (66h) to the beginning of the instruction.

Table 6. Encodings for NOP Instructions 1 to 15

Length	Encoding
1	90
2	66 90
3	0F 1F 00
4	0F 1F 40 00
5	0F 1F 44 00 00
6	66 0F 1F 44 00 00
7	0F 1F 80 00 00 00 00
8	0F 1F 84 00 00 00 00 00
9	66 0F 1F 84 00 00 00 00 00
10	66 66 0F 1F 84 00 00 00 00 00
11	66 66 66 0F 1F 84 00 00 00 00 00
12	66 66 66 66 0F 1F 84 00 00 00 00 00
13	66 66 66 66 66 0F 1F 84 00 00 00 00 00
14	66 66 66 66 66 66 0F 1F 84 00 00 00 00 00
15	66 66 66 66 66 66 66 0F 1F 84 00 00 00 00 00

The recommendation above is optimized for this processor.

Some earlier AMD processors, such as the Family 15h processor, suffer a performance penalty when decoding any instruction with more than 3 operand-size override prefixes. While this penalty is not present in the AMD Zen5 microarchitecture, it may be desirable to choose an encoding that avoids this penalty in case the code is run on a processor that does have the penalty.

The 11-byte NOP is the longest of the above encodings that uses no more than 3 operand size override prefixes (byte 66h). Beyond 11 bytes, the best single solution applicable to all AMD processors is to encode multiple NOP instructions. Except for very long sequences, this is superior to encoding a JMP around the padding.

The table below shows encodings for NOP instructions of length 12–15 formed from two NOP instructions (a NOP of length 4 followed by a NOP of length 8–11).

Table 7. Encodings for NOP Instructions 12-15

Length	Encoding
12	0F 1F 40 00 0F 1F 84 00 00 00 00 00
13	0F 1F 40 00 66 0F 1F 84 00 00 00 00 00
14	0F 1F 40 00 66 66 0F 1F 84 00 00 00 00 00
15	0F 1F 40 00 66 66 66 0F 1F 84 00 00 00 00 00

The AMD64 ISA specifies that the maximum length of any single instruction is 15 bytes. To achieve padding longer than that it is necessary to use multiple NOP instructions. For this processor, use a series of 15-byte NOP instructions followed by a shorter NOP instruction. If taking earlier AMD processor families into account, use a series of 11-byte NOPs followed by a shorter NOP instruction. Software should avoid instructions (including NOPs) longer than 10 bytes if code footprint is large and unlikely to be fetched from Op Cache. Only the first of the four instruction decoders in the AMD Zen5 microarchitecture can decode instructions longer than 10 bytes.

As a slightly more efficient alternative to inserting NOPs for padding, redundant prefixes can be used to pad existing instructions without affecting their function. This has the advantage of fewer instructions being kept in the Op Cache and maintained throughout the machine pipeline. For example, operand overrides (byte 66h) can be added to an instruction that already has operand overrides without changing its function. Whereas padding with NOPs is always possible, this method of using redundant prefixes is only practical when there are already useful instructions present that use prefixes.

2.9 Instruction Fetch and Decode

The processor fetches instructions from the instruction cache in 32-byte blocks that are 32-byte aligned. Up to two of these blocks can be independently fetched every cycle to feed the decode unit's two decode pipes. Instruction bytes from different basic blocks can be fetched and sent out-of-order to the 2 decode pipes, enabling instruction fetch-ahead which can hide latencies for TLB misses, Icache misses, and instruction decode.

Each decode pipe has a 20-entry structure called the IBQ which acts as a decoupling queue between the fetch/branch-predict unit and the decode unit. IBQ entries hold 16 byte-aligned fetch windows of the instruction byte stream.

The decode pipes each scan two IBQ entries and output up to four instructions per cycle. In single-thread mode the maximum throughput is 4 instructions per cycle. In SMT mode decode pipe 0 is dedicated to Thread 0 and decode pipe 1 is dedicated to Thread 1, supporting a maximum throughput of eight instructions per cycle.

The instruction decode window is 32 bytes, aligned on a 16-byte boundary. Having 16 byte aligned branch targets gets maximum instruction decode throughput.

Only the first decode slot (of four) can decode instructions greater than 10 bytes in length. Avoid having more than one instruction in a sequence of four that is greater than 10 bytes in length.

2.9.1 Op Cache

The Op Cache (OC) is a cache of previously decoded instructions. When instructions are being fetched from the Op Cache, normal instruction fetch and decode are bypassed. This improves pipeline latency because the Op Cache pipeline is shorter than the traditional fetch and decode pipeline. It improves bandwidth because the maximum throughput from the Op Cache is 12 instructions per cycle, whereas the maximum throughput from the traditional fetch and decode pipeline is 4 instructions per cycle per thread. Finally, it improves power because there is no need to re-decode instructions.

The Op Cache is organized as an associative cache with 64 sets and 16 ways. At each set-way intersection is an entry containing up to 6 instructions or fused instructions. This differs from Zen3 and Zen4 opcache entries, which stored macro-ops, and can improve effective Op Cache storage density. The maximum capacity of the Op Cache is 6 K instructions or fused instructions. The actual limit may be less due to efficiency considerations. Avoid hot code regions that approach this size when only one thread is running on a physical core, or half this size when two threads share a physical core. The Op Cache is physically tagged, which allows Op Cache entries to be shared between both threads when fetching shared code.

When instruction fetch misses in the Op Cache, and instructions are decoded after being read from the instruction cache (IC), they are also built into the Op Cache. Multiple instructions are built together into an Op Cache entry. Up to 6 sequential instructions or fused instructions may be cached together in an entry.

Op Cache entry limits:

- 6 instructions or fused instructions
- 6 32-bit immediate or displacement operand storage slots which can be used by any instruction. Further instructions which contain immediate, or displacement operands are not able to be included after all six slots have been used. 64-bit operands occupy 2 slots each while 8/16/32-bit operands occupy 1 slot each.
- 6 AVX-512 instructions with EVEX prefixes.
- The normal macro-op restriction does not apply when microcoded instructions are present. Instead, a limit of 4 is applied on the sum of microcoded and non-microcoded instructions.
- An Op Cache entry can only contain ops for instructions from up to two adjacent 64B cache lines.

The Op Cache is modal, and the processor can only transition between instruction cache mode (IC mode) and Op Cache mode (OC mode) at certain points. Instruction cache to Op Cache transitions can only happen at taken branches. The processor remains in Op Cache mode until an Op Cache miss is detected.

Excessive transitions between instruction cache and Op Cache mode may impact performance negatively. The size of hot code regions should be limited to the capacity of the Op Cache to minimize these transitions. This should be considered when unrolling loops.

Use of the Op Cache requires a flat memory model (64-bit or 32-bit with CS.Base = 0h and CS.Limit = FFFFFFFFh).

2.9.2 Idioms for Dependency removal

Several instructions can be used to clear a register and break the dependency without the need to load an immediate value of zero. These are referred to as Zeroing Idioms.

