Control-flow Enforcement Technology Specification

May 2019

Revision 3.0

Document Number: 334525-003

Revision History

Document Number	Revision Number	Description	Date
334525-001	1.0	Initial release of the document.	June 2016
334525-002	2.0	Numerous updates across chapters include: 1. Added CR0.WP and CR4.CET interaction, CET state save area description, and separate CPUID bits for SS and IBT. 2. Clarified that WRUSS makes the shadow stack store with user-access intent. 3. Updated the definition of the SSS bit in EPT and corresponding fault check. 4. Updated SYSCALL/SYSENTER to clear SSP instead of setting it to IA32_PL0_SSP. 5. Updated SAVESSP/RSTORSSP to close a timing window and renamed SAVESSP to SAVEPREVSSP. 6. Clarified that SETSSBSY causes a #CP exception on token check failure, and uses IA32_PL0_SSP as an implicit operand. 7. Clarified that CLRSSBSY clears SSP on completion and sets CF to indicate invalid token. 8. Updated INCSSP to accept a register source operand. 9. Updated CET MSR description to clarify that writes are always checked for machine canonicality on parts that support 64-bit mode and that bits 1:0 are reserved.	June 2017
334525-003	3.0	1. Numerous pseudocode updates across instructions and various updates across chapters, marked by change bars. 2. Added new sections on constraining speculation with CET enabled 3. Update to section 3.5 “INT3 Treatment”. 4. Added new chapter 9, “Shadow Stack, Paging and EPT”. 5. Added Intel® SGX support for CET.	May 2019

Table of Contents

Figures

1 Introduction

Return-oriented Programming (ROP), and similarly call/jmp-oriented programming (COP/JOP), have been the prevalent attack methodology for stealth exploit writers targeting vulnerabilities in programs. These attack methodology have the common elements:

- A code module with execution privilege and contain small snippets of code sequence with the characteristic: at least one instruction in the sequence being a control transfer instruction that depends on data either in the return stack or in a register for the target address,
- Diverting the control flow instruction (e.g., RET, CALL, JMP) from its original target address to a new target (via modification in the data stack or in the register).

Control-flow Enforcement Technology (CET) provides the following capabilities to defend against ROP/JOP style control-flow subversion attacks:

- Shadow Stack – return address protection to defend against Return Oriented Programming,
- Indirect branch tracking – free branch protection to defend against Jump/Call Oriented Programming.

The rest of this document is organized as follows:
After an overview of Shadow Stack and Indirect Branch Tracking in the rest of this section. Sections 2 and 3 describe the programming environment of Shadow Stack and Indirect Branch Tracking. Sections 4 and 5 describe changes to traditional control flow instructions and task switching behaviors when these new capabilities are enabled. Both Shadow Stack and Indirect Branch Tracking introduce new instruction set extensions, and are described in Sections 6 and 7.

Control-flow Enforcement Technology introduces a new exception class (#CP) with interrupt vector 21. Section 8 covers enumeration, configuration and new exception class. Sections 9 through 17 cover interactions between CET and other IA system enhancement technology, including paging, VMX, SMX, SGX.

NOTE

In sections 4 and 5, text in this color is used to illustrate the extensions to the control transfer instructions and flows for CET.

1.1 Shadow Stack

A shadow stack is a second expand down stack for the program that is used exclusively for control transfer operations. This stack is separate from the data stack and can be enabled for operation individually in user mode or supervisor mode. When shadow stacks are enabled, the CALL instruction pushes the return address on both the data and shadow stack. The RET instruction pops the return address from both stacks and compares them. If the return addresses from the two stacks do not match, the processor signals a control protection exception (#CP). Note that the shadow stack only holds the return addresses and not parameters passed to the call instruction.

The shadow stack is protected from tamper through the page table protections such that regular store instructions cannot modify the contents of the shadow stack. To provide this protection the page table protections are extended to support an additional attribute for pages to mark them as “Shadow Stack” pages. When shadow stacks are enabled, control transfer instructions/flows like near call, far call, call to interrupt/exception handlers, etc. store return addresses to the shadow stack and the access will fault if the underlying page is not marked as a “Shadow Stack” page. However stores from instructions like MOV, XSAVE, etc. will not be allowed. Likewise control transfer instructions like near ret, far ret, iret, etc. when they attempt to read from the shadow stack the access will fault if the underlying page is not marked as a “Shadow Stack” page. This paging protection detects and prevents conditions that cause an overflow or underflow of the shadow stack when the shadow stack is delimited by non-shadow stack guard pages, or any malicious attempts to redirect the processor to consume data from addresses that are not shadow stack addresses.

1.2 Indirect Branch Tracking

The ENDBRANCH (see Section 73 for details) is a new instruction that is used to mark valid jump target addresses of indirect calls and jumps in the program. This instruction opcode is selected to be one that is a NOP on legacy machines such that programs compiled with ENDBRANCH new instruction continue to function on old machines without the CET enforcement. On processors that support CET the ENDBRANCH is still a NOP and is primarily used as a marker instruction by the processor pipeline to detect control flow violations. The CPU implements a state machine that tracks indirect jmp and call instructions. When one of these instructions is seen, the state machine moves from IDLE to WAIT_FOR_ENDBRANCH state. In WAIT_FOR_ENDBRANCH state the next instruction in the program stream must be an ENDBRANCH. If an ENDBRANCH is not seen the processor causes a control protection exception (#CP), else the state machine moves back to IDLE state.

2 Shadow Stacks

A shadow stack is a second expand down stack used exclusively for control transfer operations. This stack is separate from the data stack. The shadow stack is not used to store data and hence is not explicitly writeable by software. Writes to the shadow stack are restricted to control transfer instructions and shadow stack management instructions. The shadow stack feature can be enabled separately in user mode (CPL == 3) or supervisor mode (CPL < 3).

Shadow stacks operate only in protected mode with paging enabled. Shadow stacks cannot be enabled in virtual 8086 mode.

2.1 Shadow Stack Pointer and its Operand and Address Size Attributes

When CET is enabled the processor supports a new architectural register, shadow stack pointer (SSP), when the processor supports the shadow stack feature. The SSP cannot be directly encoded as a source, destination or memory operand in instructions. The SSP points to the current top of the shadow stack.

The width of the shadow stack is 32-bit in 32-bit/compatibility mode and is 64-bit in 64-bit mode. The address-size attribute of the shadow stack is likewise 32-bit in 32-bit/compatibility mode and 64-bit in 64-bit mode.

2.2 Terminology

When shadow stacks are enabled, certain control transfer instructions/flows and shadow stack management instructions do loads/stores to the shadow stack. Such load/stores from control transfer instructions and shadow stack management instructions are termed as shadow_stack_load and shadow_stack_store to distinguish them from a load/store performed by other instructions like MOV, XSAVES, etc.

The pseudocode for the instruction operations use the notation ShadowStackEnabled(CPL) as a test of whether shadow stacks are enabled at the CPL. This term returns a TRUE or FALSE indication as follows:

```
ShadowStackEnabled(CPL):
    IF CR4.CET = 1 AND CR0.PE = 1 AND EFLAGS.VM = 0
        IF CPL = 3
            THEN
                (* Obtain the shadow stack enable from IA32_U_CET MSR (MSR address 6A0H) used to enable
                   feature for CPL = 3 *)
                SHADOW_STACK_ENABLED = IA32_U_CET.SH_STK_EN;
            ELSE
                (* Obtain the shadow stack enable from IA32_S_CET MSR (MSR address 6A2H) used to enable
                   feature for CPL < 3 *)
                SHADOW_STACK_ENABLED = IA32_S_CET.SH_STK_EN;
        FI;
        IF SHADOW_STACK_ENABLED = 1
            THEN
                return TRUE;
            ELSE
                return FALSE;
        FI;
    ELSE
        (* Shadow stacks not enabled in real mode and virtual-8086 mode or if the master CET feature
            enable in CR4 is disabled *)
        return FALSE;
    ENDIF
```

Additionally, the following terms are used:

- ShadowStackPush4B – decrements the shadow stack pointer (SSP) by 4 bytes and copies the 4 byte source operand to the top of the shadow stack.
- ShadowStackPush8B – decrements the shadow stack pointer (SSP) by 8 bytes and copies the 8 byte source operand to the top of the shadow stack.
- ShadowStackPop4B – copies 4 bytes at the current top of stack (indicated by the SSP register) to the location specified with the destination operand. It then increments the SSP register by 4 bytes to point to the new top of stack.
- ShadowStackPop8B – copies 8 bytes at the current top of stack (indicated by the SSP register) to the location specified with the destination operand. It then increments the SSP register by 8 bytes to point to the new top of stack.

2.3 Near CALL and RET Behavior with Shadow Stacks Enabled

When shadow stack is enabled, near CALL, except for calls with a displacement value equal to zero, pushes the return address on both the data stack and the shadow stack. Near RET, when shadow stack is enabled, pops the return address from both the shadow stack and data stack. The data stack pointer (ESP/RSP) is further incremented optionally by ‘n’ bytes if an optional ‘n’ operand was specified. However, the shadow stack pointer (SSP) does not increment. If the return addresses popped from the two stacks are not the same, then the processor causes a control protection exception (#CP) (NEAR-RET) exception.

2.4 Far CALL and RET

The CALL instruction can be used to call a procedure located in a different segment than the current code segment or to a segment at a different privilege level.

On a far CALL to the same privilege level, the processor pushes the CS, LIP (linear address of the return address) and the SSP on the shadow stack and on a far RET pops the SSP, LIP and the CS from the shadow stack. If the CS and LIP do not match the return address as determined by popping the CS and EIP from the data stack, the processor causes a #CP(FAR-RET/IRET) exception.

On a far CALL to a higher privilege level (inter-privilege level call), shadow stack behavior is as follows.

- When the far CALL originates at CPL3, the return addresses are not pushed onto the supervisor shadow stack. Likewise, a far RET to CPL3 from supervisor privilege level (CPL < 3) does not do any verification of the return addresses. On a CPL3 -> CPL<3 transition, the user space SSP is saved to an MSR (IA32_PL3_SSP) and on a CPL<3 -> CPL3 transition is restored from this MSR.
- On an inter-privilege-level call, the call instruction performs a stack switch. The data stack for the supervisor program is located from the current TSS. Likewise, the shadow stack is switched on such transfers. The SSP for the supervisor program is obtained from one of following MSRs depending on the target privilege level.
  - IA32_PL2_SSP if transitioning to ring 2.
  - IA32_PL1_SSP if transitioning to ring 1.
  - IA32_PL0_SSP if transitioning to ring 0.
- A far call from ring 2 to ring 1, ring 2 to ring 0, or from ring 1 to ring 0 is considered a “same privilege class” transfer for shadow stacks. Thus such far calls, subsequent to locating the SSP for the new privilege level, push the CS, LIP and SSP of the calling procedure onto the shadow stack of the called procedure. Likewise, the far RET will verify the CS and LIP from the shadow stack matches the return address as determined by the CS and EIP obtained from the data stack.

2.4.1 Supervisor Shadow Stack Token

On an inter-privilege far CALL, CET verifies a **supervisor shadow stack token** that is setup by the supervisor when creating shadow stacks intended to be used on these transfers. The supervisor shadow stack token is a 64-bit value formulated as follows.

- Bit 63:3 – Bits 63:3 of the linear address of the supervisor shadow stack token.
- Bit 2 – Reserved. Must be zero.
- Bit 1 –Reserved. Must be zero.
- Bit 0 – Busy bit. If 0, indicates this shadow stack is not active on any logical processor. If 1, indicates this shadow stack is currently active on one of the logical processors.

The following figure illustrates a supervisor shadow stack with a supervisor shadow stack token located at its base.

