Software Optimization Guide for AMD Family 16h Processors

Publication # **52128** Revision: **1.1**
Issue Date: **March 2013**

Advanced Micro Devices

Trademarks

Disclaimer

Contents

List of Figures

List of Tables

Revision History

Date	Rev.	Description
March 2013	1.1	Initial Public Release

1 Preface

About this Document

This document provides software optimization information and recommendations for programming the AMD Family 16h processor.

Audience

This guide 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).

References

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:

Title	Publication 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 useful set of guidelines for writing efficient code that have general applicability to Family 16h processors:

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

Refer to *BIOS and Kernel Developers Guide (BKDG) for AMD Family 16h Models 00h-0Fh Processors* (Order # 48751) for more information about machine-specific registers, debug, and performance profiling tools.

Notational Convention

Instruction mnemonics, micro-instructions, and example code are set in `mono-spaced font`.

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 16h processor encounters a 1-Gbyte page size, it will smash translations of that 1-Gbyte region into 2-Mbyte TLB entries, each of which translates a 2-Mbyte region of the 1-Gbyte page.

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.

2 Microarchitecture of the Family 16h Processor

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 16h processor is compatible with the industry-standard x86 instruction set.

The term microarchitecture refers to the design features used to reach the 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 AMD Family 16h 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:

2.1 Features

This topic introduces some of the key features of the AMD Family 16h Processor.

The AMD Family 16h processor implements a specific subset of the AMD64 instruction set architecture.

Instruction set architecture support includes:

- 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. The following instruction 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)
  - Half-precision floating-point conversion (F16C)
- Carry-less Multiply (CLMUL) instructions
- Advanced Encryption Standard (AES) acceleration instructions
- Bit Manipulation Instructions (BMI)
- Move Big-Endian instruction (`MOVBE`)
- `XSAVE` / `XSAVEOPT`
- `LZCNT` / `POPCNT`
- AMD Virtualization™ technology (AMD-V™)

The AMD Family 16h processor does not support the following instruction subsets:

- Fused Multiply/Add instructions (FMA3 / FMA4)
- XOP instructions
- Trailing bit manipulation (TBM) instructions
- Light-weight profiling (LWP) instructions
- Read and write fsbase and gsbase instructions
- `RDRAND`, and `INVPCID` instructions

The AMD Family 16h processor includes many features designed to improve software performance. The microarchitecture provides the following key features:

- Unified 1–2-Mbyte L2 cache shared by up to 4 cores
- Integrated memory controller with memory prefetcher
- 32-Kbyte L1 instruction cache per core
- 32-Kbyte L1 data cache per core
- Prefetchers for L2 cache, L1 data cache, and L1 instruction cache
- Advanced dynamic branch prediction
- 32-byte instruction fetch
- 2-way x86 instruction decoding with sideband stack optimizer
- Dynamic out-of-order scheduling and speculative execution
- Two-way integer execution
- Two-way address generation (1 load and 1 store)
- Two-way 128-bit wide floating-point and packed integer execution
- Integer hardware divider
- Superforwarding
- L1 Instruction TLB of 32 4-Kbyte entries and L1 Data TLB of 40 4-Kbyte entries
- Four fully-symmetric core performance counters

2.2 Instruction Decomposition

The AMD Family 16h 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 2 macro-ops). Macro-ops can normally contain up to 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`	2	4: 2 × {`store`, `FP-store-data`}	256b AVX Fastpath double
`ADDPD xmm,xmm`	1	1: `addpd`	Fastpath single
`ADDPD xmm,[mem]`	1	2: `load`, `addpd`	Fastpath single
`VADDPD ymm,ymm`	2	2: 2 × {`addpd`}	256b AVX Fastpath double
`VADDPD ymm,[mem]`	2	4: 2 × {`load`, `addpd`}	256b AVX Fastpath double

2.3 Superscalar Organization

The AMD Family 16h processor is an out-of-order, two-way superscalar AMD64 processor. It can fetch, decode, and retire up to two AMD64 instructions per cycle. The processor uses decoupled execution units to process instructions through fetch/branch-predict, decode, schedule/execute, and retirement pipelines.

The processor can fetch 32 bytes per cycle and can scan two 16-byte instruction windows for up to two instruction decodes per cycle. The decoder marks each instruction as fastpath single, fastpath double, or microcode. The dispatcher can send up to two macro-ops to the retire unit for tracking, as well as sending the corresponding micro-ops to the schedulers. These are upper limits, however. The actual number of bytes fetched or scanned, instructions decoded, or macro-ops dispatched may be lower, depending on a number of factors such as whether instructions can be broken up into 16-byte windows.

