

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 Intel® microarchitecture code name Broadwell, Intel® microarchitecture code name Haswell, Intel microarchitecture code name Ivy Bridge, Intel microarchitecture code name Sandy Bridge, processors based on the Intel Core microarchitecture, Enhanced Intel Core microarchitecture, Intel microarchitecture code name Nehalem). 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 across Intel Core processor and Intel Xeon processor families
- Hyper-Threading Technology¹ (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), Supplemental Streaming SIMD Extensions 3 (SSSE3), SSE4.1, and SSE4.2.
- Intel® Advanced Vector Extensions (Intel® AVX).
- Half-precision floating-point conversion and RDRAND
- Fused Multiply Add Extensions
- Intel® AVX2
- ADX and RDSEED

The Intel Core 2, Intel Core 2 Extreme, Intel Core 2 Quad processor family, Intel Xeon processor 3000, 3200, 5100, 5300, 7300 series are based on the high-performance and power-efficient Intel Core microarchitecture. Intel Xeon processor 3100, 3300, 5200, 5400, 7400 series, Intel Core 2 Extreme processor QX9600, QX9700 series, Intel Core 2 Quad Q9000 series, Q8000 series are based on the enhanced Intel Core microarchitecture. Intel Core i7 processor is based on Intel microarchitecture code name Nehalem. Intel® Xeon® processor 5600 series, Intel Xeon processor E7 and Intel Core i7, i5, i3 processors are based on Intel microarchitecture code name Westmere.

The Intel® Xeon® processor E5 family, Intel® Xeon® processor E3-1200 family, Intel® Xeon® processor E7-8800/4800/2800 product families, Intel® Core™ i7-3930K processor, and 2nd generation Intel® Core™ i7-2xxx, Intel® Core™ i5-2xxx, Intel® Core™ i3-2xxx processor series are based on the Intel® microarchitecture code name Sandy Bridge.

The Intel® Xeon® processor E3-1200 v2 product family and the 3rd generation Intel® Core™ processors are based on the Ivy Bridge microarchitecture and support Intel 64 architecture. The Intel® Xeon® processor E5 v2 and E7 v2 families are based on the Ivy Bridge-E microarchitecture, support Intel 64 architecture and multiple physical processor packages in a platform.

The Intel® Xeon® processor E3-1200 v3 product family and 4th Generation Intel® Core™ processors are based on the Haswell microarchitecture and support Intel 64 architecture. The Intel® Xeon® processor E5 26xx v3 family is based on the Haswell-E microarchitecture, supports Intel 64 architecture and multiple physical processor packages in a platform.

Intel® Core™ M processors are based on the Broadwell microarchitecture and support Intel 64 architecture.

1. 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 THE HASWELL MICROARCHITECTURE

The Haswell microarchitecture builds on the successes of the Sandy Bridge and Ivy Bridge microarchitectures. The basic pipeline functionality of the Haswell microarchitecture is depicted in Figure 2-1. In general, Most of the features described in Section 2.1.1 - Section 2.1.4 also apply to the Broadwell microarchitecture. Enhancements of the Broadwell microarchitecture is summarized in Section 2.1.6.

The Haswell microarchitecture offers the following innovative features:

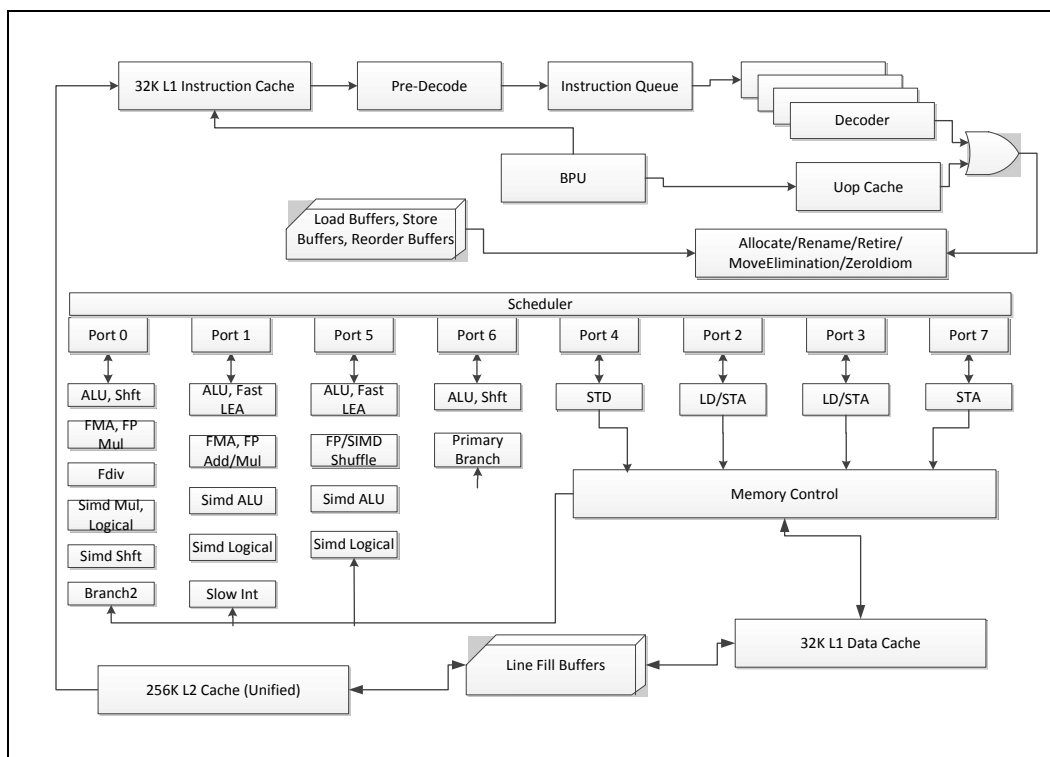


Figure 2-1. CPU Core Pipeline Functionality of the Haswell Microarchitecture

- Support for Intel® Advanced Vector Extensions 2 (AVX2), FMA
- Support for general-purpose, new instructions to accelerate integer numeric, encryption.
- Support for Intel Transactional Synchronization Extensions (TSX)
- Each core can dispatch up to 8 micro-ops per cycle
- 256-bit data path for memory operation, FMA, AVX floating-point and AVX2 integer execution units
- Improved L1D and L2 cache bandwidth
- Two FMA execution pipelines
- Four arithmetic logical units (ALUs)
- Three store address ports
- Two branch execution units
- Advanced power management features for IA processor core and uncore sub-systems
- Support for optional fourth level cache

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

example of the system integration view of four CPU cores with uncore components is illustrated in Figure 2-2.

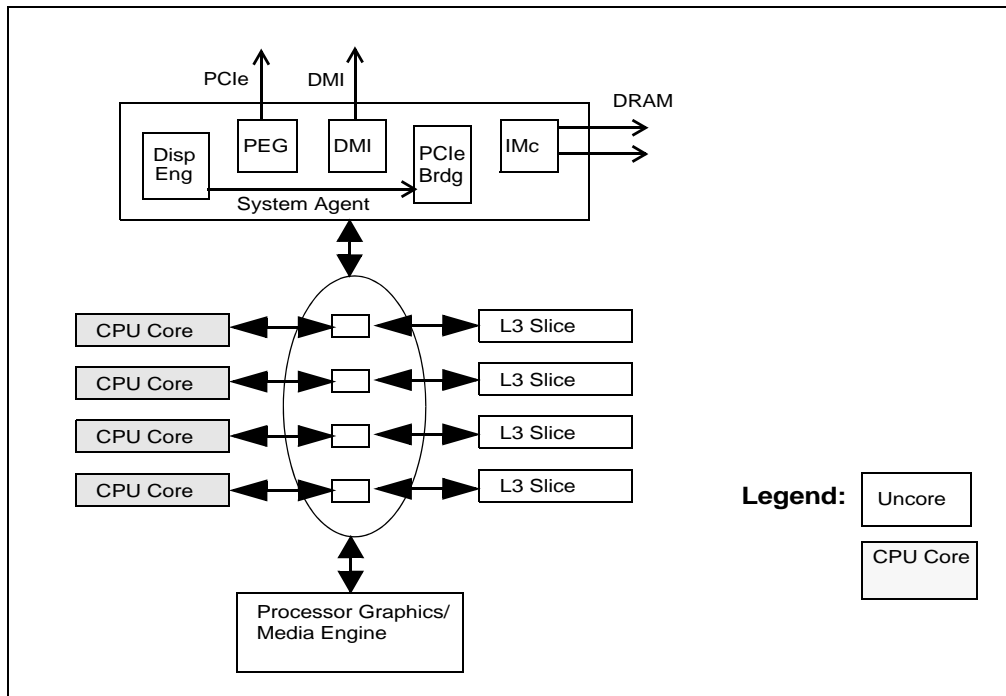


Figure 2-2. Four Core System Integration of Intel Microarchitecture Haswell

2.1.1 The Front End

The front end of Intel microarchitecture code name Haswell builds on that of Intel microarchitecture code name Sandy Bridge and Intel microarchitecture code name Ivy Bridge, see Section 2.2.2 and Section 2.2.7. Additional enhancement in the front end include:

- The uop cache (or decoded ICache) is partitioned equally between two logical processors,
- The instruction decoders will alternate between each active logical processor. If one sibling logical processor is idle, the active logical processor will use the decoders continuously.
- The LSD/micro-op queue can detect small loops up to 56 micro-ops. The 56-entry micro-op queue is shared by two logical processors if Hyper-Threading Technology is active (Intel microarchitecture Sandy Bridge provides duplicated 28-entry micro-op queue in each core).

2.1.2 The Out-of-Order Engine

The key components and significant improvements to the out-of-order engine are summarized below:

Renamer: The Renamer moves micro-ops from the micro-op queue to bind to the dispatch ports in the Scheduler with execution resources. Zero-idiom, one-idiom and zero-latency register move operations are performed by the Renamer to free up the Scheduler and execution core for improved performance.

Scheduler: The Scheduler controls the dispatch of micro-ops onto the dispatch ports. There are eight dispatch ports to support the out-of-order execution core. Four of the eight ports provided execution resources for computational operations. The other 4 ports support memory operations of up to two 256-bit load and one 256-bit store operation in a cycle.

Execution Core: The scheduler can dispatch up to eight micro-ops every cycle, one on each port. Of the four ports providing computational resources, each provides an ALU, two of these execution pipes provided dedicated FMA units. With the exception of division/square-root, STTNI/AESNI units, most

floating-point and integer SIMD execution units are 256-bit wide. The four dispatch ports servicing memory operations consist with two dual-use ports for load and store-address operation. Plus a dedicated 3rd store-address port and one dedicated store-data port. All memory ports can handle 256-bit memory micro-ops. Peak floating-point throughput, at 32 single-precision operations per cycle and 16 double-precision operations per cycle using FMA, is twice that of Intel microarchitecture code name Sandy Bridge.

The out-of-order engine can handle 192 uops in flight compared to 168 in Intel microarchitecture code name Sandy Bridge.

2.1.3 Execution Engine

The following table summarizes which operations can be dispatched on which port.

Table 2-1. Dispatch Port and Execution Stacks

Port 0	Port 1	Port 2, 3	Port 4	Port 5	Port 6	Port 7
ALU, Shift	ALU, Fast LEA,	Load_Addr, Store_addr	Store_data	ALU, Fast LEA	ALU, Shift, JEU	Store_addr, Simple_AGU
SIMD_Log, STTNI, SIMD_Shifts	SIMD_ALU, SIMD_Log			SIMD_ALU, SIMD_Log		
FMA/FP_mul, Div	FMA/FP_mul, FP_add			FP/Int Shuffle		
2nd_Jeu	slow_int					

The reservation station (RS) is expanded to 60 entries deep (compared to 54 entries in Intel microarchitecture code name Sandy Bridge). It can dispatch up to eight micro-ops in one cycle if the micro-ops are ready to execute. The RS dispatch a micro-op through an issue port to a specific execution cluster, arranged in several stacks to handle specific data types or granularity of data.

When a source of a micro-op executed in one stack comes from a micro-op executed in another stack, a delay can occur. The delay occurs also for transitions between Intel SSE integer and Intel SSE floating-point operations. In some of the cases the data transition is done using a micro-op that is added to the instruction flow. Table 2-23 describes how data, written back after execution, can bypass to micro-op execution in the following cycles.

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

From/To	INT	SSE-INT/ AVX-INT	SSE-FP/ AVX-FP_LOW	X87/ AVX-FP_High
INT		<ul style="list-style-type: none"> • micro-op (port 5) • micro-op (port 6) + 1 cycle 	<ul style="list-style-type: none"> • micro-op (port 5) • micro-op (port 6) + 1 cycle 	micro-op (port 5) + 3 cycle delay
SSE-INT/ AVX-INT	micro-op (port 1)		1 cycle delay	

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

From/To	INT	SSE-INT/ AVX-INT	SSE-FP/ AVX-FP_LOW	X87/ AVX-FP_High
SSE-FP/ AVX-FP_LOW	micro-op (port 1)	1 cycle delay		micro-op (port 5) + 1 cycle delay
X87/ AVX-FP_High	micro-op (port 1) + 3 cycle delay		micro-op (port 5) + 1 cycle delay	
Load		1 cycle delay	1 cycle delay	2 cycle delay

2.1.4 Cache and Memory Subsystem

The cache hierarchy is similar to prior generations, including an instruction cache, a first-level data cache and a second-level unified cache in each core, and a 3rd-level unified cache with size dependent on specific product configuration. The 3rd-level cache is organized as multiple cache slices, the size of each slice may depend on product configurations, connected by a ring interconnect. The exact details of the cache topology is reported by CPUID leaf 4. The 3rd level cache resides in the “uncore” sub-system that is shared by all the processor cores. In some product configuration, a fourth level cache is also supported. Table 2-21 provides more details of the cache hierarchy.

Table 2-3. Cache Parameters of the Haswell Microarchitecture

Level	Capacity/Associativity	Line Size (bytes)	Fastest Latency ¹	Throughput (clocks)	Peak Bandwidth (bytes/cyc)	Update Policy
First Level Data	32 KB/ 8	64	4 cycle	0.5 ²	64 (Load) + 32 (Store)	Writeback
Instruction	32 KB/8	64	N/A	N/A	N/A	N/A
Second Level	256KB/8	64	11 cycle	Varies	64	Writeback
Third Level (Shared L3)	Varies	64		Varies		Writeback

NOTES:

1. Software-visible latency will vary depending on access patterns and other factors.
2. First level data cache supports two load micro-ops each cycle, each micro-op can fetch up to 32-bytes of data.

The TLB hierarchy consists of dedicated level one TLB for instruction cache, TLB for L1D, plus unified TLB for L2.

Table 2-4. TLB Parameters of the Haswell Microarchitecture

Level	Page Size	Entries	Associativity	Partition
Instruction	4KB	128	4 ways	dynamic
Instruction	2MB/4MB	8 per thread		fixed
First Level Data	4KB	64	4	fixed
First Level Data	2MB/4MB	32	4	fixed
First Level Data	1GB	4	4	fixed
Second Level	Shared by 4KB and 2/4MB pages	1024	8	fixed

2.1.4.1 Load and Store Operation Enhancements

The L1 data cache can handle two 256-bit load and one 256-bit store operations each cycle. The unified L2 can service one cache line (64 bytes) each cycle. In addition, there are 72 load buffers and 42 store buffers available to support micro-ops execution in-flight.