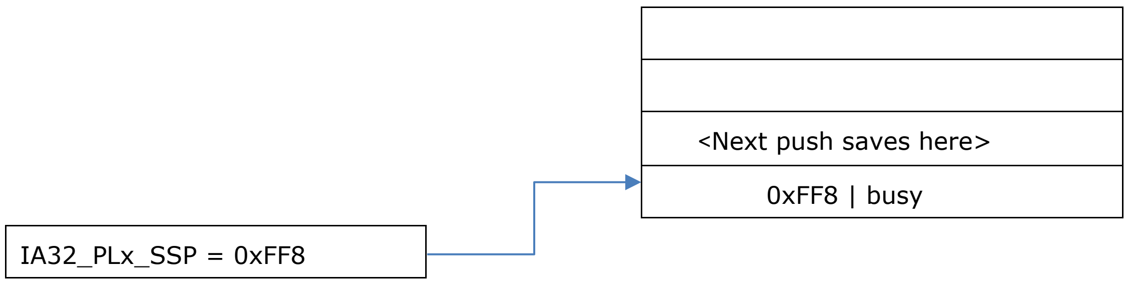

Figure 1 Supervisor Shadow Stack with a Supervisor Shadow Stack Token

The address specified in the IA32_PLx_SSP MSR is required to be 8 byte aligned. The processor does the following checks prior to switching to a supervisor shadow stack programmed into the IA32_PLx_SSP MSR. These steps are performed atomically.

1. Load the supervisor shadow stack token from the address specified in the IA32_PLx_SSP MSR using a shadow_stack_load.
2. Check if the busy bit in the token is 0; reserved bits must be 0.
3. Check if the address programmed in the MSR matches the address in the supervisor shadow stack token; reserved bits must be 0.
4. If checks 2 and 3 are successful, then set the busy bit in the token using a shadow_stack_store and switch the SSP to the value specified in the IA32_PLx_SSP MSR.
5. If checks 2 or 3 fail, then the busy bit is not set and a #GP(0) exception is raised.

On a far RET, the instruction clears the busy bit in the shadow stack token as follows. These steps are also performed atomically.

1. Load the supervisor shadow stack token from the SSP using a shadow_stack_load.
2. Check if the busy bit in the token is 1; reserved bits must be 0and reserved bits must be 0.

3. Check if the address programmed in supervisor shadow stack token matches SSP; reserved bits must be 0.
4. If checks 2 and 3 are successful, then clear the busy bit in the token using a shadow_stack_store; else continue without modifying the contents of the shadow stack pointed to by SSP.

The operations described here are also applicable to a far transfer performed when calling an interrupt or exception handler through an interrupt/trap gate in the IDT. Likewise, the IRET instruction behaves similar to the Far RET instruction.

2.5 Stack Switching on Call to Interrupt/Exception Handlers in 64-bit Mode

The 64-bit mode operation provides a stack-switching mechanism called Interrupt Stack Table (IST) wherein the 64-bit IDT descriptor can be used to specify one of seven data stack pointers in the 64-bit TSS. If the IST index specified is 0 and there is no privilege change involved then a stack switch occurs to the same stack.

To support this stack-switching mechanism, the shadow stack feature provides an MSR, IA32_INTERRUPT_SSP_TABLE, to program the linear address of a table of seven shadow stack pointers. When a non-zero IST value is specified, the MSR points to a 64 byte table in memory that is indexed using the IST index.

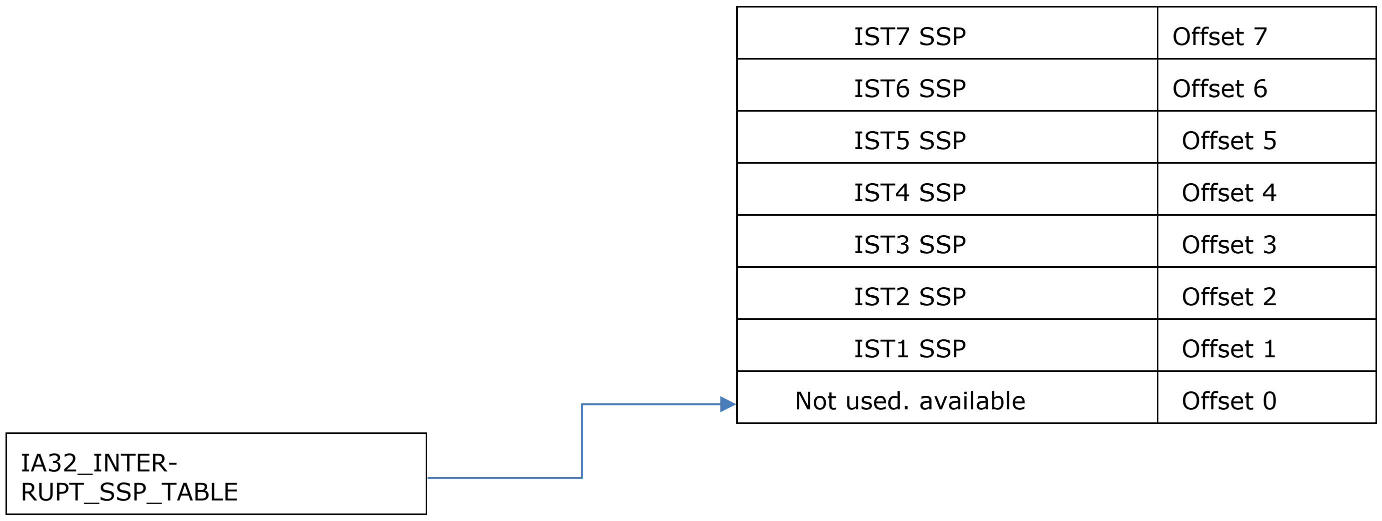

Figure 2 Interrupt Shadow Stack Table

2.6 Shadow Stack Usage on Task Switch

A task switch (see Section 5 "Task Management Interactions with CET") may be invoked by:

- JMP or CALL instruction to a TSS descriptor in the GDT.
- JMP or CALL instruction to a task-gate descriptor in the GDT or the current LDT.
- An interrupt or exception vector points to a task-gate descriptor in the IDT.

With shadow stack enabled, the new task must be associated with a 32-bit TSS and must not be in virtual-8086 mode. The 32-bit SSP for the new task is located at offset 104 in the 32-bit TSS. Thus the TSS of the new task must be at least 108 bytes. This SSP is required to be 8 byte aligned, and required to point to a "supervisor shadow stack" token (though the task may be at CPL3).

On a task switch initiated by a CALL instruction, an interrupt, or exception, the SSP of the old task is pushed onto the shadow stack of the new task along with the CS and LIP of the old task. This is true even for a nested task switch initiated by a CALL instruction. Likewise, on a task switch initiated by IRET, the SSP of the new task is restored from the shadow stack of old task. The CS and LIP on the shadow stack of the old task are matched against the return address determined by the CS and EIP of the new task. If the match fails, a #CP(FAR-RET/IRET) exception is reported.

2.7 Switching Shadow Stacks

The architecture provides a mechanism to switch shadow stacks using a pair of instructions; RSTORSSP and SAVEPREVSSP. The RSTORSSP instruction verifies a "shadow stack restore" token located at the top of the new shadow stack and referenced by the memory operand of this instruction. After RSTORSSP determines the validity of the restore point on the new shadow stack, it switches the SSP to point to the token. The "shadow stack restore" token is a 64-bit value formatted as follows.

- Bit 63:2 – Value of shadow stack pointer when this restore point was created.
- Bit 1 – Reserved. Must be zero.
- Bit 0 – Mode bit. If 0, the token is a compatibility/legacy mode "shadow stack restore" token. If 1, then this shadow stack restore token can be used with a RSTORSSP instruction in 64-bit mode.

The "shadow stack restore" token is created by the SAVEPREVSSP instruction. The operating system may also create a restore point on a shadow stack by creating a "shadow stack restore" token.

Once the shadow stack has been switched to a new shadow stack by the RSTORSSP instruction, software can create a restore point on the old shadow stack by executing the SAFEPREVSSP instruction. In order to allow the SAVEPREVSSP instruction to determine the address where to save the "shadow stack restore" token, the RSTORSSP instruction replaces the "shadow stack restore" token with a "previous ssp" token that holds the value of the SSP at the time the RSTORSSP instruction was invoked. The "previous ssp" token is formatted as follows.

- Bit 63:2 – Shadow stack pointer when the RSTORSSP instruction was invoked, i.e., the SSP of the old shadow stack.
- Bit 1 – Set to 1.
- Bit 0 – Mode bit. If 0, then this "previous ssp" token can be used with a SAVEPREVSSP instruction in compatibility/legacy mode. If 1, then this "previous ssp" token can be used with a SAVEPREVSSP instruction in 64-bit mode.

The following figure illustrates the RSTORSSP instruction operation during a shadow stack switching sequence.

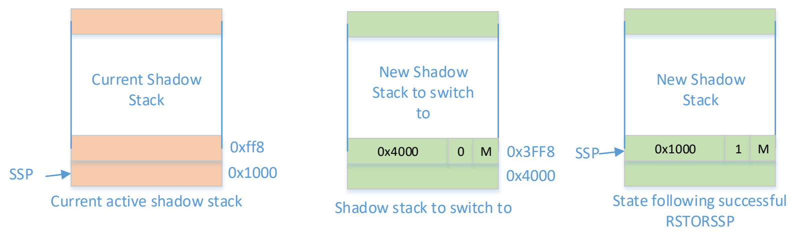
Current Shadow Stack
0xff8
SSP
0x1000
Current active shadow stack
New Shadow Stack to switch to
0x4000
0
M
0x3FF8
0x4000
Shadow stack to switch to
New Shadow Stack
SSP
0x1000
1
M
State following successful RSTORSSP

Figure 3 RSTORSSP to switch to new shadow stack

In this example, the initial SSP is 0x1000 and the "shadow stack restore" token is on a new shadow stack at address 0x3FF8. The token at address 0x3FF8 holds the SSP when this restore point was created; in this example it is 0x4000.

In order to switch to the new shadow stack, the RSTORSSP instruction is invoked with the memory operand pointing set to 0x3FF8. When the RSTORSSP instruction completes, the SSP is set to 0x3FF8 and the "shadow stack restore" token at 0x3FF8 is replaced by a "previous ssp" token that holds the address 0x1000, i.e., old SSP.

The following figure illustrates the SAVEPREVSSP instruction operation during a shadow stack switching sequence.

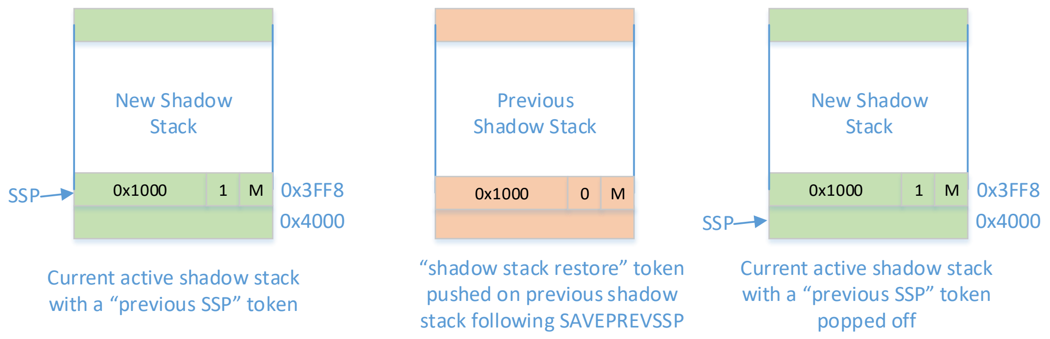
New Shadow Stack
SSP
0x1000
1
M
0x3FF8
0x4000
Current active shadow stack with a "previous SSP" token
Previous Shadow Stack
0x1000
0
M
"shadow stack restore" token pushed on previous shadow stack following SAVEPREVSSP
New Shadow Stack
SSP
0x1000
1
M
0x3FF8
0x4000
Current active shadow stack with a "previous SSP" token popped off

Figure 4 SAVEPREVSSP to save a restore point

To allow switching back to this old shadow stack, a SAVEPREVSSP instruction is now invoked. The SAVE-PREVSSP instruction does not take any memory operand and expects to find a "previous ssp" token at the top of the shadow stack, i.e., at address 0x3FF8. The SAVEPREVSSP instruction then saves a "shadow stack restore" token on the old shadow stack at address 0xFF8, and the token itself holds the address 0x1000 which is the address recorded in the "previous ssp" token. The SAVEPREVSSP instruction also pops the "previous ssp" token off the current shadow stack and thus the SSP following SAVEPREVSSP is 0x4000.

