Advanced Synchronization Facility

Proposed Architectural Specification

Publication #	**45432** Revision: **2.1**
Issue Date:	**March 2009**

Contents

Revision History

Date	Revision	Description
March 2009	2.1	Minor typographic and language corrections
August 2008	2.0	Initial public release

Chapter 1 Introduction

The Advanced Synchronization Facility (ASF) is an AMD64 extension to allow user- and system-level code to modify a set of memory objects atomically without requiring expensive traditional synchronization mechanisms.

The ASF extension provides an inexpensive primitive from which higher-level synchronization mechanisms can be synthesized: for example, multi-word compare-and-exchange, load-locked-store-conditional, lock-free data structures, lock-based data structures that do not suffer from priority inversion, and primitives for software-transactional memory.

ASF is both more flexible and less expensive than existing atomic memory modification primitives. Instead of offering new instructions with hardwired semantics (such as compare-and-exchange for two independent memory locations), ASF only exposes a mechanism for atomically updating multiple independent memory locations and allows software to implement the intended synchronization semantics.

1.1 Overview

ASF works by allowing software to declare speculative regions that specify and modify a set of protected memory locations. Modifications made to protected memory become visible to other CPUs either all at once (when the speculative region finishes successfully) or never (if the speculative region is aborted).

Unlike traditional critical sections, ASF speculative regions do not require mutual exclusion. Multiple ASF speculative regions that may access the same memory locations can be active at the same time on different processors, allowing greater parallelism. When ASF detects conflicting accesses to protected memory, it aborts the speculative region and notifies software, which can retry the operation as desired.

ASF protects memory at cache-line granularity. Despite cache-line size being an implementation detail, software does not have to be concerned with cache lines and can instead work on the level of memory objects, as long as all of the following constraints are met, which are supported by all ASF-capable CPU implementations:

- ASF-protected memory objects have a size of up to 64 bytes and are naturally aligned. (All ASF-capable implementations have a cache-line size of at least 64 bytes.)
- The speculative region does not reference more than four objects (the architecturally guaranteed minimum; more may be supported on a model-specific basis).
- Memory objects protected using ASF do not share cache lines with memory objects that should not be so protected. (False sharing may lead to unwanted protection, exceptions, and unnecessary aborts.)

1.1.1 ASF guarantees

In more detail, ASF guarantees forward progress for speculative regions, provided the following conditions hold:

- The speculative region does not exceed ASF's guaranteed capacity: up to four cacheable memory regions with a size and alignment of 64 bytes. (See Section 1.6 for details.)
- No interrupt or exception is delivered while executing the speculative region.
- There are no conflicting memory accesses from other CPUs.

If one of these conditions does not hold, the speculative region will be aborted (explained in more detail in Section 1.4).

1.1.2 ASF limitations

ASF has the following limitations:

- ASF supports only a limited form of nested speculative regions. (Refer to Section 1.5 for details.)
- ASF only operates on cacheable data and has a weakened memory-access-ordering model in certain respects. Memory ordering can be controlled as necessary via existing fence instructions. (Refer to Section 6.5 for details.)

1.2 Speculative region structure

ASF introduces a set of new instructions for denoting the beginning and end of a speculative region and for protecting memory objects. Additionally, ASF speculative regions first need to specify which memory objects should be protected using special declarator instructions.

Once a set of memory objects is protected, a speculative region can modify these memory objects speculatively. If a speculative region completes successfully, all such modifications become visible to all CPUs simultaneously and *atomically*. Otherwise, the modifications are discarded.

An ASF speculative region has the following structure:

1. The speculative region is entered with the SPECULATE instruction.
2. SPECULATE always writes an ASF status code of zero in rAX and sets the rFLAGS register accordingly. This status code distinguishes between the initial entry into a speculative region and an abort situation. SPECULATE also remembers the address of the instruction following the SPECULATE instruction as the landmark to which control is transferred on an abort.
3. SPECULATE is followed by instructions that check the status code and jump to an error handler if it is not zero (typically JNZ).
4. Declarator instructions (memory-load forms of LOCK MOVx, LOCK PREFETCH, and LOCK PREFETCHW instructions) are used to specify locations for atomic access – memory that ASF should protect. The MOV forms also perform the specified register load.
5. The speculative region (standard x86 instructions) is executed.

6. Once a memory location has been protected using a declarator, it can be read using regular x86 instructions. However, to modify protected memory locations, the speculative region uses memory-store forms of LOCK MOVx instructions. (It is an error to use regular memory-updating instructions for protected memory locations. Doing so results in a #GP exception.)
7. The COMMIT instruction denotes the end of the speculative region and causes the modifications to the protected lines to become visible to the rest of the system.
8. An ABORT instruction is available to programmatically terminate the speculative region with abort rather than commit semantics.

Note that the two declarators LOCK PREFETCH and LOCK PREFETCHW differ from non-LOCK-prefixed prefetches in that they need to check the specified memory address for translation faults and memory-access permission and generate a page fault if unsuccessful. This behavior is necessary because ASF needs to establish a valid translation before it starts monitoring the protected memory location.

Example

The following example code implements compare-and-exchange on two independent memory locations using ASF (dubbed "DCAS" for "double compare-and-swap"). (This code uses immediate retry as the recovery strategy. A real implementation might have a more elaborate recovery strategy, for example, exponential backoff.)

```
; DCAS Operation:
; IF ((mem1 = RAX) && (mem2 = RBX))
; {
;   mem1 = RDI
;   mem2 = RSI
;   RCX = 0
; }
; ELSE
; {
;   RAX = mem1
;   RBX = mem2
;   RCX = 1
; }
; (R8, R9 modified)
;
DCAS:
     MOV      R8, RAX
     MOV      R9, RBX
retry:
     SPECULATE                    ; Speculative region begins
     JNZ      retry               ; Page fault, interrupt, or contention
     MOV      RCX, 1              ; Default result, overwritten on success
     LOCK MOV RAX, [mem1]         ; Specification begins
     LOCK MOV RBX, [mem2]
     CMP      R8, RAX             ; DCAS semantics
     JNZ      out
     CMP      R9, RBX
     JNZ      out
```

```
    LOCK MOV [mem1], RDI          ; Update protected memory
    LOCK MOV [mem2], RSI
    XOR      RCX, RCX             ; Success indication
out:
    COMMIT                        ; End of speculative region
```

1.3 Unprotected memory

ASF only protects memory lines that have been specified using declarator instructions. All other memory remains unprotected and can be modified inside a speculative region using standard x86 instructions. These modifications retain their standard behavior, that is, they become visible to other CPUs immediately and in program order.

