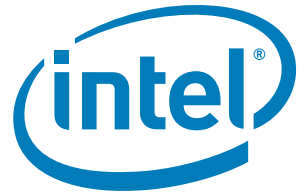

Intel® 64 and IA-32 Architectures Optimization Reference Manual

Order Number: 248966-014
November 2006

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an ordering number and are referenced in this document, or other Intel literature, may be obtained from:

Intel Corporation
P.O. Box 5937
Denver, CO 80217-9808

or call 1-800-548-4725
or visit Intel's website at http://www.intel.com

CONTENTS

PAGE

CHAPTER 3
GENERAL OPTIMIZATION GUIDELINES

CHAPTER 2
INTEL® 64 AND IA-32 PROCESSOR ARCHITECTURES

This chapter gives an overview of features relevant to software optimization for current generations of Intel 64 and IA-32 processors (processors based on the Intel Core microarchitecture, Intel NetBurst microarchitecture; including Intel Core Solo, Intel Core Duo, and Intel Pentium M processors). These features are:

- Microarchitectures that enable executing instructions with high throughput at high clock rates, a high speed cache hierarchy and high speed system bus
- Multicore architecture available in Intel Core 2 Duo, Intel Core Duo, Intel Pentium D processors, Pentium processor Extreme Edition[1], and Dual-core Intel Xeon processors
- Hyper-Threading Technology[2] (HT Technology) support
- Intel 64 architecture on Intel 64 processors
- SIMD instruction extensions: MMX technology, Streaming SIMD Extensions (SSE), Streaming SIMD Extensions 2 (SSE2), Streaming SIMD Extensions 3 (SSE3), and Supplemental Streaming SIMD Extensions 3 (SSSE3)

The Intel Pentium M processor introduced a power-efficient microarchitecture with balanced performance. Dual-core Intel Xeon processor LV, Intel Core Solo and Intel Core Duo processors incorporate enhanced Pentium M processor microarchitecture. The Intel Core 2 processor family, Intel Xeon processor 3000 series and 5100 series are based on the high-performance and power-efficient Intel Core microarchitecture. Intel Pentium 4 processors, Intel Xeon processors, Pentium D processors, and Pentium processor Extreme Editions are based on Intel NetBurst microarchitecture.

1. Dual-core platform requires an Intel Xeon processor 3000 series, Intel Xeon processor 5100 series, Intel Core 2 Duo, Intel Core 2 Extreme, Dual-core Intel Xeon processors, Intel Core Duo, Pentium D processor or Pentium processor Extreme Edition, with appropriate chipset, BIOS, and operating system. Performance varies depending on the hardware and software used.
2. Hyper-Threading Technology requires a computer system with an Intel processor supporting HT Technology and an HT Technology enabled chipset, BIOS and operating system. Performance varies depending on the hardware and software used.

2.1 INTEL® CORE™ MICROARCHITECTURE

Intel Core microarchitecture introduces the following features that enable high performance and power-efficient performance for single-threaded as well as multi-threaded workloads:

- **Intel® Wide Dynamic Execution** enables each processor core to fetch, dispatch, execute with high bandwidths and retire up to four instructions per cycle. Features include:
  - Fourteen-stage efficient pipeline
  - Three arithmetic logical units
  - Four decoders to decode up to five instruction per cycle
  - Macro-fusion and micro-fusion to improve front-end throughput
  - Peak issue rate of dispatching up to six µops per cycle
  - Peak retirement bandwidth of up to four µops per cycle
  - Advanced branch prediction
  - Stack pointer tracker to improve efficiency of executing function/procedure entries and exits
- **Intel® Advanced Smart Cache** delivers higher bandwidth from the second level cache to the core, optimal performance and flexibility for single-threaded and multi-threaded applications. Features include:
  - Optimized for multicore and single-threaded execution environments
  - 256 bit internal data path to improve bandwidth from L2 to first-level data cache
  - Unified, shared second-level cache of 4 Mbyte, 16 way (or 2 MByte, 8 way)
- **Intel® Smart Memory Access** prefetches data from memory in response to data access patterns and reduces cache-miss exposure of out-of-order execution. Features include:
  - Hardware prefetchers to reduce effective latency of second-level cache misses
  - Hardware prefetchers to reduce effective latency of first-level data cache misses
  - Memory disambiguation to improve efficiency of speculative execution execution engine

- **Intel® Advanced Digital Media Boost** improves most 128-bit SIMD instructions with single-cycle throughput and floating-point operations. Features include:
  - Single-cycle throughput of most 128-bit SIMD instructions
  - Up to eight floating-point operations per cycle
  - Three issue ports available to dispatching SIMD instructions for execution

2.1.1 Intel® Core™ Microarchitecture Pipeline Overview

The pipeline of the Intel Core microarchitecture contains:

- An in-order issue front end that fetches instruction streams from memory, with four instruction decoders to supply decoded instruction (μops) to the out-of-order execution core.
- An out-of-order superscalar execution core that can issue up to six μops per cycle (see Table 2-2) and reorder μops to execute as soon as sources are ready and execution resources are available.
- An in-order retirement unit that ensures the results of execution of μops are processed and architectural states are updated according to the original program order.

Intel Core 2 Extreme, Intel Core 2 Duo processors and Intel Xeon processor 5100 series implement two processor cores based on the Intel Core microarchitecture, the functionality of the subsystems in each core are depicted in Figure 2-1.

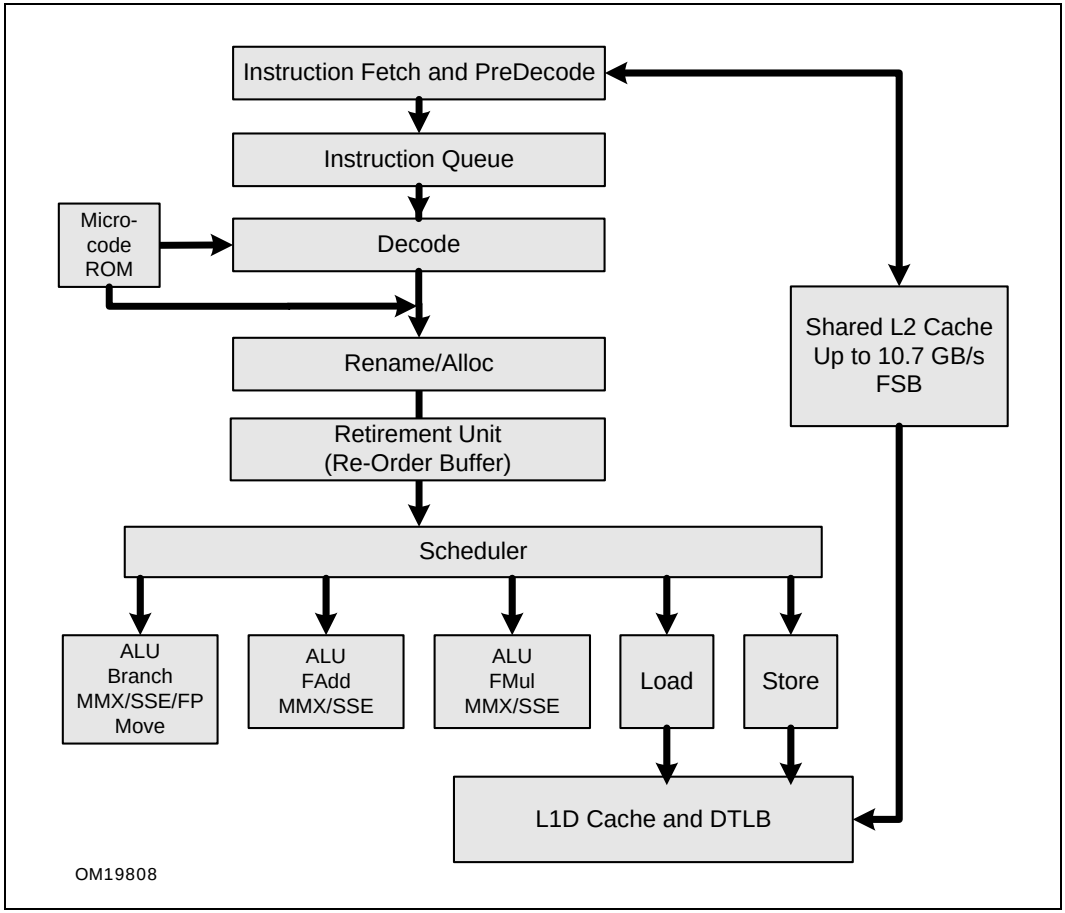

Figure 2-1. Intel Core Microarchitecture Pipeline Functionality

2.1.2 Front End

The front ends needs to supply decoded instructions (μops) and sustain the stream to a six-issue wide out-of-order engine. The components of the front end, their func-

tions, and the performance challenges to microarchitectural design are described in Table 2-1.

Table 2-1. Components of the Front End

Component	Functions	Performance Challenges
Branch Prediction Unit (BPU)	• Helps the instruction fetch unit fetch the most likely instruction to be executed by predicting the various branch types: conditional, indirect, direct, call, and return. Uses dedicated hardware for each type.	• Enables speculative execution. • Improves speculative execution efficiency by reducing the amount of code in the "non-architected path"[1] to be fetched into the pipeline.
Instruction Fetch Unit	• Prefetches instructions that are likely to be executed • Caches frequently-used instructions • Predecodes and buffers instructions, maintaining a constant bandwidth despite irregularities in the instruction stream	• Variable length instruction format causes unevenness (bubbles) in decode bandwidth. • Taken branches and misaligned targets causes disruptions in the overall bandwidth delivered by the fetch unit.
Instruction Queue and Decode Unit	• Decodes up to four instructions, or up to five with macro-fusion • Stack pointer tracker algorithm for efficient procedure entry and exit • Implements the Macro-Fusion feature, providing higher performance and efficiency • The Instruction Queue is also used as a loop cache, enabling some loops to be executed with both higher bandwidth and lower power	• Varying amounts of work per instruction requires expansion into variable numbers of μops. • Prefix adds a dimension of decoding complexity. • Length Changing Prefix (LCP) can cause front end bubbles.

NOTES:

1. Code paths that the processor thought it should execute but then found out it should go in another path and therefore reverted from its initial intention.

2.1.2.1 Branch Prediction Unit

Branch prediction enables the processor to begin executing instructions long before the branch outcome is decided. All branches utilize the BPU for prediction. The BPU contains the following features:

- 16-entry Return Stack Buffer (RSB). It enables the BPU to accurately predict RET instructions.
- Front end queuing of BPU lookups. The BPU makes branch predictions for 32 bytes at a time, twice the width of the fetch engine. This enables taken branches to be predicted with no penalty.

  Even though this BPU mechanism generally eliminates the penalty for taken branches, software should still regard taken branches as consuming more resources than do not-taken branches.

The BPU makes the following types of predictions:

- Direct Calls and Jumps. Targets are read as a target array, without regarding the taken or not-taken prediction.
- Indirect Calls and Jumps. These may either be predicted as having a monotonic target or as having targets that vary in accordance with recent program behavior.
- Conditional branches. Predicts the branch target and whether or not the branch will be taken.

For information about optimizing software for the BPU, see Section 3.4, “Optimizing the Front End”.

2.1.2.2 Instruction Fetch Unit

The instruction fetch unit comprises the instruction translation lookaside buffer (ITLB), an instruction prefetcher, the instruction cache and the predecode logic of the instruction queue (IQ).

Instruction Cache and ITLB

An instruction fetch is a 16-byte aligned lookup through the ITLB into the instruction cache and instruction prefetch buffers. A hit in the instruction cache causes 16 bytes to be delivered to the instruction predecoder. Typical programs average slightly less than 4 bytes per instruction, depending on the code being executed. Since most instructions can be decoded by all decoders, an entire fetch can often be consumed by the decoders in one cycle.