The processor supports the following Zeroing Idioms:

GPR Zeroing Idioms:

- `XOR reg, reg` (clears reg and the flags, 0-cycle operation)
- `SUB reg, reg` (clears reg and the flags, 0-cycle operation)
- `CMP reg, reg` (sets Z flag and clears other flags, 0-cycle operation)
- `SBB reg, reg` (copies the zero extended value of the carry flag into reg without a dependency on the previous value of reg, 1-cycle operation)

SIMD Zeroing Idioms (all clear destination register as a 0-cycle operation):

- `VXORP(S/D) zmm/ymm/xmm,zmm/ymm/xmm,zmm/ymm,xmm`
- `VANDNP(S/D) zmm/ymm/xmm,zmm/ymm/xmm,zmm/ymm/xmm`
- `VPCMPGT(B/W/D/Q) zmm/ymm/xmm,zmm/ymm/xmm,zmm/ymm/xmm`
- `VPANDN zmm/ymm/xmm,zmm/ymm/xmm,zmm/ymm/xmm`
- `VPXOR zmm/ymm/xmm,zmm/ymm/xmm,zmm/ymm/xmm`
- `VPSUB(B/W/D/Q) zmm/ymm/xmm,zmm/ymm/xmm,zmm/ymm/xmm`

A number of instructions can be used to set a register to all ones and break input dependencies. These are referred to as Ones Idioms.

The processor supports the following Ones Idioms:

- `PCMPEQ(B/W/D/Q) xmm,xmm`
- `VPCMPEQ(B/W/D/Q) zmm/ymm/xmm,zmm/ymm/xmm,zmm/ymm/xmm`

2.9.3 Branch Fusion

The processor’s decode unit is able to fuse conditional branch instructions with certain flag writing instructions into one fused instruction. A fused instruction only consumes one slot of an Op Cache entry and eliminates one macro-op flowing through the machine, which increases dispatch, issue, and retire bandwidth.

The following conditions need to be met for fusion to happen:

- The conditional branch needs to follow the flag writing instruction.
- The following flag writing instructions support branch fusion with their reg/reg, reg/imm and reg/mem forms. They are also supported when the instruction has an immediate and displacement that meets the criteria of 4 byte displacement and 2 byte immediate or the case of 2 byte displacement and 4 byte immediate.
  - CMP
  - TEST
  - SUB
  - ADD
  - INC
  - DEC
  - OR
  - AND
  - XOR

2.9.4 MOV + ALU Fusion

The Zen5 microarchitecture adds support to fuse reg-reg MOV Instructions with certain ALU instructions. The following conditions need to be met for fusion to happen:

- The MOV should be reg-reg mov with Opcode 0x89 or 0x8B
- The MOV is followed by an ALU instruction where the MOV and ALU destination register match.
- The ALU instruction may source only registers or immediate data. There cannot be any memory source.
- The ALU instruction sources either the source or dest of MOV instruction.
- If ALU instruction has 2 reg sources, they should be different.

The following ALU instructions can fuse with an older qualified MOV instruction:

- ADD
- ADC
- AND
- XOR
- OP
- SUB
- SBB
- INC
- DEC
- NOT
- SAL / SHL
- SHR
- SAR

2.9.5 Zero Cycle Move

The processor can execute certain register-to-register mov operations with zero cycle delay. The following instructions perform the mov operation with zero cycle delay:

- MOV r32/r64, r32/r64
- MOVSXD r32, r32
- XCHG EAX/RAX, r32/r64
- XCHG r32/r64, r32/r64
- (V)MOVAP(D/S) zmm1/ymm1/xmm1, zmm2/ymm2/xmm2
- (V)MOVDQ(U/A) zmm1/ymm1/xmm1, zmm2/ymm2/xmm2
- (V)MOVUP(D/S) zmm1/ymm1/xmm1, zmm2/ymm2/xmm2

2.9.6 Stack Pointer Tracking for Dependency Removal

The integer rename unit provides a mechanism to optimize the execution of certain implicit operations on the stack pointer and certain references to the stack pointer. The term stack pointer refers to the rSP register. When the stack pointer is tracked, supported instructions no longer have an execution dependency on older instructions which perform implicit updates of the stack pointer.

The following instructions with implicit stack pointer update support Stack Pointer Tracking:

- PUSH reg/mem/imm (excluding PUSH rSP)
- POP reg/mem (excluding POP rSP)
- CALL near rel
- CALL near abs reg
- RET near
- RET near imm

The following instructions referencing the stack pointer can take advantage of Stack Pointer Tracking:

- Memory references for load and store using rSP as base or index register.
- MOV reg, rSP
- Add/Sub RSP instructions with immediates of 12 bits (bits 31:13 are sign extensions in 32 bit instruction). Instructions supported are ADD (81 /0, 83 /0) and SUB (81 /5, 83 /5).
- This feature takes precedence over the capability to fuse the ADD or SUB if DECFG.StackEngineAddSubDisable is set to 0.
- LEA with the following memory addressing forms: [rSP + displacement], [rSP], [rSP + index × scale + displacement]
- Instructions updating rSP, not mentioned as supporting Stack Pointer Tracking, and instructions referencing rSP, not listed as using Stack Pointer Tracking, incur a penalty of an additional op and reset the tracking. Once tracking is reset there is no additional penalty until a supported update of rSP starts tracking again.

2.9.7 Dispatch

The processor may dispatch up to 8 macro-ops per cycle into the execution engine. For some instruction mixes, the processor may be limited by internal restrictions to less than 8 macro-ops of dispatch. However, software may optimize dispatch bandwidth by balancing the operations in any rolling window of 8 macro-ops. If the set of operations in such a window are a mix of operation types that can be executed concurrently in the execution engine, they may also be dispatched concurrently.

To optimize dispatch bandwidth, consider the following rules for groups of ops dispatched by the decoder:

- up to 2 Taken branches.
- up to 8 immediates or displacements.
- the 8th op cannot be a load or a store.

2.9.8 Using Pause Instruction to Optimize Spin Loops

Software may use the PAUSE instruction to reduce power consumption and other resource usage in a spin loop while waiting after failing to acquire a contended lock. The exact effect of the PAUSE instruction is implementation dependent, but in the AMD Zen5 microarchitecture the PAUSE instruction causes the executing thread to stop dispatch of macro-ops for a period of approximately 64 cycles.

2.9.9 Other Decode Considerations

Some versions of the AMD Zen5 microarchitecture include a false dependency on the destination operand of TZCNT, BLSI, BLSR, BLSMSK instructions which can prevent their out-of-order execution. Software can include a workaround (“xor dest, dest” prior to the instruction) to satisfy this dependency.

2.10 Integer Execution Unit

Figure 2 shows a diagram of the integer execution unit.