1.4 Speculative region aborts

Speculative regions can be aborted at any point because of contention, far control transfers (including those caused by interrupts and faults), or software aborts.

Speculative-region aborts discard modifications to the contents of the protected lines, causing them to be unobservable by other CPUs. However, ASF does not roll back modifications to unprotected memory. Software must be written to accommodate these modifications. In many cases this will simply be a matter of reentering the initialization sequence leading up to the speculative region.

Aborts do not roll back register state (except for the instruction and stack pointers, as described later in this section). Software must be written to handle or ignore modified register contents in case of an abort, or it must avoid modifying them in the speculative regions.

Before an interrupt or exception handler returns, operating-system code or other processes may have executed in the interim. This is of no consequence for the interrupted software as no ASF-related state is maintained across context switches. Other processes may even have executed ASF speculative regions that inspected or modified any of the locations targeted by the interrupted speculative region. The interrupted software will have its speculative region aborted and simply needs to re-inspect the state of the shared data structure as it attempts its speculative region again.

ASF is unusual in that SPECULATE has rollback semantics (much like C's setjmp interface): Speculative-region aborts reset the instruction and stack pointers to the values they had after SPECULATE was first executed. The rAX register is also written with a nonzero status code that provides details of the abort condition, and rFLAGS is set accordingly. The subsequent instructions can inspect the status code or rFLAGS register and direct the control flow (via a conditional jump) to the error handler.

1.5 Nested speculative regions

ASF supports composing a speculative region out of pseudo-nested speculative regions by flattening a hierarchy of SPECULATE-COMMIT pairs into just one speculative region. All pseudo-nested speculative regions share ASF's resources; nested COMMIT instructions do not release any protected lines. For nesting to work, all memory that is protected in the outermost SPECULATE-COMMIT pair plus all nested SPECULATE-COMMIT pairs must fit within ASF's limits for protected lines.

Because of the flattening, memory locations protected in a nested speculative region remain protected in outer speculative regions. Therefore, outer speculative regions need to use LOCK MOV for updating memory locations protected by an inner speculative region. (Use of regular memory-update instructions for protected lines results in a #GP exception.)

To detect nesting, ASF maintains an internal nesting count that is incremented by SPECULATE and decremented by COMMIT. A nested SPECULATE does not define a new checkpoint for rollback. Instead, aborts always roll back to the first SPECULATE that started the speculative region.

1.6 Capacity

A given ASF implementation will have certain capacity constraints caused by hardware limitations, such as the number of locations that can be simultaneously monitored for contention, or the number of stores that can be handled speculatively. There are two aspects to this: a minimum guaranteed capacity, and a larger reference-pattern dependent capacity.

A speculative region is guaranteed to complete, in the absence of disturbances such as faults, interrupts, or contention, as long as the number of protected locations does not exceed the minimum guaranteed capacity, regardless of where in the cacheable address space those locations are. An implementation may also provide a capacity beyond the minimum that can vary depending on which locations are referenced.

For example, an implementation may require that all protected locations simultaneously reside in the data cache for the duration of the speculative region, and if a protected line is displaced from the cache because of replacement, the speculative region is aborted. Hence, a speculative region that happened to reference N+1 locations that all mapped to the same index in an N-way associative data cache would never be able to complete. In this case, the minimum guaranteed capacity would be determined by the cache’s associativity.

For more random reference patterns, a speculative region could however reference many locations before the associativity at any one cache index is exceeded and a protected line is displaced, and hence could often operate successfully on a much larger data set than the guaranteed minimum. However, there would be no guarantee for any given set of references that it would not hit a hardware limitation. In such a scenario, software must provide an alternate means for completing the intended operation in case the ASF hardware cannot handle it – for example employing a global lock (see Section -Lock for a specific example). ASF provides an indication to software of when such a limitation has been hit, distinguishing it from transient conditions which might not be encountered on a retry. (Refer to Section 6.1 for details.)

Implementations may use monitoring and store buffering mechanisms which are not tied to cache associativity. In any event, all ASF implementations architecturally guarantee a minimum capacity of four cache lines. The actual minimum of a given implementation (which may be higher) is reported by CPUID.

For some use cases, the odds that a speculative region with a larger number of reads will succeed can be increased through the use of RELEASE instructions, which remove designated cache lines from the monitored set, lowering the chances of hitting a hardware capacity limit. This could be applicable in such cases as walking a long linked list, where each successive element can be dropped once it has been traversed without being modified.

Chapter 2 Terminology

(Terms set in *italics* are defined in a separate glossary entry.)

ABORT

Instruction that voluntarily aborts a speculative region. See also *Abort* and Section 5.5.

Abort

A condition that causes a *speculative region* to fail. Different abort conditions are distinguished by an abort status code written to rAX when the abort is signaled. In case of an abort, the contents of protected lines and the instruction and stack pointers are rolled back to the values they had when SPECULATE was executed. Aborts in nested speculative regions roll back to the SPECULATE instruction that started the outermost speculative region.

ASF configuration MSR

A model-specific register (MSR) configuring ASF.

Cache line

Aside from their use to reduce memory-access latencies, ASF uses the cache-coherency protocol for detecting *contention*. Therefore, the granularity for ASF memory protection is the size of a cache line.

See also *memory line*.

COMMIT

Instruction that denotes the end of a *speculative region*. See Section 5.4.

Contention

Conflicting memory accesses that usually cause a speculative region to abort. See Section 6.2.

CPU

In this specification, the term “CPU” refers to one logical CPU (one hardware thread executing x86 instructions), irrespective of how these logical CPUs are packaged. (Its use is synonymous to terms like “CPU core” and “x86 thread,” which are not used in this specification.)

Declarator

Instruction that declares a location for atomic access (*protected lines*) during a *speculative region*: LOCK MOVx (load), LOCK PREFETCH, and LOCK PREFETCHW. See Section 5.2.

Far control transfer

A (voluntary or involuntary) control-flow diversion to another privilege level or another code segment. Far control transfers include far-call, far-jump, far-ret, and interrupts. See Section 6.4.

