



Software Optimization Guide for AMD Family 17h Models 30h and Greater Processors

Publication No.	Revision	Date
56305	3.01	February 2020

© 2019-2020 Advanced Micro Devices, Inc. All rights reserved.

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-V, AMD Virtualization, and combinations thereof, are trademarks of Advanced Micro Devices, Inc.

Windows is a registered trademark of Microsoft Corporation.

MMX is a trademark of Intel Corporation.

Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

Contents

Chapter 1	Introduction	15
1.1	Intended Audience	15
1.2	Specialized Terminology	16
Chapter 2	Microarchitecture of AMD Family 17h Models 30h and Greater Processors ..	17
2.1	Key Microarchitecture Features	18
2.2	Cache Line, Fetch and Data Type Widths	20
2.3	Instruction Decomposition	20
2.4	Superscalar Organization	21
2.5	Processor Block Diagram	22
2.6	Processor Cache Operation	22
2.7	Memory Address Translation	25
2.8	Optimizing Branching	26
2.9	Instruction Fetch and Decode	31
2.10	Integer Execution Unit	32
2.11	Floating-Point Unit	34
2.12	Load-Store Unit	39
2.13	Optimizing Writing Data	41
2.14	Simultaneous Multi-Threading	43
2.15	LOCKS	45
Appendix A	Understanding and Using Instruction Latency Tables	47
A.1	Instruction Latency Assumptions	47
A.2	Spreadsheet Column Descriptions	48
Index		51

List of Figures

Figure 1.	Cache Line Size, Fetch and Decode Widths in Bytes.....	20
Figure 2.	Data Pipe Widths in Bytes	20
Figure 3.	Data Type Widths in Bytes	20
Figure 4.	Block Diagram - AMD Family 17h Models 30h - 3Fh	22
Figure 5.	Integer Execution Unit Block Diagram	33
Figure 6.	Floating-Point Unit Block Diagram.....	35
Figure 7.	Load-Store Unit	40
Figure 8.	Resource Sharing	44





List of Tables

Table 1.	Typical Instruction Mappings	21
Table 2.	Write-Combining Completion Events.....	42

Revision History

Date	Rev.	Description
February 2020	3.01	2nd Public Release. Chapter 2: Added Models 30h and Greater to title. Section L3 Cache: Added Models 30h and Greater to section. Execution Units section: Updated first paragraph (see change bars). Write-combining Completion Events section: Updated table (see change bars). Appendix A: Added Models 30h and Greater to first paragraph (see change bars).
August 2019	3.00	Initial Public Release.

Chapter 1 February 2020 Introduction

This guide provides optimization information and recommendations for AMD Family 17h Models 30h and greater processors. In this guide the term “the processor” is used to refer to all processors within Family 17h Models 30h and greater.

This chapter covers the following topics:

Topic	Page
Intended Audience	15
Specialized Terminology	16

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).

For complete information on the AMD64 architecture and instruction set, see the multivolume *AMD64 Architecture Programmer's Manual* available from AMD.com. Individual volumes and their order numbers are provided below.

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

The following documents provide a useful set of guidelines for writing efficient code that have general applicability to the Family 17h processor:

- *AMD Family 15h Processors Software Optimization Guide* (Order # 47414)
- *Software Optimization Guide for AMD Family 10h and 12h Processors* (Order # 40546)

Refer to the *Processor Programming Reference (PPR) for AMD Family 17h Models 30h-3Fh Processors* (Order # 55803) for more information about machine-specific registers, debug, and performance profiling tools.

1.2 Specialized Terminology

The following specialized terminology is used in this document:

- Smashing** *Smashing* (also known as Page smashing) occurs when a processor produces a TLB entry whose page size is smaller than the page size specified by the page tables for that linear address. Such TLB entries are referred to as smashed TLB entries.
- For example, when the Family 17h processor encounters a larger page size in the guest page tables which is backed by a smaller page in the host page tables, it will smash translations of the larger page size into the smaller page size found in the host.
- Superforwarding** *Superforwarding* is the capability of a processor to send (forward) the results of a load instruction to a dependent floating-point instruction bypassing the need to write and then read a register in the FPU register file.

Chapter 2 Microarchitecture of AMD Family 17h Models 30h and Greater Processors

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

The *architecture* consists of the instruction set and those features of a processor that are visible to software programs running on the processor. The architecture determines what software the processor can run. The AMD64 architecture of the AMD Family 17h processor is compatible with the industry-standard x86 instruction set.

The term *microarchitecture* refers to the design features used to reach the target cost, performance, and functionality goals of the processor.

The *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 design of the execution core allows it to implement a small number of simple instructions which 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.

This chapter covers the following topics:

Topic	Page
Key Microarchitecture Features	18
Instruction Decomposition	21
Superscalar Organization	21
Processor Block Diagram	22
Processor Cache Operation	22
Memory Address Translation	25
Optimizing Branching	26
Instruction Fetch and Decode	31
Integer Execution Unit	32
Floating-Point Unit	34
Load-Store Unit	39

Topic	Page
Optimizing Writing Data	41
Simultaneous Multi-Threading	43
LOCKs	45

2.1 Key Microarchitecture Features

The processor implements a specific subset of the 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)
- Advanced Vector Extensions 2 (AVX2)
- Advanced Encryption Standard (AES) acceleration 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

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
- INVPCID

The processor adds support for the following new instructions:

- WBNOINVD
- RDPRU
- UMIP
- CLWB

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

- Simultaneous Multi-threading
- Unified 512-Kbyte L2 cache per core
- 4-Mbyte or 16-Mbyte shared, victim L3, depending on configuration
- Integrated memory controller
- 32-Kbyte L1 instruction cache (IC) per core
- 32-Kbyte L1 data cache (DC) per core
- 4K 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
- Four-way integer execution

- Three-way address generation (2 load/store, 1 store)
- Four-way 256-bit wide floating-point and packed integer execution
- Integer hardware divider
- Superforwarding
- L1 Instruction TLB and L1 Data TLB
- Six fully-symmetric core performance counters per thread

2.2 Cache Line, Fetch and Data Type Widths

The following figures diagram the cache line size and the widths of various data pipes and registers.