Subsequently to switch back to the old shadow stack, a RSTORSSP instruction may be invoked with memory operand set to 0xFF8.

If, following a switch to a new shadow stack, it is not required to create a restore point on the old shadow stack, then the "previous ssp" token created by the RSTORSSP instruction can be popped off the shadow stack by using the INCSSP instruction.

See the SAVEPREVSSP and RSTORSSP instruction operations for the detailed algorithm.

2.8 Constraining Execution at Targets of RET

Instructions at the target of a RET instruction will not execute, even speculatively, if the RET addresses (either from normal stack or shadow stack) are speculative-only or do not match, unless the target of the RET is also predicted (e.g., by a Return Stack Buffer prediction), when CET shadow stack is enabled. A RET address would be speculative-only if it was modified by an older speculative-only store, or was an older value than the most recent value stored to that address on the logical processor.

3 Indirect Branch Tracking

When the indirect branch tracking feature is active, the indirect JMP/CALL instruction behavior changes as follows.

- JMP – If the next instruction retired after an indirect JMP is not an ENDBR32 instruction in legacy and compatibility mode, or ENDBR64 instruction in 64-bit mode, then a #CP fault is generated. Below JMP instructions are tracked to enforce an endbranch. Note that Jcc, RIP relative, and far direct JMP are not included as these have an offset encoded into the instruction and are not exploitable to create unintended control transfers.
    - JMP r/m16, JMP r/m32, JMP r/m64
    - JMP m16:16, JMP m16:32, JMP m16:64
- CALL – If the next instruction retired after an indirect CALL is not an ENDBR32 instruction in legacy and compatibility mode, or ENDBR64 in 64-bit mode, then a #CP fault is generated. Below CALL instructions are tracked to enforce an endbranch. Note that relative and zero displacement forms of CALL instructions are not included as these have an offset encoded into the instruction and are not exploitable to create unintended control transfers.
    - CALL r/m16, CALL r/m32, CALL r/m64
    - CALL m16:16, CALL m16:32, CALL m16:64

The ENDBR32 and ENDBR64 instructions will have the same effect as the NOP instruction on Intel 64 processors that do not support CET. On processors supporting CET, these instructions do not change register or flag state. This allows CET instrumented programs to execute on processors that do not support CET. Even when CET is supported and enabled, these NOP–like instructions do not affect the execution state of the program, do not cause any additional register pressure, and are minimally intrusive from power and performance perspectives.

The processor implements two dual-state machines to track indirect CALL/JMP for terminations. One state machine is maintained for user mode and one for supervisor mode. At reset the user and supervisor mode state machines are in IDLE state.

On instructions other than indirect CALL/JMP, the state machine stays in the IDLE state.

On an indirect CALL or JMP instruction, the state machine transitions to the WAIT_FOR_ENDBRANCH state.

In the WAIT_FOR_ENDBRANCH state, the indirect branch tracking state machine verifies the next instruction is an ENDBR32 instruction in legacy and compatibility mode, or ENDBR64 instruction in 64-bit mode, and either:

- Causes a #CP fault, or
- Allows the next instruction if legacy compatibility configuration allows (see section 3.6).

The priority of the #CP(ENDBRANCH) exception relative to other events is as follows.

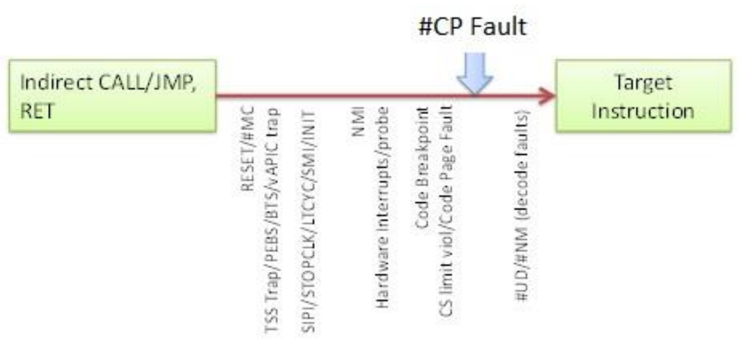
#CP Fault
Indirect CALL/JMP, RET
Target Instruction
RESET/#MC
TSS Trap/PEBS/BTS/vAPIC trap
SIPI/STOPCLK/LTCYC/SMI/INIT
NMI
Hardware interrupts/probe
Code Breakpoint
CS limit viol/Code Page Fault
#UD/#NM (decode faults)

Figure 5 Priority of Control Protection Exception on Missing Endbranch

Higher priority faults/traps/events that occur at the end of an indirect CALL/JMP are delivered ahead of any #CP(ENDBRANCH) fault. The CET state machine at the privilege level where the higher priority fault/trap/event occurred retains its state when the control transfers to the fault/trap/event handler. The instruction pointer pushed on the stack for a #CP(ENDBRANCH) fault is the address of the instruction at the target of the indirect CALL/JMP that caused the fault.

3.1 No-track Prefix for Near Indirect Call/Jmp

CET allows software to designate certain indirect CALL and JMP instructions as "non-tracked indirect control transfer instructions". When enabled by setting the NO_TRACK_EN control in the IA32_U_CET/IA32_S_CET MSR, near indirect CALL and JMP instructions when prefixed with 3EH do not modify the CET indirect branch tracker. Far CALL and JMP instructions are always tracked and ignore the 3EH prefix. When this control is 0, near indirect CALL and JMP instructions are always tracked irrespective of the presence of the 3EH prefix.

In 64-bit mode, the 3EH prefix on an indirect CALL or JMP is recognized as a no-track prefix if there isn't a 64H/65H prefix on the instruction.

In legacy/compatibility mode, the 3EH prefix on an indirect CALL or JMP is recognized as a no-track prefix when it is the last group 2 prefix on the instruction.

3.2 Terminology

The pseudocode for the instruction operations use a notation EndbranchEnabled(CPL) as a test of whether indirect branch tracking is enabled at the CPL. This term returns a TRUE or FALSE indication as follows.

```
EndbranchEnabled(CPL):
        IF CR4.CET = 1 AND CR0.PE = 1 AND EFLAGS.VM = 0
            IF CPL = 3
                THEN
                    (* Obtain the endbranch enable from MSR used to enable feature for CPL = 3 *)
                    ENDBR_ENABLED = IA32_U_CET.ENDBR_EN;
                ELSE
                    (* Obtain the endbranch enable from MSR used to enable feature for CPL < 3 *)
                    ENDBR_ENABLED = IA32_S_CET.ENDBR_EN;
            FI;
            IF ENDBR_ENABLED = 1
                THEN
                    return TRUE;
                ELSE
                    return FALSE;
            FI;
        ELSE
            (* Indirect branch tracking is not enabled in real mode and virtual-8086 mode or if the master CET feature
               enable in CR4 is disabled *)
            return FALSE;
        ENDIF
```

Likewise the notation EndbranchEnabledAndNotSuppressed is defined as follows:

```
EndbranchEnabledAndNotSuppressed(CPL):
    IF CR4.CET = 1 AND CR0.PE = 1 AND EFLAGS.VM = 0
        IF CPL = 3
            THEN
                (* Obtain the endbranch enable from MSR used to enable feature for CPL = 3 *)
                ENDBR_ENABLED = IA32_U_CET.ENDBR_EN;
                SUPPRESSED = IA32_U_CET.SUPPRESS;
            ELSE
                (* Obtain the endbranch enable from MSR used to enable feature for CPL < 3 *)
                ENDBR_ENABLED = IA32_S_CET.ENDBR_EN;
                SUPPRESSED = IA32_S_CET.SUPPRESS;
        FI;
        IF ENDBR_ENABLED = 1 AND SUPPRESSED = 0
            THEN
                return TRUE;
            ELSE
                return FALSE;
        FI;
    ELSE
        (* Indirect branch tracking is not enabled in real mode and virtual-8086 mode or if the master CET feature
           enable in CR4 is disabled *)
        return FALSE;
    ENDIF
```

3.3 Control Transfer Tracking

The hardware implements two CET indirect branch tracker state machines, one for user mode (CPL == 3) and one for supervisor mode (CPL < 3). At any time, which of the CET indirect branch trackers is in the active state depends on the CPL of the machine. When a user space program is executing, the CPL 3 CET indirect branch tracker is active. When supervisor mode software is executing, the CPL < 3 tracker is active. This section describes the various control transfer conditions and the tracker state on those transfers.

3.3.1 Control Transfers between CPL 3 and CPL < 3

Some events and instructions can cause control transfer to occur from CPL 3 to CPL < 3, and vice versa. As part of the CPL change the hardware also switches the active CET indirect branch tracker. For example, when an interrupt occurs during execution of a user mode (CPL == 3) program and it causes the CPL to switch to supervisor mode (CPL < 3) then, as part of the CPL change, the user mode CET indirect branch tracker becomes inactive and the supervisor mode CET indirect branch tracker becomes active. A subsequent iret is used by the interrupt handler to return to the interrupted user mode program. This iret causes the processor to switch the CPL to user mode (CPL ==3) and, as part of the CPL change, the supervisor mode CET indirect branch tracker becomes inactive and the user mode CET indirect branch tracker becomes active.

The CPL where the event or instruction that caused the control transfer occurs is termed the source CPL, and the CET indirect branch tracker state at that CPL is referred here as the source CET indirect branch tracker state. The CPL reached at the end of the control transfer is termed the destination CPL, and the CET indirect branch tracker state at that CPL is referred to as the destination CET indirect branch tracker state.

This section describes various cases of control transfers that occur between user mode (CPL 3) and supervisor mode (CPL < 3).

In all these cases the source CET indirect branch tracker state becomes not active and retains its state (IDLE, WAIT_FOR_ENDBRANCH), and the target CET indirect branch tracker state becomes active if there was no fault during the transfer.

- Case 1: FAR call/jmp, SYSCALL/SYSENTER
  - If indirect branch tracking is enabled, the target indirect branch tracker state becomes active and is unsuppressed and goes to WAIT_FOR_ENDBRANCH. This enforces that the subroutine invoked by a far call/jmp must begin with an endbranch.
- Case 2: Hardware interrupt/trap/exception/NMI/Software interrupt/Machine Checks
  - If indirect branch tracking is enabled, the target indirect branch tracker state becomes active and is unsuppressed and goes to WAIT_FOR_ENDBRANCH.
- Case 3: IRET
  - If indirect branch tracking enabled, the target indirect branch tracker becomes active and keeps its state. If the user mode was interrupted by a higher priority event, like an interrupt at the end of the indirect call/jmp, then when an iret is used to return to the interrupted user mode program, the user mode indirect branch tracker retains its state and a #CP fault will occur if the next instruction decoded is not an endbr32/64 according to mode of machine.

3.3.2 Control Transfers within CPL 3 or CPL < 3

Some events and instructions can cause control transfer to occur within CPL 3 or CPL < 3. For such transfers since the CPL class does not change, the same indirect branch tracker is used at the beginning and end of the control transfer.

- Case 1: FAR CALL/JMP, Near indirect call/jmp
  - FAR CALL/JMP: If indirect branch tracking is enabled, active indirect branch tracker is unsuppressed and goes to WAIT_FOR_ENDBRANCH.
  - Near indirect call/jmp: If indirect branch tracking is enabled and not suppressed, active indirect branch tracker goes to WAIT_FOR_ENDBRANCH.
- Case 2: Hardware interrupt/trap/exception/NMI/Software interrupt/Machine Checks
  - If indirect branch tracking is enabled, the active indirect branch tracker is unsuppressed and goes to WAIT_FOR_ENDBRANCH.
- Case 3: IRET
  - If indirect branch tracking is enabled, the active indirect branch tracker keeps its state.

3.4 Indirect Branch Tracking State Machine

The state machine is described by following table.