The integer execution unit for the processor consists of the following major components:

- schedulers
- execution units
- retire control

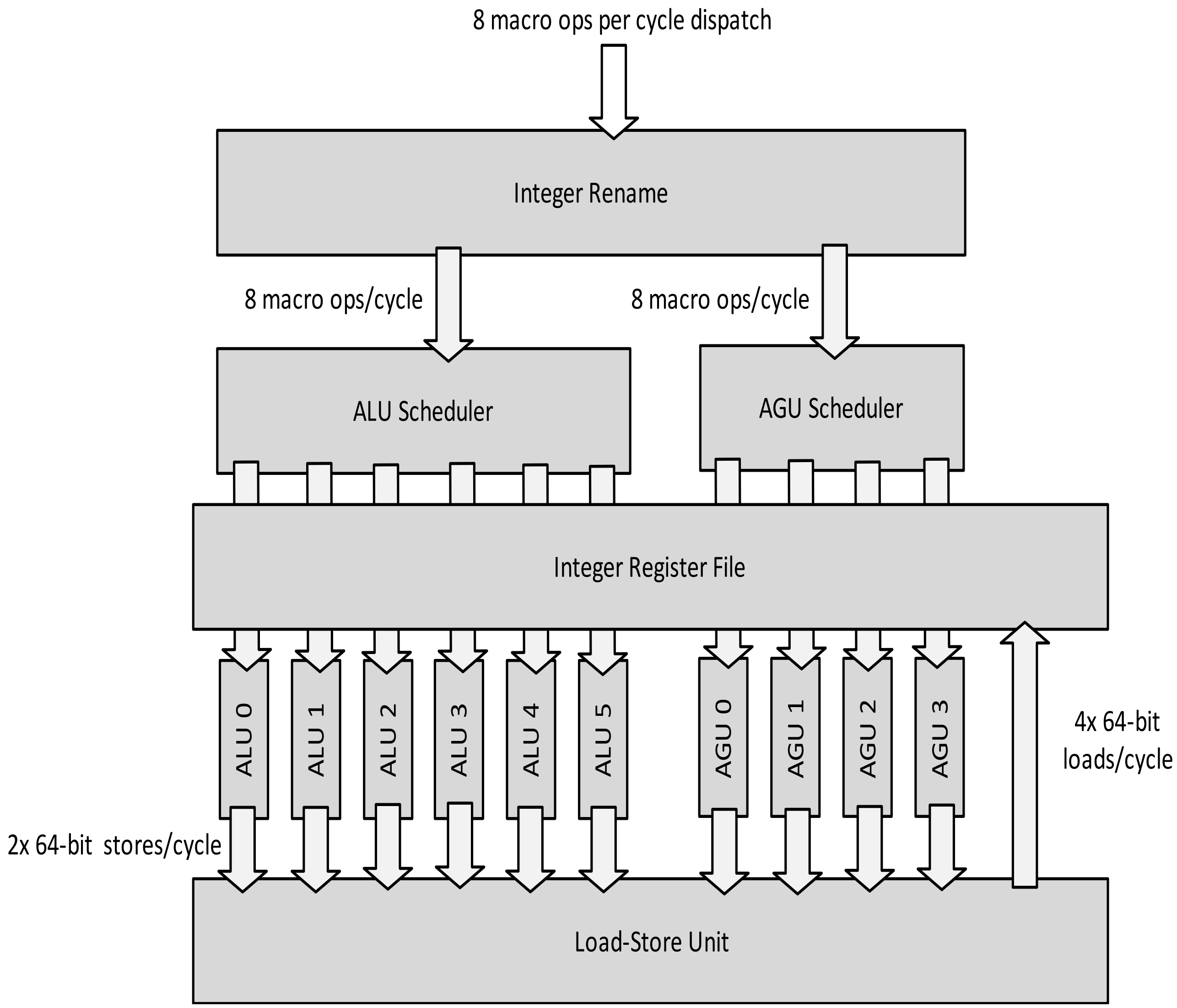

Figure 2. Integer Execution Unit Block Diagram

2.10.1 Schedulers

The schedulers can receive up to eight macro-ops per cycle, and track operand availability and dependency execution requirements of the contained micro-ops.

Micro-ops can be issued and executed out-of-order, but the scheduler ensures that older micro-ops which have been waiting for operands are issued in a timely manner. The ALU scheduler can issue six micro-ops per cycle and the AGU scheduler can issue four micro-ops per cycle.

2.10.2 Execution Units

The processor contains 6 general purpose integer execution pipes. Each pipe has an ALU capable of general-purpose integer operations. ALU0/ALU1/ALU2 additionally have multiply/CRC capability; ALU3/ALU4/ALU5 have PDEP/PEXT/shift/branch execution capability; and ALU3 additionally has divide capability. There are four Address Generation Units (AGUs) for all load and store address generation. All ALU pipelines can issue store data movement operations, with a bandwidth of two integer stores per cycle to the store queue. Many simple ALU operations, including most simple integer arithmetic, logical, branch, and conditional move instructions can be executed in a single cycle.

Shift instructions SHLD and SHRD have a three-cycle latency.

The AMD Zen5 microarchitecture has native ALU support for PDEP/PEXT, three such instructions can be sustained per cycle, with a three-cycle latency for producing the result. Software that uses different codepaths for processors with fast and slow PDEP/PEXT instructions should choose the fast PDEP/PEXT codepath for the AMD Zen5 microarchitecture.

While two-operand LEA instructions are executed as a single-cycle macro-op in the ALUs, three-operand LEA instructions are executed as two macro-ops in the ALUs.

The three integer multiply units can handle multiplies of up to 64 bits × 64 bits with 3 cycle latency, fully pipelined. If the multiply instruction has 2 destination registers, an additional one-cycle latency for the second result is required.

The hardware integer divider unit has a typical latency of 8 cycles plus 1 cycle for every 9 bits of quotient. The divider allows limited overlap between two consecutive independent divide operations. “Typical” 64-bit divides allow a throughput of one divide per 8 cycles (where the actual throughput is data dependent). For further information on instruction latencies and throughput, see Appendix A.

2.10.3 Retire Control Unit

The retire control unit (RCU) tracks the completion status of all outstanding operations (integer, load/store, and floating-point) and is the final arbiter for exception processing and recovery. The unit can receive up to 8 macro ops dispatched per cycle and track up to 448 macro ops in-flight in non- SMT mode or 224 per thread in SMT mode. In some cases, a single retire queue entry can track two macro-ops from adjacent instructions. This results in an increase of the effective retire queue capacity.

A macro-op is eligible to be committed by the retire unit when all corresponding micro-ops have finished execution. For most cases of fastpath double macro-ops, it is further required that both macro-ops have finished execution before commitment can occur. The retire unit handles in-order commit of up to eight retire queue entries, which may represent more than eight macro-ops, per cycle.

The retire unit also manages integer register mapping and renaming. The integer physical register file (PRF) consists of 240 registers, with up to 40 per thread mapped to architectural state or micro-architectural temporary state. The remaining registers are available for out-of-order renames.

The integer physical register file does not store flag information. Flag information is kept in a separate flag physical register file which provides 192 free registers that are available for out- of-order renames of flag writing instructions.

2.11 Floating-Point Unit

The processor provides native support for 32-bit single precision, 64-bit double precision and 80-bit extended precision primary floating-point data types as well as 128-, 256-, and 512-bit packed integer, single, double, and bfloat16 precision vector floating-point data types. The floating-point load and store paths are 512 bits wide.

The floating-point unit (FPU) utilizes a co-processor model for all operations that use X87, MMX™, XMM, YMM, ZMM, or floating-point control/status registers. As such, it contains its own scheduler, register file, and renamer; it does not share them with the integer units. It can handle dispatch of eight and renaming of six floating point macro-ops per cycle. When the floating-point scheduler is full or there are more than 6 floating point macro-ops dispatched in a cycle, overflow macro-ops are dispatched into the 96 entry Non-Scheduling Queue(NSQ) to allow for acceleration of load and store address calculation and support the full 8 macro-op per cycle dispatch bandwidth.