[illegible]

Figure 1. Cache Line Size, Fetch and Decode Widths in Bytes

[illegible]

Figure 2. Data Pipe Widths in Bytes

Data Types																
YMMWORD	32								32							
XMMWORD	16				16				16				16			
QWORD	8		8		8		8		8		8		8		8	
DWORD	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	

Figure 3. Data Type Widths in Bytes

2.3 Instruction Decomposition

The processor implements the 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 fastpath single (one macro-op), fastpath double (two macro ops), or microcode (greater than two (2) macro ops). Macro ops can normally contain up to two (2) 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 1. Typical Instruction Mappings

Instruction	Macro ops	Micro ops	Comments
MOV reg,[mem]	1	1: load	Fastpath single
MOV [mem],reg	1	1: store	Fastpath single
MOV [mem],imm	1	2: move-imm, 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	Fastpath single
VMOVAPD [mem],ymm	1	2: store, FP-store-data 256b AVX	Fastpath single
ADDPD xmm,xmm	1	1: addpd	Fastpath single
ADDPD xmm,[mem]	1	2: load, addpd	Fastpath single
VADDPD ymm,ymm	1	1: addpd 256b AVX	Fastpath single
VADDPD ymm,[mem]	1	2: load, addpd256b AVX	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 decoupled independent schedulers, consisting of four integer ALU schedulers, a unified AGU scheduler servicing the three AGU pipelines, and a unified floating-point scheduler servicing the four FP pipelines. These schedulers can simultaneously issue up to eleven micro ops to the four integer ALU pipes, three AGU pipes, and the four FPU pipes.

2.5 Processor Block Diagram

A block diagram of the processor is shown in Figure 4 below.

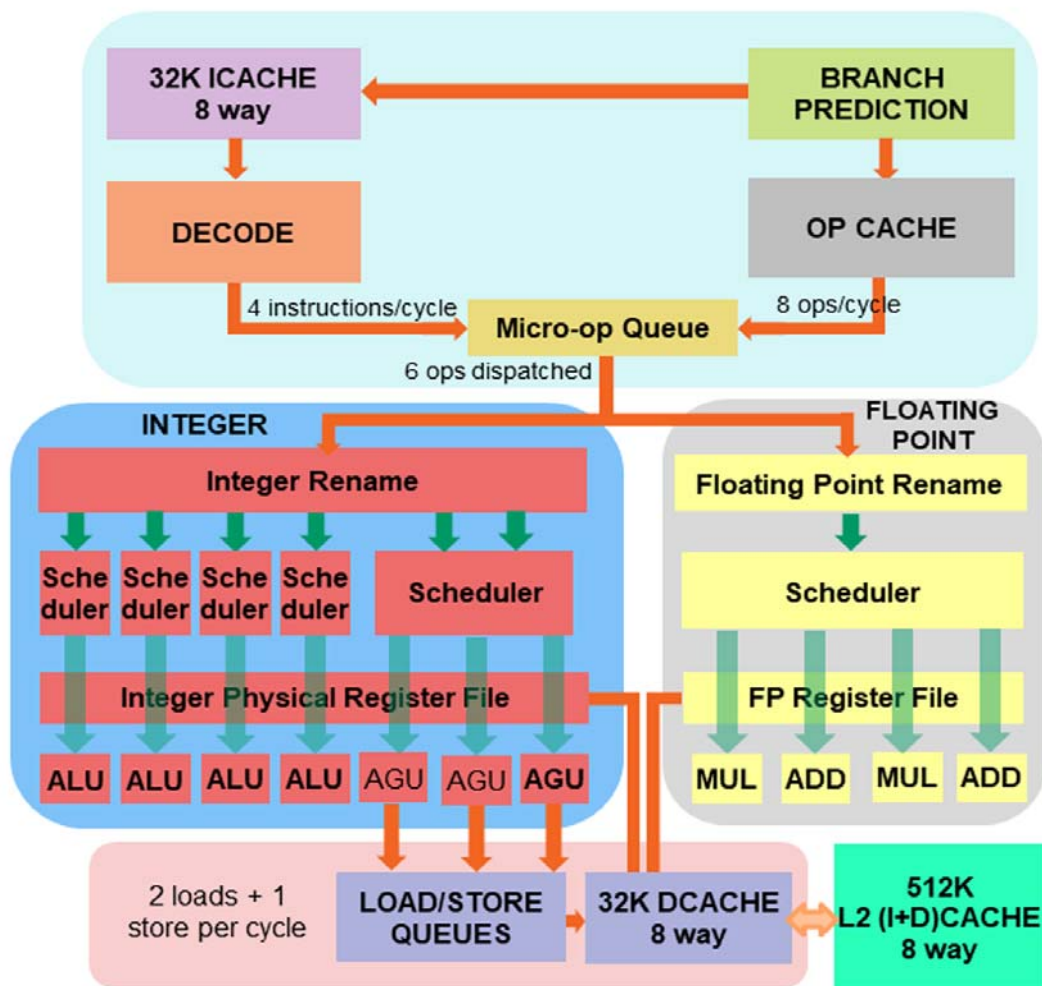


Figure 4. Block Diagram - AMD Family 17h Models 30h - 3Fh

2.6 Processor Cache Operation

The AMD Family 17h processor 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
- 4-Mbyte or 16-Mbyte L3 cache (depending on configuration)

2.6.1 L1 Instruction Cache

The processor contains a 32-Kbyte, 8-way set associative L1 instruction cache. Cache line size is 64 bytes; 32 bytes are fetched in a cycle. 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 block that includes the miss address and up to thirteen additional blocks. These blocks are prefetched from addresses generated by the Branch Predict unit. Because code typically exhibits spatial locality, prefetching is an effective technique for avoiding decode stalls. Cache-line replacement is based on a least recently-used replacement algorithm. The L1 instruction cache is protected from error through the use of parity.