Imprecise exception

Exceptions occurring in a *speculative region* cause an *abort*, rolling back execution flow to the instruction following *SPECULATE* before the exception handler is called. Consequently, the instruction and stack pointers reported to the exception handler do not correspond to the fault site, making the exception imprecise. See Section 6.4.1.1.

Memory line

A region of physical memory that has the same size and alignment as a *cache line*.

Protected line

A *memory line* that is protected during a *speculative region*. ASF maintains atomicity of updates to all protected lines as long as no other CPU contends for it (see *contention*). Otherwise, the speculative region is *aborted*.

RELEASE

Instruction that allows ASF to release one *protected line* before the end of a *speculative region*. Protected lines that have been modified cannot be released.

SPECULATE

Instruction that starts an ASF *speculative region*. In case the speculative region is aborted, the instruction and stack pointer are rolled back to the post-SPECULATE instruction values, and modifications to *protected lines* are discarded.

Speculative region

An ASF speculative region starts with the execution of the *SPECULATE* instruction and ends either when the *COMMIT* instruction is executed or when the speculative region is *aborted*.

Transactional store

Instructions that write to *protected lines*, in particular, memory-store variants of LOCK MOVx. No other instructions are allowed to write to protected lines. See Section 5.3.

Chapter 3 CPUID identification

To determine whether ASF is present and which capabilities it has, use the CPUID instruction.

3.1 Detecting ASF presence and capabilities

CPUID <= EAX = ***TBD***

Return: ASF capabilities, according to the following table:

Register	Bits	Meaning
EDX	31:1	Reserved.
EDX	0	**ASF**: Set to 1 if the CPU supports ASF.
EBX	31:16	Reserved.
EBX	15:0	**ASFCapacity**: ASF capacity. The minimum number of different protected lines in an ASF speculative region that this implementation supports. If ASF is present, this value is always greater than or equal to 4. Note that an ASF implementation might support more than the number of protected lines reported by ASFCapacity under certain conditions; see Section 1.6.

3.2 Detecting the cache-line size

CPUID <= EAX = 0000_0001h

Return: Various information (refer to the CPUID specification for details). Includes:

Register	Bits	Meaning
EBX	15:8	**CLFlush**: Cache-line size. Specifies the size of a cache line in quadwords. (A quadword has a size of eight bytes.) AMD64 implementations supporting ASF always have a cache-line size of at least 8 quadwords (64 bytes).

Chapter 4 Model-specific registers

4.1 ASF configuration MSR

MSR ***TBD*** – ASF_CFG

ASF configuration MSR

This MSR defines ASF's current operating mode.

Bits	Meaning
63:1	Reserved – MBZ
0	**ASFFault**: Fault when ASF capacity is exceeded. When this bit it set to 1, declarators generate a #GP(0) fault when software attempts to protect more lines than supported by ASF. Otherwise, the #GP is suppressed. Instead, the speculative region is aborted and the abort status code is set to *ASF_CAPACITY*.

This MSR is read–write. Its reset value is 0.

4.2 ASF exception IP MSR

MSR ***TBD*** – ASF_EXCEPTION_IP

ASF exception IP MSR

In the case of an exception in a speculative region that causes an abort, ASF saves the rIP of the original fault or trap site in this MSR before aborting the speculative region. (This rIP value is the one that would have been put in the exception frame if the rollback had not happened.) The rIP actually reported in the exception frame is the address of the instruction following the initial SPECULATE instruction due to the rollback.

A bit on the exception handler's stack frame indicates whether the ASF_EXCEPTION_IP MSR contains a valid value. Refer to Section 6.4.1.1 for details.

Bits	Meaning
63:0	**ExceptionIP**: rIP at which an exception or fault occurred before the speculative region was aborted.

This MSR is read-only.

4.3 Debug-control MSR

MSR 0000_01D9 – DebugCtlMSR

The following bit is added to this MSR:

Bits	Meaning
TBD	**DebugAbort**: If set to 1, #DB debug traps abort speculative regions. Otherwise, ASF speculative regions act as an interrupt shadow for debug traps: #DB traps in ASF speculative regions are deferred until after the speculative region has ended. Read-write. Defaults to 0.

Chapter 5 Instructions

This section describes the instructions added to the AMD64 architecture to support ASF. All of these instructions raise #UD if ASF is not implemented or if bit 0 of the ASF_CFG MSR is 0.

5.1 SPECULATE

5.1.1 Instruction

Mnemonic

```
SPECULATE
```

Opcode

TBD

5.1.2 Description

SPECULATE starts an ASF speculative region.

The exact operation of SPECULATE differs depending on whether it is the initial SPECULATE of a top-level speculative region or a nested SPECULATE.

The initial instance of SPECULATE records the (partial) checkpoint to which execution returns if the speculative region is aborted. The checkpoint consists of the values the instruction and stack pointers will have after SPECULATE has completed execution (hence on an abort, control transfers to whatever instruction follows SPECULATE). SPECULATE also clears the rAX register, sets rFLAGS accordingly, and sets the nesting level (an internal processor state variable) to 1.

If an instance of SPECULATE is encountered within an ASF speculative region, it does not checkpoint the instruction and stack pointers but it does clear rAX and set rFLAGS. It also increments the nesting level. An ASF abort will transfer control to the checkpoint recorded by the initial instance of SPECULATE.

The maximum nesting level is 256. If this level is exceeded, SPECULATE raises #GP(0).

5.1.3 Operation

```
IF (NEST_LEVEL = 256)
{
  EXCEPTION [#GP(0)]
}
rAX = 0
NEST_LEVEL += 1
IF (NEST_LEVEL = 1)
{
  SAVED_rSP = rSP
  SAVED_rIP = rIP of next instruction
}
```

5.1.4 Flags affected

SF, ZF, AF, PF, CF set according to result in rAX. OF is set to 0.

5.2 LOCK MOVx (load), PREFETCH, and PREFETCHW

5.2.1 Instruction

Mnemonic

```
LOCK MOV reg,mem
```

Opcodes

F0 8A/r, F0 8B/r, F0 A0, F0 A1

Mnemonic

```
LOCK MOV{D,DQA,DQU,Q} xmm,mem
```

Opcodes

F0 66 0F 6E/r, F0 66 0F 6F/r, F0 F3 0F 6F/r, F0 F3 0F 7E/r

Mnemonic

```
LOCK PREFETCH mem
```

Opcode

F0 0F 0D/0

Mnemonic

```
LOCK PREFETCHW mem
```

Opcode

F0 0F 0D/1

5.2.2 Description

These memory-reference instructions, called *declarators*, are used to specify locations for which atomic access is desired.