The floating-point scheduler has a 3*38 entry macro-op capacity and can issue 1 micro-op per cycle for each pipe. The floating-point unit shares the retire queue with the integer unit which holds up to 448 macro-ops or 224 per thread in SMT mode.

Figure 6 below shows a basic diagram of the floating-point unit and how it interfaces with the other units in the processor. Notice that there are four execution pipes (0 through 3) that can execute an operation every cycle. The FP unit receives up to two 512-bit loads every cycle from the load/store unit. There are also dedicated buses to enable fast moves between the floating-point registers and the general registers in the EX-unit. Store data and floating point to general purpose register transfer have two dedicated pipelines (pipe 5 and 6). FP stores are supported on two pipelines, but throughput is limited to one per cycle. For further information on instruction latencies and throughput, see Appendix A, “Understanding and Using Instruction Latency Tables,”.

The floating-point unit supports AVX-512 with full 512-bit data paths and operations.

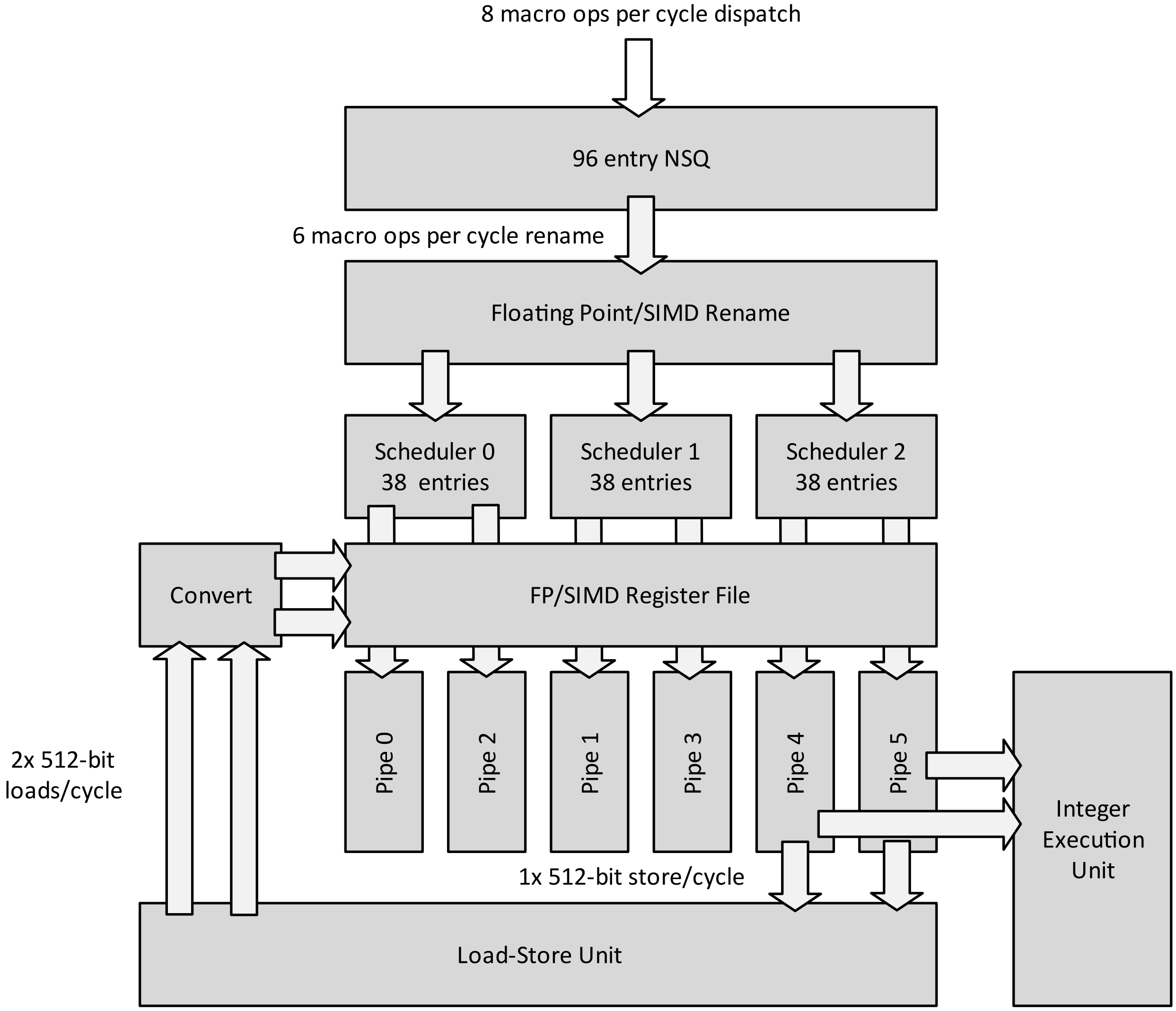

Figure 3. Floating-Point Unit Block Diagram

Pipes 0, 1, 2, and 3 support operations that require three operands. The third operand for pipes 0 and 2 contends for the source bus normally allocated to pipe 4, and the third operand for pipes 1 and 3 contend for the source bus normally allocated to pipe 5. This contention stalls the operation that loses the arbitration.

If the data for the third operand, or the pipe 4/5 source, can be bypassed from a result that same cycle, then there is no contention. AVX-512 operations with merge masking require an additional source operand, and the potential source bus contention may affect latency and throughput of these operations.

2.11.1 Floating Point Execution Resources

Unit	Pipe						Domain[4]	Ops Supported
	0	1	2	3	4	5		
FMUL	X	X					F	(v)FMUL*, (v)FMA*, Floating Point Compares, Blendv(DQ)
FADD			X	X			F	(v)FADD* Signature
FCVT			X	X			F	All convert operations except pack/unpack
FDIV[1]		X					F	All Divide and Square Root except Reciprocal Approximation
FMISC	X	X	X	X			F	Moves and Logical operations on Floating Point Data Types
STORE					X	X	S	Stores and Move to General Register (EX) Operations
VADD[2]	X	X	X	X			I	Integer Adds, Subtracts, and Compares
VMUL	X	X		X			I	Integer Multiplies, SAD
VSHUF[3]		X	X				I	Data Shuffles, Packs, Unpacks, Permute, Blendvb
VSHIFT		X	X				I	Bit Shift Left/Right operations
VMISC	X	X	X	X			I	Moves and Logical operations on Packed Integer Data Types
AES	X	X					I	*AES*
CLM	X	X						*CLM*

Notes:

1. *FDIV unit can support two simultaneous operations in flight, even though it occupies a single pipe.*