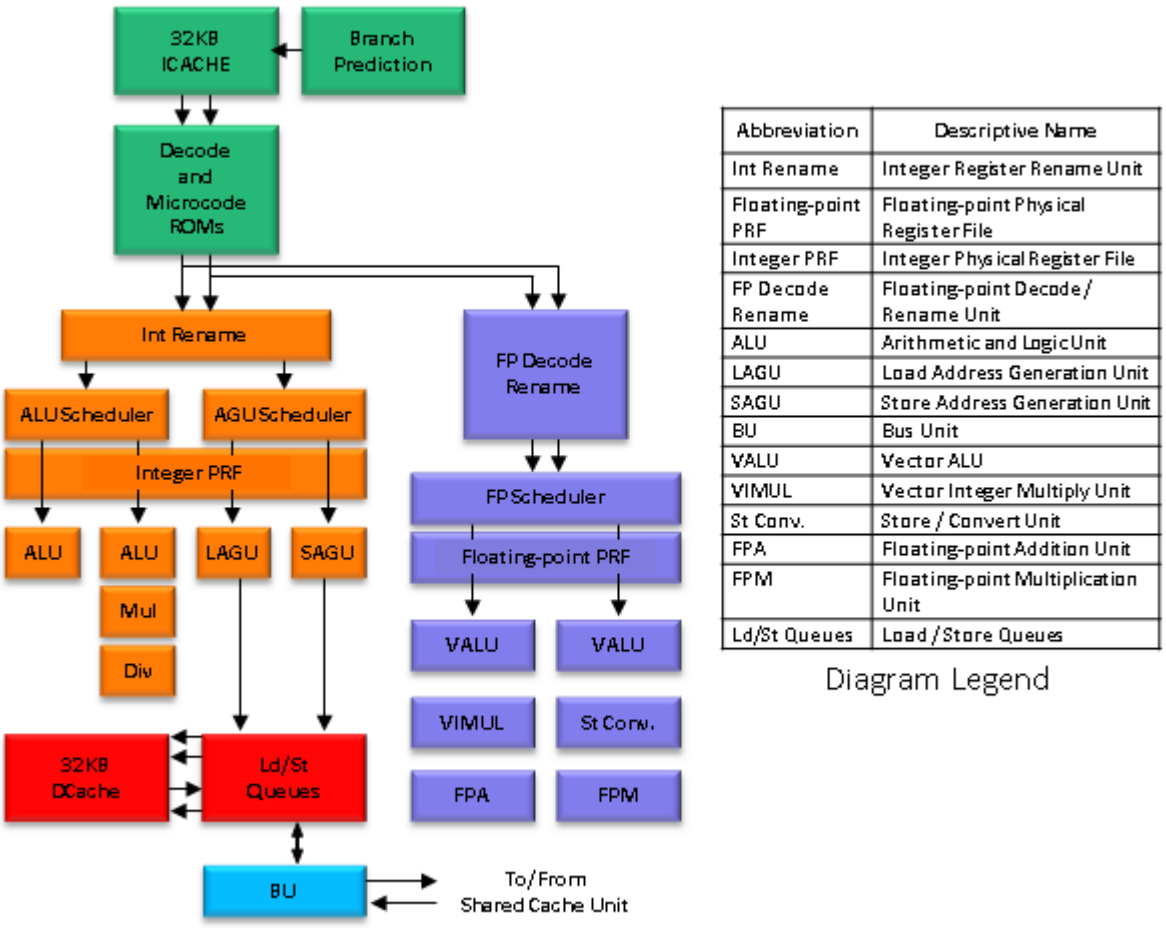
The processor uses decoupled independent schedulers, consisting of an integer ALU scheduler, an AGU scheduler, and a floating-point scheduler. These three schedulers can simultaneously issue up to six micro-ops to

the two integer ALU pipes, the load address generation pipe, the store address generation pipe, and the two FPU pipes.

A macro-op is eligible to be committed by the retire unit when all corresponding micro-ops have finished execution. The retire unit handles in-order commit of up to two macro-ops per cycle.

2.4 Processor Block Diagram

A block diagram of the AMD Family 16h processor is shown below.



Abbreviation	Descriptive Name
Int Rename	Integer Register Rename Unit
Floating-point PRF	Floating-point Physical Register File
Integer PRF	Integer Physical Register File
FP Decode Rename	Floating-point Decode / Rename Unit
ALU	Arithmetic and Logic Unit
LAGU	Load Address Generation Unit
SAGU	Store Address Generation Unit
BU	Bus Unit
VALU	Vector ALU
VIMUL	Vector Integer Multiply Unit
St Conv.	Store / Convert Unit
FPA	Floating-point Addition Unit
FPM	Floating-point Multiplication Unit
Ld/St Queues	Load / Store Queues

Figure 1. Family 16h Processor Block Diagram

2.5 Processor Cache Operations

AMD Family 16h processors use three different caches to accelerate instruction execution and data processing:

- Dedicated L1 instruction cache
- Dedicated L1 data cache
- Unified L2 cache shared by up to four cores

2.5.1 L1 Instruction Cache

The AMD Family 16h processor contains a 32-Kbyte, 2-way set associative L1 instruction cache. Cache line size is 64 bytes; however, only 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, prefetching instructions, and predicting branches. Requests that miss in the L1 instruction cache are fetched from the L2 cache or, if not resident in the L2 cache, 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 one or two sequential blocks (prefetches). 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.