2.1.5 The Haswell-E Microarchitecture

Intel processors based on the Haswell-E microarchitecture comprise of the same processor cores as described in the Haswell microarchitecture, but provide more advanced uncore and integrated I/O capabilities. Processors based on the Haswell-E microarchitecture support platforms with multiple sockets.

The Haswell-E microarchitecture supports versatile processor architectures and platform configurations for scalability and high performance. Some of capabilities provided by the uncore and integrated I/O sub-system of the Haswell-E microarchitecture include:

- Support for multiple Intel QPI interconnects in multi-socket configurations
- Up to two integrated memory controllers per physical processor
- Up to 40 lanes of PCI Express* 3.0 links per physical processor
- Up to 18 processor cores connected by two ring interconnects to the L3 in each physical processor.

An example of a possible 12-core processor implementing the Haswell-E microarchitecture is illustrated in Figure 2-3. The capabilities of the uncore and integrated I/O sub-system vary across the processor family implementing the Haswell-E microarchitecture. For details, please consult the data sheets of respective Intel Xeon E5 v3 processors.

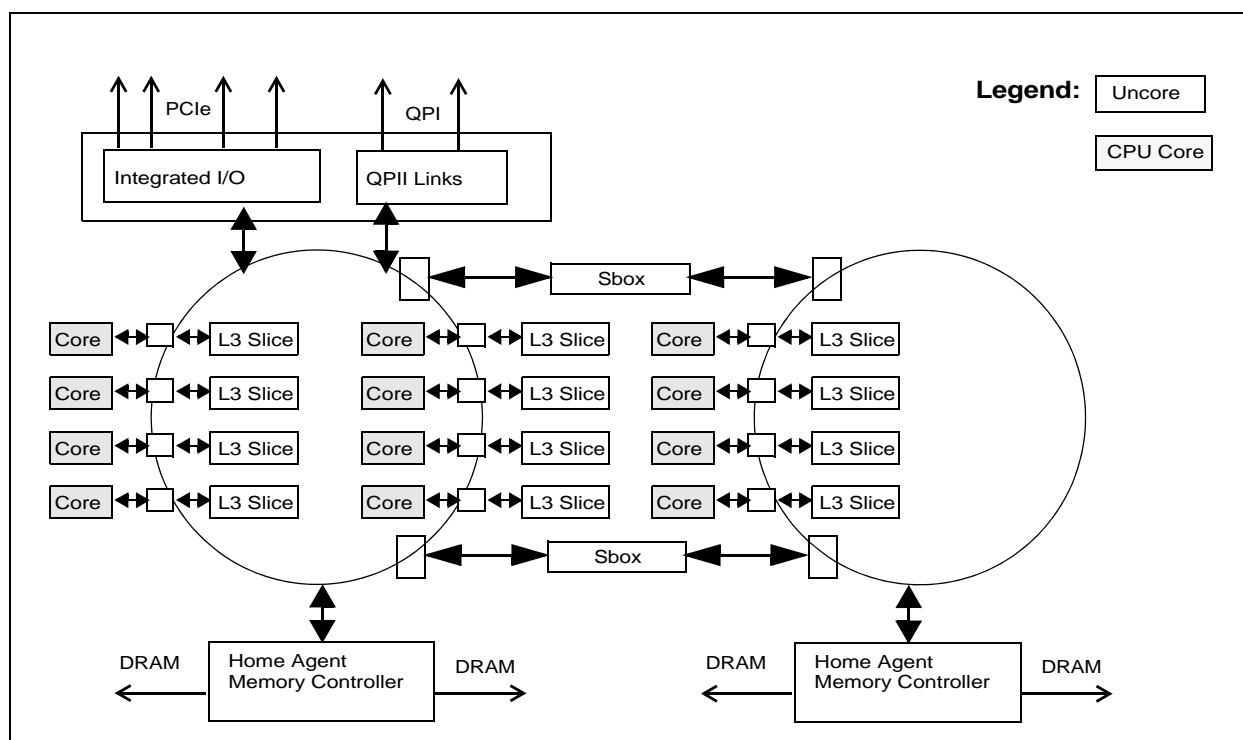


Figure 2-3. An Example of the Haswell-E Microarchitecture Supporting 12 Processor Cores

2.1.6 The Broadwell Microarchitecture

Intel Core M processors are based on the Broadwell microarchitecture. It builds from the Haswell microarchitecture and provides several enhancements. This section covers enhanced features of the Broadwell microarchitecture.

- Floating-point multiply instruction latency is improved from 5 cycles in prior generation to 3 cycle in the Broadwell microarchitecture. This applies to AVX, SSE and FP instruction sets.
- The throughput of Gather instructions has been improved significantly, see Table C-5.
- The PCLMULQDQ instruction implementation is a single uop in the Broadwell microarchitecture with improved latency and throughput.

The TLB hierarchy consists of dedicated level one TLB for instruction cache, TLB for L1D, plus unified TLB for L2.

Table 2-5. TLB Parameters of the Broadwell Microarchitecture

Level	Page Size	Entries	Associativity	Partition
Instruction	4KB	128	4 ways	dynamic
Instruction	2MB/4MB	8 per thread		fixed
First Level Data	4KB	64	4	fixed
First Level Data	2MB/4MB	32	4	fixed
First Level Data	1GB	4	4	fixed
Second Level	Shared by 4KB and 2MB pages	1536	6	fixed
Second Level	1GB pages	16	4	fixed

2.2 INTEL® MICROARCHITECTURE CODE NAME SANDY BRIDGE

Intel® microarchitecture code name Sandy Bridge builds on the successes of Intel® Core™ microarchitecture and Intel microarchitecture code name Nehalem. It offers the following innovative features:

- Intel Advanced Vector Extensions (Intel AVX)
 - 256-bit floating-point instruction set extensions to the 128-bit Intel Streaming SIMD Extensions, providing up to 2X performance benefits relative to 128-bit code.
 - Non-destructive destination encoding offers more flexible coding techniques.
 - Supports flexible migration and co-existence between 256-bit AVX code, 128-bit AVX code and legacy 128-bit SSE code.
- Enhanced front end and execution engine
 - New decoded ICache component that improves front end bandwidth and reduces branch misprediction penalty.
 - Advanced branch prediction.
 - Additional macro-fusion support.
 - Larger dynamic execution window.
 - Multi-precision integer arithmetic enhancements (ADC/SBB, MUL/IMUL).
 - LEA bandwidth improvement.
 - Reduction of general execution stalls (read ports, writeback conflicts, bypass latency, partial stalls).
 - Fast floating-point exception handling.
 - XSAVE/XRSTORE performance improvements and XSAVEOPT new instruction.
- Cache hierarchy improvements for wider data path
 - Doubling of bandwidth enabled by two symmetric ports for memory operation.
 - Simultaneous handling of more in-flight loads and stores enabled by increased buffers.
 - Internal bandwidth of two loads and one store each cycle.

- Improved prefetching.
- High bandwidth low latency LLC architecture.
- High bandwidth ring architecture of on-die interconnect.
- System-on-a-chip support
 - Integrated graphics and media engine in second generation Intel Core processors.
 - Integrated PCIE controller.
 - Integrated memory controller.
- Next generation Intel Turbo Boost Technology
 - Leverage TDP headroom to boost performance of CPU cores and integrated graphic unit.

2.2.1 Intel® Microarchitecture Code Name Sandy Bridge Pipeline Overview

Figure 2-4 depicts the pipeline and major components of a processor core that's based on Intel microarchitecture code name Sandy Bridge. The pipeline consists of

- An in-order issue front end that fetches instructions and decodes them into micro-ops (micro-operations). The front end feeds the next pipeline stages with a continuous stream of micro-ops from the most likely path that the program will execute.
- An out-of-order, superscalar execution engine that dispatches up to six micro-ops to execution, per cycle. The allocate/rename block reorders micro-ops to "dataflow" order so they can execute as soon as their sources are ready and execution resources are available.
- An in-order retirement unit that ensures that the results of execution of the micro-ops, including any exceptions they may have encountered, are visible according to the original program order.

The flow of an instruction in the pipeline can be summarized in the following progression:

1. The Branch Prediction Unit chooses the next block of code to execute from the program. The processor searches for the code in the following resources, in this order:
 - a. Decoded ICache
 - b. Instruction Cache, via activating the legacy decode pipeline
 - c. L2 cache, last level cache (LLC) and memory, as necessary

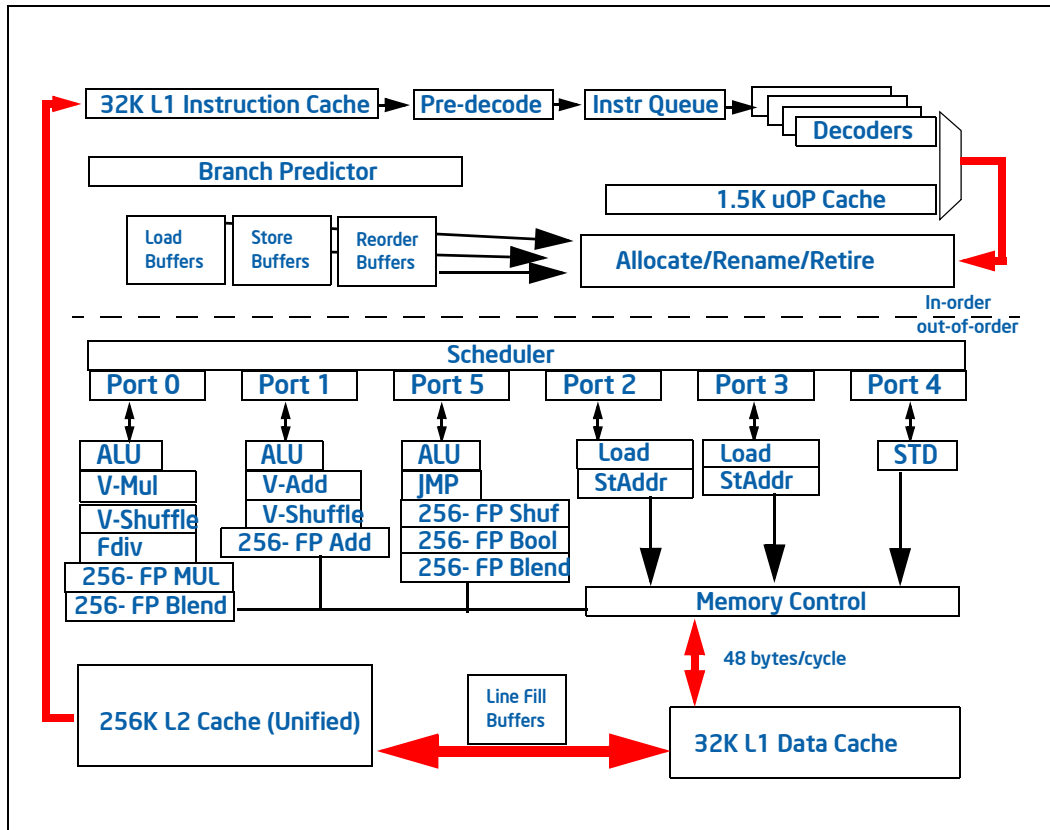


Figure 2-4. Intel Microarchitecture Code Name Sandy Bridge Pipeline Functionality

2. The micro-ops corresponding to this code are sent to the Rename/retirement block. They enter into the scheduler in program order, but execute and are de-allocated from the scheduler according to data-flow order. For simultaneously ready micro-ops, FIFO ordering is nearly always maintained.

Micro-op execution is executed using execution resources arranged in three stacks. The execution units in each stack are associated with the data type of the instruction.

Branch mispredictions are signaled at branch execution. It re-steers the front end which delivers micro-ops from the correct path. The processor can overlap work preceding the branch misprediction with work from the following corrected path.

3. Memory operations are managed and reordered to achieve parallelism and maximum performance. Misses to the L1 data cache go to the L2 cache. The data cache is non-blocking and can handle multiple simultaneous misses.
4. Exceptions (Faults, Traps) are signaled at retirement (or attempted retirement) of the faulting instruction.

Each processor core based on Intel microarchitecture code name Sandy Bridge can support two logical processor if Intel HyperThreading Technology is enabled.

2.2.2 The Front End

This section describes the key characteristics of the front end. Table 2-6 lists the components of the front end, their functions, and the problems they address.

Table 2-6. Components of the Front End of Intel Microarchitecture Code Name Sandy Bridge

Component	Functions	Performance Challenges
Instruction Cache	32-Kbyte backing store of instruction bytes	Fast access to hot code instruction bytes
Legacy Decode Pipeline	Decode instructions to micro-ops, delivered to the micro-op queue and the Decoded ICache.	Provides the same decode latency and bandwidth as prior Intel processors. Decoded ICache warm-up
Decoded ICache	Provide stream of micro-ops to the micro-op queue.	Provides higher micro-op bandwidth at lower latency and lower power than the legacy decode pipeline
MSROM	Complex instruction micro-op flow store, accessible from both Legacy Decode Pipeline and Decoded ICache	
Branch Prediction Unit (BPU)	Determine next block of code to be executed and drive lookup of Decoded ICache and legacy decode pipelines.	Improves performance and energy efficiency through reduced branch mispredictions.
Micro-op queue	Queues micro-ops from the Decoded ICache and the legacy decode pipeline.	Hide front end bubbles; provide execution micro-ops at a constant rate.

2.2.2.1 Legacy Decode Pipeline

The Legacy Decode Pipeline comprises the instruction translation lookaside buffer (ITLB), the instruction cache (ICache), instruction predecode, and instruction decode units.

Instruction Cache and ITLB

An instruction fetch is a 16-byte aligned lookup through the ITLB and into the instruction cache. The instruction cache can deliver every cycle 16 bytes to the instruction pre-decoder. Table 2-6 compares the ICache and ITLB with prior generation.

Table 2-7. ICache and ITLB of Intel Microarchitecture Code Name Sandy Bridge

Component	Intel microarchitecture code name Sandy Bridge	Intel microarchitecture code name Nehalem
ICache Size	32-Kbyte	32-Kbyte
ICache Ways	8	4
ITLB 4K page entries	128	128
ITLB large page (2M or 4M) entries	8	7

Upon ITLB miss there is a lookup to the Second level TLB (STLB) that is common to the DTLB and the ITLB. The penalty of an ITLB miss and a STLB hit is seven cycles.

Instruction PreDecode

The predecode unit accepts the 16 bytes from the instruction cache and determines the length of the instructions.

The following length changing prefixes (LCPs) imply instruction length that is different from the default length of instructions. Therefore they cause an additional penalty of three cycles per LCP during length decoding. Previous processors incur a six-cycle penalty for each 16-byte chunk that has one or more LCPs in it. Since usually there is no more than one LCP in a 16-byte chunk, in most cases, Intel microarchitecture code name Sandy Bridge introduces an improvement over previous processors.

- Operand Size Override (66H) preceding an instruction with a word/double immediate data. This prefix might appear when the code uses 16 bit data types, unicode processing, and image processing.
- Address Size Override (67H) preceding an instruction with a modr/m in real, big real, 16-bit protected or 32-bit protected modes. This prefix may appear in boot code sequences.
- The REX prefix (4xh) in the Intel® 64 instruction set can change the size of two classes of instructions: MOV offset and MOV immediate. Despite this capability, it does not cause an LCP penalty and hence is not considered an LCP.

Instruction Decode

There are four decoding units that decode instruction into micro-ops. The first can decode all IA-32 and Intel 64 instructions up to four micro-ops in size. The remaining three decoding units handle single-micro-op instructions. All four decoding units support the common cases of single micro-op flows including micro-fusion and macro-fusion.