Declarators work like their counterparts without the LOCK prefix, with the following additional operation:

Each declarator adds the memory line containing the first byte of the referenced memory object to the set of protected lines. Software must ensure that unaligned memory accesses do not span both protected and unprotected lines; otherwise, the atomicity of data accesses to these memory objects is not guaranteed.

Unlike prefetches without a LOCK prefix, LOCK PREFETCH and LOCK PREFETCHW also check the specified memory address for translation faults and memory-access permission (read or write, respectively) and, if unsuccessful, generate a page-fault or general-protection exception as appropriate. Also, LOCK PREFETCH and LOCK PREFETCHW generate a #DB exception when they reference a memory address for which a data breakpoint has been configured.

A declarator referencing a line that has already been protected is permitted and behaves like a regular memory reference. It does not change the protected status of the line.

Once a memory line has been protected using a declarator, it can be modified speculatively (but cannot be modified nonspeculatively) within the speculative region. See Section 5.3 for instructions that can update protected lines, and Section 6.5 for memory access ordering rules.

If the number of declarators issued in the current speculative region exceeds ASF's maximum supported capacity, the behavior depends on the setting of MSR ASF_CFG[ASFFault]. If that bit is set to 1, a #GP(0) is generated. Otherwise, the #GP is suppressed. Instead, the speculative region is aborted and the abort status code is set to *ASF_CAPACITY*.

Declarators are not allowed outside of speculative regions and result in #UD in this case.

LOCK MOVx from memory with a caching type other than WB (writeback) is not supported by ASF and results in #GP(0).

LOCK MOVD into MMX registers is not supported and results in #UD.

(#GP and #UD, like all interrupts, also abort the speculative region. See Section 6.4.)

5.2.3 Operation

```
IF (CPU not in speculative region)
{
  EXCEPTION [#UD]
}
IF (instruction = LOCK PREFETCHW)
{
  translate memory address and check for write permission
    // Generates #GP or #PF if necessary
}
ELSE
{
  translate memory address and check for read permission
    // Generates #GP or #PF if necessary
}
IF (line already protected)
{
  perform conventional memory reference operation
  EXIT
}
IF (address refers to non-WB memory type)
{
  EXCEPTION [#GP(0)]
}
IF (ASF capacity overflow)
{
  IF (ASF_CFG[ASFFault])
  {
    EXCEPTION [#GP(0)]
  }
  ELSE
  {
    abort speculative region (ASF_CAPACITY, 1, 0) // See Section 6.1
    EXIT
  }
}
execute memory reference and handle contention // See Section 6.2
add line to set of protected lines
```

5.2.4 Flags affected

None.

5.3 LOCK MOVx (store)

5.3.1 Instruction

Mnemonic

```
LOCK MOV mem,reg/imm
```

Opcodes

F0 88/r, F0 89/r, F0 A2, F0 A3, F0 C6/0i, F0 C7/0i

Mnemonic

```
LOCK MOV{D,DQA,DQU,Q} mem,xmm
```

Opcodes

F0 66 0F 7E/r, F0 66 0F 7F/r, F0 F3 0F 7F/r, F0 66 0F D6/r

5.3.2 Description

These memory-store instructions are used to store data into protected lines. The lines must already have been protected by a declarator instruction (see Section 5.2); if not, these store instructions result in #GP(0).

Updates to protected lines do not become visible to other CPUs until the COMMIT instruction is executed. If the speculative region is aborted, these updates will be discarded and cannot be observed from other CPUs.

There are no other instructions to store data into protected lines. Attempting to modify protected lines using regular move instructions or other memory-updating instructions results in #GP(0).

LOCK MOVx store instructions are not allowed outside of speculative regions and result in #UD in this case.

Software must ensure that unaligned memory accesses resulting from LOCK MOVx store instructions do not span both protected and unprotected lines; otherwise, #GP(0) is generated.

LOCK MOVD from MMX registers is not supported and results in #UD.

(#GP and #UD, like all interrupts, also abort the speculative region. See Section 6.4.)

5.3.3 Operation

```
IF (CPU not in speculative region)
```

```
{
  EXCEPTION [#UD]
}
translate memory address and check for write permission
  // Generates #PF if necessary
IF (access spanning protected and unprotected line
    || ! line already protected)
{
  EXCEPTION [#GP(0)]
}
execute memory reference and handle contention // See Section 6.2
```

5.3.4 Flags affected

None.

5.4 COMMIT

5.4.1 Instruction

Mnemonic

```
COMMIT
```

Opcode

TBD

5.4.2 Description

Denotes end of an ASF speculative region.

When the COMMIT belongs to a pseudo-nested speculative region (a nested SPECULATE-COMMIT pair), COMMIT decrements the nesting count and exits without releasing any protected lines.

When COMMIT ends a speculative region (nest count is equal to 1), this instruction releases all protected lines. Modified protected lines will be committed and made visible to other CPUs.

COMMIT sets the rAX register to zero and sets rFLAGS according to the value in rAX. (Future enhancements to ASF may result in COMMIT setting rAX to a value other than zero.)

When encountered outside of a speculative region, the COMMIT instruction raises #GP(0).

5.4.3 Operation

```
IF (CPU not in speculative region)  // NEST_LEVEL = 0
{
  EXCEPTION [#GP(0)]
}
rAX = 0
NEST_LEVEL -= 1
IF (NEST_LEVEL = 0)
{
  commit protected lines
  release protected lines
  end speculative region
}
```

5.4.4 Flags affected

SF, ZF, AF, PF, CF set according to result in rAX. OF is set to 0.

5.5 ABORT

5.5.1 Instruction

Mnemonic

```
ABORT
```

Opcode

TBD

5.5.2 Description

Aborts an ASF speculative region.

In a speculative region, ABORT discards modifications to previously protected lines and releases all protected lines. The contents of the AX register are copied into the SoftwareAbort field of the abort code in rAX. The abort status-code field will be set to *ASF_ABORT*.

Refer to Section 6.1 for a further description of abort behavior.