2.6.2 L1 Data Cache

The processor contains a 32-Kbyte, 8-way set associative L1 data cache. This is a write-back cache that supports two 256-bit loads and one 256-bit store per cycle. In addition, the L1 cache is protected from bit errors through the use of 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", for more information on load-to-use latency.

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. Stores have two different alignment boundaries. The alignment boundary for accessing TLB and tags is 64 bytes, and the alignment boundary for writing data to the cache or memory system is 32 bytes. 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).

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

Natural alignment for 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 is a banked structure. Two loads per cycle can access the DC if they are to different banks. The DC banks that are accessed by a load are determined by address bits 5:3, the size of the

load, and the DC way. DC way is determined using the linear-address-based utag/way-predictor (see section below). A bank conflict will result in a reflow of one of the loads, which will appear as a longer-latency load.

2.6.2.2 Linear address utag/way-predictor

The L1 data cache tags contain a linear-address-based microtag (utag) that tags each cacheline with the linear address that was used to access the cacheline 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 in which way the cacheline is located prior to a read of the cache data. This allows a load to read just a single cache way, instead of all 8. This saves power and reduces 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 cachelines. 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 cacheline.

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 (11:6), only one cacheline with a given linear hash is accessible at any time; any cachelines with matching linear hashes are marked invalid in the utag and are not accessible.

2.6.3 L2 Cache

The processor implements a unified 8-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 512 Kbytes with a variable load-to-use latency of no less than 12 cycles. The L2 to L1 data path is 32 bytes wide.

2.6.4 L3 Cache

The AMD Family 17h Models 30h or Greater processors implements a 4 MB or 16-MB L3 cache (depending on SOC configuration) that is 16-way set associative and shared by four cores inside a CPU complex. The L3 is a write-back cache populated by L2 victims. When there is an L3 hit, the line is invalidated from the L3 if the access was a store. It is invalidated from the L3 if the access was a load and the line was read by just one core. It stays valid in the L3 if it was a code fetch. It stays valid in the L3 if it was a load and the line has been read by more than one core. The L3 maintains shadow tags for each L2 cache in the complex. If a core misses in its local L2 and also 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 39 cycles. The non-temporal cache fill hint, indicated with PREFETCHNTA, reduces cache pollution for data that will only be used once. It is not suitable for cache blocking of small data sets. 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 processor 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 64 entries that hold 4-Kbyte, 2-Mbyte, or 1-Gbyte page entries.

2.7.2 L2 Translation Lookaside Buffers

The processor provides an 8-way set associative L2 instruction TLB with 512 entries capable of holding 4-Kbyte pages, and 2-Mbyte pages. 1-Gbyte pages are not held in the L2 instruction TLB; they are smashed into 2-Mbyte pages in the L2 ITLB.

The L2 data TLB provides a unified 16-way set-associative L2 data TLB with 2048 entries capable of holding 4-Kbyte pages, 2-Mbyte pages, and page-directory entries (PDEs) used to speed up table walks, and 1-Gbyte pages smashed into 2-Mbyte pages. When a 1-Gbyte smashed page held in the L2 data TLB is reloaded into the L1 data TLB, it is installed as a 1-Gbyte page in the L1 data TLB.

2.7.3 Hardware Page Table Walkers

The processor has two hardware page table walkers to handle L2 TLB misses. Misses can start speculatively from either the instruction or the data side. As was described in section 2.6.2, the L2 data TLB holds PDEs, which are used to speed up tablewalks by skipping three levels of page table reads. In addition to the PDE storage in the L2 data TLB, the table walker includes a 64-entry Page Directory Cache (PDC) which holds page-map-level-4 entries (PML4Es) and page-directory-pointer entries (PDPEs) to speed up table walks. The PDC entries and the PDE entries in the L2 data TLB 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.

In the L1TLBs, INVLPG and INVLPGA instructions cause a flush of all smashed entries corresponding to the same 1-Gbyte guest linear address page. In the L2TLBs, INVLPG and INVLPGA cause a flush of all smashed entries.

See the definition of the terms smashing and smashed in the Section 1.2 on page 16.

2.8 Optimizing Branching

Branching can reduce throughput when instruction execution must wait on the completion of the instructions prior to the branch that determine whether the branch is taken. The processor integrates logic that is designed to reduce the average cost of conditional branching by attempting to predict the outcome of a branch decision prior to the resolution of the condition upon which the decision is based.

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 or not the instructions are being fed from the op cache. The common case penalty is 16 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 64-byte fetch block. This calculation is performed every cycle to support the 64 byte per cycle fetch bandwidth of the op cache. 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 the next instruction to be executed following a branch are detailed in the following sections.

2.8.1.2 Branch Target Buffer

The branch target buffer (BTB) is a three-level structure accessed using the fetch address of the current fetch block. Each BTB entry includes information for branches and their targets. Each BTB entry can hold up to two branches if the branches reside in the same 64-byte aligned cache line and the first branch is a conditional branch.

Each level of BTB holds an increasing number of entries, and prediction from the larger BTBs have higher latencies. When possible, keep the critical working set of branches in the code as small as possible (see *Software Optimization Guide for AMD Family 15h*, Section 7.6). L0BTB holds 8 forward taken branches and 8 backward taken branches, and predicts with zero bubbles. L1BTB has 512 entries and creates one bubble if prediction differs from L0BTB. L2BTB has 7168 entries and creates four bubbles if its prediction differs from L1BTB.

2.8.1.3 Return Address Stack

The processor implements a 32-entry return address stack (RAS) to predict return addresses from a near call. One of the entries is unusable for pointer logic simplification. In dual-threaded mode, each thread is allocated 15 entries. 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 the execution hardware restores it to a consistent state.

The following sections discuss some 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.