Micro-ops emitted by the decoders are directed to the micro-op queue and to the Decoded ICache. Instructions longer than four micro-ops generate their micro-ops from the MSROM. The MSROM bandwidth is four micro-ops per cycle. Instructions whose micro-ops come from the MSROM can start from either the legacy decode pipeline or from the Decoded ICache.

MicroFusion

Micro-fusion fuses multiple micro-ops from the same instruction into a single complex micro-op. The complex micro-op is dispatched in the out-of-order execution core as many times as it would if it were not micro-fused.

Micro-fusion enables you to use memory-to-register operations, also known as the complex instruction set computer (CISC) instruction set, to express the actual program operation without worrying about a loss of decode bandwidth. Micro-fusion improves instruction bandwidth delivered from decode to retirement and saves power.

Coding an instruction sequence by using single-uop instructions will increase the code size, which can decrease fetch bandwidth from the legacy pipeline.

The following are examples of micro-fused micro-ops that can be handled by all decoders.

- All stores to memory, including store immediate. Stores execute internally as two separate functions, store-address and store-data.
- All instructions that combine load and computation operations (load+op), for example:
 - `ADDPS XMM9, QWORD PTR [RSP+40]`
 - `FADD DOUBLE PTR [RDI+RSI*8]`
 - `XOR RAX, QWORD PTR [RBP+32]`
- All instructions of the form "load and jump," for example:
 - `JMP [RDI+200]`
 - `RET`
- `CMP` and `TEST` with immediate operand and memory

An instruction with RIP relative addressing is not micro-fused in the following cases:

- An additional immediate is needed, for example:
 - `CMP [RIP+400], 27`
 - `MOV [RIP+3000], 142`
- The instruction is a control flow instruction with an indirect target specified using RIP-relative addressing, for example:
 - `JMP [RIP+5000000]`

In these cases, an instruction that can not be micro-fused will require decoder 0 to issue two micro-ops, resulting in a slight loss of decode bandwidth.

In 64-bit code, the usage of RIP Relative addressing is common for global data. Since there is no micro-fusion in these cases, performance may be reduced when porting 32-bit code to 64-bit code.

Macro-Fusion

Macro-fusion merges two instructions into a single micro-op. In Intel Core microarchitecture, this hardware optimization is limited to specific conditions specific to the first and second of the macro-fusable instruction pair.

- The first instruction of the macro-fused pair modifies the flags. The following instructions can be macro-fused:
 - In Intel microarchitecture code name Nehalem: CMP, TEST.
 - In Intel microarchitecture code name Sandy Bridge: CMP, TEST, ADD, SUB, AND, INC, DEC
 - These instructions can fuse if
 - The first source / destination operand is a register.
 - The second source operand (if exists) is one of: immediate, register, or non RIP-relative memory.
- The second instruction of the macro-fusable pair is a conditional branch. Table 3-1 describes, for each instruction, what branches it can fuse with.

Macro fusion does not happen if the first instruction ends on byte 63 of a cache line, and the second instruction is a conditional branch that starts at byte 0 of the next cache line.

Since these pairs are common in many types of applications, macro-fusion improves performance even on non-recompiled binaries.

Each macro-fused instruction executes with a single dispatch. This reduces latency and frees execution resources. You also gain increased rename and retire bandwidth, increased virtual storage, and power savings from representing more work in fewer bits.

2.2.2.2 Decoded ICache

The Decoded ICache is essentially an accelerator of the legacy decode pipeline. By storing decoded instructions, the Decoded ICache enables the following features:

- Reduced latency on branch mispredictions
- Increased micro-op delivery bandwidth to the out-of-order engine
- Reduced front end power consumption

The Decoded ICache caches the output of the instruction decoder. The next time the micro-ops are consumed for execution the decoded micro-ops are taken from the Decoded ICache. This enables skipping the fetch and decode stages for these micro-ops and reduces power and latency of the Front End. The Decoded ICache provides average hit rates of above 80% of the micro-ops; furthermore, "hot spots" typically have hit rates close to 100%.

Typical integer programs average less than four bytes per instruction, and the front end is able to race ahead of the back end, filling in a large window for the scheduler to find instruction level parallelism. However, for high performance code with a basic block consisting of many instructions, for example, Intel SSE media algorithms or excessively unrolled loops, the 16 instruction bytes per cycle is occasionally a limitation. The 32-byte orientation of the Decoded ICache helps such code to avoid this limitation.

The Decoded ICache automatically improves performance of programs with temporal and spatial locality. However, to fully utilize the Decoded ICache potential, you might need to understand its internal organization.

The Decoded ICache consists of 32 sets. Each set contains eight Ways. Each Way can hold up to six micro-ops. The Decoded ICache can ideally hold up to 1536 micro-ops.

The following are some of the rules how the Decoded ICache is filled with micro-ops:

- All micro-ops in a Way represent instructions which are statically contiguous in the code and have their EIPs within the same aligned 32-byte region.

- Up to three Ways may be dedicated to the same 32-byte aligned chunk, allowing a total of 18 micro-ops to be cached per 32-byte region of the original IA program.
- A multi micro-op instruction cannot be split across Ways.
- Up to two branches are allowed per Way.
- An instruction which turns on the MSROM consumes an entire Way.
- A non-conditional branch is the last micro-op in a Way.
- Micro-fused micro-ops (load+op and stores) are kept as one micro-op.
- A pair of macro-fused instructions is kept as one micro-op.
- Instructions with 64-bit immediate require two slots to hold the immediate.

When micro-ops cannot be stored in the Decoded ICache due to these restrictions, they are delivered from the legacy decode pipeline. Once micro-ops are delivered from the legacy pipeline, fetching micro-ops from the Decoded ICache can resume only after the next branch micro-op. Frequent switches can incur a penalty.

The Decoded ICache is virtually included in the Instruction cache and ITLB. That is, any instruction with micro-ops in the Decoded ICache has its original instruction bytes present in the instruction cache. Instruction cache evictions must also be evicted from the Decoded ICache, which evicts only the necessary lines.

There are cases where the entire Decoded ICache is flushed. One reason for this can be an ITLB entry eviction. Other reasons are not usually visible to the application programmer, as they occur when important controls are changed, for example, mapping in CR3, or feature and mode enabling in CR0 and CR4. There are also cases where the Decoded ICache is disabled, for instance, when the CS base address is NOT set to zero.

2.2.2.3 Branch Prediction

Branch prediction predicts the branch target and enables the processor to begin executing instructions long before the branch true execution path is known. All branches utilize the branch prediction unit (BPU) for prediction. This unit predicts the target address not only based on the EIP of the branch but also based on the execution path through which execution reached this EIP. The BPU can efficiently predict the following branch types:

- Conditional branches
- direct calls and jumps
- indirect calls and jumps
- returns

2.2.2.4 Micro-op Queue and the Loop Stream Detector (LSD)

The micro-op queue decouples the front end and the out-of order engine. It stays between the micro-op generation and the renamer as shown in Figure 2-4. This queue helps to hide bubbles which are introduced between the various sources of micro-ops in the front end and ensures that four micro-ops are delivered for execution, each cycle.

The micro-op queue provides post-decode functionality for certain instructions types. In particular, loads combined with computational operations and all stores, when used with indexed addressing, are represented as a single micro-op in the decoder or Decoded ICache. In the micro-op queue they are fragmented into two micro-ops through a process called un-lamination, one does the load and the other does the operation. A typical example is the following "load plus operation" instruction:

ADD RAX, [RBP+RSI] ; rax := rax + LD(RBP+RSI)

Similarly, the following store instruction has three register sources and is broken into "generate store address" and "generate store data" sub-components.

MOV [ESP+ECX*4+12345678], AL

The additional micro-ops generated by unlamination use the rename and retirement bandwidth. However, it has an overall power benefit. For code that is dominated by indexed addressing (as often happens with array processing), recoding algorithms to use base (or base+displacement) addressing can sometimes improve performance by keeping the load plus operation and store instructions fused.

The Loop Stream Detector (LSD)

The Loop Stream Detector was introduced in Intel® Core microarchitectures. The LSD detects small loops that fit in the micro-op queue and locks them down. The loop streams from the micro-op queue, with no more fetching, decoding, or reading micro-ops from any of the caches, until a branch miss-prediction inevitably ends it.

The loops with the following attributes qualify for LSD/micro-op queue replay:

- Up to eight chunk fetches of 32-instruction-bytes
- Up to 28 micro-ops (~28 instructions)
- All micro-ops are also resident in the Decoded ICache
- Can contain no more than eight taken branches and none of them can be a CALL or RET
- Cannot have mismatched stack operations. For example, more PUSH than POP instructions.

Many calculation-intensive loops, searches and software string moves match these characteristics.

Use the loop cache functionality opportunistically. For high performance code, loop unrolling is generally preferable for performance even when it overflows the LSD capability.

2.2.3 The Out-of-Order Engine

The Out-of-Order engine provides improved performance over prior generations with excellent power characteristics. It detects dependency chains and sends them to execution out-of-order while maintaining the correct data flow. When a dependency chain is waiting for a resource, such as a second-level data cache line, it sends micro-ops from another chain to the execution core. This increases the overall rate of instructions executed per cycle (IPC).

The out-of-order engine consists of two blocks, shown in Figure 2-4: Core Functional Diagram, the Rename/retirement block, and the Scheduler.

The Out-of-Order engine contains the following major components:

Renamer. The Renamer component moves micro-ops from the front end to the execution core. It eliminates false dependencies among micro-ops, thereby enabling out-of-order execution of micro-ops.

Scheduler. The Scheduler component queues micro-ops until all source operands are ready. Schedules and dispatches ready micro-ops to the available execution units in as close to a first in first out (FIFO) order as possible.

Retirement. The Retirement component retires instructions and micro-ops in order and handles faults and exceptions.

2.2.3.1 Renamer

The Renamer is the bridge between the in-order part in Figure 2-4, and the dataflow world of the Scheduler. It moves up to four micro-ops every cycle from the micro-op queue to the out-of-order engine. Although the renamer can send up to 4 micro-ops (unfused, micro-fused, or macro-fused) per cycle, this is equivalent to the issue port can dispatch six micro-ops per cycle. In this process, the out-of-order core carries out the following steps:

- Renames architectural sources and destinations of the micro-ops to micro-architectural sources and destinations.
- Allocates resources to the micro-ops. For example, load or store buffers.
- Binds the micro-op to an appropriate dispatch port.

Some micro-ops can execute to completion during rename and are removed from the pipeline at that point, effectively costing no execution bandwidth. These include:

- Zero idioms (dependency breaking idioms)
- NOP
- VZEROUPPER
- FXCHG

The renamer can allocate two branches each cycle, compared to one branch each cycle in the previous microarchitecture. This can eliminate some bubbles in execution.

Micro-fused load and store operations that use an index register are decomposed to two micro-ops, hence consume two out of the four slots the Renamer can use every cycle.

Dependency Breaking Idioms

Instruction parallelism can be improved by using common instructions to clear register contents to zero. The renamer can detect them on the zero evaluation of the destination register.

Use one of these dependency breaking idioms to clear a register when possible.

- XOR REG,REG
- SUB REG,REG
- PXOR/VPXOR XMMREG,XMMREG
- PSUBB/W/D/Q XMMREG,XMMREG
- VPSUBB/W/D/Q XMMREG,XMMREG
- XORPS/PD XMMREG,XMMREG
- VXORPS/PD YMMREG, YMMREG

Since zero idioms are detected and removed by the renamer, they have no execution latency.

There is another dependency breaking idiom - the "ones idiom".

- CMPEQ `XMM1, XMM1`; "ones idiom" set all elements to all "ones"

In this case, the micro-op must execute, however, since it is known that regardless of the input data the output data is always "all ones" the micro-op dependency upon its sources does not exist as with the zero idiom and it can execute as soon as it finds a free execution port.

2.2.3.2 Scheduler

The scheduler controls the dispatch of micro-ops onto their execution ports. In order to do this, it must identify which micro-ops are ready and where its sources come from: a register file entry, or a bypass directly from an execution unit. Depending on the availability of dispatch ports and writeback buses, and the priority of ready micro-ops, the scheduler selects which micro-ops are dispatched every cycle.

2.2.4 The Execution Core

The execution core is superscalar and can process instructions out of order. The execution core optimizes overall performance by handling the most common operations efficiently, while minimizing potential delays.

The out-of-order execution core improves execution unit organization over prior generation in the following ways:

- Reduction in read port stalls
- Reduction in writeback conflicts and delays
- Reduction in power
- Reduction of SIMD FP assists dealing with denormal inputs and underflowed outputs

Some high precision FP algorithms need to operate with FTZ=0 and DAZ=0, i.e. permitting underflowed intermediate results and denormal inputs to achieve higher numerical precision at the expense of reduced performance on prior generation microarchitectures due to SIMD FP assists. The reduction of SIMD FP assists in Intel microarchitecture code name Sandy Bridge applies to the following SSE instructions (and AVX variants): ADDPD/ADDPS, MULPD/MULPS, DIVPD/DIVPS, and CVTPD2PS.

The out-of-order core consists of three execution stacks, where each stack encapsulates a certain type of data. The execution core contains the following execution stacks:

- General purpose integer
- SIMD integer and floating-point
- X87

The execution core also contains connections to and from the cache hierarchy. The loaded data is fetched from the caches and written back into one of the stacks.

The scheduler can dispatch up to six micro-ops every cycle, one on each port. The following table summarizes which operations can be dispatched on which port.

Table 2-8. Dispatch Port and Execution Stacks

	Port 0	Port 1	Port 2	Port 3	Port 4	Port 5
Integer	ALU, Shift	ALU, Fast LEA, Slow LEA, MUL	Load_Addr, Store_addr	Load_Addr Store_addr	Store_data	ALU, Shift, Branch, Fast LEA
SSE-Int, AVX-Int, MMX	Mul, Shift, STTNI, Int-Div, 128b-Mov	ALU, Shuf, Blend, 128b-Mov			Store_data	ALU, Shuf, Shift, Blend, 128b-Mov
SSE-FP, AVX-FP_low	Mul, Div, Blend, 256b-Mov	Add, CVT			Store_data	Shuf, Blend, 256b-Mov
X87, AVX-FP_High	Mul, Div, Blend, 256b-Mov	Add, CVT			Store_data	Shuf, Blend, 256b-Mov

After execution, the data is written back on a writeback bus corresponding to the dispatch port and the data type of the result. Micro-ops that are dispatched on the same port but have different latencies may need the write back bus at the same cycle. In these cases the execution of one of the micro-ops is delayed until the writeback bus is available. For example, MULPS (five cycles) and BLENDPS (one cycle) may collide if both are ready for execution on port 0: first the MULPS and four cycles later the BLENDPS. Intel microarchitecture code name Sandy Bridge eliminates such collisions as long as the micro-ops write the results to different stacks. For example, integer ADD (one cycle) can be dispatched four cycles after MULPS (five cycles) since the integer ADD uses the integer stack while the MULPS uses the FP stack.

When a source of a micro-op executed in one stack comes from a micro-op executed in another stack, a one- or two-cycle delay can occur. The delay occurs also for transitions between Intel SSE integer and Intel SSE floating-point operations. In some of the cases the data transition is done using a micro-op that is added to the instruction flow. The following table describes how data, written back after execution, can bypass to micro-op execution in the following cycles.