2. *Some complex VADD operations are not available in all pipes.*

3. *Some simple shuffles are also available in pipes 0 and 3*

2.11.2 Code recommendations

1. Use the SIMD nature of the SSE or AVX instruction sets to achieve significantly higher throughput. The AMD Zen5 microarchitecture supports SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, SSE4a, F16C, FMA, AVX, AVX2, and AVX-512. In the AMD Zen5 microarchitecture, the datapath is 512 bits across all operations, so optimal code operates on 512-bit (ZMM registers) with every operation using the SIMD instructions. Due to the reduced instruction and macro-op overhead, expect software to have reduced power and increased performance when using 512-bit operations, compared to narrower operations.
2. Do full width loads and stores. For example, use vmovapd instead of movapd or movlpd/movhpd. Loading or storing a single register in multiple chunks is slower than doing it with a single operation. If multiple loads must be used, they should be placed back-to-back in the code.
3. Clear floating-point registers using one of the zeroing idioms listed in Section 2.9.2, "Idioms for Dependency removal" on page 27 when done using them. This allows the physical register to be freed up for speculative results and enables the machine to break merging dependencies for ops that do not write the entire result width such as scalar operations.
4. If possible, set MXCSR.DAZ (Denorm As Zero) or MXCSR.FTZ (Flush denorm To Zero) to 1. The hardware supports denormal inputs and outputs with no latency impact on most operations. A few operations execute assuming normal floating-point inputs or outputs. When the ops discover a denormal input or output, they may be re-executed with a longer latency. These ops include multiply, divide, and square root. Re-executing with the longer latency is avoided if the DAZ and FTZ flags are set.
5. Avoid branches/jumps in the calculation of values. For example, if one needs to do `if (a > 1.1) then b = 4 else b = 6`, then use VCMPGTSD followed by a predicated mov into b.
6. XMM/YMM/ZMM register-to-register moves have no latency; these instructions may be used without penalty.
7. Try to use consistent data types for instructions operating on the same data. For example, use VANDPS, VMAXPS, and other packed single precision operations when consuming the output of VMULPS.
8. Avoid GATHER instructions when the indices are known ahead of time. Vector loads followed by shuffles result in a higher load bandwidth.

2.11.3 FP performance on x87 code

1. Use `FXCH` instead of push/pop if possible as it is much faster at swapping register values.
2. Avoid instructions between `FCOM` and `FSTSW` in floating point compares.

2.11.4 Denormals

Denormal floating-point values (also called subnormals) can be created by a program either by explicitly specifying a denormal value in the source code or by calculations on normal floating-point values. In some instances, (MUL/DIV/SQRT) a small penalty may be incurred when these values are encountered. For SSE/AVX instructions, the denormal penalties are a function of the configuration of MXCSR and the instruction sequences that are executed in the presence of a denormal value.

If denormal precision is not required, it is recommended that software set both MXCSR.DAZ and MXCSR.FTZ. Setting MXCSR.DAZ or MXCSR.FTZ will cause the processor to produce results that are not compliant with the IEEE-754 standard when operating on or producing denormal values.

The x87 FCW does not provide functionality equivalent to MXCSR.DAZ or MXCSR.FTZ, so it is not possible to avoid these denormal penalties when using x87 instructions that encounter or produce denormal values.

2.11.5 XMM Register Merge Optimization

The processor implements an XMM register merge optimization. The processor keeps track of XMM registers whose upper portions have been cleared to zeros. This information can be followed through multiple operations and register destinations until non-zero data is written into a register. For certain instructions, this information can be used to bypass the usual result merging for the upper parts of the register. For instance, SQRTSS does not change the upper 96 bits of the destination register. If some instruction clears the upper 96 bits of its destination register and any arbitrary following sequence of instructions fails to write non-zero data in these upper 96 bits, then the SQRTSS instruction can proceed without waiting for any instructions that wrote to that destination register.

The instructions that benefit from this merge optimization are:

- CVTPI2PS
- CVTSI2SS(32-/64-bit)
- MOVSS xmm1,xmm2
- CVTSD2SS
- CVTSS2SD
- MOVLPS xmm1,[mem]
- CVTSI2SD (32-/64-bit)
- MOVSD xmm1,xmm2
- MOVLPD xmm1,[mem]
- RCPSS
- ROUNDSS
- ROUNDSD
- RSQRTSS
- SQRTSD
- SQRTSS

2.11.6 Mixing AVX and SSE

There is a penalty for mixing SSE and AVX instructions when the upper 384 bits of the ZMM registers contain non-zero data. The SSE operations get promoted to 256-bit or 512-bit operations to merge the data to the result. Instructions that do not naturally depend on the result operand gain an additional dependency.

2.11.7 When to use FMA instead of FMUL / FADD

Software will sometimes need to choose between using FMA (multiply accumulate) or separate FMUL / FADD operations. In those cases, the following guidelines are offered.

Do not use FMA if:

- The critical dependency is through the addend input of an FMA instruction. In this case, an FADD provides a shorter latency. However, when in a loop, unrolling to remove the dependency can result in the highest performance.

Use FMA if:

- The critical dependency is through an FMUL instruction that feeds its result as an input to an FADD instruction. In this case, FMA provides a shorter latency than the combined FMUL + FADD latency.
- None of the operands are in the critical dependency chain. In this case, an FMA instruction provides more efficient use of processor resources, reduces code footprint, and in most cases reduces power consumption.

2.12 Load-Store Unit

The load-store (LS) unit handles data accesses. The LS unit contains four largely independent pipelines enabling the execution of four memory operations per cycle. All four memory operations can be loads, with a separate maximum of two 128-, 256-, or 512-bit loads. A maximum of two of the memory operations can be stores, with a maximum of one store if the store is a 512-bit store.

The LS unit includes a load queue (LDQ). The LDQ receives load operations at dispatch. Loads leave the LDQ when the load has completed and delivered data to the integer unit or the floating-point unit. The LS can track up to 64 uncompleted loads and has no specific limit on the number of completed loads.

The LS unit utilizes a 104-entry store queue (STQ) which holds stores from dispatch until the store data can be written to the data cache.

The LS unit dynamically reorders operations, supporting both loads bypassing older loads and loads bypassing older non-conflicting stores. The LS unit ensures that the processor adheres to the architectural load and store ordering rules as defined by the AMD64 architecture.

The LS unit supports store-to-load forwarding (STLF) when there is an older store that contains all of the load's bytes, and the store's data has been produced and is available in the store queue. The load does not require any particular alignment relative to the store or to the 64B load alignment boundary as long as it is fully contained within the store.

The processor uses linear address bits 11:0 to determine STLF eligibility. Avoid having multiple stores with the same 11:0 address bits, but to different addresses (different 56:12 bits) in-flight simultaneously where a load may need STLF from one of them. Loads that follow stores to similar address space should be grouped closely together, when possible.