To get the value in the RIP register into a general-purpose register in 64-bit software, you can use 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 processor Families 15h, 16h, and 17h 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 rationale for the former recommendation, see section 6.2 in the *Software Optimization Guide for AMD Family 10h and 12h Processors*.

2.8.1.3.3 Function Inlining

Calls and returns are not eligible to be stored in the zero-bubble predictor (L0 BTB). Therefore, function calls within hot loops can 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 1024-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 at L2 BTB correction latency.

Branches that have so far always had the same target are predicted using the static target from the branch's BTB entry. This means the prediction latency for correctly predicted indirect branches is roughly $5-(3/N)$, where N is the number of different targets of the indirect branch. For these reasons, 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 fall-through behavior will use the conditional predictor. The conditional branch predictor uses a global history scheme that keeps track of the previously executed branches. Global history is not updated for not-taken branches. For this reason, dynamic branches which are biased towards not-taken are preferred. Branch behavior which depends on deep history or which does not correlate well with global history will be mispredicted often.

When possible, avoid branches which alternate between taken and not-taken. If a loop is executed twice and it is a small loop, it may be beneficial to unroll it.

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 64-entry (32 entries in SMT mode) FIFO from fetch until retirement. Each entry holds branch and cacheline information for up to a full 64-byte cacheline. If a single BTB entry is not sufficient to allow prediction to the end of the cache line, additional entries are used. If no branches are identified in a cacheline, the fetch window tracking structure will use a single entry to track the entire cacheline.

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.

2.8.2 Boundary Crossing Branches

Branches whose target crosses a half-megabyte aligned boundary are unable to be installed in the L0 BTB or to share BTB entries with other branches. Excessive occurrences of this scenario can reduce effective BTB capacity 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. Since the processor can read an aligned 64-byte fetch block every cycle, aligning the end of the loop to the last byte of a 64-byte cache line is the best thing to do, if possible.

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 per cache line entry point at two or below. Since BTB entries can hold up to two branches, predicting a third branch will require 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.

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

Length	Encoding
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 the 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 Family 16h and 17h processors, 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).

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 the 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.

As a slightly more efficient alternative to inserting NOPs for padding, redundant prefixes can be used to pad existing instructions without affecting 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 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 naturally aligned blocks. The processor can perform an instruction block fetch every cycle.

The fetch unit sends these bytes to the decode unit through a 20 entry Instruction Byte Queue (IBQ), each entry holding 16 instruction bytes. In SMT mode each thread has 10 dedicated IBQ entries. The IBQ acts as a decoupling queue between the fetch/branch-predict unit and the decode unit.

The decode unit scans two of these windows in a given cycle, decoding a maximum of four instructions. The decode unit also contains a sideband stack optimizer, which tracks the stack-pointer value. This optimization removes the dependencies that arise during chains of PUSH and POP operations on the rSP register, and thereby improves the efficiency of the PUSH and POP instructions.

The pick window is 32 bytes, aligned on a 16-byte boundary. Having 16 byte aligned branch targets gets maximum picker throughput and avoids end-of-cacheline short op cache (OC) entries.

Only the first pick slot (of 4) can pick instructions greater than eight bytes in length. Avoid having more than one instruction in a sequence of four that is greater than eight bytes in length.

2.9.1 Op Cache

The op cache (OC) is a cache of previously decoded instructions. When instructions are being served 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 8 instructions per cycle whereas the maximum throughput from the traditional fetch and decode pipeline is 4 instructions per cycle. 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 8 ways. At each set-way intersection is an entry containing up to 8 instructions, so the maximum capacity of the op cache is then 4K ops. The actual limit may be less due to efficiency considerations. Avoid hot code regions that approach this size for a single thread or half this size for two SMT threads.

When instructions are decoded, they are also built into the op cache. Multiple instructions are built together into an op cache "entry". Up to 8 sequential instructions ending in the same 64-byte aligned memory region may be cached together in an entry.

Op cache entry limits:

- 8 instructions
- 8 32-bit immediates/displacements (64-bit immediates/displacements take two slots)
- 4 microcode instructions

The op cache is modal and the machine can only transition between instruction cache mode (IC mode) and op cache mode (OC mode) at certain points. The machine can only transition from IC mode to OC mode at a branch target. Once in OC mode, the machine will generally remain in this mode until there is a fetch address for which there is no corresponding OC entry (a miss).

If there are an excess of mode transitions, IPC can be negatively impacted. Limiting hot regions of code to fit in the capacity of the op cache will minimize the possibility of mode transitions, and it is particularly important when unrolling loops to avoid exceeding the capacity of the op cache.

An OC entry terminates at the end of a 64-byte aligned memory region, so branching to one of the last few instructions in a region will result in an inefficient OC entry being built (that is less than the maximum 8 instructions that could be stored in that entry).

Use of the OC requires a flat memory model (64-bit or 32-bit with CS base of 0 and CS limit at max).

2.10 Integer Execution Unit

Figure 5 on page 33 diagrams the integer execution unit.