Due to the indexing and tagging scheme used in the instruction cache, optimal performance is obtained when two hot cache lines which need to be resident in the instruction cache simultaneously do not share the same virtual address bits [20:6].

2.5.2 L1 Data Cache

The AMD Family 16h processor contains a 32-Kbyte, 8-way set associative L1 data cache. This is a write-back cache that supports one 128-bit load and one 128-bit store per cycle. In addition, the L1 cache is protected from bit errors through the use of parity. There is a hardware prefetcher that brings data into the L1 data cache to avoid misses. The L1 data cache has a 3-cycle integer load-to-use latency, and a 5-cycle FPU load-to-use latency.

The data cache natural alignment boundary is 16 bytes. A misaligned load or store operation suffers, at minimum, a one cycle penalty in the load-store pipeline if it spans a 16-byte boundary. 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 both 128-bit and 256-bit vectors is 16 bytes. There is no advantage in aligning 256-bit vectors to a 32-byte boundary on the Family 16h processor because 256-bit vectors are loaded and stored as two 128-bit halves.

2.5.3 L2 Cache

The AMD Family 16h processor implements a unified 16-way set associative L2 cache shared by up to 4 cores. This on-die L2 cache is inclusive of the L1 caches in the cores. The L2 is a write-back cache. The L2 cache has a variable load-to-use latency of no less than 25 cycles. The L2 cache size is 1 or 2 Mbytes depending on configuration. L2 cache entries are protected from errors through the use of an error correcting code (ECC).

The L2 to L1 data path is 16 bytes wide; critical data within a cache line is forwarded first.

The L2 has four 512-Kbyte banks. Bits 7:6 of the cache line address determine which bank holds the cache line. For a large contiguous block of data, this organization will naturally spread the cache lines out over all 4 banks. The banks can operate on requests in parallel and can each deliver 16 bytes per cycle, for a total peak read bandwidth of 64 bytes per cycle for the L2. Bandwidth to any individual core is 16 bytes per cycle peak, so with four cores, the four banks can each deliver 16 bytes of data to each core simultaneously. The banking scheme provides bandwidth for all four cores in the processing complex that can achieve the level that a private per-core L2 would provide.

2.6 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 Family 16h processor utilizes a two-level TLB structure.

2.6.1 L1 Translation Lookaside Buffers

The AMD Family 16h processor contains a fully-associative L1 instruction TLB (ITLB) with 32 4-Kbyte page entries and 8 2-Mbyte page entries.

The fully-associative L1 data TLB (DTLB) provides 40 4-Kbyte page entries and 8 2-Mbyte page entries.

2.6.2 L2 Translation Lookaside Buffers

The AMD Family 16h processor provides a 4-way set-associative L2 instruction TLB with 512 4-Kbyte page entries.

The L2 data TLB provides two independent translation buffers which are accessed in parallel; a 4-way set-associative buffer with 512 4-Kbyte page entries and a 2-way set-associative buffer with 256 2-Mbyte page entries.

2.6.3 Hardware Page Table Walker

The hardware page table walker handles L2 TLB misses. Misses can start speculatively from either the instruction or the data side. The table walker includes a 16-entry Page Directory Cache (PDC) to speed up table walks.

The table walker supports 1-Gbyte pages by *smashing* the page into a 2-Mbyte window, and returning a 2-Mbyte TLB entry. In legacy mode, 4-Mbyte entries are also supported by returning a smashed 2-Mbyte TLB entry.

INVLPG and INVLPGA instructions cause a flush of the entire TLB if any 1-Gbyte smashed entries have been created since the last flush. System software may wish to avoid the use of 1-Gbyte pages. In a nested paging environment, the processor does not create smashed entries if the nested page tables use 1-Gbyte pages but the guest page tables do not use 1-Gbyte pages.

See the definition of the terms *smashing* and *smashed* in the Preface.

2.7 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 Family 16h 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 minimum branch misprediction penalty is 14 cycles.

The following topic describes the branch prediction hardware facilities of the processor. This is followed by a discussion of how to align code within a loop to use the loop optimization hardware to its fullest advantage.

2.7.1 Branch Prediction

To predict and accelerate branches the AMD Family 16h processor employs:

- next-address logic
- branch target buffer
- branch target address calculator
- out-of-page target array
- branch marker caching
- return address stack (RAS)
- indirect target predictor
- advanced conditional branch direction predictor
- fetch window tracking structure

The following sections discuss each of these facilities in turn.

2.7.1.1 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 32-byte fetch block. This calculation is performed every cycle to support the 32 byte per cycle fetch bandwidth of the processor. 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.