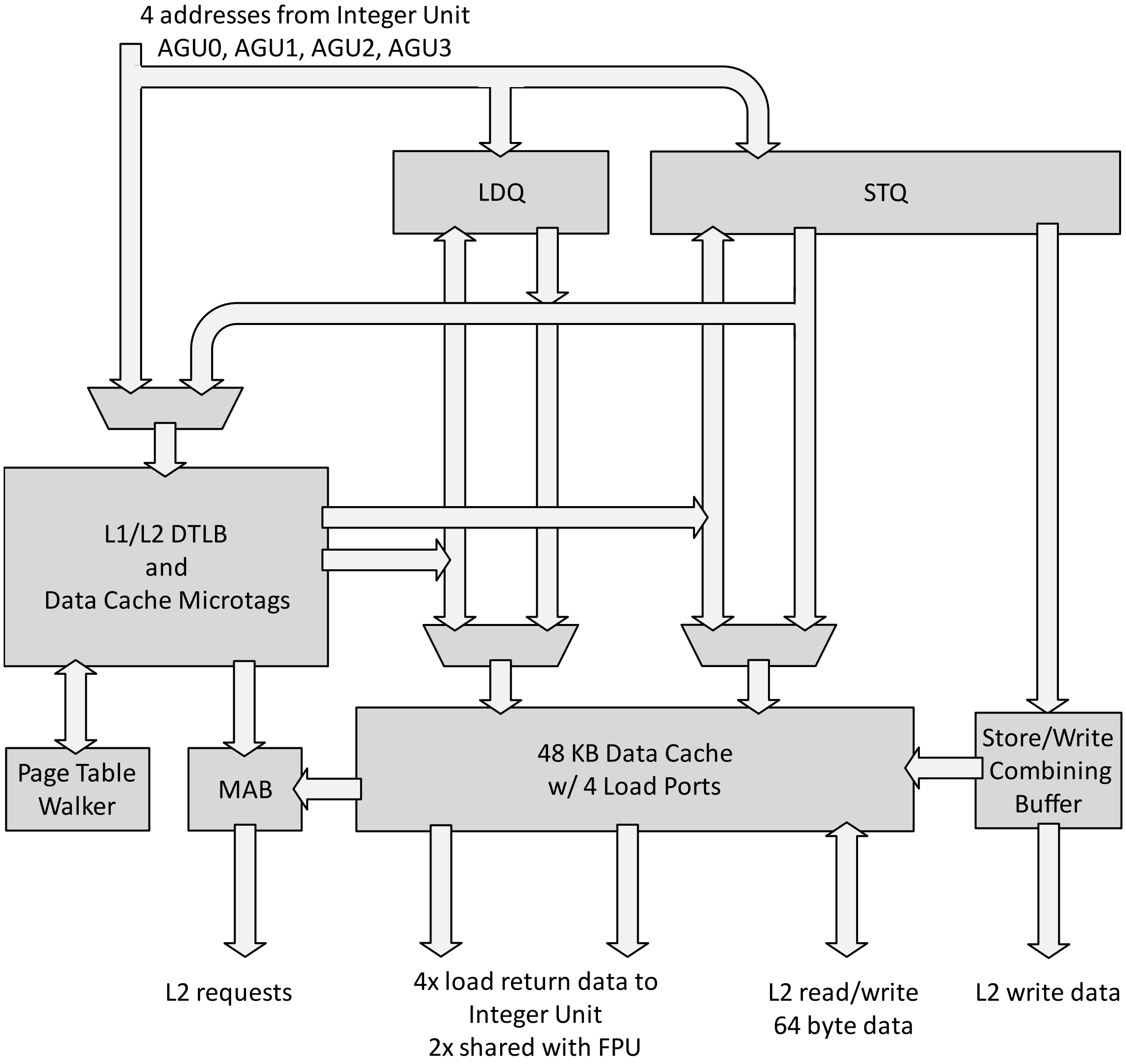

Figure 4. Load-Store Unit

The LS unit can track up to 124 outstanding in-flight cache misses.

The AGU and LS pipelines are optimized for simple address generation modes. Base+displacement, base+index, unscaled index+displacement, and displacement-only addressing modes (regardless of displacement size) are considered simple addressing modes and can achieve 4-cycle load-to-use integer load latency and 7-cycle load-to-use FP load latency. Addressing modes with base+index+displacement, and any addressing mode utilizing a scaled index (*2, *4, or *8 scales) are considered complex addressing modes and require an additional cycle of latency to compute the address. Complex addressing modes can achieve a 5-cycle (integer)/8-cycle (FP) load-to-use latency. It is recommended that compilers avoid complex addressing modes in latency-sensitive code.

The load store pipelines are optimized for zero-segment-base operations. A load or store that has a non-zero segment base suffers a one-cycle penalty in the load-store pipeline. Most modern operating systems use zero segment bases while running user processes and thus applications will not normally experience this penalty.

This segment-base latency penalty is not additive with the above-mentioned complex addressing-mode penalty. If an LS operation has both a non-zero base and a complex addressing mode, it requires just a single additional cycle of latency and can still achieve 5-cycle (integer)/8-cycle (FP) load-to- use latency.

2.12.1 Prefetching of Data

The AMD Zen5 microarchitecture implements data prefetch logic for its L1 data cache and L2 cache. In general, the L1 data prefetchers fetch lines into both the L1 data cache and the L2 cache, while the L2 data prefetchers fetch lines into the L2 cache.

The following prefetchers are included:

- L1 Stream: Uses history of memory access patterns to fetch additional sequential lines in ascending or descending order.
- L1 Stride: Uses memory access history of individual instructions to fetch additional lines when each access is a constant distance from the previous. The stride prefetcher can also recognize a "jump” in the stride pattern if the jump distance is a constant and the jump occurs at a regular interval. This pattern is commonly used when performing 2-dimensional array accesses.
- L1 Region: Uses memory access history to fetch additional lines when the data access for a given instruction tends to be followed by a consistent pattern of other accesses within a localized region.
- L2 Stream: Uses history of memory access patterns to fetch additional sequential lines in ascending or descending order.
- L2 Up/Down: Uses memory access history to determine whether to fetch the next or previous line for all memory accesses.

For workloads that miss in the L1 or L2 caches, software may get improved performance if data structures are designed such that data access patterns match one of the above listed behaviors.

While prefetcher logic has been tuned to improve performance in most cases, for some programs the access patterns may be hard to predict. This can lead to prefetching data that will not eventually be used, causing excess cache and memory bandwidth usage. This can be the case for workloads with random access patterns or less regular access patterns such as some database applications, etc. For this reason, some server variants of the AMD Zen5 microarchitecture support a Prefetch Control MSR that can individually disable or enable the prefetchers. See Processor Programming Reference for details on CPUID enumeration and MSR details.

2.13 Optimizing Writing Data

Write-combining is the merging of multiple memory write cycles that target locations within the address range of a write buffer. AMD Zen5 processors support the memory type range register (MTRR) and the page attribute table (PAT) extensions, which allow software to define ranges of memory as either writeback (WB), write-protected (WP), writethrough (WT), uncacheable (UC), or write-combining (WC).

Defining the memory type for a range of memory as WC allows the processor to conditionally combine data from multiple write cycles that are addressed within this range into a merge buffer. Merging multiple write cycles into a single write cycle reduces processor bus utilization and processor stalls. Write combining buffers are also used for streaming store instructions such as MOVNTQ and MOVNTI.

2.13.1 Software Prefetching of Data or Instructions

The AMD Zen5 microarchitecture implements improved software prefetch control allowing the PREFETCH instructions to load either data or instructions into targeted levels of the Cache Hierarchy.

Two new locality reference types (IT0 and IT1) are added to the PREFETCHlevel instruction to specify instruction prefetch operation. The following table shows the behavior of PREFETCH instructions on the Zen5 microarchitecture.

Table 8. Behavior of the PREFETCH Instructions.