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

- schedulers
- execution units
- retire control

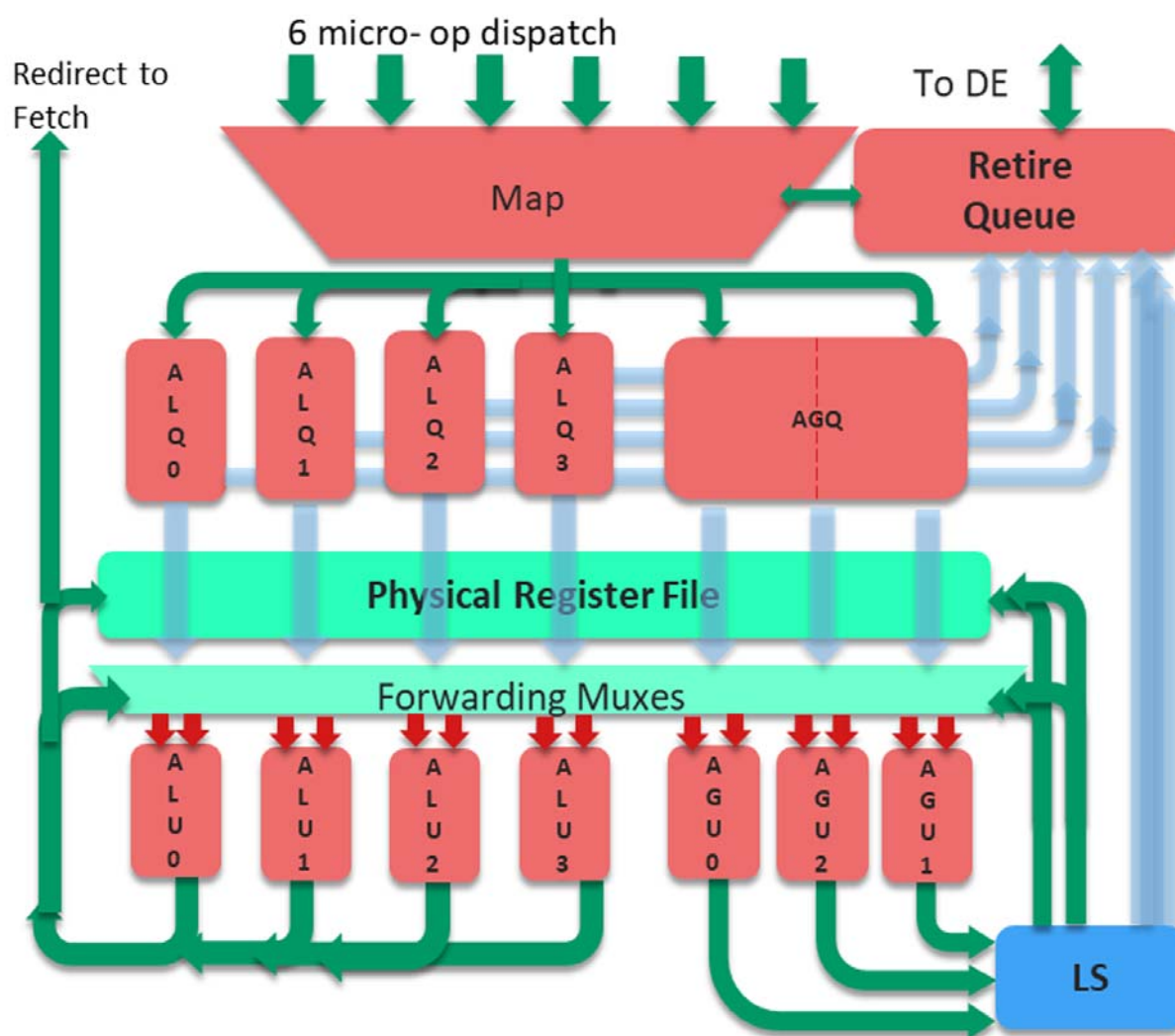


Figure 5. Integer Execution Unit Block Diagram

2.10.1 Schedulers

The schedulers can receive up to six macro ops per cycle, where they are broken down into micro ops. ALU micro ops are sent to one of four 16-entry ALU schedulers. Load and Store micro ops are sent to one 28-entry address generation unit (AGU) scheduler. Each scheduler can issue one micro op per cycle. The scheduler tracks operand availability and dependency information as part of its task of

issuing micro ops to be executed. It also ensures that older micro ops which have been waiting for operands are executed in a timely manner. Micro ops can be issued and executed out-of-order.

2.10.2 Execution Units

The processor contains 4 integer execution pipes. There are four ALUs capable of all integer operations with the exception of multiplies, divides, and CRC which are dedicated to one ALU each. There are 3 AGUs for address generation. Two of them (AGU0/AGU1) can generate load addresses, while all three (AGU0/AGU1/AGU2) can generate store addresses.

While two-operand LEA instructions are mapped as a single-cycle micro-op in the ALUs, three-operand LEA instructions are mapped to an AGU and have 2 cycle latency, with results inserted back in to either the ALU2 or ALU3 pipeline.

The integer multiply unit can handle multiplies of up to 64 bits \times 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 with a reduction in throughput to one every two cycles.

The radix-4 hardware integer divider unit can compute 2 bits of results per cycle.

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 6 macro ops dispatched per cycle and track up to 224 macro ops in-flight. 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 macro ops per cycle.

The retire control unit also manages internal integer register mapping and renaming. The integer physical register file (PRF) consists of 180 registers, with up to 38 per thread mapped to architectural state or microarchitectural temporary state. The remaining registers are available for out-of-order renames.

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 bit and 256 bit packed integer, single and double precision vector floating-point data types. The floating-point load and store paths are 256 bits wide.

The floating-point unit (FPU) utilizes a coprocessor model for all operations that use X87, MMX™, XMM, YMM, 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 and renaming of

4 floating point micro ops per cycle and the scheduler can issue 1 micro op per cycle for each pipe. The floating-point scheduler has a 36 entry micro-op capacity. The floating-point unit shares the retire queue with the integer unit. The retire queue can hold up to 224 micro ops or 112 per thread in SMT mode. Micro ops can be dispatched to the Execution unit even if floating-point scheduler is full to allow loads and stores to be accelerated.

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 4 execution pipes which can execute an operation every cycle. The FP unit receives 2 loads from the load/store unit every cycle that are up to 256b each. There are dedicated busses to enable fast moves between the floating point registers and the general registers in the EX unit. Stores are serviced out of execution pipe 2.