When encountered outside an ASF speculative region, the ABORT instruction generates #GP(0).

5.5.3 Operation

```
IF (CPU not in speculative region)
{
  EXCEPTION [#GP(0)]
}
abort speculative region (ASF_ABORT, 0, AX) // See Section 6.1
```

5.5.4 Flags affected

None.

5.6 RELEASE

5.6.1 Instruction

Mnemonic

```
RELEASE mem
```

Opcode

TBD

5.6.2 Description

The RELEASE instruction is a hint that allows ASF to remove an unmodified protected line (referenced by the specified memory address) from a speculative region's set of protected lines.

RELEASE can be used to circumvent ASF's capacity limitations when traversing potentially long chains of pointers. However, as the instruction does not guarantee that the specified protected line

will actually be released, software must be designed to fall back to a different code path when the capacity limit is reached.

RELEASE never releases protected lines that have been modified within the speculative region. The circumstances under which RELEASE releases unmodified protected lines are implementation specific.

If RELEASE does release a protected line, then another CPU accessing data contained in that memory line will no longer cause ASF contention. Otherwise, ASF continues to monitor the protected line for contention.

RELEASE does not consider the number of declarators that were used to protect the memory line. In other words, a protected line might be released even if it was specified using more than one declarator.

When attempting to release a line that is not in the current set of protected lines, the instruction is a no-op.

When encountered outside an ASF speculative region, the instruction generates #GP(0).

5.6.3 Operation

```
IF (CPU not in speculative region)
{
  EXCEPTION [#GP(0)]
}
IF (referenced line not in set of protected lines)
{
  EXIT
}
IF (referenced line has not been modified in speculative region)
{
  IF (implementation-specific conditions)
  {
    release referenced line
  }
}
```

5.6.4 Flags affected

None.

Chapter 6 Operation in ASF speculative regions

6.1 Aborts

6.1.1 Description

ASF automatically aborts a speculative region when one of the following conditions occurs:

- Contention for memory that is included in the set of protected lines (see Section 6.2)
- A condition that results in a far control transfer (see Section 6.4)
- Explicit abort (by executing ABORT)
- Other implementation-specific conditions

Aborts discard any modifications to currently locked cache lines and release all protected cache lines. Conditions that cause an abort also set the abort status code to a nonzero value. This code is passed to software in the rAX register, and rFLAGS is set accordingly, when the abort is signaled.

ASF signals aborts by rolling back rSP and rIP to the instruction following the SPECULATE instruction that initiated the speculative region. The conditional jump following SPECULATE can then jump to a recovery routine.

The other registers (general-purpose registers, floating-point registers, XMM registers) are not restored during a roll back. The only way software can rely on the contents of a register after a roll back is by not modifying it in the speculative region. Otherwise, software must be written to ignore, in the case of an abort, the contents of any registers the speculative region might have modified.

When an abort is signaled, the rAX register is always nonzero and has the following layout:

Bits	Meaning
63:32	(In 64-bit mode:) Set to zero
31:16	**SoftwareAbort**: 16-bit value passed to the ABORT instruction. Zero if no ABORT instruction was encountered.
15:8	**NestLevel**: Nesting level in which the abort occurred (equivalent to ASF's nesting count minus 1). Zero if the aborted speculative region has not been nested.
7	**HardError**: If set to 0, the speculative region has been aborted because of a transient error (such as contention) and can be retried. If set to 1, a hard error (such as a capacity overrun) has been detected, requiring a different recovery method.

6:0	**StatusCode**: The reason for the abort. See following table.

The StatusCode and HardError fields have the following meaning:

Status code	Hard error	Meaning
0	0	Success (no abort)
ASF_CONTENTION	0	Speculative region was aborted because of contention.
ASF_ABORT	0	Speculative region aborted using ABORT instruction.
ASF_FAR	0	Speculative region aborted by an exception or an interrupt.
ASF_DISALLOWED_OP	1	Speculative region aborted because of a disallowed instruction. (This status code indicates a programming error.)
ASF_CAPACITY	1	ASF capacity exceeded. The number of declarators exceeded the hardware's capacity for handling them atomically.
Other value	0	Spurious error
Other value	1	Hard error

"*ASF_CONTENTION*", "*ASF_ABORT*", and so on, are symbolic constants that will be defined in a later revision of this document.

Note that it is possible for interrupt handlers to modify the abort status code in rAX when they detect that an ASF speculative region has been aborted (through the Imprecise bit in the rFLAGS image on the stack or in the VMCB; see Section 6.4.1.1). For example, interrupt handlers can convey additional information in the SoftwareAbort field according to a software convention, and exception handlers can set the HardError flag if necessary.

6.1.2 Operation

```
abort speculative region (status_code, hard_error, software_code):
    STATUS_CODE = (status_code
                   | (hard_error << 7)
                   | ((NEST_LEVEL - 1) << 8)
                   | ((software_code & FFFFh) << 16))
    undo modifications to protected lines
    release protected lines
    NEST_LEVEL = 0
    rSP = SAVED_rSP
```

```
rIP = SAVED_rIP
rAX = STATUS_CODE
set EFLAGS.{SF,ZF,AF,PF,CF,OF} according to rAX
```

6.2 Contention

6.2.1 Description

Contention is interference that other CPUs cause when they access memory that has previously been protected. ASF aborts speculative regions under certain types of contention.

The following table summarizes how ASF handles contention in the case where CPU A performs an operation while CPU B is in a speculative region with the line protected by ASF:

CPU A mode	CPU A operation	CPU B cache-line state	
		Protected Shared	**Protected Owned ***
Speculative region	LOCK MOVx (load)	OK	B aborts
Speculative region	LOCK MOVx (store)	B aborts	B aborts
Speculative region	LOCK PREFETCH	OK	B aborts
Speculative region	LOCK PREFETCHW	B aborts	B aborts
Speculative region	COMMIT	OK	OK
Any	Read operation	OK	B aborts
Any	Write operation	B aborts	B aborts
Any	Prefetch operation	OK	B aborts
Any	PREFETCHW	B aborts	B aborts

“Owned *” – Modified or owned

6.2.2 Operation

Memory references:

```
    // "CPU A" refers to the current CPU executing the memory reference;
    // "CPU B" refers to another CPU
    IF (memory reference contends with CPU B's ASF speculative region)
    {
      CPU B -> abort speculative region (ASF_CONTENTION, 0, 0) // See
Section 6.1
    }
    execute memory reference
```

6.3 Disallowed instructions

Privileged instructions (those that must be executed at CPL = 0), instructions that cause a far control transfer or an exception, and all instructions that can be intercepted by an SVM hypervisor are not allowed in an ASF speculative region. This includes:

- FAR JMP, FAR CALL, FAR RET
- SYSCALL, SYSRET, SYSENTER, SYSEXIT
- INT, INT1, INT3, INTO, IRET, RSM
- BOUND, UD2
- PUSHF, POPF, PAUSE, HLT, CPUID, MONITOR, MWAIT, RDTSC, RDTSCP, RDPMC
- IN, OUT
- SIDT, SLDT, SGDT, STR, SMSW
- All privileged instructions
- All SVM instructions

Attempting to execute these instructions causes an #GP fault, which will be handled as a far control transfer (as described in the next section).