Current State	Trigger	Next state
TRACKER=IDLE, SUPPRESS=0, ENDBR_EN=1	Instructions other than indirect call/jmp or 3EH prefixed near indirect call/jmp and NO_TRACK_EN=1	TRACKER=IDLE, SUPPRESS=0, ENDBR_EN=1
	Indirect call/jmp without 3EH prefix Indirect call/jmp with 3EH prefix and NO_TRACK_EN=0 Far call/jmp	TRACKER=WAIT_FOR_ENDBRANCH, SUPPRESS=0, ENDBR_EN=1
TRACKER= WAIT_FOR_ENDBRANCH, SUPPRESS=0, ENDBR_EN=1	INT3/INT1	TRACKER= WAIT_FOR_ENDBRANCH, SUPPRESS=0, ENDBR_EN=1
	Endbranch instruction	TRACKER=IDLE, SUPPRESS=0, ENDBR_EN=1
	Successful ENCLU[ERESUME]	TRACKER=IDLE, SUPPRESS=0, ENDBR_EN=1
	Instructions other than endbranch, successful ENCLU[ERESUME] or int3 or int1	If legacy compatibility treatment is not enabled or if not allowed by legacy code page bitmap: • No state change and deliver #CP (ENDBRANCH) If legacy compatibility treatment is enabled and transfer allowed by legacy code page bitmap: • TRACKER=IDLE, SUPRESS=!SUPPRESS_DIS, ENDBR_EN=1
TRACKER=x, SUPPRESS=x, ENDBR_EN=0	All instructions	TRACKER=x, SUPPRESS=x, ENDBR_EN=0
TRACKER=IDLE, SUPPRESS=1, ENDBR_EN=1	FAR CALL/JMP, INTn/INT3/INTO	TRACKER=WAIT_FOR_ENDBRANCH, SUPPRESS=0, ENDBR_EN=1
	Endbranch instruction Successful ENCLU[ERESUME]	TRACKER=IDLE, SUPPRESS=0, ENDBR_EN=1
	All other instructions including indirect call/jmp	TRACKER=IDLE, SUPPRESS=1, ENDBR_EN=1
TRACKER=1, SUPPRESS=1, ENDBR_EN=1 (This state cannot be reached by hardware and is disallowed as a valid state by WRMSR/XRSTORS/VM entry/VM exit)	N/A	N/A

3.5 INT3 Treatment

INT3 are treated special in the WAIT_FOR_ENDBRANCH state. Occurrence of INT3 do not move the tracker to IDLE but instead the #BP trap from the INT3 instructions respectively is delivered as a higher priority event than the #CP exception due to missing endbranch.

Inside an enclave, INT3 delivers a fault-class exception and thus does not require the CPL to be less than DPL in the IDT gate 3. Following opt-out entry, the instruction delivers #UD. Following opt-in entry, INT3 delivers #BP. The special treatment of INT3 in WAIT_FOR_ENDBRANCH state does not apply in enclave mode following opt-out entry.

3.6 Legacy Compatibility Treatment

Endbranch Legacy compatibility treatment allows a CET enabled program to be used with legacy software that was not compiled / instrumented with endbranch. A CET enabled program enters legacy compatibility treatment when all of the below conditions are met.

1. Legacy compatibility configuration is enabled in this CPL class by setting the LEG_IW_EN bit in IA32_U_CET/IA32_S_CET.
2. Control transfer is performed using an indirect call/jmp without no-track prefix to a non-endbranch instruction.
3. The legacy code page bitmap is setup to indicate that the target of the control transfer is a legacy code page.

The legacy code page bitmap is a data structure in program memory that is used by the hardware to determine if the code page to which a legacy transfer is being performed is allowed.

When a matching endbranch instruction is not decoded at the target of an indirect call/jmp when required, the processor performs the below actions.

CET indirect branch tracking state machine violation event handler:

```
If LEG_IW_EN == 1
    LA = LIP;
    IF ENCLAVE_MODE == 1
        LA = LA – SECS.BASEADDR;
    ENDIF
    IF (EFER.LMA & CS.L) == 0
        BITMAP_BYTE = load.Asize_syslinaddr. Osize8(BITMAP_BASE + LA[31:15]]
    ELSE
        IF CR4.LA57 == 1
            BITMAP_BYTE = load.Asize_syslinaddr. Osize8(BITMAP_BASE + LA[56:15]]
        ELSE
            BITMAP_BYTE = load.Asize_syslinaddr. Osize8(BITMAP_BASE + LA[47:15]]
    FI;
    IF BITMAP_BYTE & (1 << LA[14:12]) == 0 then Deliver #CP(ENDBRANCH) fault
    IF CPL = 3
        IA32_U_CET.TRACKER = IDLE
        IA32_U_CET.SUPPRESS = IA32_U_CET.SUPPRESS_DIS == 0 ? 1 : 0
    ELSE
        IA32_S_CET.TRACKER = IDLE
        IA32_S_CET.SUPPRESS =    IA32_S_CET.SUPPRESS_DIS == 0 ? 1 : 0
    ENDIF
    Restart the instruction (handle all arch. consistency around MOV SS state machines, STI etc.)
    without opening up interrupt/trap window
ELSE
    Deliver #CP(ENDBRANCH) Fault
ENDIF
```

Faults/traps in pseudocode are delivered normally (e.g. #PF, EPT violation). On fault, active tracker holds last value (WAIT_FOR_ENDBRANCH) and address saved on stack is current IP (instruction that wasn't the ENDBRANCH).

The CET indirect branch tracking state machine is suppressed in legacy compatibility mode if the SUPPRESS_DIS control bit is 0.

Once the CET indirect branch tracking state machine has been suppressed, subsequent indirect call/jmp are not tracked for termination instruction.

Once CET indirect branch tracking has been suppressed, subsequent execution of endbranch instructions will do the following (see section 7 for details).

```
IF EndbranchEnabled(CPL) == 0
    NOP
ELSE
    SUPPRESS = 0
    TRACKER = IDLE
ENDIF
```

3.6.1 Legacy Code Page Bitmap Format

The legacy code page bitmap is a flat bitmap whose linear address is pointed to by the EB_LEG_BITMAP_BASE. Each bit in the bitmap represents a 4K page in linear memory. If the bit is 1 it indicates that the corresponding code page is a legacy code page; else it is a CET-enabled code page.
The processor uses the linear address of the instruction to which legacy transfer was attempted to lookup the bitmap. Bits of the linear address used as index in the bitmap are as follows.

- In legacy and compatibility mode – Bits 31:12
- In 64-bit mode (EFER.LMA=1 and CS.L=1)
  - If CR4.LA57 = 1, then Bits 56:12
  - If CR4.LA57 = 0, then Bits 47:12

3.7 Other Considerations

3.7.1 Intel® Transactional Synchronization Extensions (Intel® TSX) Interactions

The XBEGIN instruction encodes the relative offset to the abort handler and hence the fallback to the abort handler can be considered as a "direct" branch and the abort handler does not need to have an ENDBRANCH.

CET continues to enforce indirect call/jmp tracking within a transaction. Legacy compatibility treatment inside a transaction functions normally. If a transaction abort occurs then the processor sets the state of the indirect branch tracker to IDLE and not-suppressed.

3.7.2 #CP(ENDBRANCH) Priority w.r.t #NM and #UD

#NM, #UD and #CP(ENDBRANCH) are in the same priority class. Both #NM and #UD are opcode based faults. The #CP(endbranch) is prioritized higher than #NM and #UD as CET architecturally requires an ENDBRANCH at target of indirect call/jmp.

3.7.3 #CP(ENDBRANCH) Priority w.r.t #BP

Debug Exceptions priority is as follows.

- Traps delivered before any #CP(ENDBRANCH) fault: data breakpoint trap, IO breakpoint trap single step trap, task switch trap.
- Code Breakpoint fault detected before instruction decode and delivered before #CP(endbranch).
- GD condition fault – lower priority than #CP(endbranch).
- On IRET back from #DB/#BP the source indirect branch tracker becomes active if enabled and not suppressed.

INT3 does not cause #CP(endbranch) to support debugger usage of replacing bytes of ENDBRANCH with INT3 to set breakpoints. INT3 at target of a CALL-JMP(indirect) cause #BP(INT3) instead of #CP(endbranch), #CP(endbranch) fault is delayed. #BP caused by INT3 treated like other events that are higher priority than CET fault. On IRET back from #BP the source indirect tracker becomes active if enabled and not suppressed.

3.8 Constraining Speculation after Missing ENDBRANCH

When the CET tracker is in the WAIT_FOR_ENDBRANCH state, instruction execution will be limited or blocked, even speculatively, if the next instruction is not an ENDBRANCH.

This means that when indirect branch tracking is enabled and not suppressed, the instructions at the target of a near indirect JMP/CALL without the no-track prefix will only speculatively execute if there is an ENDBRANCH at the target. Early implementations of CET may limit the speculative execution to a small number of instructions (less than 8, with no more than 5 loads) past a missing ENDBRANCH, while later implementations will completely block the speculative execution of instructions after a missing ENDBRANCH.

This mechanism also limits or blocks speculation of the next sequential instructions after an indirect JMP or CALL, presuming the JMP/CALL puts the CET tracker into the WAIT_FOR_ENDBRANCH state and the next sequential instruction is not an ENDBRANCH.

4 Changes to Control Transfer Instructions Reference

When CET is enabled, the changes in operation of traditional control transfer instructions are described in this section.

4.1 CALL— Call Procedure

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
E8 cw	CALL rel16	M	N.S.	Valid	Call near, relative, displacement relative to next instruction.
E8 cd	CALL rel32	M	Valid	Valid	Call near, relative, displacement relative to next instruction. 32-bit displacement sign extended to 64-bits in 64-bit mode.
FF /2	CALL r/m16	M	N.E.	Valid	Call near, absolute indirect, address given in r/m16.
FF /2	CALL r/m32	M	N.E.	Valid	Call near, absolute indirect, address given in r/m32.
FF /2	CALL r/m64	M	Valid	N.E.	Call near, absolute indirect, address given in r/m64.
9A cd	CALL ptr16:16	D	Invalid	Valid	Call far, absolute, address given in operand.
9A cp	CALL ptr16:32	D	Invalid	Valid	Call far, absolute, address given in operand.
FF /3	CALL m16:16	M	Valid	Valid	Call far, absolute indirect address given in m16:16. In 32-bit mode: if selector points to a gate, then RIP = 32-bit zero extended displacement taken from gate; else RIP = zero extended 16-bit offset from far pointer referenced in the instruction.
FF /3	CALL m16:32	M	Valid	Valid	In 64-bit mode: If selector points to a gate, then RIP = 64-bit displacement taken from gate; else RIP = zero extended 32-bit offset from far pointer referenced in the instruction.
REX.W + FF /3	CALL m16:64	M	Valid	N.E.	In 64-bit mode: If selector points to a gate, then RIP = 64-bit displacement taken from gate; else RIP = 64-bit offset from far

		pointer referenced in the instruction.

Instruction Operand Encoding

Op /En	Operand 1	Operand 2	Operand 3	Operand 4
D	Offset	NA	NA	NA
M	ModRM:r/m (r)	NA	NA	NA

Description

Saves procedure linking information on the stack and branches to the called procedure specified using the target operand. The target operand specifies the address of the first instruction in the called procedure. The operand can be an immediate value, a general-purpose register, or a memory location.

This instruction can be used to execute four types of calls.

- **Near Call** — A call to a procedure in the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intra-segment call.
- **Far Call** — A call to a procedure located in a different segment than the current code segment, sometimes referred to as an inter-segment call.
- **Inter-privilege-level far call** — A far call to a procedure in a segment at a different privilege level than that of the currently executing program or procedure.
- **Task switch** — A call to a procedure located in a different task.

The latter two call types (inter-privilege-level call and task switch) can only be executed in protected mode. See "Calling Procedures Using Call and RET" in Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for additional -information on near, far, and inter-privilege-level calls. See Chapter 7, "Task Management," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for information on performing task switches with the CALL instruction.

Near Call. When executing a near call, the processor pushes the value of the EIP register (which contains the offset of the instruction following the CALL instruction) on the stack (for use later as a return-instruction pointer). The processor then branches to the address in the current code segment specified by the target operand. The target operand specifies either an absolute offset in the code segment (an offset from the base of the code segment) or a relative offset (a signed displacement relative to the current value of the instruction pointer in the EIP register; this value points to the instruction following the CALL instruction). The CS register is not changed on near calls.