A misaligned target reduces the number of instruction bytes by the amount of offset into the 16 byte fetch quantity. A taken branch reduces the number of instruction bytes delivered to the decoders since the bytes after the taken branch are not decoded. Branches are taken approximately every 10 instructions in typical integer code, which translates into a “partial” instruction fetch every 3 or 4 cycles.

Due to stalls in the rest of the machine, front end starvation does not usually cause performance degradation. For extremely fast code with larger instructions (such as SSE2 integer media kernels), it may be beneficial to use targeted alignment to prevent instruction starvation.

Instruction PreDecode

The predecode unit accepts the sixteen bytes from the instruction cache or prefetch buffers and carries out the following tasks:

- Determine the length of the instructions.
- Decode all prefixes associated with instructions.
- Mark various properties of instructions for the decoders (for example, “is branch.”).

The predecode unit can write up to six instructions per cycle into the instruction queue. If a fetch contains more than six instructions, the predecoder continues to decode up to six instructions per cycle until all instructions in the fetch are written to the instruction queue. Subsequent fetches can only enter predecoding after the current fetch completes.

For a fetch of seven instructions, the predecoder decodes the first six in one cycle, and then only one in the next cycle. This process would support decoding 3.5 instructions per cycle. Even if the instruction per cycle (IPC) rate is not fully optimized, it is higher than the performance seen in most applications. In general, software usually does not have to take any extra measures to prevent instruction starvation.

The following instruction prefixes cause problems during length decoding. These prefixes can dynamically change the length of instructions and are known as length changing prefixes (LCPs):

- Operand Size Override (66H) preceding an instruction with a word immediate data
- Address Size Override (67H) preceding an instruction with a mod R/M in real, 16-bit protected or 32-bit protected modes

When the predecoder encounters an LCP in the fetch line, it must use a slower length decoding algorithm. With the slower length decoding algorithm, the predecoder decodes the fetch in 6 cycles, instead of the usual 1 cycle.

Normal queuing within the processor pipeline usually cannot hide LCP penalties.

The REX prefix (4xh) in the Intel 64 architecture instruction set can change the size of two classes of instruction: MOV offset and MOV immediate. Nevertheless, it does not cause an LCP penalty and hence is not considered an LCP.

2.1.2.3 Instruction Queue (IQ)

The instruction queue is 18 instructions deep. It sits between the instruction predecode unit and the instruction decoders. It sends up to five instructions per cycle, and

supports one macro-fusion per cycle. It also serves as a loop cache for loops smaller than 18 instructions. The loop cache operates as described below.

A Loop Stream Detector (LSD) resides in the BPU. The LSD attempts to detect loops which are candidates for streaming from the instruction queue (IQ). When such a loop is detected, the instruction bytes are locked down and the loop is allowed to stream from the IQ until a misprediction ends it. When the loop plays back from the IQ, it provides higher bandwidth at reduced power (since much of the rest of the front end pipeline is shut off).

The LSD provides the following benefits:

- No loss of bandwidth due to taken branches
- No loss of bandwidth due to misaligned instructions
- No LCP penalties, as the pre-decode stage has already been passed
- Reduced front end power consumption, because the instruction cache, BPU and predecode unit can be idle

Software should use the loop cache functionality opportunistically. Loop unrolling and other code optimizations may make the loop too big to fit into the LSD. For high performance code, loop unrolling is generally preferable for performance even when it overflows the loop cache capability.

2.1.2.4 Instruction Decode

The Intel Core microarchitecture contains four instruction decoders. The first, Decoder 0, can decode Intel 64 and IA-32 instructions up to 4 μops in size. Three other decoders handles single-μop instructions. The microsequencer can provide up to 3 μops per cycle, and helps decode instructions larger than 4 μops.

All decoders support the common cases of single μop flows, including: micro-fusion, stack pointer tracking and macro-fusion. Thus, the three simple decoders are not limited to decoding single-μop instructions. Packing instructions into a 4-1-1-1 template is not necessary and not recommended.

Macro-fusion merges two instructions into a single μop. Intel Core microarchitecture is capable of one macro-fusion per cycle in 32-bit operation (including compatibility sub-mode of the Intel 64 architecture), but not in 64-bit mode because code that uses longer instructions (length in bytes) more often is less likely to take advantage of hardware support for macro-fusion.

2.1.2.5 Stack Pointer Tracker

The Intel 64 and IA-32 architectures have several commonly used instructions for parameter passing and procedure entry and exit: PUSH, POP, CALL, LEAVE and RET. These instructions implicitly update the stack pointer register (RSP), maintaining a combined control and parameter stack without software intervention. These instructions are typically implemented by several μops in previous microarchitectures.

The Stack Pointer Tracker moves all these implicit RSP updates to logic contained in the decoders themselves. The feature provides the following benefits:

- Improves decode bandwidth, as PUSH, POP and RET are single μop instructions in Intel Core microarchitecture.
- Conserves execution bandwidth as the RSP updates do not compete for execution resources.
- Improves parallelism in the out of order execution engine as the implicit serial dependencies between μops are removed.
- Improves power efficiency as the RSP updates are carried out on small, dedicated hardware.

2.1.2.6 Micro-fusion

Micro-fusion fuses multiple μops from the same instruction into a single complex μop. The complex μop is dispatched in the out-of-order execution core. Micro-fusion provides the following performance advantages:

- Improves instruction bandwidth delivered from decode to retirement.
- Reduces power consumption as the complex μop represents more work in a smaller format (in terms of bit density), reducing overall “bit-toggling” in the machine for a given amount of work and virtually increasing the amount of storage in the out-of-order execution engine.

Many instructions provide register flavors and memory flavors. The flavor involving a memory operand will decodes into a longer flow of μops than the register version. Micro-fusion enables software to use memory to register operations to express the actual program behavior without worrying about a loss of decode bandwidth.

2.1.3 Execution Core

The execution core of the Intel Core microarchitecture is superscalar and can process instructions out of order. When a dependency chain causes the machine to wait for a resource (such as a second-level data cache line), the execution core executes other instructions. This increases the overall rate of instructions executed per cycle (IPC).

The execution core contains the following three major components:

- **Renamer** — Moves μops from the front end to the execution core. Architectural registers are renamed to a larger set of microarchitectural registers. Renaming eliminates false dependencies known as read-after-read and write-after-read hazards.
- **Reorder buffer** (ROB) — Holds μops in various stages of completion, buffers completed μops, updates the architectural state in order, and manages ordering of exceptions. The ROB has 96 entries to handle instructions in flight.

- **Reservation station** (RS) — Queues μops until all source operands are ready, schedules and dispatches ready μops to the available execution units. The RS has 32 entries.

The initial stages of the out of order core move the μops from the front end to the ROB and RS. In this process, the out of order core carries out the following steps:

- Allocates resources to μops (for example: these resources could be load or store buffers).
- Binds the μop to an appropriate issue port.
- Renames sources and destinations of μops, enabling out of order execution.
- Provides data to the μop when the data is either an immediate value or a register value that has already been calculated.

The following list describes various types of common operations and how the core executes them efficiently:

- **Micro-ops with single-cycle latency** — Most μops with single-cycle latency can be executed by multiple execution units, enabling multiple streams of dependent operations to be executed quickly.
- **Frequently-used μops with longer latency** — These μops have pipelined execution units so that multiple μops of these types may be executing in different parts of the pipeline simultaneously.
- **Operations with data-dependent latencies** — Some operations, such as division, have data dependent latencies. Integer division parses the operands to perform the calculation only on significant portions of the operands, thereby speeding up common cases of dividing by small numbers.
- **Floating point operations with fixed latency for operands that meet certain restrictions** — Operands that do not fit these restrictions are considered exceptional cases and are executed with higher latency and reduced throughput. The lower-throughput cases do not affect latency and throughput for more common cases.
- **Memory operands with variable latency, even in the case of an L1 cache hit** — Loads that are not known to be safe from forwarding may wait until a store-address is resolved before executing. The memory order buffer (MOB) accepts and processes all memory operations. See Section 2.1.5 for more information about the MOB.

2.1.3.1 Issue Ports and Execution Units

The scheduler can dispatch up to six μops per cycle through the issue ports depicted in Table 2-2. The table provides latency and throughput data of common integer and floating-point (FP) operations for each issue port in cycles.

Table 2-2. Issue Ports of Intel Core Microarchitecture

Port	Executable operations	Latency	Through put	Writeback Port	Comment
Port 0	Integer ALU	1	1	Writeback 0	Includes 64-bit mode integer MUL.
	Integer SIMD ALU	1	1		
	Single-precision (SP) FP MUL	4	1		Mixing operations of different latencies that use the same port can result in writeback bus conflicts; this can reduce overall throughput.
	Double-precision FP MUL	5	1		
	FP MUL (X87)	5	2		
	FP/SIMD/SSE2 Move and Logic	1	1		
	QW Shuffle	1	1		
Port 1	Integer ALU	1	1	Writeback 1	Excludes 64-bit mode integer MUL.
	Integer SIMD MUL	1	1		
	FP ADD	3	1		Mixing operations of different latencies that use the same port can result in writeback bus conflicts; this can reduce overall throughput.
	FP/SIMD/SSE2 Move and Logic	1	1		
	QW Shuffle	1	1		
Port 2	Integer loads	3	1	Writeback 2	
	FP loads	4	1		
Port 3	Store address	3	1	None (flags)	Prepares the store forwarding and store retirement logic with the address of the data being stored.
Port 4	Store data			None	Prepares the store forwarding and store retirement logic with the data being stored.
Port 5	Integer ALU	1	1	Writeback 5	Excludes QW shuffles.
	Integer SIMD ALU	1	1		
	FP/SIMD/SSE2 Move and Logic	1	1		
	Shuffle	1	1		

In each cycle, the RS can dispatch up to six μops. Each cycle, up to 4 results may be written back to the RS and ROB, to be used as early as the next cycle by the RS. This

high execution bandwidth enables execution bursts to keep up with the functional expansion of the micro-fused μops that are decoded and retired.

The execution core contains the following three execution stacks:

- SIMD integer
- regular integer
- x87/SIMD floating point

The execution core also contains connections to and from the memory cluster. See Figure 2-2.

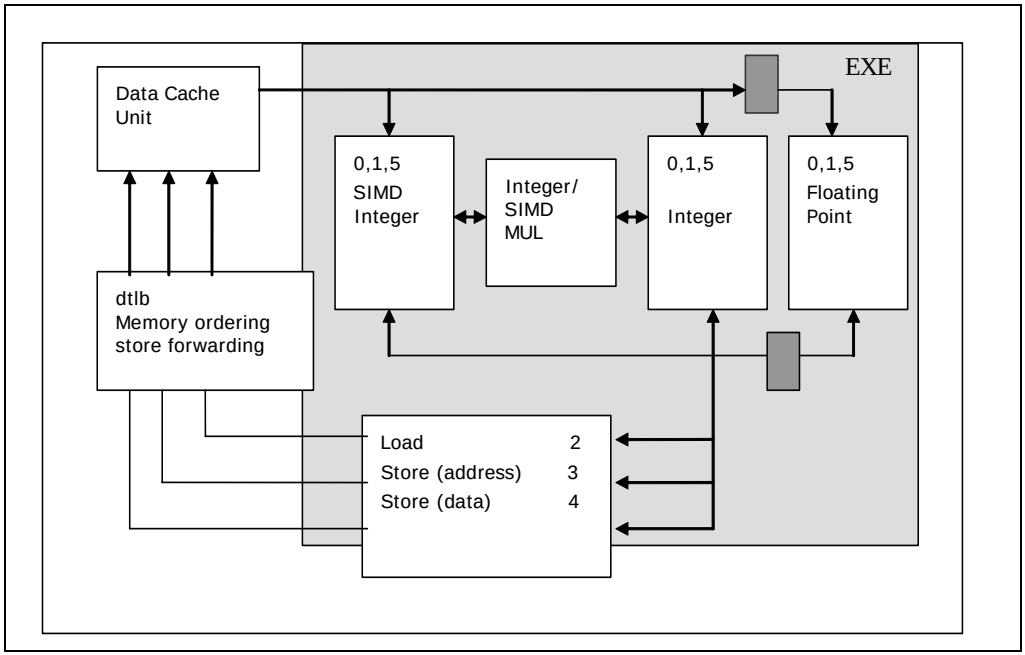

Figure 2-2. Execution Core of Intel Core Microarchitecture

Notice that the two dark squares inside the execution block (in grey color) and appear in the path connecting the integer and SIMD integer stacks to the floating point stack. This delay shows up as an extra cycle called a bypass delay. Data from the L1 cache has one extra cycle of latency to the floating point unit. The dark-colored squares in Figure 2-2 represent the extra cycle of latency.

