Intel® 64 Architecture Memory Ordering White Paper

Revision: 1.0

SKU: 318147-001

Date: August 2007

Revision History

Rev. No.	Date	Comment	
1.0	August 2007	Initial Release of memory ordering document	

Contents

Legal Information

Overview

This document depicts Intel 64 memory ordering at a level that is architecturally visible to software.[1] The principles and examples provide software writers with a clear understanding of the results that different sequences of memory access instructions may produce. This document does not provide memory ordering rules for I/O operations.

This document discusses only software-visible behavior. Hardware is allowed to perform any optimizations as long as it does not violate any of the visibility principles. (It may even execute a single memory access more than once; if it does, only the final execution is visible to software.) For example “loads are not reordered” means that “loads do not appear to be reordered,” not that the hardware is restricted in how loads are internally implemented.

The principles identified here apply to code as executed by the processor, that is, to code after all compiler optimizations have been performed and the final executable generated.

This document is subject to future revision.

[1] The term Intel 64 refers to both IA-32 and Intel® 64 processors.

1 Instructions and memory accesses

Intel 64 memory ordering guarantees that for each of the following memory-access instructions, the constituent memory operation appears to execute as a single memory access regardless of memory type:

1. Instructions that read or write a single byte.
2. Instructions that read or write a word (2 bytes) whose address is aligned on a 2 byte boundary.
3. Instructions that read or write a doubleword (4 bytes) whose address is aligned on a 4 byte boundary.
4. Instructions that read or write a quadword (8 bytes) whose address is aligned on an 8 byte boundary.

All locked instructions (the implicitly locked xchg instruction and other read-modify-write instructions with a lock prefix) are an indivisible and uninterruptible sequence of load(s) followed by store(s) regardless of memory type and alignment.

Other instructions may be implemented with multiple memory accesses. From a memory-ordering point of view, there are no guarantees regarding the relative order in which the constituent memory accesses are made. There is also no guarantee that the constituent operations of a store are executed in the same order as the constituent operations of a load.

2 Memory ordering for write-back (WB) memory

This section presents a set of memory-ordering principles for coherent write-back (WB) memory. WB memory is what is typically used for memory made available by C *malloc* library calls and by allocation primitives such as *new* in higher-level languages. These principles are illustrated with examples in which each instruction is implemented with a single memory access (see Section 1). This section does not apply to non-temporal operations, fast string operations, or instructions that are not implemented as a single memory access.

The following items explain some of the terminology used in this document:

- An instruction that comes first in program order is referred as an **older** instruction. An instruction which comes later in program order is referred as a **younger** instruction.
- The term **processor** refers to a logical processor, which may be one processing component of a hyperthreaded physical processor or of a multi-core physical package.
- Operations that write to memory have a **total order** if all processors agree on their order of execution.

Intel 64 memory ordering obeys the following principles:

1. Loads are not reordered with other loads.
2. Stores are not reordered with other stores.
3. Stores are not reordered with older loads.
4. Loads may be reordered with older stores to different locations but not with older stores to the same location.
5. In a multiprocessor system, memory ordering obeys causality (memory ordering respects transitive visibility).
6. In a multiprocessor system, stores to the same location have a total order.
7. In a multiprocessor system, locked instructions have a total order.
8. Loads and stores are not reordered with locked instructions.

This document provides a set of examples that illustrate how these principles apply to code sequences. The examples use the following conventions:

- They are written using Intel 64 assembly language syntax. The mov instruction serves as an example of a memory-access instruction; other memory-access instructions (e.g., floating point loads) also obey these principles. The xchg instruction serves as a generic locked instruction; other locked instructions (e.g., lock cmpxchg) also follow the principles for locked instructions.

- Arguments beginning with an “r”, such as r1 or r2 refer to registers visible only to the local processor.
- Variables are denoted with x, y, z.
- Stores are written as *mov [_x], val* which implies that *val* is being stored into the memory location *x*.
- Loads are written as *mov r, [_x]* which implies that the contents of the memory location *x* are being loaded into the register *r.*
- We use M1, M2 etc. to number the instructions in the examples.
- Unless otherwise noted, all registers and memory locations initially contain zero.
- Unless stated otherwise, memory locations do not overlap and are not aliased.