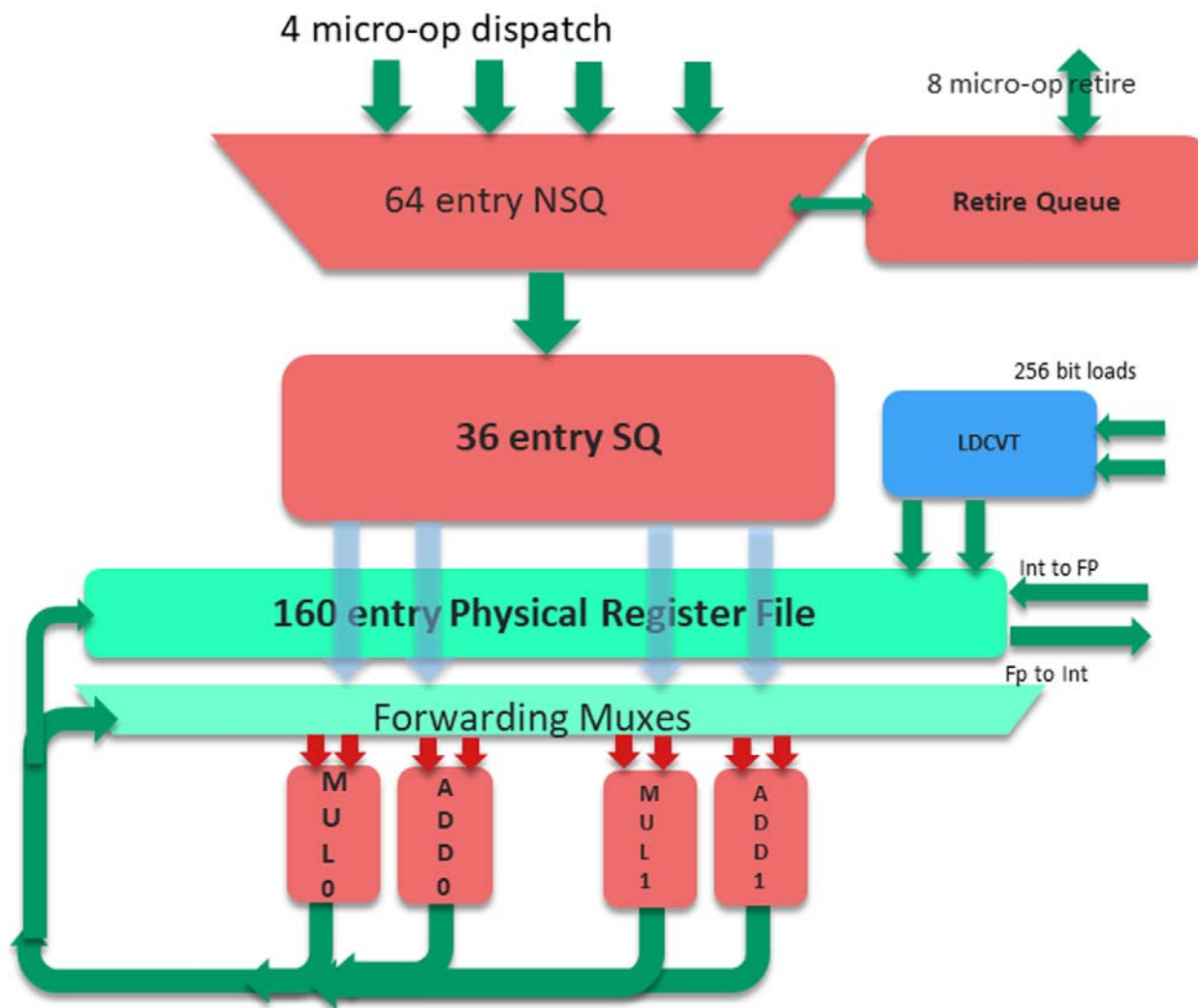


Figure 6. Floating-Point Unit Block Diagram

Pipes 0 and 1 support operations that require three operands. When three operands are required for an operation, it uses ones of the source busses normally allocated to pipe3, which can block any execution in pipe3. If data for pipe3 or the 3rd operand can be bypassed from a result generated that same cycle, then pipe3 can execute an operation even when either pipe0 or pipe1 require a third source.

2.11.1 Floating Point Execution Resources

Unit	Pipe				Domain ²	Ops Supported
	0	1	2	3		
FMUL	X	X			F	(v)FMUL*, (v)FMA*, Floating Point Compares, Blendv(DQ)
FADD			X	X	F	(v)FADD*
FCVT				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		S	Stores and Move to General Register (EX) Operations
VADD ²	X	X		X	I	Integer Adds, Subtracts, and Compares
VMUL	X				I	Integer Multiplies, SAD, Blendvb
VSHUF ³		X	X		I	Data Shuffles, Packs, Unpacks, Permute
VSHIFT			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		S				*CLM*

Notes:

1. FDIV unit can support 2 simultaneous operations in flight even though it occupies a single pipe.
2. Some complex VADD operations are not available in all pipes.
3. Some complex shuffle operations are only available in pipe1.
4. There is 1 cycle of added latency for a result to cross from F to I or I to F domain.

2.11.2 Code recommendations

1. Use the SIMD nature of the SSE or AVX instruction sets to achieve significantly higher throughput. The processor supports SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, SSE4a, F16C, FMA, AVX, and AVX2. The datapath is 256 bits across all operations, so optimal code will operate on 256b (YMM registers) with every operation using the SIMD instructions.
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 one has no choice but to use multiple loads, try to make sure they are back to back in the code.
3. Clear floating point registers 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 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 so on when consuming the output of `VMULPS`.

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`. Note that 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
- CVTSS2SS (32-/64-BIT)
- MOVSS xmm1, xmm2
- CVTSD2SS
- CVTSS2SD
- MOVLPS xmm1, [mem]
- CVTSS2SD (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 significant penalty for mixing SSE and AVX instructions when the upper 128 bits of the YMM registers contain non-zero data. Transitioning in either direction will cause a micro-fault to

spill or fill the upper 128 bits of all 16 YMM registers. There will be an approximately 100 cycle penalty to signal and handle this fault. To avoid this penalty, a VZEROUPPER or VZEROALL instruction should be used to clear the upper 128 bits of all YMM registers when transitioning from AVX code to SSE or unknown code.