2.7.1.2 Branch Target Buffer

The branch target buffer (BTB) is a two-level structure accessed using the fetch address of the current fetch block. Each BTB entry includes information about a branch and its target. The L1 BTB is a sparse branch predictor and maps up to the first two branches per instruction cache line (64 bytes), for a total of 1024 entries. The two branches in the sparse predictor can be predicted in the same cycle. The L2 BTB is a dense branch predictor and contains 1024 branch entries, mapped as up to an additional 2 branches per 8 byte instruction chunk, if located in the same 64-byte aligned block.

Predicted-taken branches incur a 1-cycle bubble in the branch prediction pipeline when they are predicted by the L1 BTB (sparse predictor). The L2 BTB (dense predictor) can predict one additional branch per cycle, with the first dense branch prediction incurring a 2-cycle bubble and subsequent predictions incurring one additional cycle per branch. Predicting long strings of branches in the same cache line through the dense predictor only occurs as long as the branches are predicted not-taken, as the string will be broken by a predicted taken branch.

An additional 3-cycle latency is incurred for branch targets predicted through the indirect target predictor or fixed up with the branch target address calculator. For example, an indirect branch predicted by the sparse predictor will incur a 4-cycle bubble. The minimum branch misprediction penalty is 14 cycles.

The dense branch predictor can predict one additional branch per cycle, with the first dense branch prediction incurring a 2-cycle bubble and subsequent predictions incurring one additional cycle per branch. Predicting long strings of branches per line through the dense predictor only occurs as long as the branches are fall-through, as the string will be broken by a predicted taken branch.

2.7.1.3 Branch Target Address Calculator

The branch target address calculator (BTAC) allows redirection if a direct branch target from the sparse or dense predictor was mispredicted. The BTAC can only correct direct branch targets after all of the immediate bytes for the branch instruction have been fetched.

2.7.1.4 Out-of-Page Target Array

The out-of-page target array (OPG) holds the high address bits ([28:12]) for 32 targets that are outside the current page for branches marked in the sparse BTB. Only sparse branches are eligible for out-of-page target prediction. Branches marked by the dense predictor are not eligible for OPG target prediction. Direct dense branches that are out-of-page will have their targets corrected by the branch target address calculator with a 4-cycle penalty. Direct sparse branch targets that cross a 28-bit address block boundary (beyond the range of the out-of-page target array) are also corrected by the branch target address calculator.

2.7.1.5 Branch Marker Caching

When a cache line is evicted, the sparse marker information for the first two branches in that cache line are slightly compressed and written out into a subset of the L2 ECC bits—but only if the line contains instructions exclusively. These markers are brought back into the core and reloaded into the sparse predictor if their L2 line is reloaded into the L1 instruction cache before eviction from L2 or before the line is the target of a store. Dense branches may or may not remain resident in the dense predictor when the L1 instruction cache is reloaded. Sparse markers in the shared L2 can be shared with other cores that fetch from the same L2 line.

Software with extremely large instruction footprints, especially those with multiple threads that share instruction cache lines, can take advantage of this property by targeting a branch density of no more than 2 branches per cache line.

2.7.1.6 Return Address Stack

The Family 16h processor implements a 16-entry return address stack (RAS) 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 of the call 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 commonly used coding practices optimized for other processor microarchitectures are not optimum for the Family 16h processor:

`CALL 0h`

In prior processor families (for example, Family 10h) a `CALL 0h` followed by a `POP` instruction was recommended for 32-bit software to get the RIP value into a general-purpose register. `CALL 0h` was recognized and treated specially, and the return address stack was kept consistent even though there was no return instruction paired with the call. On the Family 16h processor, `CALL 0h` is not treated specially, and thus this code sequence will cause the RAS to get out of sync due to the un-paired call. It is recommended to avoid the use of `CALL 0h` in 32-bit software, and instead use a true subroutine call, a `MOV reg,[RSP]` instruction, and a paired return to get the value of the RIP register into a general-purpose register.

`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. With processor Families 15h and 16h, 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.

2.7.1.7 Indirect Target Predictor

The processor implements a 512-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 a 26-bit global history structure.

Branches that have so far always had the same target are predicted by the indirect target predictor only if that target crosses a 28-bit address block boundary. Single-target branches whose target does not cross a 28-bit address block boundary are not predicted by the indirect target predictor; the out-of-page target array is used instead to achieve lower branch prediction latency.

2.7.1.8 Conditional Branch 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 the same 26-bit global history used by the indirect target predictor.

Conditional branches that have not yet been discovered to be taken are not marked in the sparse or dense predictor. 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.

2.7.1.9 Fetch Window Tracking Structure

Fetch windows are tracked in a 16-entry FIFO from fetch until retirement. Each entry holds branch and cacheline information for up to a full 64-byte cacheline (four 16-byte fetch windows). The first two branches in a cacheline (those identified by the sparse predictor) are allocated into a single entry. Each additional branch in the cacheline (those identified by the dense predictor) is allocated into a separate entry. 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.