2.1.4 Intel® Advanced Memory Access

The Intel Core microarchitecture contains an instruction cache and a first-level data cache in each core. The two cores share a 2 or 4-MByte L2 cache. All caches are writeback and non-inclusive. Each core contains:

- **L1 data cache, known as the data cache unit (DCU)** — The DCU can handle multiple outstanding cache misses and continue to service incoming stores and loads. It supports maintaining cache coherency. The DCU has the following specifications:
  - 32-KBytes size
  - 8-way set associative
  - 64-bytes line size
- **Data translation lookaside buffer (DTLB)** — The DTLB in Intel Core microarchitecture implements two levels of hierarchy. Each level of the DTLB have multiple entries and can support either 4-KByte pages or large pages. The entries of the inner level (DTLB0) is used for loads. The entries in the outer level (DTLB1) support store operations and loads that missed DTLB0. All entries are 4-way associative. Here is a list of entries in each DTLB:
  - DTLB1 for large pages: 32 entries
  - DTLB1 for 4-KByte pages: 256 entries
  - DTLB0 for large pages: 16 entries
  - DTLB0 for 4-KByte pages: 16 entries

  An DTLB0 miss and DTLB1 hit causes a penalty of 2 cycles. Software only pays this penalty if the DTLB0 is used in some dispatch cases. The delays associated with a miss to the DTLB1 and PMH are largely non-blocking due to the design of Intel Smart Memory Access.
- **Page miss handler (PMH)**
- **A memory ordering buffer (MOB)** — Which:
  - enables loads and stores to issue speculatively and out of order
  - ensures retired loads and stores have the correct data upon retirement
  - ensures loads and stores follow memory ordering rules of the Intel 64 and IA-32 architectures.

The memory cluster of the Intel Core microarchitecture uses the following to speed up memory operations:

- 128-bit load and store operations
- data prefetching to L1 caches
- data prefetch logic for prefetching to the L2 cache
- store forwarding
- memory disambiguation

- 8 fill buffer entries
- 20 store buffer entries
- out of order execution of memory operations
- pipelined read-for-ownership operation (RFO)

For information on optimizing software for the memory cluster, see Section 3.6, “Optimizing Memory Accesses.”

2.1.4.1 Loads and Stores

The Intel Core microarchitecture can execute up to one 128-bit load and up to one 128-bit store per cycle, each to different memory locations. The microarchitecture enables execution of memory operations out of order with respect to other instructions and with respect to other memory operations.

Loads can:

- issue before preceding stores when the load address and store address are known not to conflict
- be carried out speculatively, before preceding branches are resolved
- take cache misses out of order and in an overlapped manner
- issue before preceding stores, speculating that the store is not going to be to a conflicting address

Loads cannot:

- speculatively take any sort of fault or trap
- speculatively access the uncacheable memory type

Faulting or uncacheable loads are detected and wait until retirement, when they update the programmer visible state. x87 and floating point SIMD loads add 1 additional clock latency.

Stores to memory are executed in two phases:

- **Execution phase** — Prepares the store buffers with address and data for store forwarding. Consumes dispatch ports, which are ports 3 and 4.
- **Completion phase** — The store is retired to programmer-visible memory. It may compete for cache banks with executing loads. Store retirement is maintained as a background task by the memory order buffer, moving the data from the store buffers to the L1 cache.

2.1.4.2 Data Prefetch to L1 caches

Intel Core microarchitecture provides two hardware prefetchers to speed up data accessed by a program by prefetching to the L1 data cache:

- **Data cache unit (DCU) prefetcher** — This prefetcher, also known as the streaming prefetcher, is triggered by an ascending access to very recently loaded

data. The processor assumes that this access is part of a streaming algorithm and automatically fetches the next line.

- **Instruction pointer (IP)- based strided prefetcher** — This prefetcher keeps track of individual load instructions. If a load instruction is detected to have a regular stride, then a prefetch is sent to the next address which is the sum of the current address and the stride. This prefetcher can prefetch forward or backward and can detect strides of up to half of a 4KB-page, or 2 KBytes.

Data prefetching works on loads only when the following conditions are met:

- Load is from writeback memory type.
- Prefetch request is within the page boundary of 4 Kbytes.
- No fence or lock is in progress in the pipeline.
- Not many other load misses are in progress.
- The bus is not very busy.
- There is not a continuous stream of stores.

DCU Prefetching has the following effects:

- Improves performance if data in large structures is arranged sequentially in the order used in the program.
- May cause slight performance degradation due to bandwidth issues if access patterns are sparse instead of local.
- On rare occasions, if the algorithm's working set is tuned to occupy most of the cache and unneeded prefetches evict lines required by the program, hardware prefetcher may cause severe performance degradation due to cache capacity of L1.

In contrast to hardware prefetchers relying on hardware to anticipate data traffic, software prefetch instructions relies on the programmer to anticipate cache miss traffic, software prefetch act as hints to bring a cache line of data into the desired levels of the cache hierarchy. The software-controlled prefetch is intended for prefetching data, but not for prefetching code.

2.1.4.3 Data Prefetch Logic

Data prefetch logic (DPL) prefetches data to the second-level (L2) cache based on past request patterns of the DCU from the L2. The DPL maintains two independent arrays to store addresses from the DCU: one for upstreams (12 entries) and one for down streams (4 entries). The DPL tracks accesses to one 4K byte page in each entry. If an accessed page is not in any of these arrays, then an array entry is allocated.

The DPL monitors DCU reads for incremental sequences of requests, known as streams. Once the DPL detects the second access of a stream, it prefetches the next cache line. For example, when the DCU requests the cache lines A and A+1, the DPL assumes the DCU will need cache line A+2 in the near future. If the DCU then reads

A+2, the DPL prefetches cache line A+3. The DPL works similarly for “downward” loops.

The Intel Pentium M processor introduced DPL. The Intel Core microarchitecture added the following features to DPL:

- The DPL can detect more complicated streams, such as when the stream skips cache lines. DPL may issue 2 prefetch requests on every L2 lookup. The DPL in the Intel Core microarchitecture can run up to 8 lines ahead from the load request.
- DPL in the Intel Core microarchitecture adjusts dynamically to bus bandwidth and the number of requests. DPL prefetches far ahead if the bus is not busy, and less far ahead if the bus is busy.
- DPL adjusts to various applications and system configurations.

Entries for the two cores are handled separately.

2.1.4.4 Store Forwarding

If a load follows a store and reloads the data that the store writes to memory, the Intel Core microarchitecture can forward the data directly from the store to the load. This process, called store to load forwarding, saves cycles by enabling the load to obtain the data directly from the store operation instead of through memory.

The following rules must be met for store to load forwarding to occur:

- The store must be the last store to that address prior to the load.
- The store must be equal or greater in size than the size of data being loaded.
- The load cannot cross a cache line boundary.
- The load cannot cross an 8-Byte boundary. 16-Byte loads are an exception to this rule.
- The load must be aligned to the start of the store address, except for the following exceptions:
  - An aligned 64-bit store may forward either of its 32-bit halves
  - An aligned 128-bit store may forward any of its 32-bit quarters
  - An aligned 128-bit store may forward either of its 64-bit halves

Software can use the exceptions to the last rule to move complex structures without losing the ability to forward the subfields.

2.1.4.5 Memory Disambiguation

A load instruction μop may depend on a preceding store. Many microarchitectures block loads until all preceding store address are known.

The memory disambiguator predicts which loads will not depend on any previous stores. When the disambiguator predicts that a load does not have such a dependency, the load takes its data from the L1 data cache.

Eventually, the prediction is verified. If an actual conflict is detected, the load and all succeeding instructions are re-executed.

2.1.5 Intel® Advanced Smart Cache

The Intel Core microarchitecture optimized a number of features for two processor cores on a single die. The two cores share a second-level cache and a bus interface unit, collectively known as Intel Advanced Smart Cache. This section describes the components of Intel Advanced Smart Cache. Figure 2-3 illustrates the architecture of the Intel Advanced Smart Cache.

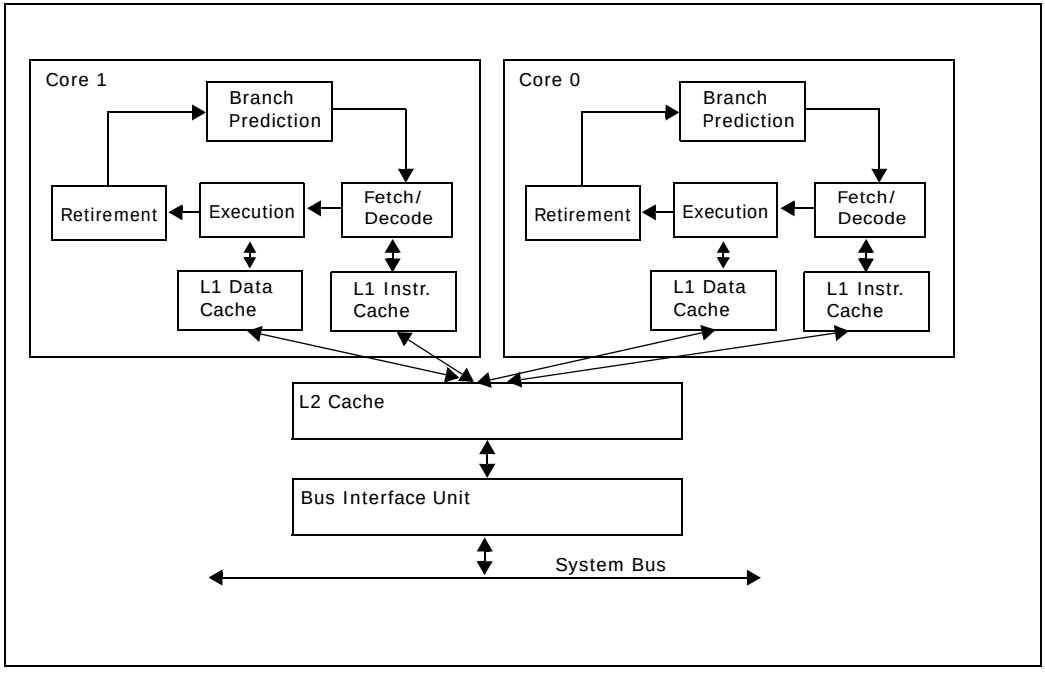

Figure 2-3. Intel Advanced Smart Cache Architecture

Table 2-3 details the parameters of caches in the Intel Core microarchitecture. For information on enumerating the cache hierarchy identification using the deterministic cache parameter leaf of CPUID instruction, see the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A.*

Table 2-3. Cache Parameters of Processors based on Intel Core Microarchitecture

Level	Capacity	Associativity (ways)	Line Size (bytes)	Access Latency (clocks)	Access Throughput (clocks)	Write Update Policy
First Level	32 KB	8	64	3	1	Writeback
Instruction	32 KB	8	N/A	N/A	N/A	N/A
Second Level (Shared L2)	2, 4 MB	8 or 16	64	14[1]	2	Writeback

NOTES:

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

2.1.5.1 Loads

When an instruction reads data from a memory location that has write-back (WB) type, the processor looks for the cache line that contains this data in the caches and memory in the following order:

1. DCU of the initiating core
2. DCU of the other core and second-level cache
3. System memory

The cache line is taken from the DCU of the other core only if it is modified, ignoring the cache line availability or state in the L2 cache.