Instruction	Description
PREFETCH mem8, PREFETCHT0 mem8	All Data Cache Levels—Move the specified data into all data cache levels.
PREFETCHT1 mem8	Level 2 Data Cache and Higher—Move the specified data into all data cache levels except 0th level (L1) cache.
PREFETCHT2 mem8	Level 3 Data Cache and Higher—Move the specified data into all data cache levels except 0th level (L1) and 1st level (L2) caches
PREFETCHW mem8	Moves the specified cacheline into the L1 data cache with a hint that its written. The processor attempts to get the line in an exclusive coherency state.
PREFETCHNTA mem8	Non-Temporal Access - Move the specified data into the processor with minimum cache pollution. This is intended for data that will be used only once, rather than repeatedly.
PREFETCHIT0 mem8	All Instruction Cache Levels—Move the specified instructions into all instruction cache levels
PREFETCHIT1 mem8	Level 2 instruction Cache and Higher—Move the specified instructions into all instruction cache levels except 0th level (L1) cache.

2.13.2 Write-Combining Definitions and Abbreviations

This section uses the following definitions and abbreviations:

- MTRR—Memory type range register
- PAT—Page attribute table
- UC—Un-cacheable memory type
- WC—Write-combining memory type
- WT—Writethrough memory type
- WP—Write-protected memory type
- WB—Writeback memory type

2.13.3 Programming Details

Write-combining regions are controlled by the MTRRs and PAT extensions. Write-combining should be enabled for the appropriate memory ranges.

For more information on the MTRRs and the PAT extensions, see the following documents:

- AMD64 Architecture Programmer's Manual, Volume 2, order# 24593
- Processor Programming Reference (PPR) for AMD Family 19h Models 00h-0Fh Processors, order# 55898

2.13.4 Write-Combining Operations

To improve system performance, AMD Zen5 processors include a Write Combining Buffer (WCB) that consists of multiple 64-byte write buffers that are aligned to cache-line boundaries. The write buffers aggressively combine multiple memory-write cycles of any data size that address locations within 64-byte aligned regions. The processor continues to combine writes to this buffer without writing the data to the system, if certain rules apply (see Table 9 for more information). The data sizes can be bytes, words, doublewords, quadwords, xmmword, ymmword, or zmmword.

- WC memory type writes can be combined in any order up to a full 64-byte write buffer.
- All other memory types for stores that go through the write buffer (UC, WP, WT and WB) cannot be combined except when the WB memory type is over-ridden for streaming store instructions such as the MOVNTQ and MOVNTI instructions, etc. These instructions use the write buffers and will be write-combined in the same way as address spaces mapped by the MTRR registers and PAT extensions. When WCB is used for streaming store instructions, the buffers are subject to the same flushing events as write-combined address spaces.

The processor may combine writes that do not store all bytes of a 64-byte write buffer. These partially filled buffers may not be closed for significant periods of time and may affect the bandwidth of remaining writes in a stream. Aligning write-combining operations to 64-byte cache line boundaries avoids having partially full buffers. When software starts a long write-combining operation on a non- cache line boundary, it may be beneficial to place a write-combining completion event (listed in Table 9) to ensure that the first partially filled buffer is closed and available to the remaining stores.

Combining continues until interrupted by one of the conditions listed in Table 9. When combining is interrupted, one or more bus commands are issued to the system for that write buffer and all older write buffers, even if not full, as described in section “Sending Write-Buffer Data to the System”.

Table 9. Write-Combining Completion Events

Event	Comment
No Write Buffers Available	If a write needs to allocate in the write buffer when no entries are available, the oldest write buffer is closed.
I/O Read or Write	Any IN/INS or OUT/OUTS instruction closes combining. The implied memory type for all IN/OUT instructions is UC, which cannot be combined.
Serializing instructions	Any serializing instruction closes combining. These instructions include: MOVCRx, MOVDRx, WRMSR, INVD, INVLPG, WBINVD, LGDT, LLDT, LIDT, LTR, CPUID, IRET, RSM, INIT, and HALT.
Flushing instructions	CLFLUSH will only close the WCB if it is for WC or UC memory type.
Locks	Any instruction or processor operation that requires a cache or bus lock closes write-combining before starting the lock. Writes within a lock can be combined.
Uncacheable Reads and Writes	A UC read or write closes write-combining. A WC read closes combining only if a cache block address match occurs between the WC read and a write in the write buffer.
Different memory type	When a store hits on a write buffer that has been written to earlier with a different memory type than that store, the buffer is closed and flushed.
Buffer full	Write-combining is closed if all 64 bytes of the write buffer are valid.
TLB AD bit set	Write-combining is closed whenever a TLB reload sets the accessed [A] or dirty [D] bits of a PDE or PTE.
Executing SFENCE (Store Fence) and MFENCE (Memory Fence) instructions.	These instructions force the completion of pending stores, including those within the WC memory type, making these globally visible and emptying the store buffer and all write-combining buffers.
An interrupt or exception occurs.	Interrupts and exceptions are serializing events that force the processor to write all results to memory before fetching the first instruction from the interrupt or exception service routine

Note: *See section LOCKs for more information on locks and memory barriers.*

2.13.5 Sending Write-Buffer Data to the System

Maximum throughput is achieved by write combining when all quadwords or doublewords are valid and the processor can use one efficient 64-byte memory write instead of multiple 16-byte memory writes. The processor can gather writes from twelve different 64B cache lines (up to eleven from one thread when SMT is active). Throughput is best when the number of simultaneous write-combining streams is low.

2.13.6 String Store Optimizations

The AMD Zen5 architecture includes several optimizations to improve the performance of stores produced by rep movs and rep stos instructions (string instructions).

At very large string sizes, sizes that are greater than or equal to the L3 size of the processor, Zen5 performs string stores using streaming-store operations. Streaming stores are non-cacheable stores that bypass the cache hierarchy of the processor and write data directly to the destination after aggregation in Write Combining Buffer. This optimization avoids replacing all cachelines in the cache hierarchy with the string data.

At smaller string sizes, in some system configurations, the AMD Zen5 architecture includes an optimization to eliminate the Read For Ownership (RFO) cache-coherence action for the destination cachelines that are fully overwritten by the string instruction. The destination cachelines are allocated into the cache hierarchy without being read, and are fully overwritten by the stores of the string instruction. The size threshold at which this optimization is active is implementation dependent. The optimization does not require that the size of the string instruction be a multiple of the cacheline size; the processor handles cachelines that are not fully-overwritten using a read-for-ownership.

In both optimizations, the stores produced by the string instruction may become visible to other processors out-of-order with respect to other stores in the string instruction. However, the processor ensures that stores older than the string instruction are visible before any stores in the string instruction, and that all stores from the string instruction are visible before stores younger than the string instruction.

2.14 Simultaneous Multi-Threading

To improve instruction throughput, the processor implements Simultaneous Multi-Threading (SMT). Single-threaded applications do not always occupy all resources of the processor at all times. The processor can take advantage of the unused resources to execute a second thread concurrently.

Resources such as queue entries, caches, pipelines, and execution units can be competitively shared, watermarked, or statically partitioned in two-threaded mode (see Table 10 below).

These categories are defined as:

- Competitively Shared: Resource entries are assigned on demand. A thread may use all resource entries.
- Watermarked: Resource entries are assigned on demand. When in two-threaded mode a thread may not use more resource entries than are specified by a watermark threshold.
- Statically Partitioned: Resource entries are partitioned when entering two-threaded mode. A thread may not use more resource entries than are available in its partition.