Table 2-9. Execution Core Writeback Latency (cycles)

	Integer	SSE-Int, AVX-Int, MMX	SSE-FP, AVX-FP_low	X87, AVX-FP_High
Integer	0	micro-op (port 0)	micro-op (port 0)	micro-op (port 0) + 1 cycle
SSE-Int, AVX-Int, MMX	micro-op (port 5) or micro-op (port 5) +1 cycle	0	1 cycle delay	0
SSE-FP, AVX-FP_low	micro-op (port 5) or micro-op (port 5) +1 cycle	1 cycle delay	0	micro-op (port 5) +1 cycle
X87, AVX-FP_High	micro-op (port 5) +1 cycle	0	micro-op (port 5) +1 cycle	0
Load	0	1 cycle delay	1 cycle delay	2 cycle delay

2.2.5 Cache Hierarchy

The cache hierarchy contains a first level instruction cache, a first level data cache (L1 DCache) and a second level (L2) cache, in each core. The L1D cache may be shared by two logical processors if the processor support Intel HyperThreading Technology. The L2 cache is shared by instructions and data. All cores in a physical processor package connect to a shared last level cache (LLC) via a ring connection.

The caches use the services of the Instruction Translation Lookaside Buffer (ITLB), Data Translation Lookaside Buffer (DTLB) and Shared Translation Lookaside Buffer (STLB) to translate linear addresses to physical address. Data coherency in all cache levels is maintained using the MESI protocol. For more information, see the Intel® 64 IA-32 Architectures Software Developer's Manual, Volume 3. Cache hierarchy details can be obtained at run-time using the CPUID instruction. see the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*.

Table 2-10. Cache Parameters

Level	Capacity	Associativity (ways)	Line Size (bytes)	Write Update Policy	Inclusive
L1 Data	32 KB	8	64	Writeback	-
Instruction	32 KB	8	N/A	N/A	-
L2 (Unified)	256 KB	8	64	Writeback	No
Third Level (LLC)	Varies, query CPUID leaf 4	Varies with cache size	64	Writeback	Yes

2.2.5.1 Load and Store Operation Overview

This section provides an overview of the load and store operations.

Loads

When an instruction reads data from a memory location that has write-back (WB) type, the processor looks for it in the caches and memory. Table 2-11 shows the access lookup order and best case latency. The actual latency can vary depending on the cache queue occupancy, LLC ring occupancy, memory components, and their parameters.

Table 2-11. Lookup Order and Load Latency

Level	Latency (cycles)	Bandwidth (per core per cycle)
L1 Data	4 ¹	2 x16 bytes
L2 (Unified)	12	1 x 32 bytes
Third Level (LLC)	26-31 ²	1 x 32 bytes
L2 and L1 DCache in other cores if applicable	43- clean hit; 60 - dirty hit	

NOTES:

1. Subject to execution core bypass restriction shown in Table 2-9.
2. Latency of L3 varies with product segment and sku. The values apply to second generation Intel Core processor families.

The LLC is inclusive of all cache levels above it - data contained in the core caches must also reside in the LLC. Each cache line in the LLC holds an indication of the cores that may have this line in their L2 and L1 caches. If there is an indication in the LLC that other cores may hold the line of interest and its state might have to modify, there is a lookup into the L1 DCache and L2 of these cores too. The lookup is called "clean" if it does not require fetching data from the other core caches. The lookup is called "dirty" if modified data has to be fetched from the other core caches and transferred to the loading core.

The latencies shown above are the best-case scenarios. 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 memory 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. Memory access latencies vary based on occupancy of the memory controller queues, DRAM configuration, DDR parameters, and DDR paging behavior (if the requested page is a page-hit, page-miss or page-empty).

Stores

When an instruction writes data to a memory location that has a write back memory type, the processor first ensures that it has the line containing this memory location in its L1 DCache, in Exclusive or Modified MESI state. If the cache line is not there, in the right state, the processor fetches it from the next levels of the memory hierarchy using a Read for Ownership request. The processor looks for the cache line in the following locations, in the specified order:

1. L1 DCache
2. L2
3. Last Level Cache
4. L2 and L1 DCache in other cores, if applicable
5. Memory

Once the cache line is in the L1 DCache, the new data is written to it, and the line is marked as Modified.

Reading for ownership and storing the data happens after instruction retirement and follows the order of store instruction retirement. Therefore, the store latency usually does not affect the store instruction itself. However, several sequential stores that miss the L1 DCache may have cumulative latency that can affect performance. As long as the store does not complete, its entry remains occupied in the store buffer. When the store buffer becomes full, new micro-ops cannot enter the execution pipe and execution might stall.

2.2.5.2 L1 DCache

The L1 DCache is the first level data cache. It manages all load and store requests from all types through its internal data structures. The L1 DCache:

- enables loads and stores to issue speculatively and out of order
- ensures that retired loads and stores have the correct data upon retirement
- ensures that loads and stores follow the memory ordering rules of the IA-32 and Intel 64 instruction set architecture.

Table 2-12. L1 Data Cache Components

Component	Intel microarchitecture code name Sandy Bridge	Intel microarchitecture code name Nehalem
Data Cache Unit (DCU)	32KB, 8 ways	32KB, 8 ways
Load buffers	64 entries	48 entries
Store buffers	36 entries	32 entries
Line fill buffers (LFB)	10 entries	10 entries

The DCU is organized as 32 KBytes, eight-way set associative. Cache line size is 64-bytes arranged in eight banks.

Internally, accesses are up to 16 bytes, with 256-bit Intel AVX instructions utilizing two 16-byte accesses. Two load operations and one store operation can be handled each cycle.

The L1 DCache maintains requests which cannot be serviced immediately to completion. Some reasons for requests that are delayed: cache misses, unaligned access that splits across cache lines, data not ready to be forwarded from a preceding store, loads experiencing bank collisions, and load block due to cache line replacement.

The L1 DCache can maintain up to 64 load micro-ops from allocation until retirement. It can maintain up to 36 store operations from allocation until the store value is committed to the cache, or written to the line fill buffers (LFB) in the case of non-temporal stores.

The L1 DCache can handle multiple outstanding cache misses and continue to service incoming stores and loads. Up to 10 requests of missing cache lines can be managed simultaneously using the LFB.

The L1 DCache is a write-back write-allocate cache. Stores that hit in the DCU do not update the lower levels of the memory hierarchy. Stores that miss the DCU allocate a cache line.

Loads

The L1 DCache architecture can service two loads per cycle, each of which can be up to 16 bytes. Up to 32 loads can be maintained at different stages of progress, from their allocation in the out of order engine until the loaded value is returned to the execution core.

Loads can:

- Read data before preceding stores when the load address and store address ranges 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.

Loads cannot:

- Speculatively take any sort of fault or trap.
- Speculatively access uncacheable memory.

The common load latency is five cycles. When using a simple addressing mode, base plus offset that is smaller than 2048, the load latency can be four cycles. This technique is especially useful for pointer-chasing code. However, overall latency varies depending on the target register data type due to stack bypass. See Section 2.2.4 for more information.

The following table lists overall load latencies. These latencies assume the common case of flat segment, that is, segment base address is zero. If segment base is not zero, load latency increases.

Table 2-13. Effect of Addressing Modes on Load Latency

Data Type/Addressing Mode	Base + Offset > 2048; Base + Index [+ Offset]	Base + Offset < 2048
Integer	5	4
MMX, SSE, 128-bit AVX	6	5
X87	7	6
256-bit AVX	7	7

Stores

Stores to memory are executed in two phases:

- Execution phase. Fills the store buffers with linear and physical address and data. Once store address and data are known, the store data can be forwarded to the following load operations that need it.
- Completion phase. After the store retires, the L1 DCache moves its data from the store buffers to the DCU, up to 16 bytes per cycle.

Address Translation

The DTLB can perform three linear to physical address translations every cycle, two for load addresses and one for a store address. If the address is missing in the DTLB, the processor looks for it in the STLB, which holds data and instruction address translations. The penalty of a DTLB miss that hits the STLB is seven cycles. Large page support include 1G byte pages, in addition to 4K and 2M/4M pages.

The DTLB and STLB are four way set associative. The following table specifies the number of entries in the DTLB and STLB.

Table 2-14. DTLB and STLB Parameters

TLB	Page Size	Entries
DTLB	4KB	64
	2MB/4MB	32
	1GB	4
STLB	4KB	512

Store Forwarding

If a load follows a store and reloads the data that the store writes to memory, the data can forward directly from the store operation 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. You can take advantage of store forwarding to quickly move complex structures without losing the ability to forward the subfields. The memory control unit can handle store forwarding situations with less restrictions compared to previous micro-architectures.

The following rules must be met to enable store to load forwarding:

- The store must be the last store to that address, prior to the load.
- The store must contain all data being loaded.
- The load is from a write-back memory type and neither the load nor the store are non-temporal accesses.

Stores cannot forward to loads in the following cases:

- Four byte and eight byte loads that cross eight byte boundary, relative to the preceding 16- or 32-byte store.
- Any load that crosses a 16-byte boundary of a 32-byte store.

Table 2-15 to Table 2-18 detail the store to load forwarding behavior. For a given store size, all the loads that may overlap are shown and specified by 'F'. Forwarding from 32 byte store is similar to forwarding from each of the 16 byte halves of the store. Cases that cannot forward are shown as 'N'.

Table 2-15. Store Forwarding Conditions (1 and 2 byte stores)

		Load Alignment															
Store Size	Load Size	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	F															
2	1	F	F														
	2	F	N														

Table 2-16. Store Forwarding Conditions (4-16 byte stores)

		Load Alignment															
Store Size	Load Size	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
4	1	F	F	F	F												
	2	F	F	F	N												
	4	F	N	N	N												
8	1	F	F	F	F	F	F	F	F								
	2	F	F	F	F	F	F	F	N								
	4	F	F	F	F	F	N	N	N								
	8	F	N	N	N	N	N	N	N								
16	1	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
	2	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	N
	4	F	F	F	F	F	N	N	N	F	F	F	F	F	N	N	N
	8	F	N	N	N	N	N	N	N	F	N	N	N	N	N	N	N
	16	F	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N

Table 2-17. 32-byte Store Forwarding Conditions (0-15 byte alignment)

		Load Alignment															
Store Size	Load Size	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
32	1	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
	2	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	N
	4	F	F	F	F	F	N	N	N	F	F	F	F	F	N	N	N
	8	F	N	N	N	N	N	N	N	F	N	N	N	N	N	N	N
	16	F	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
	32	F	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N

Table 2-18. 32-byte Store Forwarding Conditions (16-31 byte alignment)

Store Size	Load Size	Load Alignment															
		16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
32	1	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
	2	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	N
	4	F	F	F	F	F	N	N	N	F	F	F	F	F	N	N	N
	8	F	N	N	N	N	N	N	N	F	N	N	N	N	N	N	N
	16	F	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
	32	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N

Memory Disambiguation

A load operation may depend on a preceding store. Many microarchitectures block loads until all preceding store addresses 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 even when the store address is unknown. This hides the load latency. Eventually, the prediction is verified. If an actual conflict is detected, the load and all succeeding instructions are re-executed.

The following loads are not disambiguated. The execution of these loads is stalled until addresses of all previous stores are known.

- Loads that cross the 16-byte boundary
- 32-byte Intel AVX loads that are not 32-byte aligned.

The memory disambiguator always assumes dependency between loads and earlier stores that have the same address bits 0:11.

Bank Conflict

Since 16-byte loads can cover up to three banks, and two loads can happen every cycle, it is possible that six of the eight banks may be accessed per cycle, for loads. A bank conflict happens when two load accesses need the same bank (their address has the same 2-4 bit value) in different sets, at the same time. When a bank conflict occurs, one of the load accesses is recycled internally.

In many cases two loads access exactly the same bank in the same cache line, as may happen when popping operands off the stack, or any sequential accesses. In these cases, conflict does not occur and the loads are serviced simultaneously.

2.2.5.3 Ring Interconnect and Last Level Cache

The system-on-a-chip design provides a high bandwidth bi-directional ring bus to connect between the IA cores and various sub-systems in the uncore. In the second generation Intel Core processor 2xxx series, the uncore subsystem include a system agent, the graphics unit (GT) and the last level cache (LLC).

The LLC consists of multiple cache slices. The number of slices is equal to the number of IA cores. Each slice has logic portion and data array portion. The logic portion handles data coherency, memory ordering, access to the data array portion, LLC misses and writeback to memory, and more. The data array portion stores cache lines. Each slice contains a full cache port that can supply 32 bytes/cycle.

The physical addresses of data kept in the LLC data arrays are distributed among the cache slices by a hash function, such that addresses are uniformly distributed. The data array in a cache block may have 4/8/12/16 ways corresponding to 0.5M/1M/1.5M/2M block size. However, due to the address distribution among the cache blocks from the software point of view, this does not appear as a normal N-way cache.

From the processor cores and the GT view, the LLC act as one shared cache with multiple ports and bandwidth that scales with the number of cores. The LLC hit latency, ranging between 26-31 cycles, depends on the core location relative to the LLC block, and how far the request needs to travel on the ring.

The number of cache-slices increases with the number of cores, therefore the ring and LLC are not likely to be a bandwidth limiter to core operation.

The GT sits on the same ring interconnect, and uses the LLC for its data operations as well. In this respect it is very similar to an IA core. Therefore, high bandwidth graphic applications using cache bandwidth and significant cache footprint, can interfere, to some extent, with core operations.

All the traffic that cannot be satisfied by the LLC, such as LLC misses, dirty line writeback, non-cacheable operations, and MMIO/IO operations, still travels through the cache-slice logic portion and the ring, to the system agent.

In the Intel Xeon Processor E5 Family, the uncore subsystem does not include the graphics unit (GT). Instead, the uncore subsystem contains many more components, including an LLC with larger capacity and snooping capabilities to support multiple processors, Intel® QuickPath Interconnect interfaces that can support multi-socket platforms, power management control hardware, and a system agent capable of supporting high-bandwidth traffic from memory and I/O devices.

In the Intel Xeon processor E5 2xxx or 4xxx families, the LLC capacity generally scales with the number of processor cores with 2.5 MBytes per core.

2.2.5.4 Data Prefetching

Data can be speculatively loaded to the L1 DCache using software prefetching, hardware prefetching, or any combination of the two.

You can use the four Streaming SIMD Extensions (SSE) prefetch instructions to enable software-controlled prefetching. These instructions are 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.

The rest of this section describes the various hardware prefetching mechanisms provided by Intel micro-architecture code name Sandy Bridge and their improvement over previous processors. The goal of the prefetchers is to automatically predict which data the program is about to consume. If this data is not close-by to the execution core or inner cache, the prefetchers bring it from the next levels of cache hierarchy and memory. Prefetching has the following effects:

- Improves performance if data 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.

Data Prefetch to L1 Data Cache

Data prefetching is triggered by load operations when the following conditions are met:

- Load is from writeback memory type.
- The prefetched data is within the same 4K byte page as the load instruction that triggered it.
- No fence is in progress in the pipeline.
- Not many other load misses are in progress.
- There is not a continuous stream of stores.

Two hardware prefetchers load data to the L1 DCache:

- **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 stride 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 2K bytes.

Data Prefetch to the L2 and Last Level Cache

The following two hardware prefetchers fetched data from memory to the L2 cache and last level cache:

Spatial Prefetcher: This prefetcher strives to complete every cache line fetched to the L2 cache with the pair line that completes it to a 128-byte aligned chunk.