For a near call absolute, an absolute offset is specified indirectly in a general-purpose register or a memory location (*r/m16*, *r/m32, or r/m64*). The operand-size attribute determines the size of the target operand (16, 32 or 64 bits). When in 64-bit mode, the operand size for near call (and all near branches) is forced to 64-bits. Absolute offsets are loaded directly into the EIP(RIP) register. If the operand size attribute is 16, the upper two bytes of the EIP register are cleared, resulting in a maximum instruction pointer size of 16 bits. When accessing an absolute offset indirectly using the stack pointer [ESP] as the base register, the base value used is the value of the ESP before the instruction executes.

A relative offset (*rel16* or *rel32*) is generally specified as a label in assembly code. But at the machine code level, it is encoded as a signed, 16- or 32-bit immediate value. This value is added to the value in the EIP(RIP) register. In 64-bit mode the relative offset is always a 32-bit immediate value which is sign extended to 64-bits before it is added to the value in the RIP register for the target calculation. As with absolute offsets, the operand-size attribute determines the size of the target operand (16, 32, or 64 bits). In 64-bit mode the target operand will always be 64-bits because the operand size is forced to 64-bits for near branches.

Far Calls in Real-Address or Virtual-8086 Mode. When executing a far call in real- address or virtual-8086 mode, the processor pushes the current value of both the CS and EIP registers on the stack for use as a return-instruction pointer. The processor then performs a “far branch” to the code segment and offset specified with the target operand for the called procedure. The target operand specifies an absolute far address either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). With the pointer method, the segment and offset of the called procedure is encoded in the instruction using a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address immediate. With the indirect method, the target operand specifies a memory location that contains a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address. The operand-size attribute determines the size of the offset (16 or 32 bits) in the far address. The far address is loaded directly into the CS and EIP registers. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared.

Far Calls in Protected Mode. When the processor is operating in protected mode, the CALL instruction can be used to perform the following types of far calls.

- Far call to the same privilege level.
- Far call to a different privilege level (inter-privilege level call).
- Task switch (far call to another task).

In protected mode, the processor always uses the segment selector part of the far address to access the corresponding descriptor in the GDT or LDT. The descriptor type (code segment, call gate, task gate, or TSS) and access rights determine the type of call operation to be performed.

If the selected descriptor is for a code segment, a far call to a code segment at the same privilege level is performed. (If the selected code segment is at a different privilege level and the code segment is non-conforming, a general-protection exception is generated.) A far call to the same privilege level in protected mode is very similar to one carried out in real-address or virtual-8086 mode. The target operand specifies an absolute far address either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). The operand- size attribute determines the size of the offset (16 or 32 bits) in the far address. The new code segment selector and its descriptor are loaded into CS register; the offset from the instruction is loaded into the EIP register.

A call gate (described in the next paragraph) can also be used to perform a far call to a code segment at the same privilege level. Using this mechanism provides an extra level of indirection and is the preferred method of making calls between 16-bit and 32-bit code segments.

When executing an inter-privilege-level far call, the code segment for the procedure being called must be accessed through a call gate. The segment selector specified by the target operand identifies the call gate. The target operand can specify the call gate segment selector either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). The processor obtains the segment selector for the new code segment and the new instruction pointer (offset) from the call gate descriptor. (The offset from the target operand is ignored when a call gate is used.)

On inter-privilege-level calls, the processor switches to the stack for the privilege level of the called procedure. The segment selector for the new stack segment is specified in the TSS for the currently running task. The branch to the new code segment occurs after the stack switch. (Note that when using a call gate to perform a far call to a segment at the same privilege level, no stack switch occurs.) On the new stack, the processor pushes the segment selector and stack pointer for the calling procedure’s stack, an optional set of parameters from the calling procedures stack, and the segment selector and instruction pointer for the calling procedure’s code segment. (A value in the call gate descriptor determines how many parameters to copy to the new stack.) Finally, the processor branches to the address of the procedure being called within the new code segment.

Executing a task switch with the CALL instruction is similar to executing a call through a call gate. The target operand specifies the segment selector of the task gate for the new task activated by the switch (the offset in the target operand is ignored). The task gate in turn points to the TSS for the new task, which contains the segment selectors for the task's code and stack segments. Note that the TSS also contains the EIP value for the next instruction that was to be executed before the calling task was suspended. This instruction pointer value is loaded into the EIP register to re-start the calling task.

The CALL instruction can also specify the segment selector of the TSS directly, which eliminates the indirection of the task gate. See Chapter 7, "Task Management," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for information on the mechanics of a task switch.

When you execute at task switch with a CALL instruction, the nested task flag (NT) is set in the EFLAGS register and the new TSS's previous task link field is loaded with the old task's TSS selector. Code is expected to suspend this nested task by executing an IRET instruction which, because the NT flag is set, automatically uses the previous task link to return to the calling task. (See "Task Linking" in Chapter 7 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for information on nested tasks.) Switching tasks with the CALL instruction differs in this regard from JMP instruction. JMP does not set the NT flag and therefore does not expect an IRET instruction to suspend the task.

Mixing 16-Bit and 32-Bit Calls. When making far calls between 16-bit and 32-bit code segments, use a call gate. If the far call is from a 32-bit code segment to a 16-bit code segment, the call should be made from the first 64 KBytes of the 32-bit code segment. This is because the operand-size attribute of the instruction is set to 16, so only a 16-bit return address offset can be saved. Also, the call should be made using a 16-bit call gate so that 16-bit values can be pushed on the stack. See Chapter 21, "Mixing 16-Bit and 32-Bit Code," in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3B*, for more information.

Far Calls in Compatibility Mode. When the processor is operating in compatibility mode, the CALL instruction can be used to perform the following types of far calls.

- Far call to the same privilege level, remaining in compatibility mode.
- Far call to the same privilege level, transitioning to 64-bit mode.
- Far call to a different privilege level (inter-privilege level call), transitioning to 64-bit mode.

Note that a CALL instruction cannot be used to cause a task switch in compatibility mode since task switches are not supported in IA-32e mode.

In compatibility mode, the processor always uses the segment selector part of the far address to access the corresponding descriptor in the GDT or LDT. The descriptor type (code segment, call gate) and access rights determine the type of call operation to be performed.

If the selected descriptor is for a code segment, a far call to a code segment at the same privilege level is performed. (If the selected code segment is at a different privilege level and the code segment is non-conforming, a general-protection exception is generated.) A far call to the same privilege level in compatibility mode is very similar to one carried out in protected mode. The target operand specifies an absolute far address either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). The operand-size attribute determines the size of the offset (16 or 32 bits) in the far address. The new code segment selector and its descriptor are loaded into CS register and the offset from the instruction is loaded into the EIP register. The difference is that 64-bit mode may be entered. This specified by the L bit in the new code segment descriptor.

Note that a 64-bit call gate (described in the next paragraph) can also be used to perform a far call to a code segment at the same privilege level. However, using this mechanism requires that the target code segment descriptor have the L bit set, causing an entry to 64-bit mode.

When executing an inter-privilege-level far call, the code segment for the procedure being called must be accessed through a 64-bit call gate. The segment selector specified by the target operand identifies the call gate. The target operand can specify the call gate segment selector either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). The processor obtains the segment selector for the new code segment and the new instruction pointer (offset) from the 16-byte call gate descriptor. (The offset from the target operand is ignored when a call gate is used.)

On inter-privilege-level calls, the processor switches to the stack for the privilege level of the called procedure. The segment selector for the new stack segment is set to NULL. The new stack pointer is specified in the TSS for the currently running task. The branch to the new code segment occurs after the stack switch. (Note that when using a call gate to perform a far call to a segment at the same privilege level, an implicit stack switch occurs as a result of entering 64-bit mode. The SS selector is unchanged, but stack segment accesses use a segment base of 0x0, the limit is ignored, and the default stack size is 64-bits. The full value of RSP is used for the offset, of which the upper 32-bits are undefined.) On the new stack, the processor pushes the segment selector and stack pointer for the calling procedure's stack and the segment selector and instruction pointer for the calling procedure's code segment. (Parameter copy is not supported in IA-32e mode.) Finally, the processor branches to the address of the procedure being called within the new code segment.

Near/(Far) Calls in 64-bit Mode. When the processor is operating in 64-bit mode, the CALL instruction can be used to perform the following types of far calls

- Far call to the same privilege level, transitioning to compatibility mode.
- Far call to the same privilege level, remaining in 64-bit mode.
- Far call to a different privilege level (inter-privilege level call), remaining in 64-bit mode.

Note that in this mode the CALL instruction cannot be used to cause a task switch in 64-bit mode since task switches are not supported in IA-32e mode.

In 64-bit mode, the processor always uses the segment selector part of the far address to access the corresponding descriptor in the GDT or LDT. The descriptor type (code segment, call gate) and access rights determine the type of call operation to be performed.

If the selected descriptor is for a code segment, a far call to a code segment at the same privilege level is performed. (If the selected code segment is at a different privilege level and the code segment is non-conforming, a general-protection exception is generated.) A far call to the same privilege level in 64-bit mode is very similar to one carried out in compatibility mode. The target operand specifies an absolute far address indirectly with a memory location (*m16:16, m16:32* or *m16:64*). The form of CALL with a direct specification of absolute far address is not defined in 64-bit mode. The operand-size attribute determines the size of the offset (16, 32, or 64 bits) in the far address. The new code segment selector and its descriptor are loaded into the CS register; the offset from the instruction is loaded into the EIP register. The new code segment may specify entry either into compatibility or 64-bit mode, based on the L bit value.

A 64-bit call gate (described in the next paragraph) can also be used to perform a far call to a code segment at the same privilege level. However, using this mechanism requires that the target code segment descriptor have the L bit set.

When executing an inter-privilege-level far call, the code segment for the procedure being called must be accessed through a 64-bit call gate. The segment selector specified by the target operand identifies the call gate. The target operand can only specify the call gate segment selector indirectly with a memory location (*m16:16, m16:32* or *m16:64*). The processor obtains the segment selector for the new code segment and the new instruction pointer (offset) from the 16-byte call gate descriptor. (The offset from the target operand is ignored when a call gate is used.)

On inter-privilege-level calls, the processor switches to the stack for the privilege level of the called procedure. The segment selector for the new stack segment is set to NULL. The new stack pointer is specified in the TSS for the currently running task. The branch to the new code segment occurs after the stack switch.

Note that when using a call gate to perform a far call to a segment at the same privilege level, an implicit stack switch occurs as a result of entering 64-bit mode. The SS selector is unchanged, but stack segment accesses use a segment base of 0x0, the limit is ignored, and the default stack size is 64-bits. (The full value of RSP is used for the offset.) On the new stack, the processor pushes the segment selector and stack pointer for the calling procedure's stack and the segment selector and instruction pointer for the calling procedure's code segment. (Parameter copy is not supported in IA-32e mode.) Finally, the processor branches to the address of the procedure being called within the new code segment.