Table 2-4 shows the characteristics of fetching the first four bytes of different localities from the memory cluster. The latency column provides an estimate of access latency. However, the actual latency can vary depending on the load of cache, memory components, and their parameters.

Table 2-4. Characteristics of Load and Store Operations in Intel Core Microarchitecture

	Load		Store	
Data Locality	**Latency**	**Throughput**	**Latency**	**Throughput**
DCU	3	1	2	1
DCU of the other core in modified state	14 + 5.5 bus cycles	14 + 5.5 bus cycles	14 + 5.5 bus cycles	
2nd-level cache	14	3	14	3
Memory	14 + 5.5 bus cycles + memory	Depends on bus read protocol	14 + 5.5 bus cycles + memory	Depends on bus write protocol

Sometimes a modified cache line has to be evicted to make space for a new cache line. The modified cache line is evicted in parallel to bringing the new data and does not require additional latency. However, when data is written back to memory, the eviction uses cache bandwidth and possibly bus bandwidth as well. Therefore, when multiple cache misses require the eviction of modified lines within a short time, there is an overall degradation in cache response time.

2.1.5.2 Stores

When an instruction writes data to a memory location that has WB memory type, the processor first ensures that the line is in Exclusive or Modified state in its own DCU. The processor looks for the cache line in the following locations, in the specified order:

1. DCU of initiating core
2. DCU of the other core and L2 cache
3. System memory

The cache line is taken from the DCU of the other core only if it is modified, ignoring the cache line availability or state in the L2 cache. After reading for ownership is completed, the data is written to the first-level data cache and the line is marked as modified.

Reading for ownership and storing the data happens after instruction retirement and follows the order of retirement. Therefore, the store latency does not effect the store instruction itself. However, several sequential stores may have cumulative latency that can affect performance. Table 2-4 presents store latencies depending on the initial cache line location.

2.2 INTEL NETBURST® MICROARCHITECTURE

The Pentium 4 processor, Pentium 4 processor Extreme Edition supporting Hyper-Threading Technology, Pentium D processor, and Pentium processor Extreme Edition implement the Intel NetBurst microarchitecture. Intel Xeon processors that implement Intel NetBurst microarchitecture can be identified using CPUID (family encoding 0FH).

This section describes the features of the Intel NetBurst microarchitecture and its operation common to the above processors. It provides the technical background required to understand optimization recommendations and the coding rules discussed in the rest of this manual. For implementation details, including instruction latencies, see Appendix C, “Instruction Latency and Throughput.”

Intel NetBurst microarchitecture is designed to achieve high performance for integer and floating-point computations at high clock rates. It supports the following features:

- hyper-pipelined technology that enables high clock rates

- high-performance, quad-pumped bus interface to the Intel NetBurst microarchitecture system bus
- rapid execution engine to reduce the latency of basic integer instructions
- out-of-order speculative execution to enable parallelism
- superscalar issue to enable parallelism
- hardware register renaming to avoid register name space limitations
- cache line sizes of 64 bytes
- hardware prefetch

2.2.1 Design Goals

The design goals of Intel NetBurst microarchitecture are:

- To execute legacy IA-32 applications and applications based on single-instruction, multiple-data (SIMD) technology at high throughput
- To operate at high clock rates and to scale to higher performance and clock rates in the future

Design advances of the Intel NetBurst microarchitecture include:

- A deeply pipelined design that allows for high clock rates (with different parts of the chip running at different clock rates).
- A pipeline that optimizes for the common case of frequently executed instructions; the most frequently-executed instructions in common circumstances (such as a cache hit) are decoded efficiently and executed with short latencies.
- Employment of techniques to hide stall penalties; Among these are parallel execution, buffering, and speculation. The microarchitecture executes instructions dynamically and out-of-order, so the time it takes to execute each individual instruction is not always deterministic.

Chapter 3, “General Optimization Guidelines,” lists optimizations to use and situations to avoid. The chapter also gives a sense of relative priority. Because most optimizations are implementation dependent, the chapter does not quantify expected benefits and penalties.

The following sections provide more information about key features of the Intel NetBurst microarchitecture.

2.2.2 Pipeline

The pipeline of the Intel NetBurst microarchitecture contains:

- an in-order issue front end
- an out-of-order superscalar execution core
- an in-order retirement unit

The front end supplies instructions in program order to the out-of-order core. It fetches and decodes instructions. The decoded instructions are translated into µops. The front end's primary job is to feed a continuous stream of µops to the execution core in original program order.

The out-of-order core aggressively reorders µops so that µops whose inputs are ready (and have execution resources available) can execute as soon as possible. The core can issue multiple µops per cycle.

The retirement section ensures that the results of execution are processed according to original program order and that the proper architectural states are updated.

Figure 2-4 illustrates a diagram of the major functional blocks associated with the Intel NetBurst microarchitecture pipeline. The following subsections provide an overview for each.

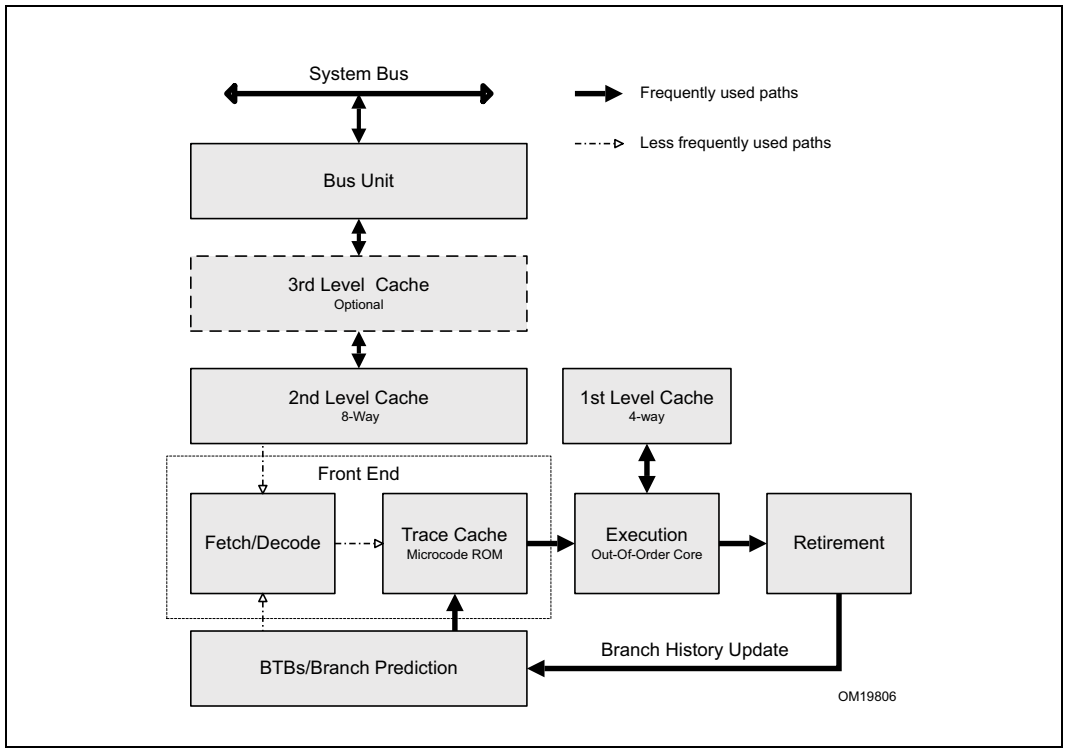

Figure 2-4. The Intel NetBurst Microarchitecture

2.2.2.1 Front End

The front end of the Intel NetBurst microarchitecture consists of two parts:

- fetch/decode unit
- execution trace cache

It performs the following functions:

- prefetches instructions that are likely to be executed
- fetches required instructions that have not been prefetched
- decodes instructions into µops
- generates microcode for complex instructions and special-purpose code
- delivers decoded instructions from the execution trace cache
- predicts branches using advanced algorithms

The front end is designed to address two problems that are sources of delay:

- time required to decode instructions fetched from the target
- wasted decode bandwidth due to branches or a branch target in the middle of a cache line

Instructions are fetched and decoded by a translation engine. The translation engine then builds decoded instructions into µop sequences called traces. Next, traces are then stored in the execution trace cache.

The execution trace cache stores µops in the path of program execution flow, where the results of branches in the code are integrated into the same cache line. This increases the instruction flow from the cache and makes better use of the overall cache storage space since the cache no longer stores instructions that are branched over and never executed.

The trace cache can deliver up to 3 µops per clock to the core.

The execution trace cache and the translation engine have cooperating branch prediction hardware. Branch targets are predicted based on their linear address using branch prediction logic and fetched as soon as possible. Branch targets are fetched from the execution trace cache if they are cached, otherwise they are fetched from the memory hierarchy. The translation engine's branch prediction information is used to form traces along the most likely paths.

2.2.2.2 Out-of-order Core

The core's ability to execute instructions out of order is a key factor in enabling parallelism. This feature enables the processor to reorder instructions so that if one µop is delayed while waiting for data or a contended resource, other µops that appear later in the program order may proceed. This implies that when one portion of the pipeline experiences a delay, the delay may be covered by other operations executing in parallel or by the execution of µops queued up in a buffer.

The core is designed to facilitate parallel execution. It can dispatch up to six µops per cycle through the issue ports (Figure 2-5). Note that six µops per cycle exceeds the trace cache and retirement µop bandwidth. The higher bandwidth in the core allows for peak bursts of greater than three µops and to achieve higher issue rates by allowing greater flexibility in issuing µops to different execution ports.

Most core execution units can start executing a new µop every cycle, so several instructions can be in flight at one time in each pipeline. A number of arithmetic logical unit (ALU) instructions can start at two per cycle; many floating-point instructions start one every two cycles. Finally, µops can begin execution out of program order, as soon as their data inputs are ready and resources are available.

2.2.2.3 Retirement

The retirement section receives the results of the executed µops from the execution core and processes the results so that the architectural state is updated according to the original program order. For semantically correct execution, the results of Intel 64 and IA-32 instructions must be committed in original program order before they are retired. Exceptions may be raised as instructions are retired. For this reason, exceptions cannot occur speculatively.

When a µop completes and writes its result to the destination, it is retired. Up to three µops may be retired per cycle. The reorder buffer (ROB) is the unit in the processor which buffers completed µops, updates the architectural state and manages the ordering of exceptions.

The retirement section also keeps track of branches and sends updated branch target information to the branch target buffer (BTB). This updates branch history.
Figure 2-9 illustrates the paths that are most frequently executing inside the Intel NetBurst microarchitecture: an execution loop that interacts with multilevel cache hierarchy and the system bus.

The following sections describe in more detail the operation of the front end and the execution core. This information provides the background for using the optimization techniques and instruction latency data documented in this manual.

2.2.3 Front End Pipeline Detail

The following information about the front end operation is be useful for tuning software with respect to prefetching, branch prediction, and execution trace cache operations.

2.2.3.1 Prefetching

The Intel NetBurst microarchitecture supports three prefetching mechanisms:

- a hardware instruction fetcher that automatically prefetches instructions

- a hardware mechanism that automatically fetches data and instructions into the unified second-level cache
- a mechanism fetches data only and includes two distinct components: (1) a hardware mechanism to fetch the adjacent cache line within a 128-byte sector that contains the data needed due to a cache line miss, this is also referred to as adjacent cache line prefetch (2) a software controlled mechanism that fetches data into the caches using the prefetch instructions.

The hardware instruction fetcher reads instructions along the path predicted by the branch target buffer (BTB) into instruction streaming buffers. Data is read in 32-byte chunks starting at the target address. The second and third mechanisms are described later.

2.2.3.2 Decoder

The front end of the Intel NetBurst microarchitecture has a single decoder that decodes instructions at the maximum rate of one instruction per clock. Some complex instructions must enlist the help of the microcode ROM. The decoder operation is connected to the execution trace cache.

2.2.3.3 Execution Trace Cache

The execution trace cache (TC) is the primary instruction cache in the Intel NetBurst microarchitecture. The TC stores decoded instructions (µops).