6.4 Far control transfers

6.4.1 Description

All far control transfers lead to an abort of the ASF speculative region. Far control transfers include traps, faults, exceptions, NMIs, SMIs, unmasked interrupts, and disallowed instructions converted into an exception (see previous section).

Instructions that directly or indirectly cause a far control transfer, described in Section 6.3, are not allowed inside ASF speculative regions and will generate a #GP exception.

After aborting the speculative region, discarding any modified protected lines, and rolling back rIP and rSP, ASF changes the abort status code to *ASF_FAR* and then executes the far control transfer. Upon return from the far control transfer (or the fault handler invoked by the #GP caused by a disallowed instruction), the conditional jump following SPECULATE can jump to a recovery routine.

6.4.1.1 Imprecise exception reporting

Exceptions, like all other far control transfers, cause ASF speculative regions to be aborted. Therefore, the rIP of the interrupted program that is pushed to the exception-handler stack does not correspond to the instruction that caused the fault or trap (unless a fault occurred at the first instruction after SPECULATE). For this reason, exceptions occurring while an ASF speculative region was active are called *imprecise exceptions*.

ASF saves the rIP of the original fault or trap site in the ASF_EXCEPTION_IP MSR.

To signal that an imprecise exception has occurred, ASF uses a flag (Imprecise) in the rFLAGS register image in the exception-handler stack frame or, in case the exception was intercepted, in the VMCB.RFLAGS state-save-area field. The Imprecise flag bit in the rFLAGS register cannot be set (it always reads as zero). Instructions that read rFLAGS from memory (such as IRET, POPF, and VMRUN) mask out the Imprecise bit when restoring the rFLAGS register. VMRUN uses VMCB.RFLAGS[Imprecise] only when injecting an exception or interrupt into a virtual machine: If the bit is set, the injected event will be marked as imprecise.

Specifically, the rFLAGS image on the exception-handler stack and VMCB.RFLAGS are extended as follows:

Bit	Meaning
31	**Imprecise**: When this bit is set, the exception has aborted an ASF speculative region, and the rIP pushed to the stack or saved to the VMCB may not correspond to the fault site. ASF sets this bit when rolling back an aborted speculative region; in that case, the rIP points to the instruction following the SPECULATE instruction. The ASF_EXCEPTION_IP MSR reports the original fault or trap site. This bit is not available to interrupt handlers invoked through a 16-bit interrupt or trap gate.

The ASF_EXCEPTION_IP MSR will be overwritten every time an imprecise exception occurs. To fully support ASF applications, operating systems should read this value as soon as possible and pass it on to user-level exception handlers.

6.4.1.2 Debug traps

When MSR DebugCtlMSR[DebugAbort] is cleared to 0, debug traps (#DB, caused by hardware breakpoints or single stepping) are deferred until the speculative region ends. Otherwise, they behave like the other far control transfers and abort the speculative region. (Please note that debug traps that abort a speculative region are signaled as imprecise exceptions – see previous subsection.)

In case of a debug trap, the debug-status register (DR6) reflects the conditions valid when the configured breakpoint was hit. If the #DB exception is deferred, DR6 reflects the condition immediately and is not protected from being overwritten before the exception is delivered. In addition, DR6 can accumulate additional breakpoint information throughout the rest of the speculative region.

6.4.1.3 Page faults

In case of page faults, CR2 (page-fault linear address) contains the actual page-fault address before the rollback.

6.4.2 Operation

```
    IF (far control transfer because of a disallowed instruction)
    {
      tmp_hard_error = 1
      tmp_status_code = ASF_DISALLOWED_OP
    }
    ELSE
    {
      tmp_hard_error = 0
      tmp_status_code = ASF_FAR
    }
    IF (CPU in speculative region)
    {
      tmp_rIP = rIP
      abort speculative region (tmp_status_code, tmp_hard_error, 0) // See
Section 6.1
      IF (far control transfer = exception)
      {
        MSR ASF_EXCEPTION_IP = tmp_RIP
        if (exception intercepted)
        {
          VMCB.RFLAGS |= 2^31
        }
        ELSE
        {
          rFLAGS stack-image value |= 2^31
        }
      }
```

```
}
execute far control transfer
```

6.5 Memory access ordering

ASF speculative regions have a different memory access ordering model in that modifications to protected lines cannot be observed from other CPUs until successful completion of a COMMIT instruction, at which point they become visible at once.

While writes to memory locations in unprotected lines become visible in program order, the total order of memory accesses in both protected and unprotected lines after COMMIT or RELEASE (observed from another CPU) is implementation specific. If a stronger ordering model is desired, software needs to insert LFENCE, SFENCE, and MFENCE instructions. For example, if all unprotected memory writes should become visible before all protected ones, software can use SFENCE immediately before COMMIT.

For all other memory modifications, the standard ordering rules apply. In particular, writes occurring before SPECULATE always become visible before all writes in the speculative region – both protected and unprotected ones (not considering incompatible caching types).

Example

Consider the following program:

```
MOV      [mem1], 0
SPECULATE
JNZ      error
LOCK MOV RAX, [mem3]
MOV      [mem2], 0
LOCK MOV [mem3], 0
MOV      [mem4], 0
COMMIT
MOV      [mem5], 0
```

This program can expose any of the following memory write orders (assuming the speculative region is not aborted):

1. mem1, mem2, mem3, mem4, mem5
2. mem1, mem2, mem4, mem3, mem5

Inserting SFENCE just before COMMIT forces the order to be the second one.