Operation

```
IF near call
   THEN IF near relative call
       THEN
           IF OperandSize = 64
               THEN
                   tempDEST ← SignExtend(DEST); (* DEST is rel32 *)
                   tempRIP ← RIP    + tempDEST;
                   IF stack not large enough for a 8-byte return address
                       THEN #SS(0); FI;
                   Push(RIP);
                       IF ShadowStackEnabled(CPL) AND DEST != 0
                           ShadowStackPush8B(RIP);
                       FI;
                   RIP ← tempRIP;
           FI;
           IF OperandSize = 32
               THEN
                   tempEIP ← EIP    + DEST; (* DEST is rel32 *)
                   IF tempEIP is not within code segment limit THEN #GP(0); FI;
                   IF stack not large enough for a 4-byte return address
                       THEN #SS(0); FI;
                   Push(EIP);
                       IF ShadowStackEnabled(CPL) AND DEST != 0
                           ShadowStackPush4B(EIP);
                       FI;
                   EIP ← tempEIP;
           FI;
           IF OperandSize = 16
               THEN
                   tempEIP ← (EIP    + DEST) AND 0000FFFFH; (* DEST is rel16 *)
                   IF tempEIP is not within code segment limit THEN #GP(0); FI;
                   IF stack not large enough for a 2-byte return address
                       THEN #SS(0); FI;
                   Push(IP);
                       IF ShadowStackEnabled(CPL) AND DEST != 0
                           (* IP is zero extended and pushed as a 32 bit value on shadow stack *)
                           ShadowStackPush4B(IP);
                       FI;
                   EIP ← tempEIP;
           FI;
       ELSE (* Near absolute call *)
           IF OperandSize = 64
               THEN
                   tempRIP ← DEST; (* DEST is r/m64 *)
                   IF stack not large enough for a 8-byte return address
                       THEN #SS(0); FI;
                   Push(RIP);
                       IF ShadowStackEnabled(CPL)
                           ShadowStackPush8B(RIP);
```

```
                    FI;
                RIP ← tempRIP;
        FI;
        IF OperandSize = 32
            THEN
                tempEIP ← DEST; (* DEST is r/m32 *)
                IF tempEIP is not within code segment limit THEN #GP(0); FI;
                IF stack not large enough for a 4-byte return address
                    THEN #SS(0); FI;
                Push(EIP);
                    IF ShadowStackEnabled(CPL)
                        ShadowStackPush4B(EIP);
                    FI;
                EIP ← tempEIP;
        FI;
        IF OperandSize = 16
            THEN
                tempEIP ← DEST AND 0000FFFFH; (* DEST is r/m16 *)
                IF tempEIP is not within code segment limit THEN #GP(0); FI;
                IF stack not large enough for a 2-byte return address
                    THEN #SS(0); FI;
                Push(IP);
                    IF ShadowStackEnabled(CPL)
                        (* IP is zero extended and pushed as a 32 bit value on shadow stack *)
                        ShadowStackPush4B(IP);
                    FI;
                EIP ← tempEIP;
        FI;
    FI; rel/abs
    IF (Call near indirect, absolute indirect)
        IF EndbranchEnabledAndNotSuppressed(CPL)
            IF CPL = 3
                THEN
                    IF ( no 3EH prefix OR IA32_U_CET.NO_TRACK_EN == 0 )
                        THEN
                            IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH
                        FI
                ELSE
                    IF ( no 3EH prefix OR IA32_S_CET.NO_TRACK_EN == 0 )
                        THEN
                            IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
                        FI
                FI;
        FI;
    FI;
FI; near
```

```
IF far call and (PE = 0 or (PE = 1 and VM = 1)) (* Real-address or virtual-8086 mode *)
   THEN

       IF OperandSize = 32
           THEN
               IF stack not large enough for a 6-byte return address
                   THEN #SS(0); FI;
               IF DEST[31:16] is not zero                         THEN #GP(0); FI;
               Push(CS); (* Padded with 16 high-order bits *)
               Push(EIP);
               CS ← DEST[47:32]; (* DEST is ptr16:32 or [m16:32] *)
               EIP ← DEST[31:0]; (* DEST is ptr16:32 or [m16:32] *)
           ELSE (* OperandSize = 16 *)
               IF stack not large enough for a 4-byte return address
                   THEN #SS(0); FI;
               Push(CS);
               Push(IP);
               CS ← DEST[31:16]; (* DEST is ptr16:16 or [m16:16] *)
               EIP ← DEST[15:0]; (* DEST is ptr16:16 or [m16:16]; clear upper 16 bits *)
       FI;
FI;

IF far call and (PE = 1 and VM = 0) (* Protected mode or IA-32e Mode, not virtual-8086 mode*)
   THEN
       IF segment selector in target operand NULL
           THEN #GP(0); FI;
       IF segment selector index not within descriptor table limits
           THEN #GP(new code segment selector); FI;
       Read type and access rights of selected segment descriptor;
       IF IA32_EFER.LMA = 0
           THEN
               IF segment type is not a conforming or nonconforming code segment, call
               gate, task gate, or TSS
                   THEN #GP(segment selector); FI;
           ELSE
               IF segment type is not a conforming or nonconforming code segment or
               64-bit call gate,
                   THEN #GP(segment selector); FI;
       FI;
       Depending on type and access rights:
           GO TO CONFORMING-CODE-SEGMENT;
           GO TO NONCONFORMING-CODE-SEGMENT;
           GO TO CALL-GATE;
           GO TO TASK-GATE;
           GO TO TASK-STATE-SEGMENT;
FI;

CONFORMING-CODE-SEGMENT:
   IF L bit = 1 and D bit = 1 and IA32_EFER.LMA = 1
       THEN GP(new code segment selector); FI;
   IF DPL > CPL
```

```
        THEN #GP(new code segment selector); FI;
IF segment not present
        THEN #NP(new code segment selector); FI;
IF stack not large enough for return address
        THEN #SS(0); FI;
tempEIP ←DEST(Offset);
IF OperandSize = 16
        THEN
                tempEIP ← tempEIP AND 0000FFFFH; FI; (* Clear upper 16 bits *)
IF (EFER.LMA = 0 or target mode = Compatibility mode) and (tempEIP outside new code
segment limit)
        THEN #GP(0); FI;
IF tempEIP is non-canonical
        THEN #GP(0); FI;

 IF ShadowStackEnabled(CPL)
          IF OperandSize = 32
                    THEN
                              tempPushLIP = CSBASE + EIP;
                    ELSE
                              IF OperandSize = 16
                                        THEN
                                                  tempPushLIP = CSBASE + IPEIP;
                                        ELSE (* OperandSize = 64 *)
                                                  tempPushLIP = RIP;
                              FI;
          FI;
        tempPushCS = CS;
 FI;
IF OperandSize = 32
        THEN
                Push(CS); (* Padded with 16 high-order bits *)
                Push(EIP);
                CS ← DEST(CodeSegmentSelector);
                (* Segment descriptor information also loaded *)
                CS(RPL) ← CPL;
                EIP ← tempEIP;
        ELSE
                IF OperandSize = 16
                        THEN
                                Push(CS);
                                Push(IP);
                                CS ← DEST(CodeSegmentSelector);
                                (* Segment descriptor information also loaded *)
                                CS(RPL) ← CPL;
                                EIP ← tempEIP;
                        ELSE (* OperandSize = 64 *)
                    Push(CS); (* Padded with 48 high-order bits *)
                    Push(RIP);
                    CS ← DEST(CodeSegmentSelector);
                    (* Segment descriptor information also loaded *)
                    CS(RPL) ← CPL;
                    RIP ← tempEIP;
            FI;
    FI;
     IF ShadowStackEnabled(CPL)
          IF (EFER.LMA and DEST(CodeSegmentSelector).L) = 0
               (* If target is legacy or compatibility mode then the SSP must be in low 4G *)
            IF (SSP & 0xFFFFFFFF00000000 != 0)
                THEN #GP(0); FI;
        FI;
          (* align to 8 byte boundary if not already aligned *)
        tempSSP = SSP;
          Shadow_stack_store 4 bytes of 0 to (SSP – 4)
        SSP = SSP & 0xFFFFFFFFFFFFFFF8H
        ShadowStackPush8B(tempPushCS);              (* Padded with 48 high-order bits of 0 *)
        ShadowStackPush8B(tempPushLIP);             (* Padded with 32 high-order bits of 0 for 32 bit LIP*)
        ShadowStackPush8B(tempSSP);
     FI;
    IF EndbranchEnabled(CPL)
        IF CPL = 3
            THEN
                IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH
                IA32_U_CET.SUPPRESS = 0
            ELSE
                IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
                IA32_S_CET.SUPPRESS = 0
        FI;
     FI;
END;

NONCONFORMING-CODE-SEGMENT:
    IF L-Bit = 1 and D-BIT = 1 and IA32_EFER.LMA = 1
        THEN GP(new code segment selector); FI;
    IF (RPL > CPL) or (DPL != CPL)
        THEN #GP(new code segment selector); FI;
    IF segment not present
        THEN #NP(new code segment selector); FI;
    IF stack not large enough for return address
        THEN #SS(0); FI;
    tempEIP ← DEST(Offset);
    IF OperandSize = 16
        THEN tempEIP ← tempEIP AND 0000FFFFH; FI; (* Clear upper 16 bits *)
    IF (EFER.LMA = 0 or target mode = Compatibility mode) and (tempEIP outside new code
    segment limit)
        THEN #GP(0); FI;
    IF tempEIP is non-canonical
        THEN #GP(0); FI;
```

```
 IF ShadowStackEnabled(CPL)
     IF OperandSize = 32
         THEN
             tempPushLIP = CSBASE + EIP;
         ELSE
             IF OperandSize = 16
                 THEN
                     tempPushLIP = CSBASE + EIP;
                 ELSE (* OperandSize = 64 *)
                     tempPushLIP = RIP;
             FI;
     FI;
     tempPushCS = CS;
 FI;
IF OperandSize = 32
    THEN
        Push(CS); (* Padded with 16 high-order bits *)
        Push(EIP);

        CS ← DEST(CodeSegmentSelector);
        (* Segment descriptor information also loaded *)
        CS(RPL) ← CPL;
        EIP ← tempEIP;
    ELSE
        IF OperandSize = 16
            THEN
                Push(CS);
                Push(IP);
                CS ← DEST(CodeSegmentSelector);
                (* Segment descriptor information also loaded *)
                CS(RPL) ← CPL;
                EIP ← tempEIP;
            ELSE (* OperandSize = 64 *)
                Push(CS); (* Padded with 48 high-order bits *)
Push(RIP);
                CS ← DEST(CodeSegmentSelector);
                (* Segment descriptor information also loaded *)
                CS(RPL) ← CPL;
                RIP ← tempEIP;
        FI;
FI;
IF ShadowStackEnabled(CPL)
     IF (EFER.LMA and DEST(CodeSegmentSelector).L) = 0
         (* If target is legacy or compatibility mode then the SSP must be in low 4G *)
        IF (SSP & 0xFFFFFFFF00000000 != 0)
            THEN #GP(0); FI;
    FI;
```

```
            (* align to 8 byte boundary if not already aligned *)
          tempSSP = SSP;
            Shadow_stack_store 4 bytes of 0 to (SSP – 4)
          SSP = SSP & 0xFFFFFFFFFFFFFFF8H
          ShadowStackPush8B(tempPushCS); (* Padded with 48 high-order 0 bits *)
          ShadowStackPush8B(tempPushLIP); (* Padded 32 high-order bits of 0 for 32 bit LIP*)
          ShadowStackPush8B(tempSSP);
    FI;
    IF EndbranchEnabled(CPL)
          IF CPL = 3
                THEN
                      IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH
                      IA32_U_CET.SUPPRESS = 0
                ELSE
                      IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
                      IA32_S_CET.SUPPRESS = 0
          FI;
     FI;
END;

CALL-GATE:
    IF call gate (DPL < CPL) or (RPL > DPL)
          THEN #GP(call-gate selector); FI;
    IF call gate not present
          THEN #NP(call-gate selector); FI;
    IF call-gate code-segment selector is NULL
          THEN #GP(0); FI;
    IF call-gate code-segment selector index is outside descriptor table limits
          THEN #GP(call-gate code-segment selector); FI;
    Read call-gate code-segment descriptor;
    IF call-gate code-segment descriptor does not indicate a code segment
    or call-gate code-segment descriptor DPL > CPL
          THEN #GP(call-gate code-segment selector); FI;
    IF IA32_EFER.LMA = 1 AND (call-gate code-segment descriptor is
    not a 64-bit code segment       or call-gate code-segment descriptor has both L-bit and D-bit set)
          THEN #GP(call-gate code-segment selector); FI;
    IF call-gate code segment not present
          THEN #NP(call-gate code-segment selector); FI;
    IF call-gate code segment is non-conforming and DPL < CPL
          THEN go to MORE-PRIVILEGE;
          ELSE go to SAME-PRIVILEGE;
    FI;
END;

MORE-PRIVILEGE:
    IF current TSS is 32-bit
          THEN
                TSSstackAddress ← (new code-segment DPL * 8) + 4;
                IF (TSSstackAddress + 5) > current TSS limit
                      THEN #TS(current TSS selector); FI;
                NewSS ← 2 bytes loaded from (TSS base + TSSstackAddress + 4);
                NewESP ← 4 bytes loaded from (TSS base + TSSstackAddress);
```

```
        ELSE
            IF current TSS is 16-bit
                THEN
                    TSSstackAddress ← (new code-segment DPL * 4) + 2
                    IF (TSSstackAddress     + 3) > current TSS limit
                        THEN #TS(current TSS selector); FI;
                    NewSS ← 2 bytes loaded from (TSS base + TSSstackAddress + 2);
                    NewESP ← 2 bytes loaded from (TSS base + TSSstackAddress);
                ELSE (* current TSS is 64-bit *)
                    TSSstackAddress ← (new code-segment DPL * 8) + 4;
                    IF (TSSstackAddress     + 7) > current TSS limit
                        THEN #TS(current TSS selector); FI;
                    NewSS ← new code-segment DPL;                   (* NULL selector with RPL = new CPL *)
                    NewRSP ← 8 bytes loaded from (current TSS base + TSSstackAddress);
            FI;
    FI;
    IF IA32_EFER.LMA = 0 and NewSS is NULL
        THEN #TS(NewSS); FI;
    Read new code-segment descriptor and new stack-segment descriptor;
    IF IA32_EFER.LMA = 0 and (NewSS RPL != new code-segment DPL
    or new stack-segment DPL != new code-segment DPL or new stack segment is not a
    writable data segment)
        THEN #TS(NewSS); FI
    IF IA32_EFER.LMA = 0 and new stack segment not present
        THEN #SS(NewSS); FI;
    IF CallGateSize = 32
        THEN
            IF new stack does not have room for parameters plus 16 bytes
                THEN #SS(NewSS); FI;
            IF CallGate(InstructionPointer) not within new code-segment limit
                THEN #GP(0); FI;
            SS ← newSS;                 (* Segment descriptor information also loaded *)
            ESP ← newESP;
            CS:EIP ← CallGate(CS:InstructionPointer);
            (* Segment descriptor information also loaded *)
            Push(oldSS:oldESP); (* From calling procedure *)
            temp ← parameter count from call gate, masked to 5 bits;
            Push(parameters from calling procedure's stack, temp)
            Push(oldCS:oldEIP); (* Return address to calling procedure *)
        ELSE
            IF CallGateSize = 16
                THEN
                    IF new stack does not have room for parameters plus 8 bytes
                        THEN #SS(NewSS); FI;
                    IF (CallGate(InstructionPointer) AND FFFFH) not in new code-segment limit
                        THEN #GP(0); FI;
                    SS ← newSS;                 (* Segment descriptor information also loaded *)
                    ESP ← newESP;
                    CS:IP ← CallGate(CS:InstructionPointer);
                    (* Segment descriptor information also loaded *)
                    Push(oldSS:oldESP); (* From calling procedure *)
                    temp ← parameter count from call gate, masked to 5 bits;
                    Push(parameters from calling procedure's stack, temp)
                    Push(oldCS:oldEIP); (* Return address to calling procedure *)
                ELSE (* CallGateSize = 64 *)
                    IF pushing 32 bytes on the stack would use a non-canonical address
                        THEN #SS(NewSS); FI;
                    IF (CallGate(InstructionPointer) is non-canonical)
                        THEN #GP(0); FI;
                    SS ← NewSS; (* NewSS is NULL)
                    RSP ← NewESP;
                    CS:IP ← CallGate(CS:InstructionPointer);
                    (* Segment descriptor information also loaded *)
                    Push(oldSS:oldESP); (* From calling procedure *)
                    Push(oldCS:oldEIP); (* Return address to calling procedure *)
            FI;
    FI;
     IF ShadowStackEnabled(CPL)
        THEN
            IF CPL = 3
                THEN IA32_PL3_SSP ←SSP; FI;
    FI;
    CPL ← CodeSegment(DPL)
    CS(RPL) ← CPL
     IF ShadowStackEnabled(CPL)
         oldSSP ← SSP
         SSP ← IA32_PLi_SSP; (* where i is the CPL *)
         IF SSP & 0x07 != 0      (* if SSP not aligned to 8 bytes then #GP *)
            THEN #GP(0); FI;
        Fault = 0
         Atomic Start
             SSPToken = 8 bytes locked loaded with shadow stack semantics from SSP
            IF (SSPToken AND 0x01)
                    THEN fault ← 1; FI;
                IF ((EFER.LMA and CS.L) = 0 AND SSPToken[63:32] != 0)
                    THEN fault ← 1; FI;
              IF ((SSPToken AND 0xFFFFFFFFFFFFFFFE) != SSP)
                    THEN fault ← 1; FI;
            IF fault = 0
                    THEN SSPToken = SSPToken OR 0x01; FI;
              Store 8 bytes of SSPToken and unlock with shadow stack semantics to SSP;
          Atomic End
          If fault = 1
            THEN #GP(0); FI;
          IF oldSS.DPL != 3
              ShadowStackPush8B(oldCS);                        (* Padded with 48 high-order bits of 0 *)
                ShadowStackPush8B(oldCSBASE+oldRIP); (* Padded with 32 high-order bits of 0 for 32 bit LIP*)
                ShadowStackPush8B(oldSSP);
            FI;
      FI
    IF EndbranchEnabled (CPL)
        IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
        IA32_S_CET.SUPPRESS = 0
      FI;
END;

SAME-PRIVILEGE:
    IF CallGateSize = 32
        THEN
            IF stack does not have room for 8 bytes
                THEN #SS(0); FI;
            IF CallGate(InstructionPointer) not within code segment limit
                THEN #GP(0); FI;
            CS:EIP ← CallGate(CS:EIP) (* Segment descriptor information also loaded *)
            Push(oldCS:oldEIP); (* Return address to calling procedure *)

        ELSE
            If CallGateSize = 16
                THEN
                    IF stack does not have room for 4 bytes
                        THEN #SS(0); FI;
                    IF CallGate(InstructionPointer) not within code segment limit
                        THEN #GP(0); FI;
                    CS:IP ← CallGate(CS:instruction pointer);
                    (* Segment descriptor information also loaded *)
                    Push(oldCS:oldIP); (* Return address to calling procedure *)

                ELSE (* CallGateSize = 64)
                    IF pushing 16 bytes on the stack touches non-canonical addresses
                        THEN #SS(0); FI;
                    IF RIP non-canonical
                        THEN #GP(0); FI;
                    CS:RIP ← CallGate(CS:instruction pointer);
                    (* Segment descriptor information also loaded *)
                    Push(oldCS:oldRIP); (* Return address to calling procedure *)                FI;
    FI;

    CS(RPL) ← CPL
    IF ShadowStackEnabled(CPL)
         (* Align to next 8 byte boundary *)
         tempSSP = SSP;
         Shadow_stack_store 4 bytes of 0 to (SSP – 4)
         SSP = SSP & 0xFFFFFFFFFFFFFFF8H;
         (* push cs:lip:ssp on shadow stack *)
         ShadowStackPush8B(oldCS); (* Padded with 48 high-order bits of 0 *)
         ShadowStackPush8B(oldCSBASE + oldRIP); (* Padded with 32 high-order bits of 0 for 32 bit LIP*)
```

```
            ShadowStackPush8B(tempSSP);
    FI;
    IF EndbranchEnabled (CPL)
        IF CPL = 3
            THEN
                IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH;
                IA32_U_CET.SUPPRESS = 0
            ELSE
                IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH;
                IA32_S_CET.SUPPRESS = 0
        FI;
    FI;
END;

TASK-GATE:
    IF task gate DPL < CPL or RPL
        THEN #GP(task gate selector); FI;
    IF task gate not present
        THEN #NP(task gate selector); FI;
    Read the TSS segment selector in the task-gate descriptor;
    IF TSS segment selector local/global bit is set to local
    or index not within GDT limits
        THEN #GP(TSS selector); FI;
    Access TSS descriptor in GDT;
    IF TSS descriptor specifies that the TSS is busy (low-order 5 bits set to 00001)
        THEN #GP(TSS selector); FI;
    IF TSS not present
        THEN #NP(TSS selector); FI;
    SWITCH-TASKS (with nesting) to TSS;
    IF EIP not within code segment limit
        THEN #GP(0); FI;
END;

TASK-STATE-SEGMENT:
    IF TSS DPL < CPL or RPL
    or TSS descriptor indicates TSS not available
        THEN #GP(TSS selector); FI;
    IF TSS is not present
        THEN #NP(TSS selector); FI;
    SWITCH-TASKS (with nesting) to TSS;
    IF EIP not within code segment limit
        THEN #GP(0); FI;
END;
```