In the Pentium 4 processor implementation, TC can hold up to 12-Kbyte µops and can deliver up to three µops per cycle. TC does not hold all of the µops that need to be executed in the execution core. In some situations, the execution core may need to execute a microcode flow instead of the µop traces that are stored in the trace cache.

The Pentium 4 processor is optimized so that most frequently-executed instructions come from the trace cache while only a few instructions involve the microcode ROM.

2.2.3.4 Branch Prediction

Branch prediction is important to the performance of a deeply pipelined processor. It enables the processor to begin executing instructions long before the branch outcome is certain. Branch delay is the penalty that is incurred in the absence of correct prediction. For Pentium 4 and Intel Xeon processors, the branch delay for a correctly predicted instruction can be as few as zero clock cycles. The branch delay for a mispredicted branch can be many cycles, usually equivalent to the pipeline depth.

Branch prediction in the Intel NetBurst microarchitecture predicts near branches (conditional calls, unconditional calls, returns and indirect branches). It does not predict far transfers (far calls, irets and software interrupts).

Mechanisms have been implemented to aid in predicting branches accurately and to reduce the cost of taken branches. These include:

- ability to dynamically predict the direction and target of branches based on an instruction's linear address, using the branch target buffer (BTB)
- if no dynamic prediction is available or if it is invalid, the ability to statically predict the outcome based on the offset of the target: a backward branch is predicted to be taken, a forward branch is predicted to be not taken
- ability to predict return addresses using the 16-entry return address stack
- ability to build a trace of instructions across predicted taken branches to avoid branch penalties

The Static Predictor. Once a branch instruction is decoded, the direction of the branch (forward or backward) is known. If there was no valid entry in the BTB for the branch, the static predictor makes a prediction based on the direction of the branch. The static prediction mechanism predicts backward conditional branches (those with negative displacement, such as loop-closing branches) as taken. Forward branches are predicted not taken.

To take advantage of the forward-not-taken and backward-taken static predictions, code should be arranged so that the likely target of the branch immediately follows forward branches (see also Section 3.4.1, "Branch Prediction Optimization").

Branch Target Buffer. Once branch history is available, the Pentium 4 processor can predict the branch outcome even before the branch instruction is decoded. The processor uses a branch history table and a branch target buffer (collectively called the BTB) to predict the direction and target of branches based on an instruction's linear address. Once the branch is retired, the BTB is updated with the target address.

Return Stack. Returns are always taken; but since a procedure may be invoked from several call sites, a single predicted target does not suffice. The Pentium 4 processor has a Return Stack that can predict return addresses for a series of procedure calls. This increases the benefit of unrolling loops containing function calls. It also mitigates the need to put certain procedures inline since the return penalty portion of the procedure call overhead is reduced.

Even if the direction and target address of the branch are correctly predicted, a taken branch may reduce available parallelism in a typical processor (since the decode bandwidth is wasted for instructions which immediately follow the branch and precede the target, if the branch does not end the line and target does not begin the line). The branch predictor allows a branch and its target to coexist in a single trace cache line, maximizing instruction delivery from the front end.

2.2.4 Execution Core Detail

The execution core is designed to optimize overall performance by handling common cases most efficiently. The hardware is designed to execute frequent operations in a

common context as fast as possible, at the expense of infrequent operations using rare contexts.

Some parts of the core may speculate that a common condition holds to allow faster execution. If it does not, the machine may stall. An example of this pertains to store-to-load forwarding (see “Store Forwarding” in this chapter). If a load is predicted to be dependent on a store, it gets its data from that store and tentatively proceeds. If the load turned out not to depend on the store, the load is delayed until the real data has been loaded from memory, then it proceeds.

2.2.4.1 Instruction Latency and Throughput

The superscalar out-of-order core contains hardware resources that can execute multiple μops in parallel. The core’s ability to make use of available parallelism of execution units can enhanced by software’s ability to:

- Select instructions that can be decoded in less than 4 μops and/or have short latencies
- Order instructions to preserve available parallelism by minimizing long dependence chains and covering long instruction latencies
- Order instructions so that their operands are ready and their corresponding issue ports and execution units are free when they reach the scheduler

This subsection describes port restrictions, result latencies, and issue latencies (also referred to as throughput). These concepts form the basis to assist software for ordering instructions to increase parallelism. The order that μops are presented to the core of the processor is further affected by the machine’s scheduling resources.

It is the execution core that reacts to an ever-changing machine state, reordering μops for faster execution or delaying them because of dependence and resource constraints. The ordering of instructions in software is more of a suggestion to the hardware.

Appendix C, “Instruction Latency and Throughput,” lists some of the more-commonly-used Intel 64 and IA-32 instructions with their latency, their issue throughput, and associated execution units (where relevant). Some execution units are not pipelined (meaning that μops cannot be dispatched in consecutive cycles and the throughput is less than one per cycle). The number of μops associated with each instruction provides a basis for selecting instructions to generate. All μops executed out of the microcode ROM involve extra overhead.

2.2.4.2 Execution Units and Issue Ports

At each cycle, the core may dispatch μops to one or more of four issue ports. At the microarchitecture level, store operations are further divided into two parts: store data and store address operations. The four ports through which μops are dispatched to execution units and to load and store operations are shown in Figure 2-5. Some ports can dispatch two μops per clock. Those execution units are marked Double Speed.

Port 0. In the first half of the cycle, port 0 can dispatch either one floating-point move µop (a floating-point stack move, floating-point exchange or floating-point store data) or one arithmetic logical unit (ALU) µop (arithmetic, logic, branch or store data). In the second half of the cycle, it can dispatch one similar ALU µop.

Port 1. In the first half of the cycle, port 1 can dispatch either one floating-point execution (all floating-point operations except moves, all SIMD operations) µop or one normal-speed integer (multiply, shift and rotate) µop or one ALU (arithmetic) µop. In the second half of the cycle, it can dispatch one similar ALU µop.

Port 2. This port supports the dispatch of one load operation per cycle.

Port 3. This port supports the dispatch of one store address operation per cycle.

The total issue bandwidth can range from zero to six µops per cycle. Each pipeline contains several execution units. The µops are dispatched to the pipeline that corresponds to the correct type of operation. For example, an integer arithmetic logic unit and the floating-point execution units (adder, multiplier, and divider) can share a pipeline.

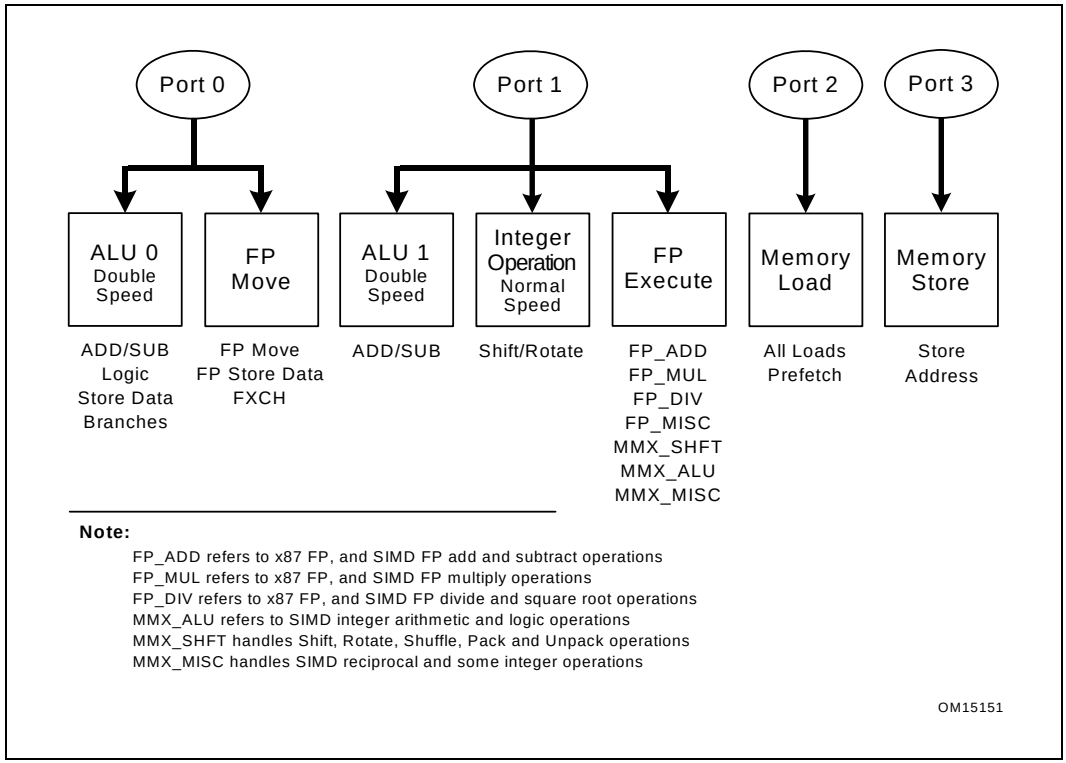

Figure 2-5. Execution Units and Ports in Out-Of-Order Core

2.2.4.3 Caches

The Intel NetBurst microarchitecture supports up to three levels of on-chip cache. At least two levels of on-chip cache are implemented in processors based on the Intel NetBurst microarchitecture. The Intel Xeon processor MP and selected Pentium and Intel Xeon processors may also contain a third-level cache.

The first level cache (nearest to the execution core) contains separate caches for instructions and data. These include the first-level data cache and the trace cache (an advanced first-level instruction cache). All other caches are shared between instructions and data.

Levels in the cache hierarchy are not inclusive. The fact that a line is in level *i* does not imply that it is also in level *i*+1. All caches use a pseudo-LRU (least recently used) replacement algorithm.

Table 2-5 provides parameters for all cache levels for Pentium and Intel Xeon Processors with CPUID model encoding equals 0, 1, 2 or 3.

Table 2-5. Pentium 4 and Intel Xeon Processor Cache Parameters

Level (Model)	Capacity	Associativity (ways)	Line Size (bytes)	Access Latency, Integer/ floating-point (clocks)	Write Update Policy
First (Model 0, 1, 2)	8 KB	4	64	2/9	write through
First (Model 3)	16 KB	8	64	4/12	write through
TC (All models)	12K µops	8	N/A	N/A	N/A
Second (Model 0, 1, 2)	256 KB or 512 KB[1]	8	64[2]	7/7	write back
Second (Model 3, 4)	1 MB	8	64[2]	18/18	write back
Second (Model 3, 4, 6)	2 MB	8	64[2]	20/20	write back
Third (Model 0, 1, 2)	0, 512 KB, 1 MB or 2 MB	8	64[2]	14/14	write back

NOTES:

1. Pentium 4 and Intel Xeon processors with CPUID model encoding value of 2 have a second level cache of 512 KB.
2. Each read due to a cache miss fetches a sector, consisting of two adjacent cache lines; a write operation is 64 bytes.

On processors without a third level cache, the second-level cache miss initiates a transaction across the system bus interface to the memory sub-system. On processors with a third level cache, the third-level cache miss initiates a transaction across the system bus. A bus write transaction writes 64 bytes to cacheable memory, or separate 8-byte chunks if the destination is not cacheable. A bus read transaction from cacheable memory fetches two cache lines of data.

The system bus interface supports using a scalable bus clock and achieves an effective speed that quadruples the speed of the scalable bus clock. It takes on the order of 12 processor cycles to get to the bus and back within the processor, and 6-12 bus cycles to access memory if there is no bus congestion. Each bus cycle equals several processor cycles. The ratio of processor clock speed to the scalable bus clock speed is referred to as bus ratio. For example, one bus cycle for a 100 MHz bus is equal to 15 processor cycles on a 1.50 GHz processor. Since the speed of the bus is implementation-dependent, consult the specifications of a given system for further details.

2.2.4.4 Data Prefetch

The Pentium 4 processor and other processors based on the NetBurst microarchitecture have two type of mechanisms for prefetching data: software prefetch instructions and hardware-based prefetch mechanisms.