Streamer: This prefetcher monitors read requests from the L1 cache for ascending and descending sequences of addresses. Monitored read requests include L1 DCache requests initiated by load and store operations and by the hardware prefetchers, and L1 ICache requests for code fetch. When a forward or backward stream of requests is detected, the anticipated cache lines are prefetched. Prefetched cache lines must be in the same 4K page.

The streamer and spatial prefetcher prefetch the data to the last level cache. Typically data is brought also to the L2 unless the L2 cache is heavily loaded with missing demand requests.

Enhancement to the streamer includes the following features:

- The streamer may issue two prefetch requests on every L2 lookup. The streamer can run up to 20 lines ahead of the load request.
- Adjusts dynamically to the number of outstanding requests per core. If there are not many outstanding requests, the streamer prefetches further ahead. If there are many outstanding requests it prefetches to the LLC only and less far ahead.
- When cache lines are far ahead, it prefetches to the last level cache only and not to the L2. This method avoids replacement of useful cache lines in the L2 cache.
- Detects and maintains up to 32 streams of data accesses. For each 4K byte page, you can maintain one forward and one backward stream can be maintained.

2.2.6 System Agent

The system agent implemented in the second generation Intel Core processor family contains the following components:

- An arbiter that handles all accesses from the ring domain and from I/O (PCIe* and DMI) and routes the accesses to the right place.
- PCIe controllers connect to external PCIe devices. The PCIe controllers have different configuration possibilities that varies with product segment specifics: x16+x4, x8+x8+x4, x8+x4+x4+x4.
- DMI controller connects to the PCH chipset.
- Integrated display engine, Flexible Display Interconnect, and Display Port, for the internal graphic operations.
- Memory controller

All main memory traffic is routed from the arbiter to the memory controller. The memory controller in the second generation Intel Core processor 2xxx series support two channels of DDR, with data rates of 1066MHz, 1333MHz and 1600MHz, and 8 bytes per cycle, depending on the unit type, system configuration and DRAMs. Addresses are distributed between memory channels based on a local hash function that attempts to balance the load between the channels in order to achieve maximum bandwidth and minimum hotspot collisions.

For best performance, populate both channels with equal amounts of memory, preferably the exact same types of DIMMs. In addition, using more ranks for the same amount of memory, results in somewhat better memory bandwidth, since more DRAM pages can be open simultaneously. For best performance, populate the system with the highest supported speed DRAM (1333MHz or 1600MHz data rates, depending on the max supported frequency) with the best DRAM timings.

The two channels have separate resources and handle memory requests independently. The memory controller contains a high-performance out-of-order scheduler that attempts to maximize memory band-

width while minimizing latency. Each memory channel contains a 32 cache-line write-data-buffer. Writes to the memory controller are considered completed when they are written to the write-data-buffer. The write-data-buffer is flushed out to main memory at a later time, not impacting write latency.

Partial writes are not handled efficiently on the memory controller and may result in read-modify-write operations on the DDR channel if the partial-writes do not complete a full cache-line in time. Software should avoid creating partial write transactions whenever possible and consider alternative, such as buffering the partial writes into full cache line writes.

The memory controller also supports high-priority isochronous requests (such as USB isochronous, and Display isochronous requests). High bandwidth of memory requests from the integrated display engine takes up some of the memory bandwidth and impacts core access latency to some degree.

2.2.7 Intel® Microarchitecture Code Name Ivy Bridge

Third generation Intel Core processors are based on Intel microarchitecture code name Ivy Bridge. Most of the features described in Section 2.2.1 - Section 2.2.6 also apply to Intel microarchitecture code name Ivy Bridge. This section covers feature differences in microarchitecture that can affect coding and performance.

Support for new instructions enabling include:

- Numeric conversion to and from half-precision floating-point values,
- Hardware-based random number generator compliant to NIST SP 800-90A,
- Reading and writing to FS/GS base registers in any ring to improve user-mode threading support.

For details about using the hardware based random number generator instruction RDRAND, please refer to the article available from Intel Software Network at <http://software.intel.com/en-us/articles/download-the-latest-bull-mountain-software-implementation-guide/?wapkw=bull+mountain>.

A small number of microarchitectural enhancements that can be beneficial to software:

- Hardware prefetch enhancement: A next-page prefetcher (NPP) is added in Intel microarchitecture code name Ivy Bridge. The NPP is triggered by sequential accesses to cache lines approaching the page boundary, either upwards or downwards.
- Zero-latency register move operation: A subset of register-to-register MOV instructions are executed at the front end, conserving scheduling and execution resource in the out-of-order engine.
- Front end enhancement: In Intel microarchitecture code name Sandy Bridge, the micro-op queue is statically partitioned to provide 28 entries for each logical processor, irrespective of software executing in single thread or multiple threads. If one logical processor is not active in Intel microarchitecture code name Ivy Bridge, then a single thread executing on that processor core can use the 56 entries in the micro-op queue. In this case, the LSD can handle larger loop structure that would require more than 28 entries.
- The latency and throughput of some instructions have been improved over those of Intel microarchitecture code name Sandy Bridge. For example, 256-bit packed floating-point divide and square root operations are faster; ROL and ROR instructions are also improved.

2.3 INTEL® CORE™ MICROARCHITECTURE AND ENHANCED 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 micro-ops per cycle
- Peak retirement bandwidth of up to four micro-ops 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 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 (except 128-bit shuffle, pack, unpack operations)
 - Up to eight floating-point operations per cycle
 - Three issue ports available to dispatching SIMD instructions for execution.

The Enhanced Intel Core microarchitecture supports all of the features of Intel Core microarchitecture and provides a comprehensive set of enhancements.

- **Intel® Wide Dynamic Execution** includes several enhancements:
 - A radix-16 divider replacing previous radix-4 based divider to speedup long-latency operations such as divisions and square roots.
 - Improved system primitives to speedup long-latency operations such as RDTSC, STI, CLI, and VM exit transitions.
- **Intel® Advanced Smart Cache** provides up to 6 MBytes of second-level cache shared between two processor cores (quad-core processors have up to 12 MBytes of L2); up to 24 way/set associativity.
- **Intel® Smart Memory Access** supports high-speed system bus up 1600 MHz and provides more efficient handling of memory operations such as split cache line load and store-to-load forwarding situations.
- **Intel® Advanced Digital Media Boost** provides 128-bit shuffler unit to speedup shuffle, pack, unpack operations; adds support for 47 SSE4.1 instructions.

In the sub-sections of 2.1.x, most of the descriptions on Intel Core microarchitecture also applies to Enhanced Intel Core microarchitecture. Differences between them are note explicitly.

2.3.1 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 (micro-ops) to the out-of-order execution core.

- An out-of-order superscalar execution core that can issue up to six micro-ops per cycle (see Table 2-20) and reorder micro-ops 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 micro-ops are processed and architectural states are updated according to the original program order.

Intel Core 2 Extreme processor X6800, Intel Core 2 Duo processors and Intel Xeon processor 3000, 5100 series implement two processor cores based on the Intel Core microarchitecture. Intel Core 2 Extreme quad-core processor, Intel Core 2 Quad processors and Intel Xeon processor 3200 series, 5300 series implement four processor cores. Each physical package of these quad-core processors contains two processor dies, each die containing two processor cores. The functionality of the subsystems in each core are depicted in Figure 2-5.

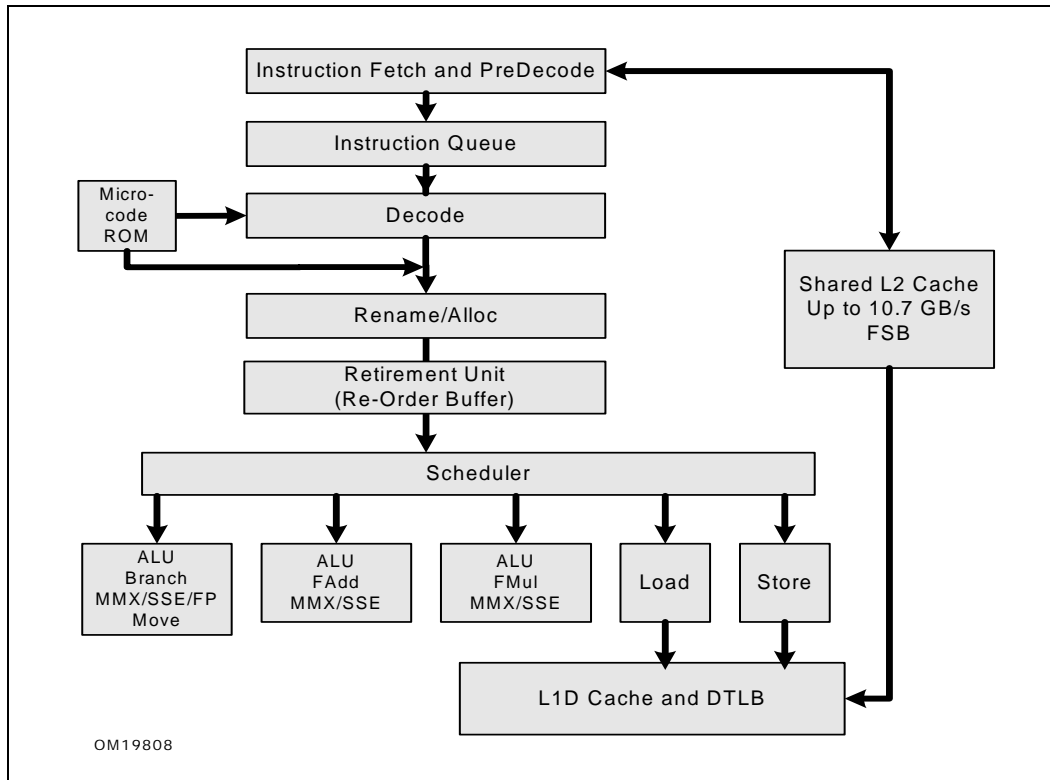


Figure 2-5. Intel Core Microarchitecture Pipeline Functionality

2.3.2 Front End

The front ends needs to supply decoded instructions (micro-ops) and sustain the stream to a six-issue wide out-of-order engine. The components of the front end, their functions, and the performance challenges to microarchitectural design are described in Table 2-19.

Table 2-19. Components of the Front End

Component	Functions	Performance Challenges
Branch Prediction Unit (BPU)	<ul style="list-style-type: none"> 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. 	<ul style="list-style-type: none"> 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.

Table 2-19. Components of the Front End

Component	Functions	Performance Challenges
Instruction Fetch Unit	<ul style="list-style-type: none"> • 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 	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • 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 	<ul style="list-style-type: none"> • Varying amounts of work per instruction requires expansion into variable numbers of micro-ops. • 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.

2.3.2.1 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.”

2.3.2.2 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.

2.3.2.3 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.

2.3.2.4 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 micro-ops in size. Three other decoders handle single micro-op instructions. The microsequencer can provide up to 3 micro-ops per cycle, and helps decode instructions larger than 4 micro-ops.

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

Macro-fusion merges two instructions into a single micro-op. 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.

2.3.2.5 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 micro-ops 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 micro-op 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 micro-ops are removed.
- Improves power efficiency as the RSP updates are carried out on small, dedicated hardware.

2.3.2.6 Micro-fusion

Micro-fusion fuses multiple micro-ops from the same instruction into a single complex micro-op. The complex micro-op 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 micro-op 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 decode into a longer flow of micro-ops 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.

2.3.3 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 micro-ops 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 micro-ops in various stages of completion, buffers completed micro-ops, 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 micro-ops until all source operands are ready, schedules and dispatches ready micro-ops to the available execution units. The RS has 32 entries.

The initial stages of the out of order core move the micro-ops 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 micro-ops (for example: these resources could be load or store buffers).
- Binds the micro-op to an appropriate issue port.
- Renames sources and destinations of micro-ops, enabling out of order execution.
- Provides data to the micro-op 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 micro-ops with single-cycle latency can be executed by multiple execution units, enabling multiple streams of dependent operations to be executed quickly.
- **Frequently-used μ ops with longer latency** — These micro-ops have pipelined execution units so that multiple micro-ops 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 2.3.4 for more information about the MOB.

2.3.3.1 Issue Ports and Execution Units

The scheduler can dispatch up to six micro-ops per cycle through the issue ports. The issue ports of Intel Core microarchitecture and Enhanced Intel Core microarchitecture are depicted in Table 2-20, the former is denoted by its CPUID signature of DisplayFamily_DisplayModel value of 06_0FH, the latter denoted by the corresponding signature value of 06_17H. The table provides latency and throughput data of common integer and floating-point (FP) operations for each issue port in cycles.

Table 2-20. Issue Ports of Intel Core Microarchitecture and Enhanced Intel Core Microarchitecture

Executable operations	Latency, Throughput		Comment ¹
	Signature = 06_0FH	Signature = 06_17H	
Integer ALU Integer SIMD ALU FP/SIMD/SSE2 Move and Logic	1, 1 1, 1 1, 1	1, 1 1, 1 1, 1	Includes 64-bit mode integer MUL; Issue port 0; Writeback port 0;
Single-precision (SP) FP MUL Double-precision FP MUL	4, 1 5, 1	4, 1 5, 1	Issue port 0; Writeback port 0
FP MUL (X87) FP Shuffle DIV/SQRT	5, 2 1, 1	5, 2 1, 1	Issue port 0; Writeback port 0 FP shuffle does not handle QW shuffle.
Integer ALU Integer SIMD ALU FP/SIMD/SSE2 Move and Logic	1, 1 1, 1 1, 1	1, 1 1, 1 1, 1	Excludes 64-bit mode integer MUL; Issue port 1; Writeback port 1;
FP ADD QW Shuffle	3, 1 1, 1 ²	3, 1 1, 1 ³	Issue port 1; Writeback port 1;
Integer loads FP loads	3, 1 4, 1	3, 1 4, 1	Issue port 2; Writeback port 2;
Store address ⁴	3, 1	3, 1	Issue port 3;
Store data ⁵ .			Issue Port 4;
Integer ALU Integer SIMD ALU FP/SIMD/SSE2 Move and Logic	1, 1 1, 1 1, 1	1, 1 1, 1 1, 1	Issue port 5; Writeback port 5;
QW shuffles 128-bit Shuffle/Pack/Unpack	1, 1 ² 2-4, 2-4 ⁶	1, 1 ³ 1-3, 1 ⁷	Issue port 5; Writeback port 5;

NOTES:

1. Mixing operations of different latencies that use the same port can result in writeback bus conflicts; this can reduce overall throughput
2. 128-bit instructions executes with longer latency and reduced throughput
3. Uses 128-bit shuffle unit in port 5.
4. Prepares the store forwarding and store retirement logic with the address of the data being stored.
5. Prepares the store forwarding and store retirement logic with the data being stored
6. Varies with instructions; 128-bit instructions are executed using QW shuffle units
7. Varies with instructions, 128-bit shuffle unit replaces QW shuffle units in Intel Core microarchitecture.

In each cycle, the RS can dispatch up to six micro-ops. 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 micro-ops 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 2-6.