2.7.2 Loop Alignment

For the Family 16h 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 32-byte fetch block every cycle, to achieve maximum fetch bandwidth the loop start point should be aligned to 32 bytes. For very hot loops, it may be useful to further consider branch placement. The branch predictor can process the first two branches in a cache line in a single cycle through the sparse predictor. For best performance, any branches in the first cache line of the hot loop should be in the sparse predictor. The simplest way to guarantee this for very hot loops is to align the start point to a cache line (64-byte) boundary.

2.7.2.1 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
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 AMD Family 16h processor.

Some earlier AMD processors 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 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 AMD Family 16h processors 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.

2.7.2.2 Aligning Loops to Reduce Power Consumption

The Family 16h processor includes a loop buffer which can reduce power consumption when hot loops fit entirely within it. The loop buffer is composed of four 32-byte chunks and is essentially a subset of the instruction cache. Two of the 32-byte chunks must be in the lower 32 bytes of a 64-byte cache line, and the other two must be from the upper 32 bytes of a cache line. Hot code loops that can fit within the loop buffer can save power by not requiring the full instruction cache lookup.

Compilers may choose to align known-hot loops to a 32-byte boundary if doing so ensures that they fit completely within the loop buffer. The loop buffer is a power optimization, not an instruction throughput optimization, although in a system with Bidirectional Application Power Management or performance boost enabled, this feature may allow sufficient power budget savings to enable boosting the clock rate to a higher frequency.

2.8 Instruction Fetch and Decode

The AMD Family 16h processor fetches instructions in 32-byte naturally aligned blocks. The processor can perform an instruction block fetch every cycle. The first two branches in a 64-byte cache line are typically allocated into the same fetch window tracking structure entry. Each additional branch will be allocated into a separate fetch window tracking structure entry.

The fetch unit sends these bytes to the decode unit through a 16-entry Instruction Byte Buffer (IBB) in two 16-byte windows. The IBB 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 two 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.

2.9 Integer Unit

The *integer unit* consists of the following components:

- schedulers
- execution units
- retire control unit

The schedulers feed integer micro-ops to the execution units. The execution units carry out various types of operations further described below. The retire control unit serves as the final arbiter for exception processing versus instruction retirement.

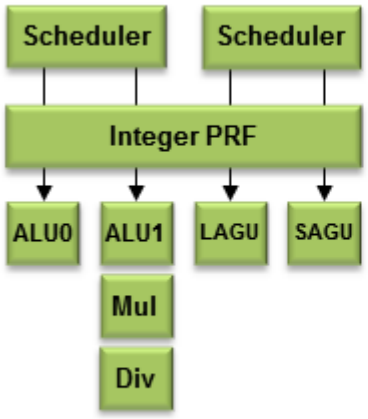
2.9.1 Integer Schedulers

The schedulers can receive up to two macro-ops per cycle, where they are broken down into micro-ops. ALU micro-ops are sent to the 20-entry ALU scheduler. Load and Store micro-ops are sent to the 12-entry address generation unit (AGU) scheduler. Each scheduler can issue up to two micro-ops per cycle. The scheduler tracks operand availability and dependency information as part of its task of issuing micro-ops to be executed. It also assures 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.

2.9.2 Integer Execution Units

The AMD Family 16h processor contains 4 integer execution pipes. There are 2 ALUs connected to the ALU scheduler, one of which can also handle integer multiplies and divides. There are 2 AGUs connected to the AGU scheduler, one AGU dedicated for load address generation handling (LAGU), and the other AGU dedicated for store address generation handling (SAGU).

Figure 2 below provides a block diagram of the integer schedulers and execution units for the AMD Family 16h processor core.



Abbreviation	Descriptive Name
Integer PRF	Integer Physical Register File
ALU0	Arithmetic and Logic Unit 0
ALU1	Arithmetic and Logic Unit 1
Mul	Multiply Unit
Div	Divide Unit
LAGU	Load Address Generation Unit
SAGU	Store Address Generation Unit

Diagram Legend

Figure 2. Integer Schedulers and Execution Units

All integer operations can be handled in the ALUs (ALU0 and 1 are fully symmetrical) with the exception of integer multiply, integer divide, and three-operand LEA instructions. While two-operand LEA instructions are mapped as a single-cycle micro-op in the ALUs, three-operand LEA instructions are mapped to the store AGU and have 2 cycle latency, with results inserted back in to the ALU1 pipeline.

The integer multiply unit can handle multiplies of up to 32 bits × 32 bits with 3 cycle latency, fully pipelined. 64-bit × 64-bit multiplies require data pumping and have a 6-cycle latency with a throughput rate of 1 every 4 cycles. If the multiply instruction has 2 destination registers, an additional one cycle latency and one cycle reduction in throughput is required.

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

2.9.3 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 2 macro-ops dispatched per cycle and track up to 64 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 (like when an AVX 256-bit instruction is broken into two 128-bit 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 two macro-ops per cycle.