2.12 Load-Store Unit

The processor load-store (LS) unit handles data accesses. The LS unit contains three largely independent pipelines enabling the execution of two 256-bit load memory operations and one 256-bit store memory operation per cycle.

The LS unit includes a 44-entry 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 unit utilizes a 48-entry store queue which holds stores from dispatch until the store data can be written to the data cache.

The LS unit dynamically reorders operations, supporting both load operations 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 address bits 11:0 to determine STLF eligibility. Avoid having multiple stores with the same 11:0 address bits, but to different addresses (different 47:12 bits) in-flight simultaneously where a load may need STLF from one of them. Loads that follow stores to similar address space should use the same registers and accesses should be grouped closely together, avoiding intervening modifications or writes to the base or index register used by the store and load when possible. Also, minimize displacement values such that the range will fit within 8 bits when possible.

The LS unit can track up to 22 outstanding in-flight cache misses.

The AGU and LS pipelines are optimized for simple address generation modes. Base+displacement, base+index, 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 where both an index and displacement are present (most commonly 3-source 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 (scaled-index, or index+displacement) 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.

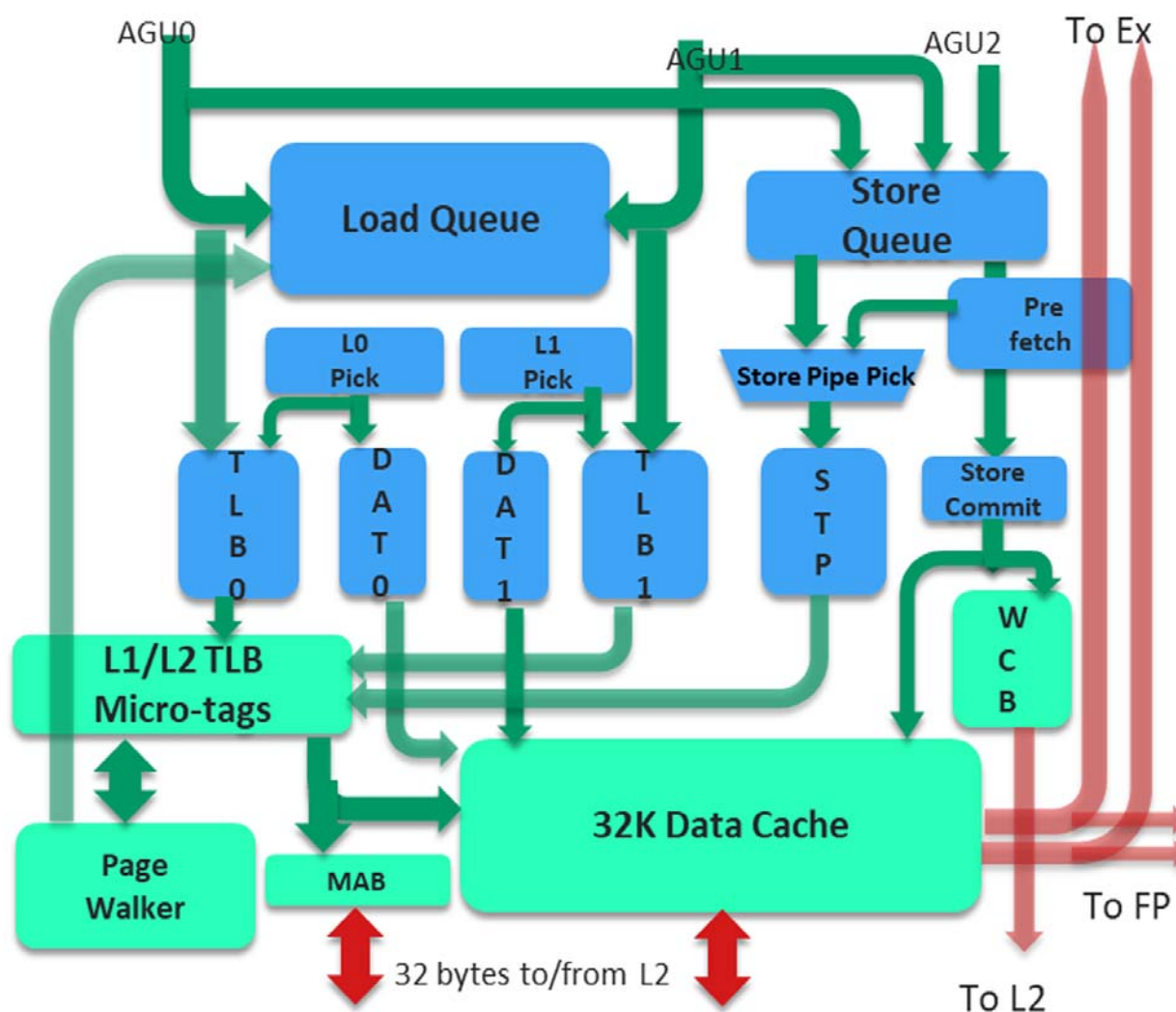


Figure 7. Load-Store Unit

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 Family 17h processor supports 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 Write-Combining Definitions and Abbreviations

This appendix uses the following definitions and abbreviations:

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

2.13.2 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 17h Models 30h-3Fh Processors*, order# 55803

2.13.3 Write-Combining Operations

To improve system performance, AMD Family 17h processor aggressively combines multiple memory-write cycles of any data size that address locations within a 64-byte write buffer that is aligned to a cache-line boundary. The processor continues to combine writes to this buffer without writing the data to the system, as long as certain rules apply (see Table 2 for more information). The data sizes can be bytes, words, doublewords, or quadwords.

- 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 MTTR 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 the 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 2 below) 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 2. 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 they are not full, as described in “Sending Write-Buffer Data to the System” on page 43.

Table 2. Write-Combining Completion Events

Event	Comment
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.

Table 2. Write-Combining Completion Events (Continued)

Event	Comment
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 2.15 on page 45 for more info on locks and memory barriers.