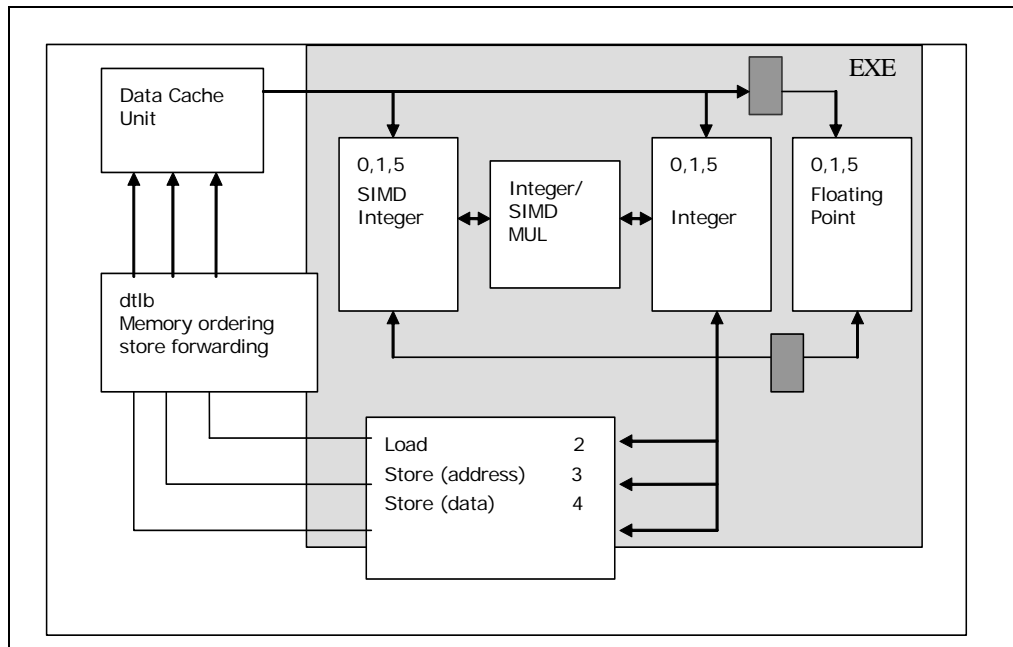


Figure 2-6. Execution Core of Intel Core Microarchitecture

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 2-6 represent the extra cycle of latency.

2.3.4 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.”

2.3.4.1 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.

2.3.4.2 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.

2.3.4.3 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.

2.3.4.4 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.

In Enhanced Intel Core microarchitecture, the alignment restrictions to permit store forwarding to proceed have been relaxed. Enhanced Intel Core microarchitecture permits store-forwarding to proceed in several situations that the succeeding load is not aligned to the preceding store. Figure 2-7 shows six situations (in gradient-filled background) of store-forwarding that are permitted in Enhanced Intel Core microarchitecture but not in Intel Core microarchitecture. The cases with backward slash background depicts store-forwarding that can proceed in both Intel Core microarchitecture and Enhanced Intel Core microarchitecture.

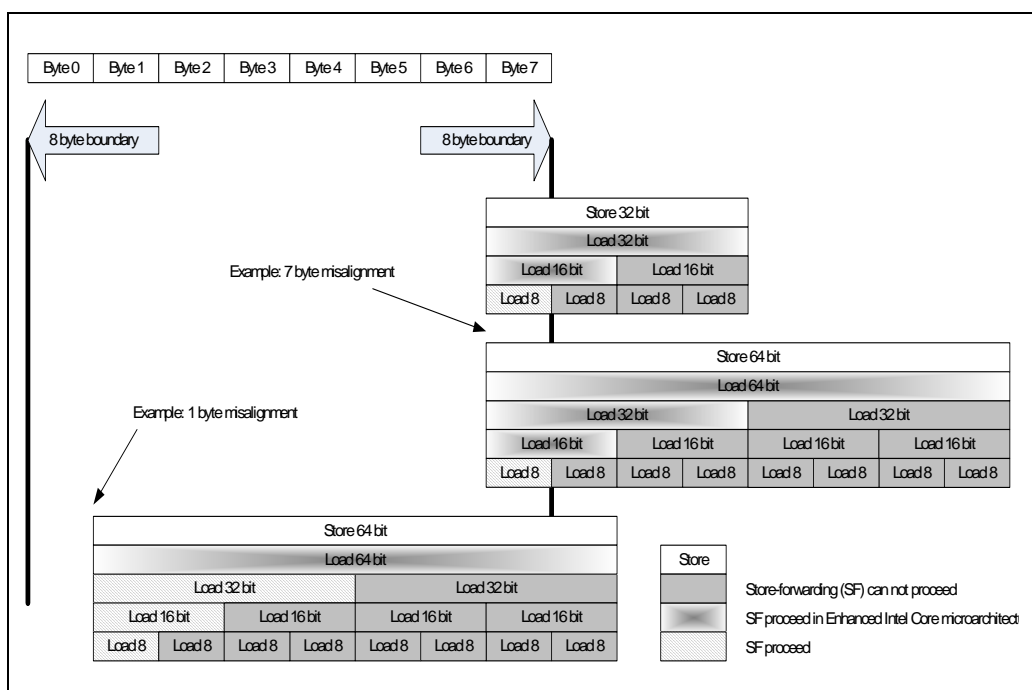


Figure 2-7. Store-Forwarding Enhancements in Enhanced Intel Core Microarchitecture

2.3.4.5 Memory Disambiguation

A load instruction micro-op 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.

2.3.5 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 2-8 illustrates the architecture of the Intel Advanced Smart Cache.

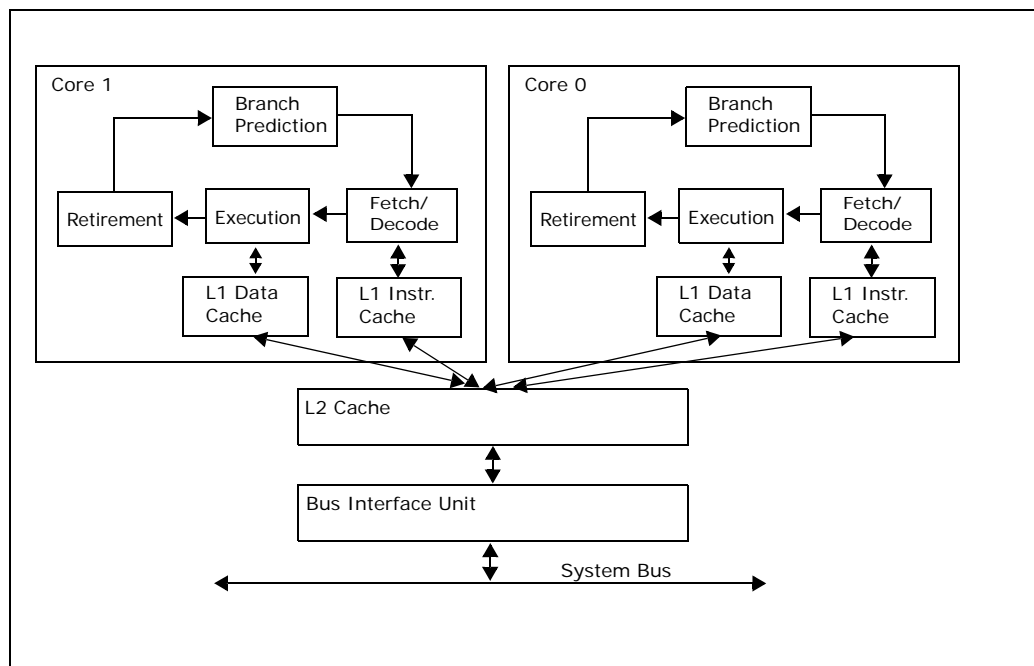


Figure 2-8. Intel Advanced Smart Cache Architecture

Table 2-21 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 2-21. 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) ¹	2, 4 MB	8 or 16	64	14 ²	2	Writeback
Second Level (Shared L2) ³	3, 6MB	12 or 24	64	15 ²	2	Writeback
Third Level ⁴	8, 12, 16 MB	16	64	~110	12	Writeback

NOTES:

1. Intel Core microarchitecture (CPUID signature DisplayFamily = 06H, DisplayModel = 0FH).
2. Software-visible latency will vary depending on access patterns and other factors.
3. Enhanced Intel Core microarchitecture (CPUID signature DisplayFamily = 06H, DisplayModel = 17H or 1DH).
4. Enhanced Intel Core microarchitecture (CPUID signature DisplayFamily = 06H, DisplayModel = 1DH).

2.3.5.1 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 2-22 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 2-22. Characteristics of Load and Store Operations in Intel Core Microarchitecture

Data Locality	Load		Store	
	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.

2.3.5.2 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 2-22 presents store latencies depending on the initial cache line location.

2.4 INTEL® MICROARCHITECTURE CODE NAME NEHALEM

Intel microarchitecture code name Nehalem provides the foundation for many innovative features of Intel Core i7 processors and Intel Xeon processor 3400, 5500, and 7500 series. It builds on the success of 45nm enhanced Intel Core microarchitecture and provides the following feature enhancements:

- **Enhanced processor core**
 - Improved branch prediction and recovery from misprediction.
 - Enhanced loop streaming to improve front end performance and reduce power consumption.
 - Deeper buffering in out-of-order engine to extract parallelism.
 - Enhanced execution units to provide acceleration in CRC, string/text processing and data shuffling.
- **Hyper-Threading Technology**
 - Provides two hardware threads (logical processors) per core.
 - Takes advantage of 4-wide execution engine, large L3, and massive memory bandwidth.
- **Smart Memory Access**
 - Integrated memory controller provides low-latency access to system memory and scalable memory bandwidth
 - New cache hierarchy organization with shared, inclusive L3 to reduce snoop traffic
 - Two level TLBs and increased TLB size.
 - Fast unaligned memory access.
- **Dedicated Power management Innovations**
 - Integrated microcontroller with optimized embedded firmware to manage power consumption.
 - Embedded real-time sensors for temperature, current, and power.
 - Integrated power gate to turn off/on per-core power consumption
 - Versatility to reduce power consumption of memory, link subsystems.

Intel microarchitecture code name Westmere is a 32nm version of Intel microarchitecture code name Nehalem. All of the features of latter also apply to the former.

2.4.1 Microarchitecture Pipeline

Intel microarchitecture code name Nehalem continues the four-wide microarchitecture pipeline pioneered by the 65nm Intel Core microarchitecture. Figure 2-9 illustrates the basic components of the pipeline of Intel microarchitecture code name Nehalem as implemented in Intel Core i7 processor, only two of the four cores are sketched in the Figure 2-9 pipeline diagram.

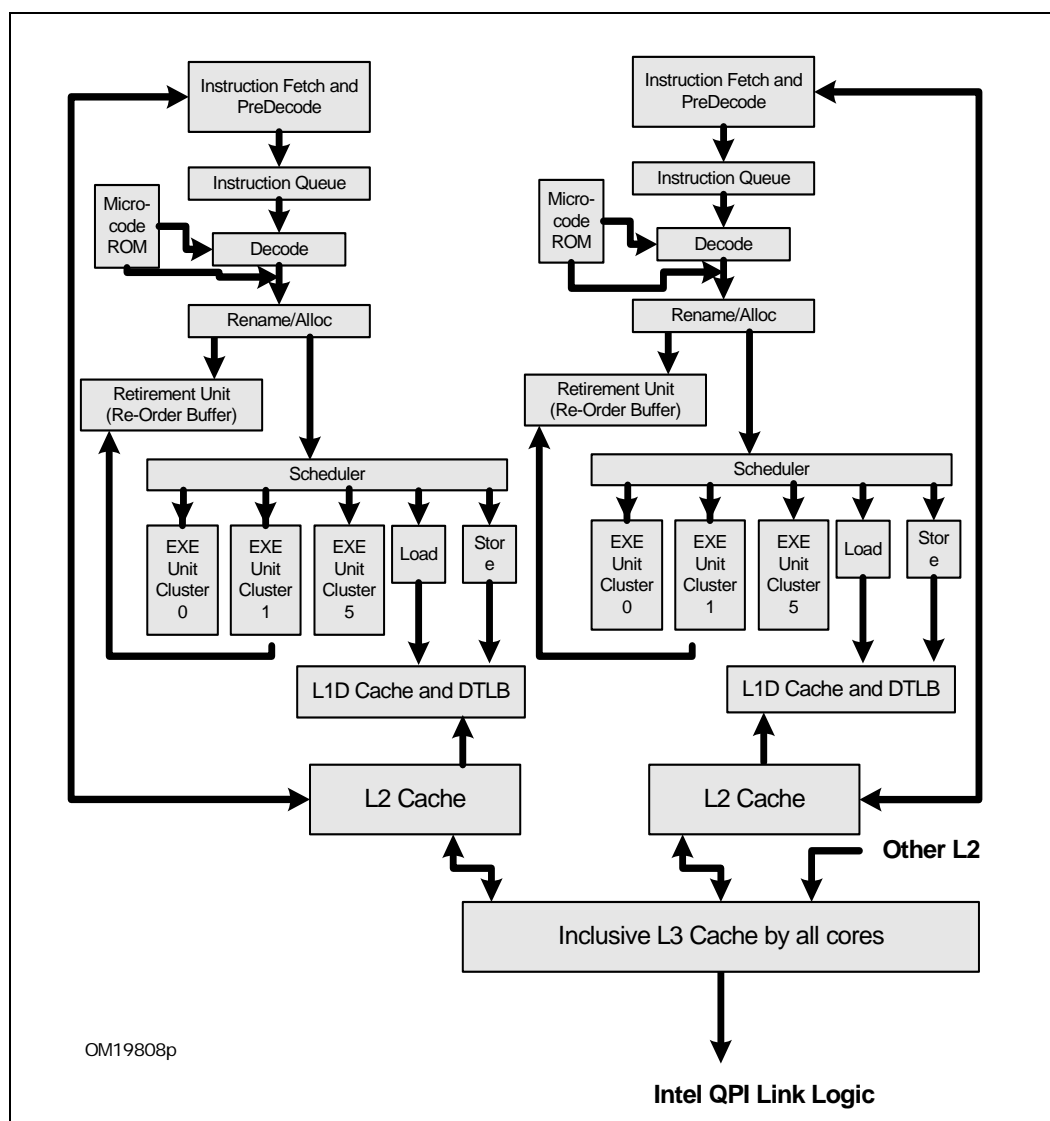


Figure 2-9. Intel Microarchitecture Code Name Nehalem Pipeline Functionality

The length of the pipeline in Intel microarchitecture code name Nehalem is two cycles longer than its predecessor in 45nm Intel Core 2 processor family, as measured by branch misprediction delay. The front end can decode up to 4 instructions in one cycle and supports two hardware threads by decoding the instruction streams between two logical processors in alternate cycles. The front end includes enhancement in branch handling, loop detection, MSROM throughput, etc. These are discussed in subsequent sections.

The scheduler (or reservation station) can dispatch up to six micro-ops in one cycle through six issue ports (five issue ports are shown in Figure 2-9; store operation involves separate ports for store address and store data but is depicted as one in the diagram).

The out-of-order engine has many execution units that are arranged in three execution clusters shown in Figure 2-9. It can retire four micro-ops in one cycle, same as its predecessor.

2.4.2 Front End Overview

Figure 2-10 depicts the key components of the front end of the microarchitecture. The instruction fetch unit (IFU) can fetch up to 16 bytes of aligned instruction bytes each cycle from the instruction cache to the instruction length decoder (ILD). The instruction queue (IQ) buffers the ILD-processed instructions and can deliver up to four instructions in one cycle to the instruction decoder.