The retire control unit also manages internal integer register mapping and renaming. The integer physical register file (PRF) consists of 64 registers, with between 20 to 31 mapped to architectural state or micro-architectural temporary state. The remaining 44 to 33 registers are available for out-of-order renames. Generally physical register renames are needed for instructions that write to an integer register destination (for example, ADD), but not for those instructions that only write flags (for example, CMP) or perform stores to memory.

2.10 Floating-Point Unit

The AMD Family 16h 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 packed single and double precision vector floating-point data types. The 256-bit packed single and double precision vector floating-point data types are fully supported through the use of two 128-bit macro-ops per instruction. The floating-point load and store

paths are 128 bits wide. As a result, the maximum throughput of both single-precision and double-precision floating-point SSE vector operations has improved by a factor of two over the AMD Family 14h processor.

The floating-point unit (FPU) utilizes a coprocessor model. As such it contains its own scheduler, register files, and renamers and does not share them with the integer units. It can handle dispatch and renaming of 2 floating-point macro-ops per cycle, and the scheduler can issue 1 micro-op per cycle for each pipe. The floating-point scheduler has an 18-entry micro-op capacity.

The floating-point retire queue holds up to 44 floating-point micro-ops between dispatch and retire. Any macro-op that has a floating-point micro-op component, and that is dispatched into the integer retire control unit, will be held in the floating-point retire queue until the macro-op retires from the integer retire control unit. Thus a maximum of 44 macro-ops which have floating-point micro-op components can be in-flight in the 64-macro-op in-flight window that the integer retire control unit provides.

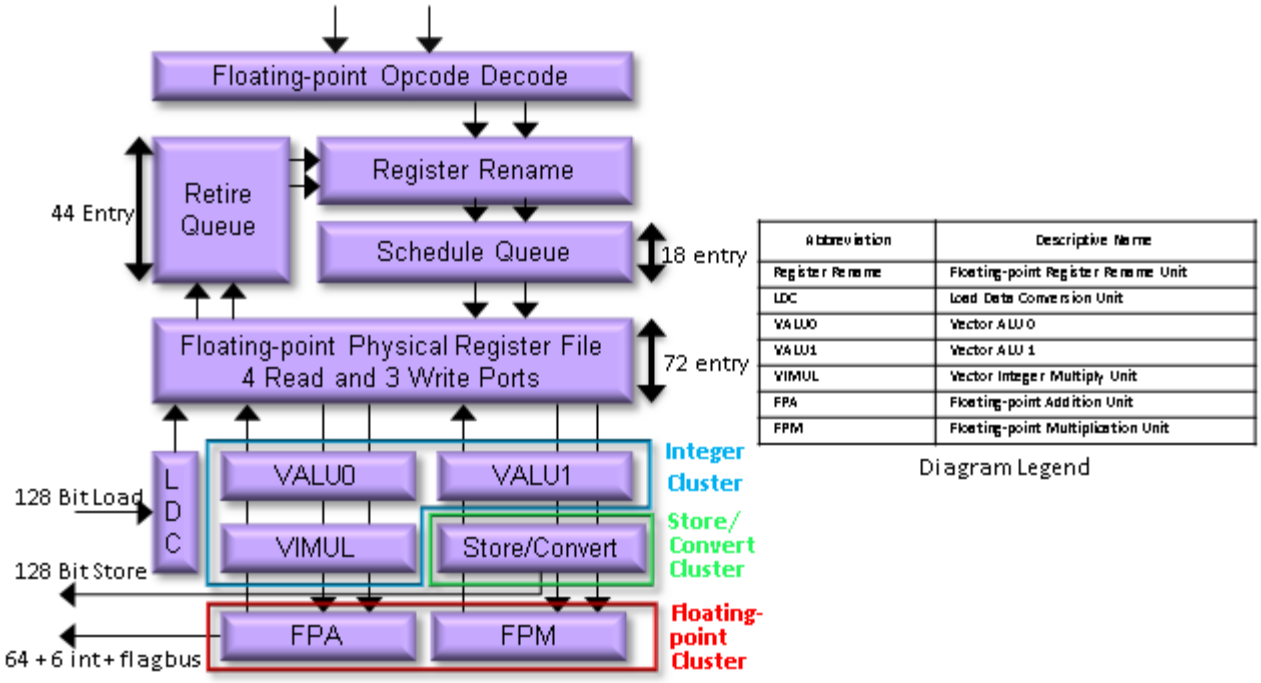


Figure 3. Floating-point Unit Block Diagram

The FPU contains a 128-bit floating-point multiply unit (FPM) and a 128-bit floating-point adder unit (FPA). The FPM contains two 76-bit × 27-bit multipliers, which means that double precision (64-bit) and extended precision (80-bit) floating-point multiplication computations require iteration. A few selected floating-point micro-ops, primarily logical/move/shuffle micro-ops, can execute in either the FPM or the FPA. The FPU also contains two 128-bit vector arithmetic / logical units (VALUs) which perform arithmetic and logical operations on AVX, SSE, and legacy MMX packed integer data, and a 128-bit integer multiply unit (VIMUL). The store/convert unit (STC) primarily handles stores (up to 128-bit operand size), floating-point / integer conversions, and integer / floating-point conversions. The register file and bypass network can also accept one 128-bit load per cycle from the load-store unit.