Flags Affected

All flags are affected if a task switch occurs; no flags are affected if a task switch does not occur.

Protected Mode Exceptions

#GP(0)	If the target offset in destination operand is beyond the new code segment limit.
	If the segment selector in the destination operand is NULL.

If the code segment selector in the gate is NULL.

If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.

If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.

If target mode is compatibility mode and SSP is not in low 4G.

If SSP in IA32_PLi_SSP (where i is the new CPL) is not 8 byte aligned.

If "supervisor Shadow Stack" token on new shadow stack is marked busy.

If destination mode is 32-bit or compatibility mode, but SSP address in "supervisor shadow stack" token is beyond 4G.

If SSP address in "supervisor shadow stack" token does not match SSP address in IA32_PLi_SSP (where i is the new CPL).

#GP(selector) If a code segment or gate or TSS selector index is outside descriptor table limits.

If the segment descriptor pointed to by the segment selector in the destination operand is not for a conforming-code segment, nonconforming-code segment, call gate, task gate, or task state segment.

If the DPL for a nonconforming-code segment is not equal to the CPL or the RPL for the segment's segment selector is greater than the CPL.

If the DPL for a conforming-code segment is greater than the CPL.

If the DPL from a call-gate, task-gate, or TSS segment descriptor is less than the CPL or than the RPL of the call-gate, task-gate, or TSS's segment selector.

If the segment descriptor for a segment selector from a call gate does not indicate it is a code segment.

If the segment selector from a call gate is beyond the descriptor table limits.

If the DPL for a code-segment obtained from a call gate is greater than the CPL.

If the segment selector for a TSS has its local/global bit set for local.

If a TSS segment descriptor specifies that the TSS is busy or not available.

#SS(0) If pushing the return address, parameters, or stack segment pointer onto the stack exceeds the bounds of the stack segment, when no stack switch occurs.

If a memory operand effective address is outside the SS segment limit.

#SS(selector) If pushing the return address, parameters, or stack segment pointer onto the stack exceeds the bounds of the stack segment, when a stack switch occurs.

If the SS register is being loaded as part of a stack switch and the segment pointed to is marked not present.

If stack segment does not have room for the return address, parameters, or stack segment pointer, when stack switch occurs.

#NP(selector) If a code segment, data segment, stack segment, call gate, task gate, or TSS is not present.

#TS(selector) If the new stack segment selector and ESP are beyond the end of the TSS.

If the new stack segment selector is NULL.

If the RPL of the new stack segment selector in the TSS is not equal to the DPL of the code segment being accessed.

If DPL of the stack segment descriptor for the new stack segment is not equal to the DPL of the code segment descriptor.

If the new stack segment is not a writable data segment.

If segment-selector index for stack segment is outside descriptor table limits.

#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the target offset is beyond the code segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the target offset is beyond the code segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

#GP(selector)	If a memory address accessed by the selector is in non-canonical space.
#GP(0)	If the target offset in the destination operand is non-canonical.

64-Bit Mode Exceptions

#GP(0)	If a memory address is non-canonical.
	If target offset in destination operand is non-canonical.
	If the segment selector in the destination operand is NULL.
	If the code segment selector in the 64-bit gate is NULL.
	If target mode is compatibility mode and SSP is not in low 4G.
	If SSP in IA32_PLi_SSP (where i is the new CPL) is not 8 byte aligned.
	If "supervisor Shadow Stack" token on new shadow stack is marked busy.
	If destination mode is 32-bit mode or compatibility mode, but SSP address in "supervisor shadow" stack token is beyond 4G.
	If SSP address in "supervisor shadow stack" token does not match SSP address in IA32_PLi_SSP (where i is the new CPL).
#GP(selector)	If code segment or 64-bit call gate is outside descriptor table limits.
	If code segment or 64-bit call gate overlaps non-canonical space.
	If the segment descriptor pointed to by the segment selector in the destination operand is not for a conforming-code segment, nonconforming-code segment, or 64-bit call gate.
	If the segment descriptor pointed to by the segment selector in the destination operand is a code segment and has both the D-bit and the L- bit set.
	If the DPL for a nonconforming-code segment is not equal to the CPL, or the RPL for the segment's segment selector is greater than the CPL.
	If the DPL for a conforming-code segment is greater than the CPL.
	If the DPL from a 64-bit call-gate is less than the CPL or than the RPL of the 64-bit call-gate.
	If the upper type field of a 64-bit call gate is not 0x0.