Even though writes to protected memory are held pending and do not become visible to other CPUs before COMMIT, all writes (to protected or unprotected memory) appear to be in program order on the executing CPU.

Conventional LOCK-prefixed instructions (such as LOCK CMPXCHG; or XCHG, which has implicit LOCK semantics) have unchanged behavior, including fencing semantics. However, note that it is not possible to use conventional LOCK-prefixed instructions to manipulate ASF-protected memory (only LOCK MOV store instructions can be used to update protected memory).

6.6 Updating Accessed and Dirty bits in page-table entries

When executing an ASF speculative region, the CPU updates the Accessed and Dirty bits of the referenced page-table entries as it would if no speculative region were active. Speculative modifications to protected memory locations thus leads to a set Dirty bit even if the modifications are later discarded because of an abort.

The behavior caused by protecting memory lines (using declarator instructions) containing active page tables (memory lines accessed and updated by the CPU's page-table walker) is undefined.

Chapter 7 ASF usage models

ASF provides considerable flexibility in the construction of synchronization methods. Some basic usage examples are provided here for illustration.

7.1 Lock-free synchronization primitives

7.1.1 Double-word compare and swap

Double compare-and-swap (DCAS) is a primitive that allows atomic manipulation of pointer-based data structures such as doubly linked lists, queues, and trees.

Unlike the example in Section 1.2, this version of DCAS does not implicitly retry in case of contention or aborts, but leaves the retry (and the backoff) to application code. (Also, in that case it does not return the current memory values.)

```
; DCAS Operation:
; IF ((mem1 = RAX) && (mem2 = RBX))
; {
;   mem1 = RDI
;   mem2 = RSI
;   RCX = 0
; }
; ELSE
; {
;   RCX = 1
; }
; (RAX, RBX, R8, R9 modified)
;
DCAS:
    MOV      R8, RAX
    MOV      R9, RBX
    MOV      RCX, 1
    SPECULATE                   ; speculative region begins
    JNZ      fail               ; Bail out if rolled back
    LOCK MOV RAX, [mem1]        ; Specification begins
    LOCK MOV RBX, [mem2]
    CMP      R8, RAX            ; DCAS semantics
    JNZ      out
    CMP      R9, RBX
    JNZ      out
    LOCK MOV [mem1], RDI        ; Update protected memory
    LOCK MOV [mem2], RSI
    XOR      RCX, RCX
```

```
out:
    COMMIT                      ; End of speculative region
fail:
```

7.1.2 Load locked, store conditional

Load locked (LL) and store conditional (SC) are a pair of primitives that allow a store to occur only if a previously loaded memory operand has not been changed.

Typical LL and SC instructions cannot directly be translated to ASF because ASF always rolls back program flow to a point before the first memory reference. However, programs using LL and SC can be expressed using ASF as follows:

```
    SPECULATE                   ; LL/SC section begins
    JNZ      ll_sc_failed
    LOCK MOV RAX, [mem]
    ...
 ; compute new value for mem in RAX
    ...
    LOCK MOV [mem], RAX
    COMMIT                      ; End of speculative region
    ...
; Error handling
ll_sc_failed:
    ...
```

In addition to traditional LL/SC semantics, ASF also supports pipelined LL/SC sequences:

```
    SPECULATE                   ; LL/SC section begins
    JNZ      ll_sc_failed
    LOCK MOV RAX, [mem1]
    LOCK MOV RBX, [mem2]
    LOCK MOV RCX, [mem3]
    ...
    LOCK MOV [mem1], RAX
    LOCK MOV [mem2], RBX
    LOCK MOV [mem3], RCX
    COMMIT                      ; End of speculative region
```

7.2 Lock-free data structures

ASF can be used to construct large speculative regions for manipulating lock-free data structures for which simple primitives such as DCAS are not sufficient or not convenient.

7.2.1 LIFO list manipulation

Lock-free LIFO lists are common linked-list structures where elements can be added or removed at the front of the list by manipulating a list header structure with a single compare-and-exchange instruction, such as CMPXCHG8B. One typical use of such structures is for maintaining a pool of buffers, where a buffer can be popped off the list as needed, and pushed back on when done with. The compare-and-exchange manipulation of the header structure allows the list to be manipulated simultaneously by competing threads without needing a global lock to provide mutually-exclusive access.

This benefit comes with a couple of constraints:

1. The list header must include a version number along with the pointer to the top-most element of the list in order to avoid the *A–B–A problem* where a concurrently executing, or interrupted, pop operation could erroneously modify the header and break the list: When popping off a list element A, the header is updated to point to the second element B, the pointer for which is read from the link field of element A. However, in the time between reading the pointer to B and updating the list header, the header might have been modified multiple times. It might again point to A, but this time A's next pointer might reference a different second element C. A compare-and-exchange operation comparing the list header to A would succeed and change it to B, which might not even be an element of the list anymore. A version number that is incremented each time the header is manipulated, and is included in the compare-and-exchange operation, prevents such erroneous matching on stale values. This requires a compare-and-exchange operation that is larger than the pointer size, and a version field that is large enough that wrap-around causing a false match is sufficiently unlikely.
2. Multiple elements cannot be removed from the list in one operation (although multiple elements can be pushed on in one operation). This latter constraint comes from the fact that one cannot safely walk the list to find the Nth element to point the header to when removing N-1 elements, because other threads can be altering the list at the same time, pushing elements on and/or popping them off.

ASF solves both of these issues. ASF eliminates the need for a version number because it allows the header to be monitored while the top-most element’s link value is read and ultimately used to update the header. Any intervening manipulation of the header, or interrupt of the sequence, causes the operation to abort. Because element A’s link to B is read and used to update the pointer atomically, the A-B-A problem does not exist.

ASF also allows multiple elements to be removed from the list in a single operation. Because the header can be continuously monitored while the list is being walked to find the Nth element, any manipulation of the list during this time will be detected and the operation aborted. With the use of the RELEASE instruction, there is a better chance that the list could be walked without exceeding the hardware's minimum guaranteed capacity. In any event, a suitable response to a hardware capacity limitation, or high contention, would be to simply resort to popping elements from the list one at a time, as is done today.