Software controlled prefetch is enabled using the four prefetch instructions (PREFETCHh) introduced with SSE. The software prefetch is not intended for prefetching code. Using it can incur significant penalties on a multiprocessor system if code is shared.

Software prefetch can provide benefits in selected situations. These situations include when:

- the pattern of memory access operations in software allows the programmer to hide memory latency
- a reasonable choice can be made about how many cache lines to fetch ahead of the line being execute
- a choice can be made about the type of prefetch to use

SSE prefetch instructions have different behaviors, depending on cache levels updated and the processor implementation. For instance, a processor may implement the non-temporal prefetch by returning data to the cache level closest to the processor core. This approach has the following effect:

- minimizes disturbance of temporal data in other cache levels
- avoids the need to access off-chip caches, which can increase the realized bandwidth compared to a normal load-miss, which returns data to all cache levels

Situations that are less likely to benefit from software prefetch are:

- For cases that are already bandwidth bound, prefetching tends to increase bandwidth demands.

- Prefetching far ahead can cause eviction of cached data from the caches prior to the data being used in execution.
- Not prefetching far enough can reduce the ability to overlap memory and execution latencies.

Software prefetches are treated by the processor as a hint to initiate a request to fetch data from the memory system, and consume resources in the processor and the use of too many prefetches can limit their effectiveness. Examples of this include prefetching data in a loop for a reference outside the loop and prefetching in a basic block that is frequently executed, but which seldom precedes the reference for which the prefetch is targeted.

See: Chapter 7, “Optimizing Cache Usage.”

Automatic hardware prefetch is a feature in the Pentium 4 processor. It brings cache lines into the unified second-level cache based on prior reference patterns.

Software prefetching has the following characteristics:

- handles irregular access patterns, which do not trigger the hardware prefetcher
- handles prefetching of short arrays and avoids hardware prefetching start-up delay before initiating the fetches
- must be added to new code; so it does not benefit existing applications

Hardware prefetching for Pentium 4 processor has the following characteristics:

- works with existing applications
- does not require extensive study of prefetch instructions
- requires regular access patterns
- avoids instruction and issue port bandwidth overhead
- has a start-up penalty before the hardware prefetcher triggers and begins initiating fetches

The hardware prefetcher can handle multiple streams in either the forward or backward directions. The start-up delay and fetch-ahead has a larger effect for short arrays when hardware prefetching generates a request for data beyond the end of an array (not actually utilized). The hardware penalty diminishes if it is amortized over longer arrays.

Hardware prefetching is triggered after two successive cache misses in the last level cache and requires these cache misses to satisfy a condition that the linear address distance between these cache misses is within a threshold value. The threshold value depends on the processor implementation (see Table 2-6). However, hardware prefetching will not cross 4-KByte page boundaries. As a result, hardware prefetching can be very effective when dealing with cache miss patterns that have small strides and that are significantly less than half the threshold distance to trigger hardware prefetching. On the other hand, hardware prefetching will not benefit cache miss patterns that have frequent DTLB misses or have access strides that cause successive cache misses that are spatially apart by more than the trigger threshold distance.

Software can proactively control data access pattern to favor smaller access strides (e.g., stride that is less than half of the trigger threshold distance) over larger access strides (stride that is greater than the trigger threshold distance), this can achieve additional benefit of improved temporal locality and reducing cache misses in the last level cache significantly.

Software optimization of a data access pattern should emphasize tuning for hardware prefetch first to favor greater proportions of smaller-stride data accesses in the workload; before attempting to provide hints to the processor by employing software prefetch instructions.

2.2.4.5 Loads and Stores

The Pentium 4 processor employs the following techniques to speed up the execution of memory operations:

- speculative execution of loads
- reordering of loads with respect to loads and stores
- multiple outstanding misses
- buffering of writes
- forwarding of data from stores to dependent loads

Performance may be enhanced by not exceeding the memory issue bandwidth and buffer resources provided by the processor. Up to one load and one store may be issued for each cycle from a memory port reservation station. In order to be dispatched to a reservation station, there must be a buffer entry available for each memory operation. There are 48 load buffers and 24 store buffers[3]. These buffers hold the μop and address information until the operation is completed, retired, and deallocated.

The Pentium 4 processor is designed to enable the execution of memory operations out of order with respect to other instructions and with respect to each other. Loads can be carried out speculatively, that is, before all preceding branches are resolved. However, speculative loads cannot cause page faults.

Reordering loads with respect to each other can prevent a load miss from stalling later loads. Reordering loads with respect to other loads and stores to different addresses can enable more parallelism, allowing the machine to execute operations as soon as their inputs are ready. Writes to memory are always carried out in program order to maintain program correctness.

A cache miss for a load does not prevent other loads from issuing and completing. The Pentium 4 processor supports up to four (or eight for Pentium 4 processor with CPUID signature corresponding to family 15, model 3) outstanding load misses that can be serviced either by on-chip caches or by memory.

3. Pentium 4 processors with CPUID model encoding equal to 3 have more than 24 store buffers.

Store buffers improve performance by allowing the processor to continue executing instructions without having to wait until a write to memory and/or cache is complete. Writes are generally not on the critical path for dependence chains, so it is often beneficial to delay writes for more efficient use of memory-access bus cycles.

2.2.4.6 Store Forwarding

Loads can be moved before stores that occurred earlier in the program if they are not predicted to load from the same linear address. If they do read from the same linear address, they have to wait for the store data to become available. However, with store forwarding, they do not have to wait for the store to write to the memory hierarchy and retire. The data from the store can be forwarded directly to the load, as long as the following conditions are met:

- **Sequence** — Data to be forwarded to the load has been generated by a programmatically-earlier store which has already executed.
- **Size** — Bytes loaded must be a subset of (including a proper subset, that is, the same) bytes stored.
- **Alignment** — The store cannot wrap around a cache line boundary, and the linear address of the load must be the same as that of the store.

2.3 INTEL® PENTIUM® M PROCESSOR MICROARCHITECTURE

Like the Intel NetBurst microarchitecture, the pipeline of the Intel Pentium M processor microarchitecture contains three sections:

- in-order issue front end
- out-of-order superscalar execution core
- in-order retirement unit

Intel Pentium M processor microarchitecture supports a high-speed system bus (up to 533 MHz) with 64-byte line size. Most coding recommendations that apply to the Intel NetBurst microarchitecture also apply to the Intel Pentium M processor.

The Intel Pentium M processor microarchitecture is designed for lower power consumption. There are other specific areas of the Pentium M processor microarchitecture that differ from the Intel NetBurst microarchitecture. They are described next. A block diagram of the Intel Pentium M processor is shown in Figure 2-6.

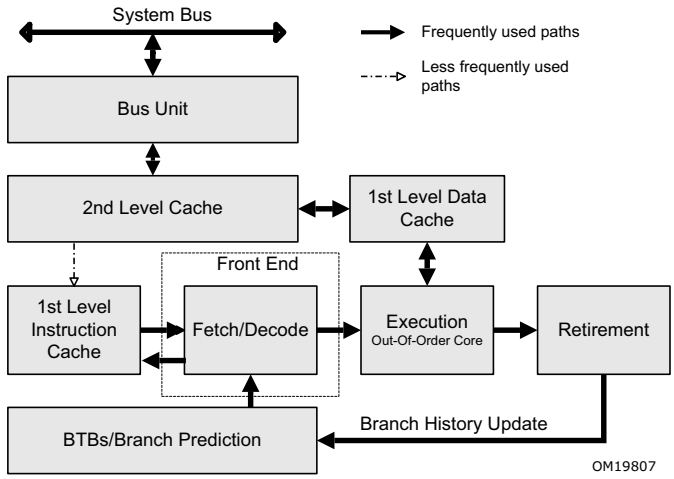

Figure 2-6. The Intel Pentium M Processor Microarchitecture

2.3.1 Front End

The Intel Pentium M processor uses a pipeline depth that enables high performance and low power consumption. It's shorter than that of the Intel NetBurst microarchitecture.

The Intel Pentium M processor front end consists of two parts:

- fetch/decode unit
- instruction cache

The fetch and decode unit includes a hardware instruction prefetcher and three decoders that enable parallelism. It also provides a 32-KByte instruction cache that stores un-decoded binary instructions.

The instruction prefetcher fetches instructions in a linear fashion from memory if the target instructions are not already in the instruction cache. The prefetcher is designed to fetch efficiently from an aligned 16-byte block. If the modulo 16 remainder of a branch target address is 14, only two useful instruction bytes are fetched in the first cycle. The rest of the instruction bytes are fetched in subsequent cycles.

The three decoders decode instructions and break them down into µops. In each clock cycle, the first decoder is capable of decoding an instruction with four or fewer

µops. The remaining two decoders each decode a one µop instruction in each clock cycle.

The front end can issue multiple µops per cycle, in original program order, to the out-of-order core.

The Intel Pentium M processor incorporates sophisticated branch prediction hardware to support the out-of-order core. The branch prediction hardware includes dynamic prediction, and branch target buffers.

The Intel Pentium M processor has enhanced dynamic branch prediction hardware. Branch target buffers (BTB) predict the direction and target of branches based on an instruction's address.

The Pentium M Processor includes two techniques to reduce the execution time of certain operations:

- **ESP folding** — This eliminates the ESP manipulation µops in stack-related instructions such as PUSH, POP, CALL and RET. It increases decode rename and retirement throughput. ESP folding also increases execution bandwidth by eliminating µops which would have required execution resources.
- **Micro-ops (µops) fusion** — Some of the most frequent pairs of µops derived from the same instruction can be fused into a single µops. The following categories of fused µops have been implemented in the Pentium M processor:
  - — "Store address" and "store data" µops are fused into a single "Store" µop. This holds for all types of store operations, including integer, floating-point, MMX technology, and Streaming SIMD Extensions (SSE and SSE2) operations.
  - — A load µop in most cases can be fused with a successive execution µop.This holds for integer, floating-point and MMX technology loads and for most kinds of successive execution operations. Note that SSE Loads can not be fused.

2.3.2 Data Prefetching

The Intel Pentium M processor supports three prefetching mechanisms:

- The first mechanism is a hardware instruction fetcher and is described in the previous section.
- The second mechanism automatically fetches data into the second-level cache. The implementation of automatic hardware prefetching in Pentium M processor family is basically similar to those described for NetBurst microarchitecture. The trigger threshold distance for each relevant processor models is shown in Table 2-6. The third mechanism is a software mechanism that fetches data into the caches using the prefetch instructions.

Table 2-6. Trigger Threshold and CPUID Signatures for Processor Families

Trigger Threshold Distance (Bytes)	Extended Model ID	Extended Family ID	Family ID	Model ID
512	0	0	15	3, 4, 6
256	0	0	15	0, 1, 2
256	0	0	6	9, 13, 14

Data is fetched 64 bytes at a time; the instruction and data translation lookaside buffers support 128 entries. See Table 2-7 for processor cache parameters.

Table 2-7. Cache Parameters of Pentium M, Intel Core Solo, and Intel Core Duo Processors

Level	Capacity	Associativity (ways)	Line Size (bytes)	Access Latency (clocks)	Write Update Policy
First	32 KByte	8	64	3	Writeback
Instruction	32 KByte	8	N/A	N/A	N/A
Second (mode 9)	1 MByte	8	64	9	Writeback
Second (model 13)	2 MByte	8	64	10	Writeback
Second (model 14)	2 MByte	8	64	14	Writeback

2.3.3 Out-of-Order Core

The processor core dynamically executes µops independent of program order. The core is designed to facilitate parallel execution by employing many buffers, issue ports, and parallel execution units.

The out-of-order core buffers µops in a Reservation Station (RS) until their operands are ready and resources are available. Each cycle, the core may dispatch up to five µops through the issue ports.

2.3.4 In-Order Retirement

The retirement unit in the Pentium M processor buffers completed µops is the reorder buffer (ROB). The ROB updates the architectural state in order. Up to three µops may be retired per cycle.