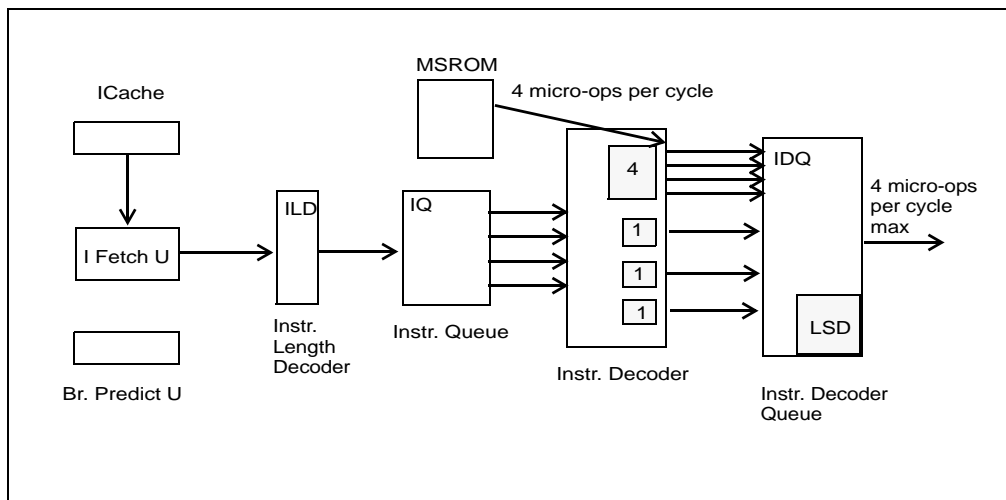


Figure 2-10. Front End of Intel Microarchitecture Code Name Nehalem

The instruction decoder has three decoder units that can decode one simple instruction per cycle per unit. The other decoder unit can decode one instruction every cycle, either simple instruction or complex instruction made up of several micro-ops. Instructions made up of more than four micro-ops are delivered from the MSROM. Up to four micro-ops can be delivered each cycle to the instruction decoder queue (IDQ).

The loop stream detector is located inside the IDQ to improve power consumption and front end efficiency for loops with a short sequence of instructions.

The instruction decoder supports micro-fusion to improve front end throughput, increase the effective size of queues in the scheduler and re-order buffer (ROB). The rules for micro-fusion are similar to those of Intel Core microarchitecture.

The instruction queue also supports macro-fusion to combine adjacent instructions into one micro-ops where possible. In previous generations of Intel Core microarchitecture, macro-fusion support for CMP/Jcc sequence is limited to the CF and ZF flag, and macrofusion is not supported in 64-bit mode.

In Intel microarchitecture code name Nehalem, macro-fusion is supported in 64-bit mode, and the following instruction sequences are supported:

- CMP or TEST can be fused when comparing (unchanged):
 - REG-REG. For example: `CMP EAX,ECX; JZ label`
 - REG-IMM. For example: `CMP EAX,0x80; JZ label`
 - REG-MEM. For example: `CMP EAX,[ECX]; JZ label`
 - MEM-REG. For example: `CMP [EAX],ECX; JZ label`
- TEST can fused with all conditional jumps (unchanged).

- CMP can be fused with the following conditional jumps. These conditional jumps check carry flag (CF) or zero flag (ZF). The list of macro-fusion-capable conditional jumps are (unchanged):

JA or JNBE
 JAE or JNB or JNC
 JE or JZ
 JNA or JBE
 JNAE or JC or JB
 JNE or JNZ

- CMP can be fused with the following conditional jumps in Intel microarchitecture code name Nehalem, (this is an enhancement):

JL or JNGE
 JGE or JNL
 JLE or JNG
 JG or JNLE

The hardware improves branch handling in several ways. Branch target buffer has increased to increase the accuracy of branch predictions. Renaming is supported with return stack buffer to reduce mispredictions of return instructions in the code. Furthermore, hardware enhancement improves the handling of branch misprediction by expediting resource reclamation so that the front end would not be waiting to decode instructions in an architected code path (the code path in which instructions will reach retirement) while resources were allocated to executing mispredicted code path. Instead, new micro-ops stream can start forward progress as soon as the front end decodes the instructions in the architected code path.

2.4.3 Execution Engine

The IDQ (Figure 2-10) delivers micro-op stream to the allocation/renaming stage (Figure 2-9) of the pipeline. The out-of-order engine supports up to 128 micro-ops in flight. Each micro-ops must be allocated with the following resources: an entry in the re-order buffer (ROB), an entry in the reservation station (RS), and a load/store buffer if a memory access is required.

The allocator also renames the register file entry of each micro-op in flight. The input data associated with a micro-op are generally either read from the ROB or from the retired register file.

The RS is expanded to 36 entry deep (compared to 32 entries in previous generation). It can dispatch up to six micro-ops in one cycle if the micro-ops are ready to execute. The RS dispatch a micro-op through an issue port to a specific execution cluster, each cluster may contain a collection of integer/FP/SIMD execution units.

The result from the execution unit executing a micro-op is written back to the register file, or forwarded through a bypass network to a micro-op in-flight that needs the result. Intel microarchitecture code name Nehalem can support write back throughput of one register file write per cycle per port. The bypass network consists of three domains of integer/FP/SIMD. Forwarding the result within the same bypass domain from a producer micro-op to a consumer micro is done efficiently in hardware without delay. Forwarding the result across different bypass domains may be subject to additional bypass delays. The bypass delays may be visible to software in addition to the latency and throughput characteristics of individual execution units. The bypass delays between a producer micro-op and a consumer micro-op across different bypass domains are shown in Table 2-23.

Table 2-23. Bypass Delay Between Producer and Consumer Micro-ops (cycles)

	FP	Integer	SIMD
FP	0	2	2
Integer	2	0	1

Table 2-23. Bypass Delay Between Producer and Consumer Micro-ops (cycles)

	FP	Integer	SIMD
SIMD	2	1	0

2.4.3.1 Issue Ports and Execution Units

Table 2-24 summarizes the key characteristics of the issue ports and the execution unit latency/throughputs for common operations in the microarchitecture.

Table 2-24. Issue Ports of Intel Microarchitecture Code Name Nehalem

Port	Executable operations	Latency	Throughput	Domain	Comment
Port 0	Integer ALU	1	1	Integer	
	Integer Shift	1	1		
Port 0	Integer SIMD ALU	1	1	SIMD	
	Integer SIMD Shuffle	1	1		
Port 0	Single-precision (SP) FP MUL	4	1	FP	
	Double-precision FP MUL	5	1		
	FP MUL (X87)	5	1		
	FP/SIMD/SSE2 Move and Logic	1	1		
	FP Shuffle	1	1		
	DIV/SQRT				
Port 1	Integer ALU	1	1	Integer	
	Integer LEA	1	1		
	Integer Mul	3	1		
Port 1	Integer SIMD MUL	1	1	SIMD	
	Integer SIMD Shift	1	1		
	PSAD	3	1		
	StringCompare				
Port 1	FP ADD	3	1	FP	
Port 2	Integer loads	4	1	Integer	
Port 3	Store address	5	1	Integer	
Port 4	Store data			Integer	
Port 5	Integer ALU	1	1	Integer	
	Integer Shift	1	1		
	Jmp	1	1		
Port 5	Integer SIMD ALU	1	1	SIMD	
	Integer SIMD Shuffle	1	1		
Port 5	FP/SIMD/SSE2 Move and Logic	1	1	FP	

2.4.4 Cache and Memory Subsystem

Intel microarchitecture code name Nehalem contains an instruction cache, a first-level data cache and a second-level unified cache in each core (see Figure 2-9). Each physical processor may contain several processor cores and a shared collection of sub-systems that are referred to as “uncore”. Specifically in Intel Core i7 processor, the uncore provides a unified third-level cache shared by all cores in the physical processor, Intel QuickPath Interconnect links and associated logic. The L1 and L2 caches are writeback and non-inclusive.

The shared L3 cache is writeback and inclusive, such that a cache line that exists in either L1 data cache, L1 instruction cache, unified L2 cache also exists in L3. The L3 is designed to use the inclusive nature to minimize snoop traffic between processor cores. Table 2-25 lists characteristics of the cache hierarchy. The latency of L3 access may vary as a function of the frequency ratio between the processor and the uncore sub-system.

Table 2-25. Cache Parameters of Intel Core i7 Processors

Level	Capacity	Associativity (ways)	Line Size (bytes)	Access Latency (clocks)	Access Throughput (clocks)	Write Update Policy
First Level Data	32 KB	8	64	4	1	Writeback
Instruction	32 KB	4	N/A	N/A	N/A	N/A
Second Level	256KB	8	64	10 ¹	Varies	Writeback
Third Level (Shared L3) ²	8MB	16	64	35-40+ ²	Varies	Writeback

NOTES:

1. Software-visible latency will vary depending on access patterns and other factors.
2. Minimal L3 latency is 35 cycles if the frequency ratio between core and uncore is unity.

The Intel microarchitecture code name Nehalem implements two levels of translation lookaside buffer (TLB). The first level consists of separate TLBs for data and code. DTLB0 handles address translation for data accesses, it provides 64 entries to support 4KB pages and 32 entries for large pages. The ITLB provides 64 entries (per thread) for 4KB pages and 7 entries (per thread) for large pages.

The second level TLB (STLB) handles both code and data accesses for 4KB pages. It support 4KB page translation operation that missed DTLB0 or ITLB. All entries are 4-way associative. Here is a list of entries in each DTLB:

- STLB for 4-KByte pages: 512 entries (services both data and instruction look-ups)
- DTLB0 for large pages: 32 entries
- DTLB0 for 4-KByte pages: 64 entries

An DTLB0 miss and STLB hit causes a penalty of 7cycles. Software only pays this penalty if the DTLB0 is used in some dispatch cases. The delays associated with a miss to the STLB and PMH are largely non-blocking.

2.4.5 Load and Store Operation Enhancements

The memory cluster of Intel microarchitecture code name Nehalem provides the following enhancements to speed up memory operations:

- Peak issue rate of one 128-bit load and one 128-bit store operation per cycle
- Deeper buffers for load and store operations: 48 load buffers, 32 store buffers and 10 fill buffers
- Fast unaligned memory access and robust handling of memory alignment hazards
- Improved store-forwarding for aligned and non-aligned scenarios

- Store forwarding for most address alignments

2.4.5.1 Efficient Handling of Alignment Hazards

The cache and memory subsystems handles a significant percentage of instructions in every workload. Different address alignment scenarios will produce varying performance impact for memory and cache operations. For example, 1-cycle throughput of L1 (see Table 2-26) generally applies to naturally-aligned loads from L1 cache. But using unaligned load instructions (e.g. MOVUPS, MOVUPD, MOVDQU, etc.) to access data from L1 will experience varying amount of delays depending on specific microarchitectures and alignment scenarios.

Table 2-26. Performance Impact of Address Alignments of MOVDQU from L1

Throughput (cycle)	Intel Core i7 Processor	45 nm Intel Core Microarchitecture	65 nm Intel Core Microarchitecture
Alignment Scenario	06_1AH	06_17H	06_0FH
16B aligned	1	2	2
Not-16B aligned, not cache split	1	~2	~2
Split cache line boundary	~4.5	~20	~20

Table 2-26 lists approximate throughput of issuing MOVDQU instructions with different address alignment scenarios to load data from the L1 cache. If a 16-byte load spans across cache line boundary, previous microarchitecture generations will experience significant software-visible delays.

Intel microarchitecture code name Nehalem provides hardware enhancements to reduce the delays of handling different address alignment scenarios including cache line splits.

2.4.5.2 Store Forwarding Enhancement

When a load follows a store and reloads the data that the store writes to memory, the microarchitecture can forward the data directly from the store to the load in many cases. This situation, called store to load forwarding, saves several cycles by enabling the load to obtain the data directly from the store operation instead of through the memory system.

Several general rules must be met for store to load forwarding to proceed without delay:

- 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 data must be completely contained in the preceding store.

Specific address alignment and data sizes between the store and load operations will determine whether a store-forward situation may proceed with data forwarding or experience a delay via the cache/memory sub-system. The 45 nm Enhanced Intel Core microarchitecture offers more flexible address alignment and data sizes requirement than previous microarchitectures. Intel microarchitecture code name Nehalem offers additional enhancement with allowing more situations to forward data expeditiously.

The store-forwarding situations for with respect to store operations of 16 bytes are illustrated in Figure 2-11.

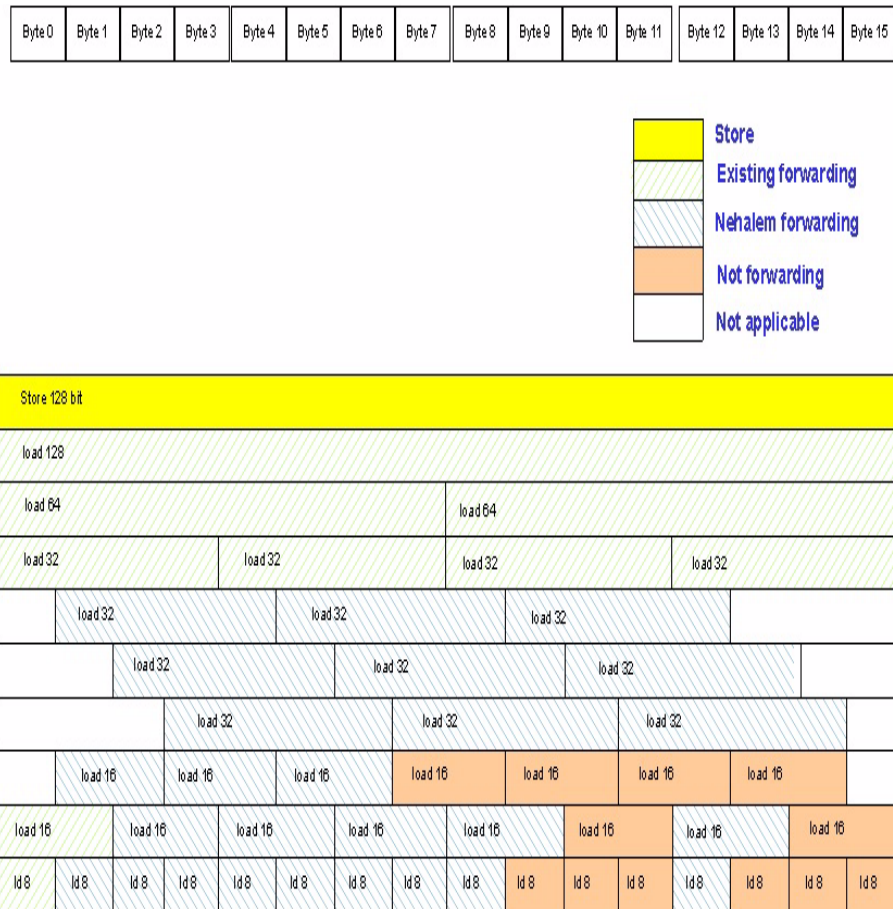


Figure 2-11. Store-Forwarding Scenarios of 16-Byte Store Operations

Intel microarchitecture code name Nehalem allows store-to-load forwarding to proceed regardless of store address alignment (The white space in the diagram does not correspond to an applicable store-to-load scenario). Figure 2-12 illustrates situations for store operation of 8 bytes or less.