2.1 Loads are not reordered with other loads and stores are not reordered with other stores

Intel 64 memory ordering ensures that loads are seen in program order, and that stores are seen in program order.

Processor 0	Processor 1
mov [_x], 1 // M1 mov [_y], 1 // M2	mov r1,[_y] // M3 mov r2, [_x] // M4
Initially x == y == 0 r1 == 1 and r2 == 0 is not allowed	

Table 2.1: Stores are not reordered with other stores

The stores M1 and M2 cannot be reordered at processor 0 and the loads M3 and M4 cannot be reordered at processor 1. If r1 == 1, M2 appears to execute before M3. This implies that M1 must appear to execute before M4, which implies that r2 == 1.

2.2 Stores are not reordered with older loads

Intel 64 memory ordering ensures that stores do not appear to execute before older loads.

Processor 0	Processor 1
mov r1, [_x] // M1 mov [_y], 1 // M2	mov r2, [_y] // M3 mov [_x], 1 // M4
Initially x == y == 0 r1 == 1 and r2 == 1 is not allowed	

Table 2.2: Stores are not reordered with older loads

Intel 64 memory ordering prevents the store M2 from being reordered with the load M1. Similarly, M4 is not reordered with M3. If r1 == 1, then M1 must see the store M4. This implies that the load M3 appears to be ordered before the store M2, implying r2 == 0.

2.3 Loads may be reordered with older stores to different locations

Intel 64 memory ordering allows load instructions to be reordered with prior stores to a different location. However, loads are not reordered with prior stores to the same location.

The first example in this section illustrates the case in which a load may be reordered with an older store – i.e. if the store and load are to different non-overlapping locations.

Processor 0	Processor 1
mov [_x], 1 // M1 mov r1, [_y] // M2	mov [_y], 1 // M3 mov r2, [_x] // M4
Initially x == y == 0 r1 == 0 and r2 == 0 is allowed	

Table 2.3.a: Loads may be reordered with older stores

At processor 0, the load M2 and the store M1 are to different locations and hence may be reordered. The same is true for M3 and M4. Any interleaving of the operations is thus allowed. One such interleaving has the two loads preceding the two stores. This would result in each load returning value 0.

The following example illustrates that Intel 64 memory ordering prevents loads from being reordered with a prior store to the same location.

Processor 0	Processor 1
mov [_x], 1 // M1 mov r1, [_x] // M2	mov [_y], 1 // M3 mov r2, [_y] // M4
Initially x == y == 0 Must have r1 == 1 and r2 == 1	

Table 2.3.b: Loads are not reordered with older stores to the same location

Intel 64 memory ordering does not allow M2 to be reordered with M1 or M4 to be reordered with M3 because each pair of memory accesses is to the same location. Therefore, r1 == 1 and r2 == 1 must hold.

2.4 Intra-processor forwarding is allowed

Intel 64 memory ordering allows stores by two processors to be seen in different orders by those two processors.

Processor 0	Processor 1
mov [_x], 1 // M1 mov r1, [_x] // M2 mov r2, [_y] // M3	mov [_y], 1 // M4 mov r3, [_y] // M5 mov r4, [_x] // M6
Initially x == y == 0 r2 == 0 and r4 == 0 is allowed	

Table 2.4: Intra-processor forwarding is allowed

Intel 64 memory ordering does not impose any constraint between stores M1 and M4 as they are seen by processor 0 and processor 1. This fact allows processor 0 to see its store M1 before seeing processor 1's M4, while processor 1 sees its store M4 before seeing processor 0's M1. This ensures that an individual processor is self consistent. This allows r2 == 0 and r4 == 0.

In practice, the reordering in Table 2.4 can arise as a result of store-buffer forwarding. While a store is temporarily held in a processor's store buffer, it can satisfy the processor's own loads but is not visible to (and cannot satisfy) loads on other processors.

2.5 Stores are transitively visible

Intel 64 memory ordering ensures transitive visibility of stores – i.e. stores that are causally related appear to execute in an order consistent with the causal relation.