2.4 MICROARCHITECTURE OF INTEL® CORE™ SOLO AND INTEL® CORE™ DUO PROCESSORS

Intel Core Solo and Intel Core Duo processors incorporate an microarchitecture that is similar to the Pentium M processor microarchitecture, but provides additional enhancements for performance and power efficiency. Enhancements include:

- **Intel Smart Cache** — This second level cache is shared between two cores in an Intel Core Duo processor to minimize bus traffic between two cores accessing a single-copy of cached data. It allows an Intel Core Solo processor (or when one of the two cores in an Intel Core Duo processor is idle) to access its full capacity.
- **Stream SIMD Extensions 3** — These extensions are supported in Intel Core Solo and Intel Core Duo processors.
- **Decoder improvement** — Improvement in decoder and μop fusion allows the front end to see most instructions as single μop instructions. This increases the throughput of the three decoders in the front end.
- **Improved execution core** — Throughput of SIMD instructions is improved and the out-of-order engine is more robust in handling sequences of frequently-used instructions. Enhanced internal buffering and prefetch mechanisms also improve data bandwidth for execution.
- **Power-optimized bus** — The system bus is optimized for power efficiency; increased bus speed supports 667 MHz.
- **Data Prefetch** — Intel Core Solo and Intel Core Duo processors implement improved hardware prefetch mechanisms: one mechanism can look ahead and prefetch data into L1 from L2. These processors also provide enhanced hardware prefetchers similar to those of the Pentium M processor (see Table 2-6).

2.4.1 Front End

Execution of SIMD instructions on Intel Core Solo and Intel Core Duo processors are improved over Pentium M processors by the following enhancements:

- **Micro-op fusion** — Scalar SIMD operations on register and memory have single μop flows comparable to X87 flows. Many packed instructions are fused to reduce its μop flow from four to two μops.
- **Eliminating decoder restrictions** — Intel Core Solo and Intel Core Duo processors improve decoder throughput with micro-fusion and macro-fusion, so that many more SSE and SSE2 instructions can be decoded without restriction. On Pentium M processors, many single μop SSE and SSE2 instructions must be decoded by the main decoder.
- **Improved packed SIMD instruction decoding** — On Intel Core Solo and Intel Core Duo processors, decoding of most packed SSE instructions is done by all three decoders. As a result the front end can process up to three packed SSE instructions every cycle. There are some exceptions to the above; some shuffle/unpack/shift operations are not fused and require the main decoder.

2.4.2 Data Prefetching

Intel Core Solo and Intel Core Duo processors provide hardware mechanisms to prefetch data from memory to the second-level cache. There are two techniques:

1. One mechanism activates after the data access pattern experiences two cache-reference misses within a trigger-distance threshold (see Table 2-6). This mechanism is similar to that of the Pentium M processor, but can track 16 forward data streams and 4 backward streams.
2. The second mechanism fetches an adjacent cache line of data after experiencing a cache miss. This effectively simulates the prefetching capabilities of 128-byte sectors (similar to the sectoring of two adjacent 64-byte cache lines available in Pentium 4 processors).

Hardware prefetch requests are queued up in the bus system at lower priority than normal cache-miss requests. If bus queue is in high demand, hardware prefetch requests may be ignored or cancelled to service bus traffic required by demand cache-misses and other bus transactions. Hardware prefetch mechanisms are enhanced over that of Pentium M processor by:

- Data stores that are not in the second-level cache generate read for ownership requests. These requests are treated as loads and can trigger a prefetch stream.
- Software prefetch instructions are treated as loads, they can also trigger a prefetch stream.

2.5 INTEL® HYPER-THREADING TECHNOLOGY

Intel® Hyper-Threading Technology (HT Technology) is supported by specific members of the Intel Pentium 4 and Xeon processor families. The technology enables software to take advantage of task-level, or thread-level parallelism by providing multiple logical processors within a physical processor package. In its first implementation in Intel Xeon processor, Hyper-Threading Technology makes a single physical processor appear as two logical processors.

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

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

Figure 2-7 shows a typical bus-based symmetric multiprocessor (SMP) based on processors supporting HT Technology. Each logical processor can execute a software thread, allowing a maximum of two software threads to execute simultaneously on one physical processor. The two software threads execute simultaneously, meaning that in the same clock cycle an “add” operation from logical processor 0 and another

"add" operation and load from logical processor 1 can be executed simultaneously by the execution engine.

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

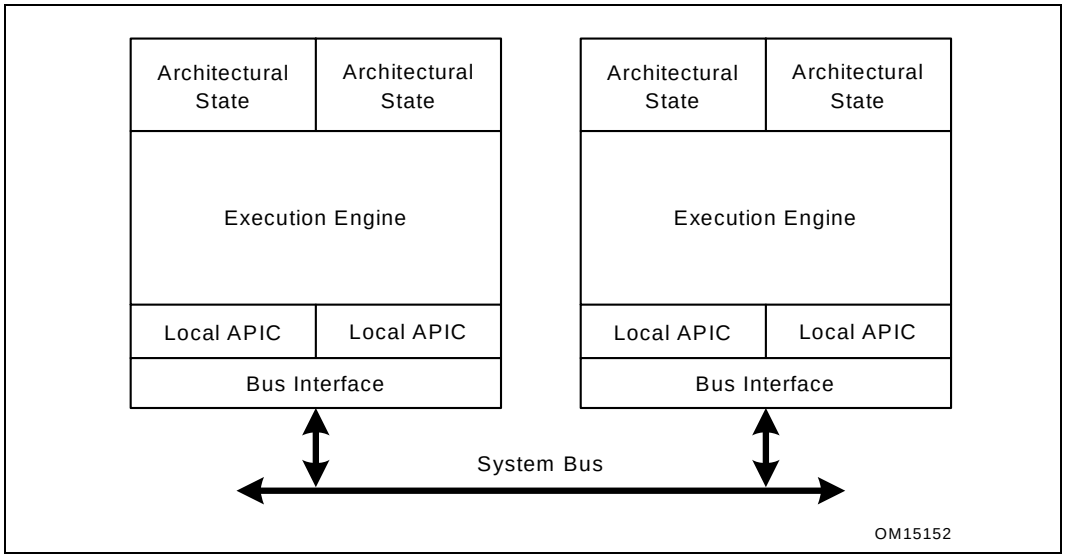

Figure 2-7. Hyper-Threading Technology on an SMP

The performance potential due to HT Technology is due to:

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

2.5.1 Processor Resources and HT Technology

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

2.5.1.1 Replicated Resources

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

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

In addition, a few buffers (for example, the 2-entry instruction streaming buffers) were replicated to reduce complexity.

2.5.1.2 Partitioned Resources

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

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

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

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

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

2.5.1.3 Shared Resources

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

The first level cache can operate in two modes depending on a context-ID bit:

- Shared mode: The L1 data cache is fully shared by two logical processors.
- Adaptive mode: In adaptive mode, memory accesses using the page directory is mapped identically across logical processors sharing the L1 data cache.

The other resources are fully shared.

2.5.2 Microarchitecture Pipeline and HT Technology

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

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

2.5.3 Front End Pipeline

The execution trace cache is shared between two logical processors. Execution trace cache access is arbitrated by the two logical processors every clock. If a cache line is fetched for one logical processor in one clock cycle, the next clock cycle a line would be fetched for the other logical processor provided that both logical processors are requesting access to the trace cache.

If one logical processor is stalled or is unable to use the execution trace cache, the other logical processor can use the full bandwidth of the trace cache until the initial logical processor's instruction fetches return from the L2 cache.

After fetching the instructions and building traces of µops, the µops are placed in a queue. This queue decouples the execution trace cache from the register rename pipeline stage. As described earlier, if both logical processors are active, the queue is partitioned so that both logical processors can make independent forward progress.

2.5.4 Execution Core

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

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

2.5.5 Retirement

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

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

2.6 MULTICORE PROCESSORS

The Intel Pentium D processor and the Pentium Processor Extreme Edition introduce multicore features. These processors enhance hardware support for multithreading by providing two processor cores in each physical processor package. The Dual-core Intel Xeon and Intel Core Duo processors also provide two processor cores in a physical package. The multicore topology of Intel Core 2 Duo processors are similar to those of Intel Core Duo processor.

The Intel Pentium D processor provides two logical processors in a physical package, each logical processor has a separate execution core and a cache hierarchy. The Dual-core Intel Xeon processor and the Intel Pentium Processor Extreme Edition provide four logical processors in a physical package that has two execution cores. Each core provides two logical processors sharing an execution core and a cache hierarchy.

The Intel Core Duo processor provides two logical processors in a physical package. Each logical processor has a separate execution core (including first-level cache) and a smart second-level cache. The second-level cache is shared between two logical processors and optimized to reduce bus traffic when the same copy of cached data is used by two logical processors. The full capacity of the second-level cache can be used by one logical processor if the other logical processor is inactive.

The functional blocks of these processors are shown in Figure 2-8.

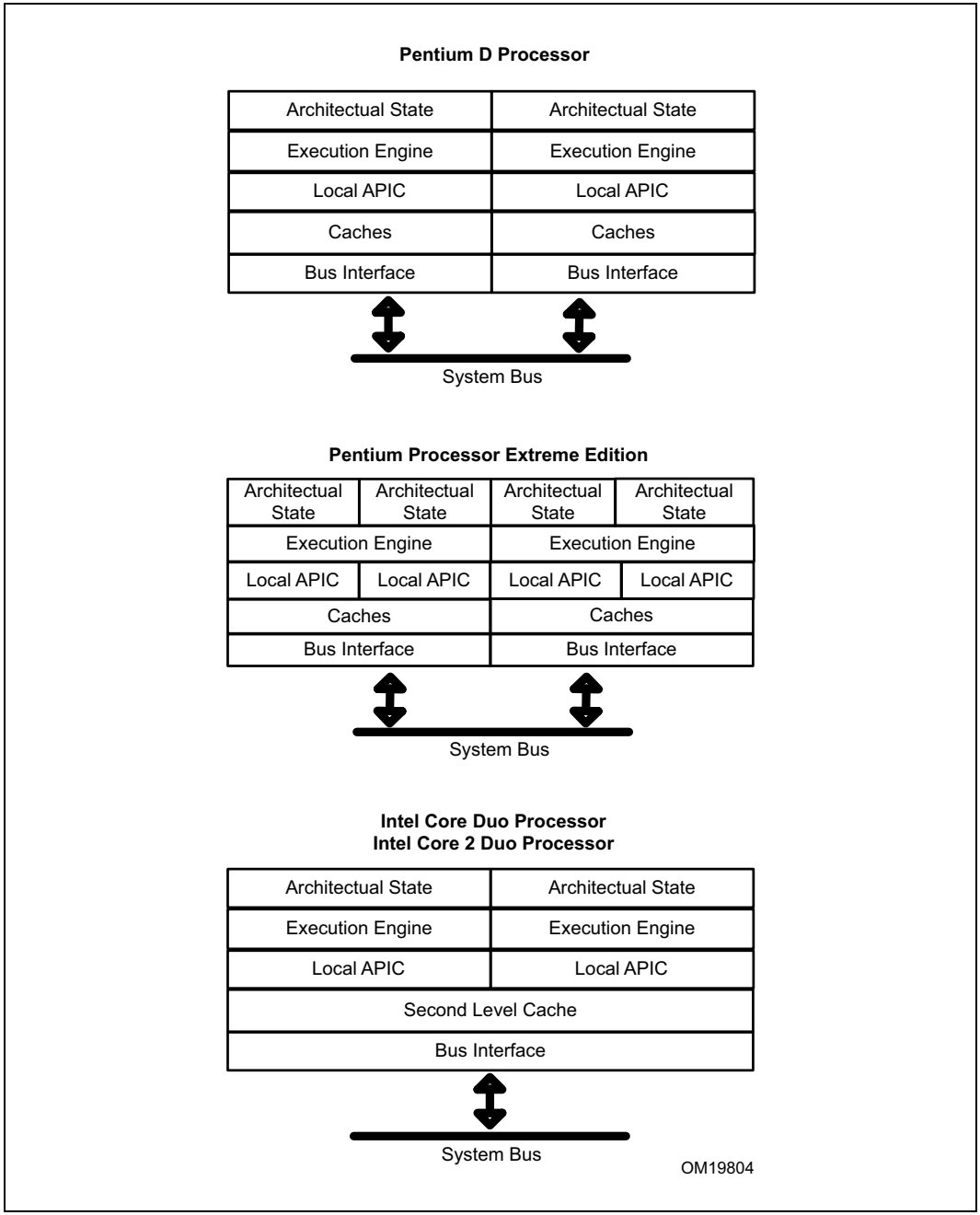

Figure 2-8. Pentium D Processor, Pentium Processor Extreme Edition, Intel Core Duo Processor, and Intel Core 2 Duo Processor

2.6.1 Microarchitecture Pipeline and MultiCore Processors

In general, each core in a multicore processor resembles a single-core processor implementation of the underlying microarchitecture. The implementation of the cache hierarchy in a dual-core or multicore processor may be the same or different from the cache hierarchy implementation in a single-core processor.