There are two important organizational dimensions to understand with respect to the execution units. The first is the pipeline binding. Pipe 0 contains vector integer ALU 0 (VALU0), the vector integer multiplier (VIMUL), and the floating-point adder (FPA). Pipe 1 contains vector integer ALU 1 (VALU1), the store/convert unit, and

the floating-point multiply unit (FPM). The floating-point scheduler can issue one micro-op to one unit per pipe per cycle and provides logic to prevent pipeline hazards like resource contention on the result bus.

The second organizational dimension for execution units is forwarding domains. The FPU is divided into three clusters, and forwarding between clusters requires an extra cycle in the bypass network. The three clusters are the Floating-point Cluster (composed of the FPM and FPA units), the Integer Cluster (composed of the VALU0, VALU1, and VIMUL units), and the Store / Convert Cluster (STC). When the result of an instruction executing in one domain is consumed as input by a subsequent instruction executing in a different domain there is a one cycle forwarding delay. This delay does not increase the time that either of the instructions is occupying the execution units, but the scheduler will not attempt to schedule the second instruction earlier. Most FPU instructions support local forwarding, which eliminates this delay when the consuming instruction executes in the same domain. However some instructions (marked with the note "local forwarding disabled" in the latency spreadsheet) do not support local forwarding and experience the forwarding delay even when the consuming instruction executes in the same domain.

The following table summarizes the majority of instruction latencies in the FPU.

Table 2. Summary of Floating-point Instruction Latencies

Instruction Class	Latency	Throughput	Execution Pipe	Unit(s)	Cluster
SIMD ALU (most)	1	2 / cycle	Either	VALU0, VALU1	Integer
Floating-point logical	1	2 / cycle	Either	FPA, FPM	Floating-pt
SIMD IMUL	2	1 / cycle	Pipe 0	VIM	Integer
Floating-point multiply single-precision	2	1 / cycle	Pipe 1	FPM	Floating-pt
Floating-point add	3	1 / cycle	Pipe 0	FPA	Floating-pt
Store/Convert (many)	3	1 / cycle	Pipe 1	Store/Convert	STC
Floating-point multiply double-precision	4	1 / 2 cycles	Pipe 1	FPM	Floating-pt
Floating-point multiply extended-precision (x87)	5	1 / 3 cycles	Pipe 1	FPM	Floating-pt
Floating-point `DIV`/ `SQRT`	Iterative	Iterative	Pipe 1	FPM	Floating-pt

Refer to the AMD64_16h_InstrLatency.xlsx spreadsheet described in Appendix A for more instruction latency and throughput information.

2.10.1 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. A significant performance cost (more than 100 processor cycles) 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. Denormal penalties may occur in two phases: usage of a denormal in a computation (pre-computation penalty), and production of a denormal during the execution of an instruction (post-computation penalty).

A sequence of floating-point compute instructions may incur a pre-computation penalty when a denormal value is encountered as an input. This penalty occurs on a floating-point computation instruction, such as `[V]ADDPS`,

if the denormal value was loaded into an XMM or YMM register from memory by a pure load instruction (such as [V]MOVUPS), or was produced by a vector-integer or logical instruction. The penalty will only occur once per new denormal value in a sequence of floating-point instructions. A similar penalty does not occur when the floating-point compute instruction is in load-op form and the memory operand is denormal, for example on [V]ADDPS xmm0,[mem] where [mem] is a denormal value.

If a compiler can determine that a memory input to a floating-point sequence is denormal, it can avoid this pre-computation penalty using a sequence such as: XORPS xmm0,xmm0; ADDPS xmm0,[mem] instead of MOVUPS xmm0,[mem].

Vector ALU and logical instructions will also incur a pre-computation penalty if they encounter a denormal input that was produced by a floating-point instruction.

If software does not require the precision that denormals provide, it can set MXCSR.DAZ (bit 6). Any denormal input will then be treated as a zero without a pre-computation penalty.

Post-computation penalties occur when a floating-point compute instruction produces a denormal result and both the precision exception and the underflow exception are masked in the MXCSR (that is, both bits 11 **Precision Mask** and bit 12 **Underflow Mask** are set). If software does not require the precision that denormals provide, it can set MXCSR.FTZ (bit 15). Any denormal output will then be converted to zero without a post-computation penalty. Post-computation penalties generally cannot be eliminated by compilers.

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.