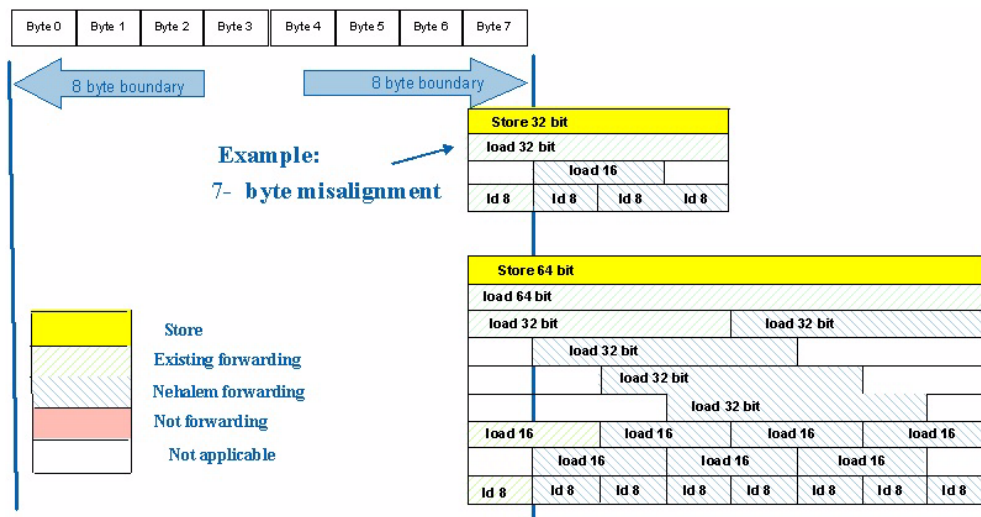


Figure 2-12. Store-Forwarding Enhancement in Intel Microarchitecture Code Name Nehalem

2.4.6 REP String Enhancement

REP prefix in conjunction with MOVSB/STOS instruction and a count value in ECX are frequently used to implement library functions such as memcpy()/memset(). These are referred to as "REP string" instructions. Each iteration of these instruction can copy/write constant a value in byte/word/dword/qword granularity. The performance characteristics of using REP string can be attributed to two components: startup overhead and data transfer throughput.

The two components of performance characteristics of REP String varies further depending on granularity, alignment, and/or count values. Generally, MOVSB is used to handle very small chunks of data. Therefore, processor implementation of REP MOVSB is optimized to handle ECX < 4. Using REP MOVSB with ECX > 3 will achieve low data throughput due to not only byte-granular data transfer but also additional startup overhead. The latency for MOVSB, is 9 cycles if ECX < 4; otherwise REP MOVSB with ECX > 9 have a 50-cycle startup cost.

For REP string of larger granularity data transfer, as ECX value increases, the startup overhead of REP String exhibit step-wise increase:

- Short string (ECX ≤ 12): the latency of REP MOVSW/MOVSQ is about 20 cycles,
- Fast string (ECX ≥ 76: excluding REP MOVSB): the processor implementation provides hardware optimization by moving as many pieces of data in 16 bytes as possible. The latency of REP string latency will vary if one of the 16-byte data transfer spans across cache line boundary:
 - Split-free: the latency consists of a startup cost of about 40 cycles and each 64 bytes of data adds 4 cycles,
 - Cache splits: the latency consists of a startup cost of about 35 cycles and each 64 bytes of data adds 6cycles.
- Intermediate string lengths: the latency of REP MOVSW/MOVSQ has a startup cost of about 15 cycles plus one cycle for each iteration of the data movement in word/dword/qword.

Intel microarchitecture code name Nehalem improves the performance of REP strings significantly over previous microarchitectures in several ways:

- Startup overhead have been reduced in most cases relative to previous microarchitecture,
- Data transfer throughput are improved over previous generation

- In order for REP string to operate in “fast string” mode, previous microarchitectures requires address alignment. In Intel microarchitecture code name Nehalem, REP string can operate in “fast string” mode even if address is not aligned to 16 bytes.

2.4.7 Enhancements for System Software

In addition to microarchitectural enhancements that can benefit both application-level and system-level software, Intel microarchitecture code name Nehalem enhances several operations that primarily benefit system software.

Lock primitives: Synchronization primitives using the Lock prefix (e.g. XCHG, CMPXCHG8B) executes with significantly reduced latency than previous microarchitectures.

VMM overhead improvements: VMX transitions between a Virtual Machine (VM) and its supervisor (the VMM) can take thousands of cycle each time on previous microarchitectures. The latency of VMX transitions has been reduced in processors based on Intel microarchitecture code name Nehalem.

2.4.8 Efficiency Enhancements for Power Consumption

Intel microarchitecture code name Nehalem is not only designed for high performance and power-efficient performance under wide range of loading situations, it also features enhancement for low power consumption while the system idles. Intel microarchitecture code name Nehalem supports processor-specific C6 states, which have the lowest leakage power consumption that OS can manage through ACPI and OS power management mechanisms.

2.4.9 Hyper-Threading Technology Support in Intel® Microarchitecture Code Name Nehalem

Intel microarchitecture code name Nehalem supports Hyper-Threading Technology (HT). Its implementation of HT provides two logical processors sharing most execution/cache resources in each core. The HT implementation in Intel microarchitecture code name Nehalem differs from previous generations of HT implementations using Intel NetBurst microarchitecture in several areas:

- Intel microarchitecture code name Nehalem provides four-wide execution engine, more functional execution units coupled to three issue ports capable of issuing computational operations.
- Intel microarchitecture code name Nehalem supports integrated memory controller that can provide peak memory bandwidth of up to 25.6 GB/sec in Intel Core i7 processor.
- Deeper buffering and enhanced resource sharing/partition policies:
 - Replicated resource for HT operation: register state, renamed return stack buffer, large-page ITLB
 - Partitioned resources for HT operation: load buffers, store buffers, re-order buffers, small-page ITLB are statically allocated between two logical processors.
 - Competitively-shared resource during HT operation: the reservation station, cache hierarchy, fill buffers, both DTLB0 and STLB.
 - Alternating during HT operation: front end operation generally alternates between two logical processors to ensure fairness.
 - HT unaware resources: execution units.

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 2-13 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.

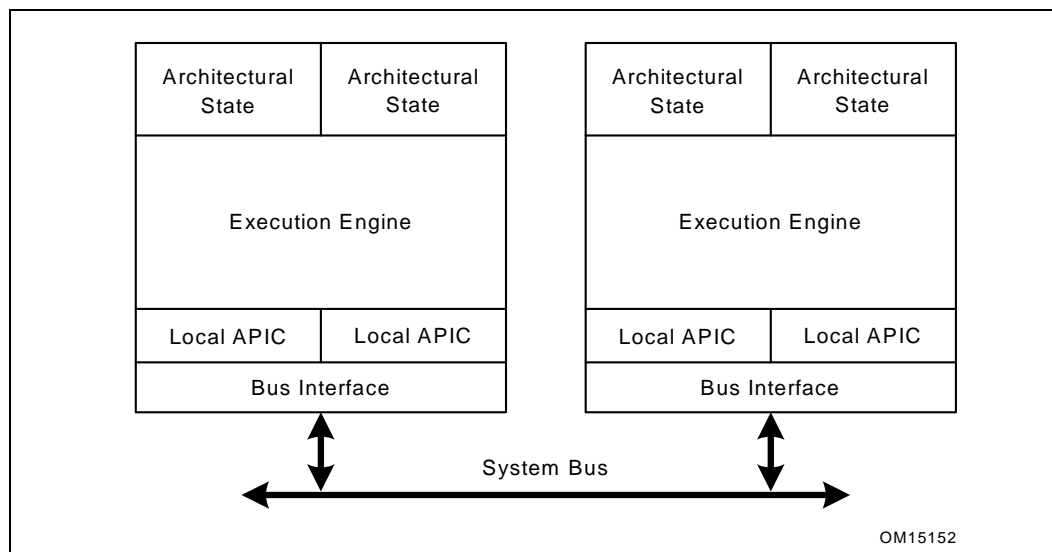


Figure 2-13. Hyper-Threading Technology on an SMP

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 & 3C*.

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 pop 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 μ ops, the μ ops 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 μ ops per cycle, provided the μ ops are ready to execute. Once the μ ops 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 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 (XMM) for streaming SIMD extensions (SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, AESNI, PCLMULQDQ)

- Sixteen 256-bit YMM registers (whose lower 128 bits are overlaid to the respective XMM registers) if AVX, F16C, AVX2 or FMA are supported.
- 64-bit-wide GPRs and instruction pointers
- uniform byte-register addressing
- fast interrupt-prioritization mechanism
- a new instruction-pointer relative-addressing mode

2.7 SIMD TECHNOLOGY

SIMD computations (see Figure 2-14) 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 2-15).

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 2-15). SSE also extended SIMD computational capability by adding additional 64-bit MMX instructions.

Figure 2-14 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.

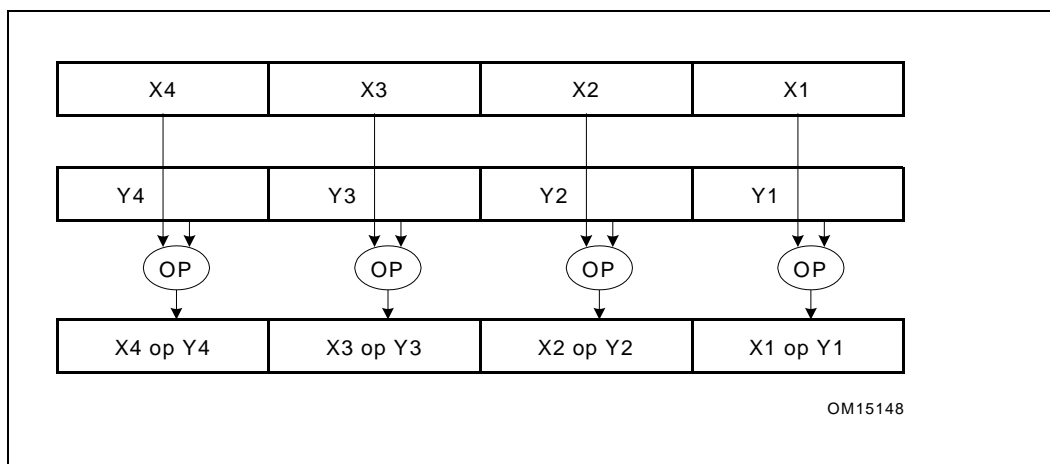


Figure 2-14. Typical SIMD Operations

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 2-15.

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

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

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

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

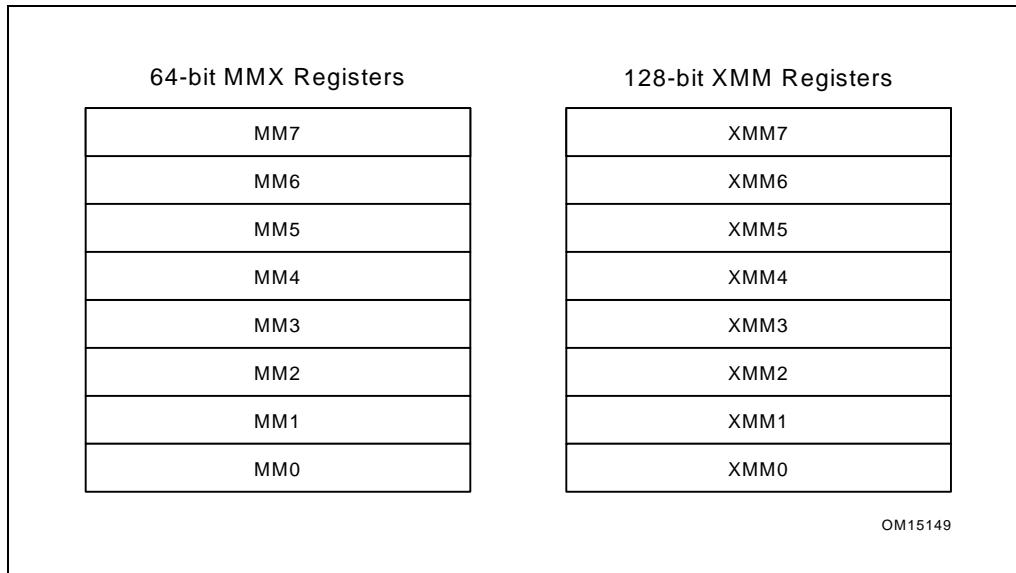


Figure 2-15. SIMD Instruction Register Usage

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

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

2.8 SUMMARY OF SIMD TECHNOLOGIES AND APPLICATION LEVEL EXTENSIONS

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

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

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

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

- Chapter 9, “Programming with Intel® MMX™ Technology”
- Chapter 10, “Programming with Streaming SIMD Extensions (SSE)”
- Chapter 11, “Programming with Streaming SIMD Extensions 2 (SSE2)”
- Chapter 12, “Programming with SSE3, SSSE3 and SSE4”
- Chapter 14, “Programming with AVX, FMA and AVX2”
- Chapter 15, “Programming with Intel® Transactional Synchronization Extensions”

2.8.1 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.

2.8.2 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.

2.8.3 Streaming SIMD Extensions 2

Streaming SIMD extensions 2 add the following:

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

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

2.8.4 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.

2.8.5 Supplemental Streaming SIMD Extensions 3

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

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

2.8.6 SSE4.1

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

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

2.8.7 SSE4.2

SSE4.2 introduces 7 new instructions. These include:

- A 128-bit SIMD integer instruction for comparing 64-bit integer data elements.
- Four string/text processing instructions providing a rich set of primitives, these primitives can accelerate:
 - basic and advanced string library functions from `strlen`, `strcmp`, to `strcspn`,
 - delimiter processing, token extraction for lexing of text streams,

- Parser, schema validation including XML processing.
- A general-purpose instruction for accelerating cyclic redundancy checksum signature calculations.
- A general-purpose instruction for calculating bit count population of integer numbers.

2.8.8 AESNI and PCLMULQDQ

AESNI introduces 7 new instructions, six of them are primitives for accelerating algorithms based on AES encryption/decryption standard, referred to as AESNI.

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

Typically, algorithm based on AES standard involve transformation of block data over multiple iterations via several primitives. The AES standard supports cipher key of sizes 128, 192, and 256 bits. The respective cipher key sizes correspond to 10, 12, and 14 rounds of iteration.

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

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

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

2.8.9 Intel® Advanced Vector Extensions

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

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

Intel AVX instruction set and 256-bit register state management detail are described in IA-32 Intel® Architecture Software Developer’s Manual, Volumes 2A, 2B and 3A. Optimization techniques for Intel AVX is discussed in Chapter 11, “Optimization for Intel AVX, FMA, and AVX2”.

2.8.10 Half-Precision Floating-Point Conversion (F16C)

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

2.8.11 RDRAND

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

2.8.12 Fused-Multiply-ADD (FMA) Extensions

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

2.8.13 Intel AVX2

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

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

2.8.14 General-Purpose Bit-Processing Instructions

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

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

2.8.15 Intel® Transactional Synchronization Extensions

The fourth generation Intel Core processor family introduces Intel® Transactional Synchronization Extensions (Intel TSX), which aim to improve the performance of lock-protected critical sections of multithreaded applications while maintaining the lock-based programming model.

For background and details, see Chapter 15, "Programming with Intel® Transactional Synchronization Extensions" of *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*.

Software tuning recommendations for using Intel TSX on lock-protected critical sections of multithreaded applications are described in Chapter 12, "Intel® TSX Recommendations".


2.8.16 RDSEED

The Intel Core M processor family introduces the RDSEED, ADCX and ADOX instructions.

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

2.8.17 ADCX and ADOX Instructions

The ADCX and ADOX instructions, in conjunction with MULX instruction, enable software to speed up calculations that require large integer numerics. Details can be found at <https://www.intel.com/content/www/us/en/processors/core-m/adc-adox-instructions.html>.



ssl.intel.com/content/www/us/en/intelligent-systems/intel-technology/ia-large-integer-arithmetic-paper.html? and <http://www.intel.com/content/www/us/en/intelligent-systems/intel-technology/large-integer-squaring-ia-paper.html>.