CPUID should be used to determine cache-sharing topology information in a processor implementation and the underlying microarchitecture. The former is obtained by querying the deterministic cache parameter leaf (see Chapter 7, "Optimizing Cache Usage"); the latter by using the encoded values for extended family, family, extended model, and model fields. See Table 2-8.

Table 2-8. Family And Model Designations of Microarchitectures

Dual-Core Processor	Micro-architecture	Extended Family	Family	Extended Model	Model
Pentium D processor	NetBurst	0	15	0	3, 4, 6
Pentium processor Extreme Edition	NetBurst	0	15	0	3, 4, 6
Intel Core Duo processor	Improved Pentium M	0	6	0	14
Intel Core 2 Duo processor/ Intel Xeon processor 5100	Intel Core Microarchitecture	0	6	0	15

2.6.2 Shared Cache in Intel® Core™ Duo Processors

The Intel Core Duo processor has two symmetric cores that share the second-level cache and a single bus interface (see Figure 2-8). Two threads executing on two cores in an Intel Core Duo processor can take advantage of shared second-level cache, accessing a single-copy of cached data without generating bus traffic.

2.6.2.1 Load and Store Operations

When an instruction needs to read data from a memory address, the processor looks for it in caches and memory. When an instruction writes data to a memory location (write back) the processor first makes sure that the cache line that contains the memory location is owned by the first-level data cache of the initiating core (that is,

the line is in exclusive or modified state). Then the processor looks for the cache line in the cache and memory sub-systems. The look-ups for the locality of load or store operation are in the following order:

1. DCU of the initiating core
2. DCU of the other core and second-level cache
3. System memory

The cache line is taken from the DCU of the other core only if it is modified, ignoring the cache line availability or state in the L2 cache. Table 2-9 lists the performance characteristics of generic load and store operations in an Intel Core Duo processor. Numeric values of Table 2-9 are in terms of processor core cycles.

Table 2-9. Characteristics of Load and Store Operations in Intel Core Duo Processors

	Load		Store	
Data Locality	**Latency**	**Throughput**	**Latency**	**Throughput**
DCU	3	1	2	1
DCU of the other core in "Modified" state	14 + bus transaction	14 + bus transaction	14 + bus transaction	~10
2nd-level cache	14	<6	14	<6
Memory	14 + bus transaction	Bus read protocol	14 + bus transaction	Bus write protocol

Throughput is expressed as the number of cycles to wait before the same operation can start again. The latency of a bus transaction is exposed in some of these operations, as indicated by entries containing "+ bus transaction". On Intel Core Duo processors, a typical bus transaction may take 5.5 bus cycles. For a 667 MHz bus and a core frequency of 2.167GHz, the total of 14 + 5.5 * 2167 /(667/4) ~ 86 core cycles.

Sometimes a modified cache line has to be evicted to make room for a new cache line. The modified cache line is evicted in parallel to bringing in new data and does not require additional latency. However, when data is written back to memory, the eviction consumes cache bandwidth and bus bandwidth. For multiple cache misses that require the eviction of modified lines and are within a short time, there is an overall degradation in response time of these cache misses.

For store operation, reading for ownership must be completed before the data is written to the first-level data cache and the line is marked as modified. Reading for ownership and storing the data happens after instruction retirement and follows the order of retirement. The bus store latency does not affect the store instruction itself. However, several sequential stores may have cumulative latency that can effect performance.

2.7 INTEL® 64 ARCHITECTURE

Intel 64 architecture supports almost all features in the IA-32 Intel architecture and extends support to run 64-bit OS and 64-bit applications in 64-bit linear address space. Intel 64 architecture provides a new operating mode, referred to as IA-32e mode, and increases the linear address space for software to 64 bits and supports physical address space up to 40 bits.

IA-32e mode consists of two sub-modes: (1) compatibility mode enables a 64-bit operating system to run most legacy 32-bit software unmodified, (2) 64-bit mode enables a 64-bit operating system to run applications written to access 64-bit linear address space.

In the 64-bit mode of Intel 64 architecture, software may access:

- 64-bit flat linear addressing
- 8 additional general-purpose registers (GPRs)
- 8 additional registers for streaming SIMD extensions (SSE, SSE2, SSE3 and SSSE3)
- 64-bit-wide GPRs and instruction pointers
- uniform byte-register addressing
- fast interrupt-prioritization mechanism
- a new instruction-pointer relative-addressing mode

For optimizing 64-bit applications, the features that impact software optimizations include:

- using a set of prefixes to access new registers or 64-bit register operand
- pointer size increases from 32 bits to 64 bits
- instruction-specific usages

2.8 SIMD TECHNOLOGY

SIMD computations (see Figure 2-9) were introduced to the architecture with MMX technology. MMX technology allows SIMD computations to be performed on packed byte, word, and doubleword integers. The integers are contained in a set of eight 64-bit registers called MMX registers (see Figure 2-10).

The Pentium III processor extended the SIMD computation model with the introduction of the Streaming SIMD Extensions (SSE). SSE allows SIMD computations to be performed on operands that contain four packed single-precision floating-point data elements. The operands can be in memory or in a set of eight 128-bit XMM registers (see Figure 2-10). SSE also extended SIMD computational capability by adding additional 64-bit MMX instructions.

Figure 2-9 shows a typical SIMD computation. Two sets of four packed data elements (X1, X2, X3, and X4, and Y1, Y2, Y3, and Y4) are operated on in parallel, with the

same operation being performed on each corresponding pair of data elements (X1 and Y1, X2 and Y2, X3 and Y3, and X4 and Y4). The results of the four parallel computations are sorted as a set of four packed data elements.

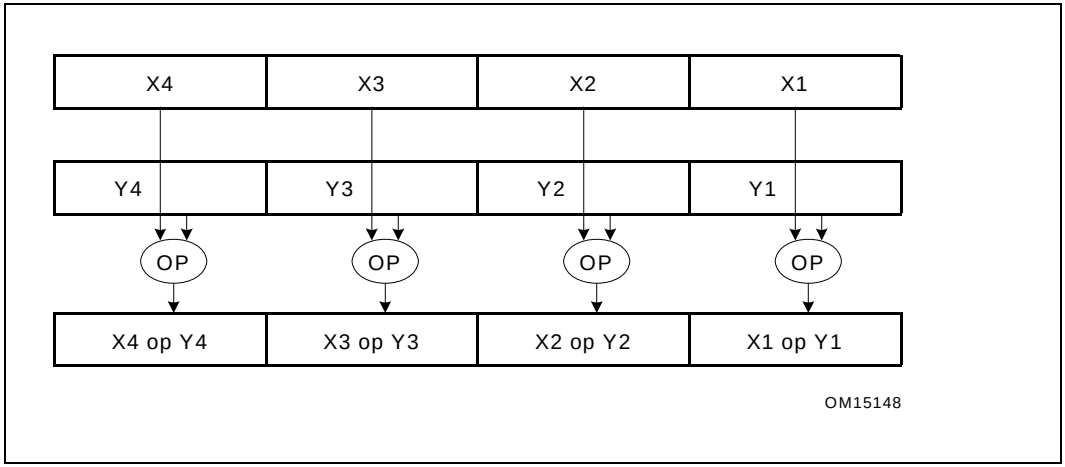

Figure 2-9. Typical SIMD Operations

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

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

SSE3 enhances x87, SSE and SSE2 by providing 13 instructions that can accelerate application performance in specific areas. These include video processing, complex arithmetics, and thread synchronization. SSE3 complements SSE and SSE2 with instructions that process SIMD data asymmetrically, facilitate horizontal computation, and help avoid loading cache line splits. See Figure 2-10.

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

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

- 128-bit SIMD instructions referencing XMM register can access 16 XMM registers in 64-bit mode.

- Instructions that reference 32-bit general purpose registers can access 16 general purpose registers in 64-bit mode.

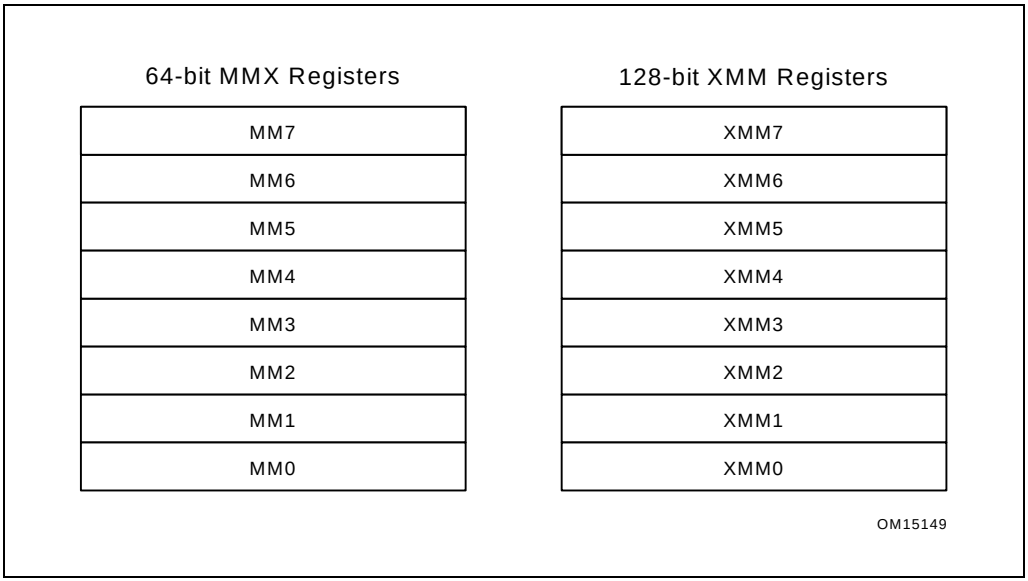

Figure 2-10. SIMD Instruction Register Usage

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

- inherently parallel
- recurring memory access patterns
- localized recurring operations performed on the data
- data-independent control flow

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

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

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

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

- Chapter 9, "Programming with Intel® MMX™ Technology"
- Chapter 10, "Programming with Streaming SIMD Extensions (SSE)"
- Chapter 11, "Programming with Streaming SIMD Extensions 2 (SSE2)"
- Chapter 12, "Programming with SSE3 and Supplemental SSE3"

2.8.1 Summary of SIMD Technologies

2.8.1.1 MMX™ Technology

MMX Technology introduced:

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

MMX instructions are useful for multimedia and communications software.

2.8.1.2 Streaming SIMD Extensions

Streaming SIMD extensions introduced:

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

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

2.8.1.3 Streaming SIMD Extensions 2

Streaming SIMD extensions 2 add the following:

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

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

2.8.1.4 Streaming SIMD Extensions 3

Streaming SIMD extensions 3 add the following:

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

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

2.8.1.5 Supplemental Streaming SIMD Extensions 3

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

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