Competitively Shared is the L3 cache default protocol, but sharing policy can be configured using the AMD64 Technology Platform Quality of Service Extensions. For more detail, see the following:

- AMD64 Technology Platform Quality of Service Extensions, order# 56375
- Processor Programming Reference (PPR) for AMD Family 17h Models A0h-AFh Processors, order# 57243
- Processor Programming Reference (PPR) for AMD Family 19h Models 10h-1Fh Processors, order# 55901

Table 10. Resource Sharing

Resource	Competitively Shared	Watermarked	Statically Partitioned
L1 Instruction Cache	X		
ITLB	X		
Op Cache	X		
Dispatch Interface	X		
L1 Data Cache	X		
DTLB	X		
L2 Cache	X		
L3 Cache	X		
Integer Scheduler		X	
Integer Register File		X	
Load Queue		X	
Floating Point Physical Register		X	
Floating Point Scheduler		X	
Memory Request Buffers		X	
Op Queue			X
Store Queue			X
Write Combining Buffer		X	
Retire Queue			X

To reduce BTB collisions, if the two threads are running different code, they should run in different linear pages. If BTB and Op Cache sharing is desired, such as for two threads running the same code, the code should run at the same linear and physical addresses. Operating system features which randomize the address layout such as Windows® ASLR should be configured appropriately.

2.15 LOCKs

The processor implements logic to improve the performance of LOCKed instructions. In order to benefit from this logic, the following guidelines are recommended:

- Ensure that LOCKed memory accesses do not cross 64-byte aligned boundaries.
- Following a LOCKed instruction, refrain from using floating point instructions as long as possible.
- Ensure that Last Branch Record is disabled (DBG_CTL_MSR.LBR = 0h)

Appendix A Understanding and Using Instruction Latency Tables

The companion file, `Zen5_Instruction_Latencies_version_1-00.xlsx`, distributed with this Software Optimization Guide, provides additional information for the processor. This appendix explains the columns and definitions used in the table of latencies. Information in the spreadsheet is based on estimates and is subject to change.

A.1 Instruction Latency Assumptions

The term *instruction latency* refers to the number of processor clock cycles required to complete the execution of a particular instruction from the time that it is issued until a dependent instruction can be issued. Throughput refers to the number of results that can be generated in a unit of time given the repeated execution of a given instruction, assuming that all dependencies are resolved.

Many factors affect instruction execution time. For instance, when a source operand must be loaded from a memory location, the time required to read the operand from system memory adds to the execution time. Furthermore, latency is highly variable since a memory operand may or may not be found in one of the levels of data cache. In some cases, the target memory location may not even be resident in system memory due to being paged out to backing storage.

In estimating the instruction latency and reciprocal throughput, the following assumptions are necessary:

- The instruction is an L1 I-cache hit that has already been fetched and decoded, with the operations loaded into the scheduler.
- Memory operands are in the L1 data cache.
- There is no contention for execution resources or load-store unit resources.

Each latency value in the spreadsheet denotes the typical execution time of the instruction when run in isolation on a processor. For real programs executed on this highly aggressive super-scalar processor, multiple instructions can execute simultaneously; therefore, the effective latency for any given instruction's execution may be overlapped with the latency of other instructions executing in parallel.

Latencies in the spreadsheet reflect the number of cycles from instruction issuance to instruction retirement. This includes the time to write results to registers or the write buffer, but not the time for results to be written from the write buffer to L1 D-cache, which may not occur until after the instruction is retired.

For most instructions, the latency shown refers to the shortest latency version of the instruction which in most cases is its register-to-register version. To calculate the latency for instructions that load from memory the following additional latencies need to be added:

- Simple addressing mode ([base], [base + index], [base + displacement] or [displacement]) and zero segment base with aligned memory operand
  - GPR destination register.
    - L1 Data Cache hit: add 4 cycles
    - L2 Cache hit: add 14 cycles (may vary by product)
    - L3 Cache hit: add an average of 46 cycles (may vary by product)
  - FP/SIMD destination register.
    - L1 Data Cache hit: add 7 cycles
    - L2 Cache hit: add 17 cycles (may vary by product)
    - L3 Cache hit: add an average of 53 cycles (may vary by product)
- Complex addressing mode ([base + index + displacement] or any scaled index register) or non-zero segment base: add an additional cycle
- Misaligned memory operands: add an additional cycle

AVX-512 instructions using merge masking may encounter higher latency and/or lower throughput than listed in the spreadsheet.

To measure the latency of an instruction that stores data to memory, it is necessary to define an end- point at which the instruction is said to be complete. This guide uses the availability of the store data as the end point, and under that definition, writes add no additional latency. Choosing another end point, such as the point at which the data has been written to the L1 cache, would result in variable latencies and would not be meaningful without considering the context in which the instruction is executed.

There are cases where additional latencies may be incurred in a real program that are not described in the spreadsheet, such as delays caused by L1 cache misses or contention for execution or load-store unit resources.

A.2 Spreadsheet Column Descriptions

The following table describes the information provided in each column of the spreadsheet:

Table 11. Spreadsheet Column Descriptions

Cols	Label	Description
A	Instruction	Instruction mnemonic
B	Instruction operands	The following notations are used in this column: I – an immediate operand(value range left unspecified) x – any XMM register y – any YMM register z – any ZMM register m – a memory operand (value range left unspecified) M – any 64-bit MMX register r/g – any general purpose(integer) register k – any AVX-512 mask register s – any 80-bit x87 stack register A slash denotes an alternative. For example: x,x/m,I – operand 1 is XMM, while operand 2 is either XMM or memory. xyz, xyz, xyz/m – operands 1, 2 is either XMM or YMM or ZMM, operand 3 is either XMM/mem or YMM/mem or ZMM/mem
C-F	Operand1-Operand4	These are the operands divided further into individual operands
G	Opmask	Opmask values - Lists whether the instructions use {k}/{kz} opmasks
H	APM Vol	AMD64 Programmer’s Manual Volume that describes the instruction
I	Cpuid Flag	CPUID feature flag for this instruction
J	Macro Ops	Number of macro-ops for the instruction Any number greater than implies that the instruction is microcoded with the given number of macro–ops in the micro-program If the entry in this column is simply ‘ucode’ then the instruction is microcoded but the exact number of macro ops is variable. Note that stores and integer instructions using a memory operand that are listed as 1 op in the spreadsheet will require 2 ops when using an addressing mode with two register sources.

Table 11. Spreadsheet Column Descriptions (continued)

Cols	Label	Description
K	Unit	Execution Units. The following abbreviations are used. Ucode – Instruction is implemented using a variable number of macro-ops FPn – Instruction can only execute in FP pipe n FPn/FPm – Instruction can execute either in FP pipe n or m FPn, FPm – Instruction execution uses FP pipe n followed by FP pipe m ALU – Instruction can flow in any of 6 ALU pipes ALU n/m – Instruction can flow in either ALU pipe n or m BRN – Instruction can flow in any of 2 BRN pipes MUL – Integer multiply functional element within the integer unit DIV – Integer divide functional element within the integer unit NA – Instruction is not supported
L	Latency	Instruction latency in processor cycles
M	Throughput	Throughout of the instruction
N	Notes	Additional information about the instruction