For x87 instructions both pre-computation and post-computation penalties are incurred when denormals are encountered. A pre-computation penalty is incurred when loading denormal values from memory onto the x87 floating-point stack. A post-computation penalty is incurred when a floating-point compute instruction produces a denormal result and both the precision exception and underflow exception are masked in the x87 floating-point control word (FCW).

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. Programs that call x87 floating-point routines that internally produce denormal values will potentially incur this penalty as well. To completely avoid this penalty, ensure that programs written using legacy x87 instructions do not produce denormal values.

2.11 XMM Register Merge Optimization

The AMD Family 16h 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`

2.12 Load Store Unit

The AMD Family 16h processor load-store (LS) unit handles data accesses. The LS unit contains two largely independent pipelines enabling the execution of one 128-bit load memory operation and one 128-bit store memory operation per cycle.

The LS unit includes a 16-entry memory ordering queue (MOQ). The MOQ receives both load and store operations at dispatch. Loads leave the MOQ when the load has completed and delivered data to the integer unit or the floating-point unit. Stores leave the MOQ when their address has been translated.

The LS unit utilizes a 20-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 all of the following conditions are met:

- the store address and load address both start on the exact same byte
- the store operation size is the same or larger than the load operation size
- neither the load nor the store operation are misaligned

One STLF pitfall to avoid is aliases where store/load virtual address bits [15:4] match, but mismatch in the range [47:16] because it can delay execution of the load.

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

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.

Appendix A Instruction Latencies

The companion file AMD64_16h_InstrLatency_1.1.xlsx distributed with this *Software Optimization Guide* provides additional detailed information for the AMD Family 16h processor. The first worksheet in the spreadsheet, "Overview," provides some useful reference information which is related to the second worksheet, "Latencies." 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 family of processors, 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 describes the information provided in each column of the spreadsheet:

Column A	**Instruction**
	Instruction opcodes

Columns B–E **Op*n***

Instruction operands. The following notations are used in these columns:

- 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
- m*N*—an *N*-bit memory operand
- r—any general purpose (integer) register
- r*N*—an *N*-bit general purpose register
- xmm*N*—any `xmm` register, the *N* distinguishes among multiple operands of the same type
- ymm*N*—any `ymm` register, the *N* distinguishes among multiple operands of the same type

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.

Column F **Cpuid flag**

CPUID feature flag for the instruction

Column G **Macro Ops**

Number of macro-ops for the instruction.

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 either has not been determined or is variable.

Column H **Unit**

Execution units. The following abbreviations are used:

- ALU—Arithmetic / logical unit.
- FPA—Floating-point add functional element within the floating-point cluster of the floating-point unit.
- FPM—Floating-point multiply functional element in the floating-point cluster of the floating-point unit.
- DIV—Integer divide functional element within the integer unit
- MUL—Integer multiply functional element within the integer unit.
- SAGU—Store address generation unit within the integer unit.
- STC—Store/convert functional element in the store/convert cluster of the floating point unit.
- VALU—Either of the vector ALUs (VALU0 or VALU1) within the integer cluster of the floating-point unit.
- VIMUL—Vector integer multiply functional element within the integer cluster of the floating-point unit.
- ST—Store unit.

In this column, a vertical bar indicates that the instruction can use either of two alternative resources. A comma indicates that both of the comma-separated resources are required.

A number of instructions are floating-point load-ops which combine a transfer of data from the integer unit to the floating-point unit with a floating point operation. This transfer is implemented by storing the data from the integer unit to a private scratch memory location, then loading it back into the floating point unit. The Unit column indicates this with "ST,LD-*fpunit*" where *fpunit* is the floating point unit required for the load-op.

The notation *x2* or *x3* appended to one of the above specifies the number of macro-ops executed on that unit for the instruction. For example, FPMx2 indicates the instruction requires two serialized macro-ops using the floating-point multiply unit.

Column I **Latency**

Instruction latency in processor cycles.

Where the latency is given as a number with a "+i", the latency listed is the latency of the floating-point operation. The +i represents an additional 1 cycle transfer operation which moves the floating point result to the integer unit. During this additional cycle the floating point unit execution resources are not occupied and ALU0 in the integer unit is occupied instead.

Where the latency is given as an "f+" with a number, the latency listed is the latency of the floating-point operation. The f+ represents an additional 6 cycle transfer operation which moves a floating point operation input value from the integer unit to the floating point unit. During these additional 6 cycles the floating point unit execution resources are not occupied and ST and LD in the integer unit are occupied instead.

Refer to the section "Instruction Latency Assumptions" above for more information about this column.

Column J **1/Throughput**

Reciprocal throughput of the instruction.

This is the number of cycles from when an instruction starts to execute until a second instruction of the same type can begin to execute. A value of 0.5 in the spreadsheet 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.

Refer to the following section "Instruction Latency Assumptions" for more information.

Column K **Notes**

Additional information about the entry.

For a discussion of the note "local forwarding disabled," see "local forwarding disabled."