2.13.4 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 8 different 64B cache lines (up to 7 from one thread). Throughput will be best when the number of simultaneous write-combining streams is low.

2.14 Simultaneous Multi-Threading

In order 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 Figure 8 below).

Resource	Competitively Shared	Watermarked	Statically Partitioned
ICACHE	x		
ITLB	x		
Micro-op Cache	x		
Dispatch Interface	x		
Microcode ROM	x		
DCACHE	x		
DTLB	x		
L2 Cache	x		
L3 Cache	x		
Scheduler tokens	x		
Physical Registers	x		
Load Queue	x		
Floating Point Physical Registers	x		
Floating Point Scheduler		x	
Memory Request Buffers		x	
Micro-op Queue			x
Store Queue			x
Retire Queue			x

Figure 8. Resource Sharing

For partitioned resources, arbitration between threads is generally round-robin unless a given thread is stalled.

It is expensive to transition between single-threaded (1T) mode and dual-threaded (2T) mode and vice versa, so software should restrict the number of transitions. If running in 2T mode, and one thread finishes execution, it may be beneficial to avoid transitioning to 1T mode if the second thread is also about to finish execution.

If the two threads are running different code, they should run in different linear pages to reduce BTB collisions.

Two threads which concurrently run the same 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. This is to facilitate BTB sharing between threads.

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 16-byte aligned boundaries.
- Following a LOCKed instruction, refrain from using floating point instructions as long as possible.
- Ensure that the Last Branch Record is disabled (DBG_CTL_MSR.LBR)

Appendix A Understanding and Using Instruction Latency Tables

The companion file **Family 17h Models 30h and Greater Instruction Latencies version_1.00.xlsx** distributed with this Software Optimization Guide provides additional detailed information for the AMD Family 17h Models 30h and Greater processors. 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. Throughput refers to the number of results that can be generated in a unit of time given the repeated execution of a given instruction.

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 due to the fact that 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 listed 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.

The 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 only forms listed are the ones without memory operands. The latency for instruction forms that load from memory can be calculated by adding the load latencies listed on the

overview worksheet to the latency for the register-only form. To measure the latency of an instruction which stores data to memory, it is necessary to define an end-point at which the instruction is said to be complete. This guide has chosen instruction retirement 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 taking into account 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:

Cols	Label	Description
A	Instruction	• Instruction mnemonic
B–E	Instruction operands	<p>The following notations are used in these columns:</p> <ul style="list-style-type: none"> • imm—an immediate operand (value range left unspecified) • imm8—an 8-bit immediate operand • m—an 8, 16, 32 or 64-bit memory operand (128 and 256 bit memory operands are always explicitly specified as m128 or m256) • mm—any 64-bit MMX register • mN—an N-bit memory operand • r—any general purpose (integer) register • N—an N-bit general purpose register • xmmN—any xmm register, the N distinguishes among multiple operands of the same type • ymmN—any ymm register, the N distinguishes among multiple operands of the same type <p>A slash denotes an alternative, for example m64/m32 is a 32-bit or 64-bit memory operand. The notation "<xmm0>" denotes that the register xmm0 is an implicit operand of the instruction.</p>
F	APM Vol	AMD64 Programmer's Manual Volume that describes the instruction.
G	Cpuid flag	CPUID feature flag for the instruction.
H	Macro Ops	<p>Number of macro ops for the instruction.</p> <p>Any number greater than 2 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.</p>

Cols	Label	Description
I	Unit	<p>Execution units. The following abbreviations are used:</p> <ul style="list-style-type: none"> • ucode—instruction is implemented using a variable number of macro ops. • ucode(<i>n</i>)—instruction is implemented using exactly <i>n</i> macro ops. • ALU—instruction can execute in any of the 4 ALU pipes. • ALU_{<i>n</i>}—instruction can only execute in ALU pipe <i>n</i>. • FPU—instruction can execute in any of the 4 FPU pipes. • FP_{<i>n</i>}—instruction can only execute in FP pipe <i>n</i>. • FP_{<i>n+m</i>}—instruction requires both FP pipes <i>n</i> and <i>m</i>. • FP_{<i>n</i>}/FP_{<i>m</i>}—instruction can execute in either FP pipe <i>n</i> or <i>m</i>. • FP_{<i>n</i>},FP_{<i>m</i>}—instruction execution uses FP pipe <i>n</i> followed by FP pipe <i>m</i>. • DIV—Integer divide functional element within the integer unit • MUL—Integer multiply functional element within the integer unit. • ST—instruction utilizes the LD/ST unit to execute a store. • LD—instruction utilizes the LD/ST unit to execute a load. • LD/ST—Load/Store unit. • (dash)—instruction does not utilize an execution pipe. • NA—instruction is not supported.
J	Latency	<p>Instruction latency in processor cycles.</p> <p>Refer to the section "Instruction Latency Assumptions" above for more information about this column.</p>
K	Throughput	<p>Throughput of the instruction.</p> <p>A value of 2 indicates that two such instructions can be retired in the same clock cycle. This value is subject to the same assumptions as the latency values.</p> <p>Refer to the section "Instruction Latency Assumptions" above for more information.</p>
L	Notes	Additional information about the entry.

Index

A

address translation 25

B

branch

 optimization 26

 prediction 26

C

cache

 L1 data 23

 L1 instruction 23

 L2 24

 L3 24

D

data and pipe widths 20

E

execution unit

 floating-point 34

 integer 32

F

floating-point

 block diagram 35

 denormals 37

 unit 34

 x87 code 37

H

hardware page table walker 25

I

instruction

 fetch and decoding 31

 integer execution 32

L

linear address tagging 24

load/store unit 39

M

memory address translation 25

microarchitecture features 18

O

optimizing writing 41

P

processor, microarchitecture 18

S

smashing, definition 16

superforwarding, definition 16

superscalar processor 21

T

terminology, specialized 16

TLB

 L1 25

 L2 25

W

write combining 43