The following example code demonstrates single-element push and pop using ASF.

```
; PUSH_ELEM Operation:
;   (INPUT: element ptr in RAX)
;   (INPUT: list ptr in RBX)
; RAX->next = RBX->head
; RBX->head = RAX
; (RDX modified)
;
PUSH_ELEM:
retry:
    SPECULATE
    JNZ      retry
    LOCK MOV RDX, [RBX + head]
    MOV      [RAX + next], RDX
    LOCK MOV [RBX + head], RAX
    COMMIT
    RET

; POP_ELEM Operation:
;   (INPUT: list ptr in RBX)
;   (RETURN: element ptr in RAX)
; RAX = RBX->head
; IF (RBX->head != 0)
; {
;   RBX->head = RAX->next
; }
; (RDX modified)
;
POP_ELEM:
retry:
    SPECULATE
    JNZ      retry
    LOCK MOV RAX, [RBX + head]
    TEST     RAX, RAX
    JZ       end
    MOV      RDX, [RAX + next]
    LOCK MOV [RBX + head], RDX
end:
    COMMIT
    RET
```

7.2.2 FIFO queue

The following example presents a FIFO queue implemented using ASF. Without ASF, lock-free FIFO queues supporting multiple readers and writers have considerably higher overhead.

Note that ASF speculative regions can safely dereference pointers once they have been protected: Pointer modifications on other CPUs (for example when elements are removed from the list) will abort the speculative region.

```
; ENQUEUE Operation:
;   (INPUT: element ptr in RAX)
;   (INPUT: list ptr in RBX)
; RAX->next = 0
; IF (RBX->tail != 0)
; {
;   tmp_ptr_next = & RBX->tail->next
; } ELSE {
;   tmp_ptr_next = & RBX->head
; }
; *tmp_ptr_next = RAX
; RBX->tail = RAX
;
ENQUEUE:
     MOV      [RAX + next], 0
retry:
     SPECULATE
     JNZ      retry
     LOCK PREFETCH [RBX + head]
     LOCK MOV RCX, [RBX + tail]
     TEST     RCX, RCX
     JZ       empty_list
     LOCK PREFETCHW [RCX + next]
     LEA      RCX, [RCX + next]
     JMP      ok
empty_list:
     LEA      RCX, [RBX + head]
ok:
     LOCK MOV [RCX], RAX
     LOCK MOV [RBX + tail], RAX
     COMMIT
     RET

; DEQUEUE Operation:
;   (INPUT: list ptr in RBX)
;   (RETURN: element ptr in RAX)
; RAX = RBX->head
; IF (RBX->head != 0)
; {
;   RBX->head = RAX->next
;   IF (RBX->head = 0)
;   {
;     RBX->tail = 0
```

```
;   }
; }
;
DEQUEUE:
retry:
    SPECULATE
    JNZ      retry
    LOCK MOV RAX, [RBX + head]
    LOCK PREFETCH [RBX + tail]
    TEST     RAX, RAX
    JZ       end
    LOCK MOV RDX, [RAX + next]
    LOCK MOV [RBX + head], RDX
    TEST     RDX, RDX
    JNZ      end
    LOCK MOV [RBX + tail], RDX
end:
    COMMIT
    RET
```

7.2.3 Speculative region composition

ASF allows composing large speculative regions out of smaller ones. In effect, ASF flattens the hierarchy of SPECULATE-COMMIT pairs into one large speculative region.

For example, a speculative region that removes a piece of data from one FIFO queue and puts it on another one can be composed of the routines presented in the previous subsections as follows:

```
; DEQUEUE_ENQUEUE Operation:
;   (INPUT: remove-list ptr in RBX)
;   (INPUT: insert-list ptr in RAX)
DEQUEUE_ENQUEUE:
retry_spec:
    SPECULATE
    JNZ      retry_spec
    PUSH     RAX
    CALL     DEQUEUE
    POP      RBX
    CALL     ENQUEUE
    COMMIT
    RET
```

7.3 Coexistence with lock-based critical sections

ASF can be used in conjunction with traditional non-ASF lock-based critical sections by including a read declarator that refers to the lock variable and checking the value of the variable before proceeding. The ASF speculative region tests and monitors the lock variable without modifying it.

For example, consider a data structure such as a B-tree. Concurrent users of the B-tree perform frequent insert and delete operations in a lock-free manner using ASF. Occasionally the B-tree needs rebalancing for efficiency, but such an operation would be beyond ASF's capacity. A global lock associated with the B-tree solves this problem in a straightforward manner: Each ASF speculative region that operates on the B-tree first initiates monitoring of the lock variable with a LOCK MOV and examines the current value of the lock. If the lock variable is set (indicating that some other thread is rebalancing the B-tree), the speculative region commits without doing any modifications (or programmatically aborts using the ABORT instruction) and then retries, effectively spinning on the lock until it clears.

The code that implements the rebalancing operation does not use ASF. It is a traditional lock-based critical section. It acquires the lock with (for example) a test-and-set-bit operation on the lock variable. The resulting write to the lock variable forces any active ASF speculative regions to abort, and upon retry they see that the lock variable is set and wait for it to clear. The rebalancing procedure need not be concerned with other operations that may be in progress and can be executed at any time.

```
; Delete operation
del_btree:
retry:
    SPECULATE
    JNZ      retry
    LOCK MOV EAX, btree_lock     ; Check and monitor global lock
    TEST     EAX, EAX            ; Rebalance in process?
    JE       nolock              ; No
    MOV      EAX, ABORT_REBALANCING ; Software abort code
    COMMIT                       ; Abort speculative region
    JMP      retry
nolock:
 ; Do the real work of deleting, using ASF
 ; If a rebalance starts, this section aborts
    ...
    COMMIT                       ; Delete finished
;
;-------------------------------------------------------------------
;
; Rebalance (does not use ASF)
rebalance_btree:
    LOCK BTS btree_lock, 0       ; Acquire rebalance lock
    JC       done                ; Another thread is rebalancing
 ; Do the rebalancing work
    ...
    MOV      btree_lock, 0       ; Release the lock
done:
```

This technique can also be used more generally as a fallback position for handling reference-pattern-dependent capacity limitations or even contention situations. Depending on the specific use case, it may be subject to some limitations of a traditional critical section, such as not being able to (easily) abort a partially completed update, or provide strong isolation in the face of non-mutex-based and non-ASF-based accesses to the shared data.