Processor 0	Processor 1	Processor 2
mov [_x], 1 // M1	mov r1, [_x] // M2 mov [_y], 1 // M3	mov r2, [_y] // M4 mov r3, [_x] // M5
Initially x == y == 0 r1 == 1, r2 == 1, r3 == 0 is not allowed		

Table 2.5: Stores are transitively visible

If r1 == 1, store M1 causally precedes load M2 and, therefore, the subsequent store M3. Intel 64 memory ordering therefore ensures that M1 appears to execute before M3 with respect to processor 2. It also prevents the loads M4 and M5 from being reordered (see Section 2.1). Since r2 == 1, M3 appears to precede M4 and, therefore, M1 must appear to precede M5 at processor 2. This implies that r3 == 1.

2.6 Total order on stores to the same location

Accesses to write-back memory are coherent. This implies that any two stores to the same memory location (even by different processors) must appear to all processors to execute in the same order.

Processor 0	Processor 1	Processor 2	Processor 3
mov [_x], 1 // M1	mov [_x], 2 // M2	mov r1, [_x] // M3 mov r2, [_x] // M4	mov r3, [_x] // M5 mov r4, [_x] // M6
Initially x == 0 r1 == 1, r2 == 2, r3 == 2, r4 == 1 is not allowed			

Table 2.6: Total order on stores to same location

Processor 2 and processor 3 must agree on the order of the stores M1 and M2. Suppose M1 appears to execute first (i.e., M2 overwrites x with 2). By the principles discussed in Section 2.1, the loads M3 and M4 at processor 2 cannot be reordered and the loads M5 and

M6 cannot be reordered. Since r3 == 2, the store M2 must have preceded the load M5. Since M1 appears to execute before M2, M6 cannot return 1, the value written by M1.

2.7 Locked instructions have a total order

Intel 64 memory ordering ensures that all processors agree on a single execution order of all locked instructions, including those that access different locations.

Processor 0	Processor 1	Processor 2	Processor 3
xchg [_x], r1 // M1	xchg [_y], r2 // M2	mov r3, [_x] //M3 mov r4, [_y] //M4	mov r5, [_y] //M5 mov r6, [_x] //M6
Initially x == y == 0, r1 == r2 == 1 r3 == 1, r4 == 0, r5 == 1, r6 == 0 is not allowed			

Table 2.7 Total order on locks

By the principles discussed in Section 2.1, the loads M3 and M4 at processor 2 cannot be reordered and the loads M5 and M6 cannot be reordered. If r3 == 1 and r4 == 0, xchg M1 appears to precede xchg M2 with respect to processor 2. Similarly, r5 == 1 and r6 == 0 imply that M2 appears to precede M1 with respect to processor 1. Because Intel 64 memory ordering ensures that all processors agree on a single execution order of locked instructions, this set of return values is not allowed,

2.8 Loads and stores are not reordered with locks

The examples in this section illustrate only cases in which a locked instruction is older than a load or a store. The reader should note that the reordering is also forbidden if the locked instruction is the younger instruction.

Processor 0	Processor 1
xchg [_x], r1 // M1 mov r2, [_y] // M2	xchg [_y], r3 // M3 mov r4, [_x] // M4
Initially x == y == 0, r1 == r3 == 1 r2 == 0 and r4 == 0 is not allowed	

Table 2.8.a: Loads are not reordered with locks

Intel 64 memory ordering prevents M2 from being reordered with M1. Similarly, it prevents M4 from being reordered with M3. As illustrated in Section 2.7, locked instructions are totally ordered: processor 0 and processor 1 must agree on the order of the locked operations M1 and M3. If M1 appears to execute first, r4 == 1. Alternatively, if M3 appears to execute first, then r2 == 1. Thus, it cannot be that r2 == r4 == 0.

Processor 0	Processor 1
xchg [_x], r1 // M1 mov [_y], 1 // M2	mov r2, [_y] // M3 mov r3, [_x] // M4
Initially x == y == 0, r1 == 1 r2 == 1 and r3 == 0 is not allowed	

Table 2.8.b: Stores are not reordered with locks

Intel 64 memory ordering prevents store M2 from being reordered with xchg M1. As illustrated in Section 2.1, Intel 64 memory ordering prevents loads M3 and M4 from being reordered. One can easily deduce from this that if r2 == 1, then r3 == 1.