	If the segment selector from a 64-bit call gate is beyond the descriptor table limits.
	If the DPL for a code-segment obtained from a 64-bit call gate is greater than the CPL.
	If the code segment descriptor pointed to by the selector in the 64-bit gate doesn't have the L-bit set and the D-bit clear.
	If the segment descriptor for a segment selector from the 64-bit call gate does not indicate it is a code segment.
#SS(0)	If pushing the return offset or CS selector onto the stack exceeds the bounds of the stack segment when no stack switch occurs.
	If a memory operand effective address is outside the SS segment limit.
	If the stack address is in a non-canonical form.
#SS(selector)	If pushing the old values of SS selector, stack pointer, EFLAGS, CS selector, offset, or error code onto the stack violates the canonical boundary when a stack switch occurs.
#NP(selector)	If a code segment or 64-bit call gate is not present.
#TS(selector)	If the load of the new RSP exceeds the limit of the TSS.
#UD	(64-bit mode only) If a far call is direct to an absolute address in memory.
	If the LOCK prefix is used.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.

4.2 INT n/INTO/INT3 – Call to Interrupt Procedure

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
CC	INT3	NP	Valid	Valid	Interrupt 3 – trap to debugger.
CD *ib*	INT *imm8*	I	Valid	Valid	Interrupt vector specified by immediate byte.
CE	INTO	NP	Invalid	Valid	Interrupt 4 – if overflow flag is 1.

Instruction Operand Encoding

Op/ En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA
I	Imm8	NA	NA	NA

Description

The INT n instruction generates a call to the interrupt or exception handler specified with the destination operand (see the section titled “Interrupts and Exceptions” in Chapter 6 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 1). The destination operand specifies a vector from 0 to 255, encoded as an 8-bit unsigned intermediate value. Each vector provides an index to a gate descriptor in the IDT. The first 32 vectors are reserved by Intel for system use. Some of these vectors are used for internally generated exceptions.

The INT n instruction is the general mnemonic for executing a software-generated call to an interrupt handler. The INTO instruction is a special mnemonic for calling overflow exception (#OF), exception 4. The overflow interrupt checks the OF flag in the EFLAGS register and calls the overflow interrupt handler if the OF flag is set to 1. (The INTO instruction cannot be used in 64-bit mode.)

The INT 3 instruction generates a special one byte opcode (CC) that is intended for calling the debug exception handler. (This one byte form is valuable because it can be used to replace the first byte of any instruction with a breakpoint, including other one byte instructions, without over-writing other code). To further support its function as a debug breakpoint, the interrupt generated with the CC opcode also differs from the regular software interrupts as follows.

- Interrupt redirection does not happen when in VME mode; the interrupt is handled by a protected-mode handler.
- The virtual-8086 mode IOPL checks do not occur. The interrupt is taken without faulting at any IOPL level.

Note that the “normal” 2-byte opcode for INT 3 (CD03) does not have these special features. Intel and Microsoft assemblers will not generate the CD03 opcode from any mnemonic, but this opcode can be created by direct numeric code definition or by self-modifying code.

The action of the INT n instruction (including the INTO and INT 3 instructions) is similar to that of a far call made with the CALL instruction. The primary difference is that with the INT n instruction, the EFLAGS register is pushed onto the stack before the return address. (The return address is a far address consisting of the current values of the CS and EIP registers.) Returns from interrupt procedures are handled with the IRET instruction, which pops the EFLAGS information and return address from the stack.

The vector specifies an interrupt descriptor in the interrupt descriptor table (IDT); that is, it provides index into the IDT. The selected interrupt descriptor in turn contains a pointer to an interrupt or exception handler procedure. In protected mode, the IDT contains an array of 8-byte descriptors, each of which is an interrupt gate, trap gate, or task gate. In real-address mode, the IDT is an array of 4-byte far pointers (2-byte code segment selector and a 2-byte instruction pointer), each of which point directly to a procedure in the selected segment. (Note that in real-address mode, the IDT is called the interrupt vector table, and its pointers are called interrupt vectors.)

The following decision table indicates which action in the lower portion of the table is taken given the conditions in the upper portion of the table. Each Y in the lower section of the decision table represents a procedure defined in the "Operation" section for this instruction (except #GP).

Decision Table

PE	0	1	1	1	1	1	1	1
VM	-	-	-	-	-	0	1	1
IOPL	-	-	-	-	-	-	<3	<3
DPL/CPL RELATIONSHIP	-	DPL< CPL	-	DPL> CPL	DPL= CPL or C	DPL < CPL & NC	-	-
INTERRUPT TYPE	-	S/W	-	-	-	-	-	-
GATE TYPE	-	-	Task	Trap or Interrupt	Trap or Interrupt	Trap or Interrupt	Trap or Interrupt	Trap or Interrupt
REAL-ADDRESS-MODE	Y							
PROTECTED-MODE		Y	Y	Y	Y	Y	Y	Y
INTER-PRIVILEGE-LEVEL-INTERRUPT						Y		
INTRA-PRIVILEGE-LEVEL-INTERRUPT					Y			
INTERRUPT-FROM-VIRTUAL-8086-MODE								Y
TASK-GATE			Y					
#GP		Y		Y			Y	

NOTES:

- - Don't Care
- Y Yes, Action taken

Blank Action not taken

When the processor is executing in virtual-8086 mode, the IOPL determines the action of the INT n instruction. If the IOPL is less than 3, the processor generates a #GP(selector) exception; if the IOPL is 3, the processor executes a protected mode interrupt to privilege level 0. The interrupt gate's DPL must be set to 3 and the target CPL of the interrupt handler procedure must be 0 to execute the protected mode interrupt to privilege level 0.

The interrupt descriptor table register (IDTR) specifies the base linear address and limit of the IDT. The initial base address value of the IDTR after the processor is powered up or reset is 0.

Operation

The following operational description applies not only to the INT n and INTO instructions, but also to external interrupts, nonmaskable interrupts (NMIs), and exceptions. Some of these events push onto the stack an error code.

The operational description specifies numerous checks whose failure may result in delivery of a nested exception. In these cases, the original event is not delivered.

The operational description specifies the error code delivered by any nested exception. In some cases, the error code is specified with a pseudofunction error_code(num,idt,ext), where idt and ext are bit values. The pseudofunction produces an error code as follows: (1) if idt is 0, the error code is (num & FCH) | ext; (2) if idt is 1, the error code is (num « 3) | 2 | ext.

In many cases, the pseudofunction error_code is invoked with a pseudovariable EXT. The value of EXT depends on the nature of the event whose delivery encountered a nested exception: if that event is a software interrupt, EXT is 0; otherwise, EXT is 1.

```
IF PE = 0
    THEN
        GOTO REAL-ADDRESS-MODE;
    ELSE (* PE = 1 *)
        IF (VM = 1 and IOPL < 3 AND INT n)
            THEN
                #GP(0); (* Bit 0 of error code is 0 because INT n *)
            ELSE (* Protected mode, IA-32e mode, or virtual-8086 mode interrupt *)
                IF (IA32_EFER.LMA = 0)
                    THEN (* Protected mode, or virtual-8086 mode interrupt *)
                        GOTO PROTECTED-MODE;
                ELSE (* IA-32e mode interrupt *)
                    GOTO IA-32e-MODE;
                FI;
        FI;
FI;
REAL-ADDRESS-MODE:
    IF ((vector_number « 2) + 3) is not within IDT limit
        THEN #GP; FI;
    IF stack not large enough for a 6-byte return information
        THEN #SS; FI;
    Push (EFLAGS[15:0]);
    IF ← 0; (* Clear interrupt flag *)
    TF ← 0; (* Clear trap flag *)
    AC ← 0; (* Clear AC flag *)
    Push(CS);
    Push(IP);
    (* No error codes are pushed in real-address mode*)
    CS ← IDT(Descriptor (vector_number « 2), selector));
    EIP ← IDT(Descriptor (vector_number « 2), offset)); (* 16 bit offset AND 0000FFFFH *)
END;
PROTECTED-MODE:
    IF ((vector_number « 3) + 7) is not within IDT limits
    or selected IDT descriptor is not an interrupt-, trap-, or task-gate type
        THEN #GP(error_code(vector_number,1,EXT)); FI;
        (* idt operand to error_code set because vector is used *)
    IF software interrupt (* Generated by INT n, INT3, or INTO *)
```

```
        THEN
            IF gate DPL < CPL (* PE = 1, DPL < CPL, software interrupt *)
                THEN #GP(error_code(vector_number,1,0)); FI;
                (* idt operand to error_code set because vector is used *)
                (* ext operand to error_code is 0 because INT n, INT3, or INTO*)
FI;
IF gate not present
    THEN #NP(error_code(vector_number,1,EXT)); FI;
    (* idt operand to error_code set because vector is used *)
IF task gate (* Specified in the selected interrupt table descriptor *)
    THEN GOTO TASK-GATE;
    ELSE GOTO TRAP-OR-INTERRUPT-GATE; (* PE = 1, trap/interrupt gate *)
FI;
END;
IA-32e-MODE:
  IF INTO and CS.L = 1 (64-bit mode)
      THEN #UD;
   FI;
  IF ((vector_number « 4) + 15) is not in IDT limits
  or selected IDT descriptor is not an interrupt-, or trap-gate type
      THEN #GP(error_code(vector_number,1,EXT));
       (* idt operand to error_code set because vector is used *)
  FI;
  IF software interrupt (* Generated by INT n, INT 3, or INTO *)
      THEN
          IF gate DPL < CPL (* PE = 1, DPL < CPL, software interrupt *)
                  THEN #GP(error_code(vector_number,1,0));
                (* idt operand to error_code set because vector is used *)
              (* ext operand to error_code is 0 because INT n, INT3, or INTO*)
          FI;
  FI;
  IF gate not present
      THEN #NP(error_code(vector_number,1,EXT));
      (* idt operand to error_code set because vector is used *)
  FI;
  GOTO TRAP-OR-INTERRUPT-GATE; (* Trap/interrupt gate *)
END;
TASK-GATE: (* PE = 1, task gate *)
  Read TSS selector in task gate (IDT descriptor);
  IF local/global bit is set to local or index not within GDT limits
      THEN #GP(error_code(TSS selector,0,EXT)); FI;
      (* idt operand to error_code is 0 because selector is used *)
      Access TSS descriptor in GDT;
      IF TSS descriptor specifies that the TSS is busy (low-order 5 bits set to 00001)
          THEN #GP(TSS selector,0,EXT)); FI;
          (* idt operand to error_code is 0 because selector is used *)
      IF TSS not present
          THEN #NP(TSS selector,0,EXT)); FI;
```

```
            (* idt operand to error_code is 0 because selector is used *)
        SWITCH-TASKS (with nesting) to TSS;
        IF interrupt caused by fault with error code
            THEN
                IF stack limit does not allow push of error code
                    THEN #SS(EXT); FI;
                Push(error code);
        FI;
        IF EIP not within code segment limit
            THEN #GP(EXT); FI;
END;
TRAP-OR-INTERRUPT-GATE:
   Read new code-segment selector for trap or interrupt gate (IDT descriptor);
   IF new code-segment selector is NULL
        THEN #GP(EXT); FI; (* Error code contains NULL selector *)
   IF new code-segment selector is not within its descriptor table limits
        THEN #GP(error_code(new code-segment selector,0,EXT)); FI;
        (* idt operand to error_code is 0 because selector is used *)
   Read descriptor referenced by new code-segment selector;
   IF descriptor does not indicate a code segment or new code-segment DPL > CPL
        THEN #GP(error_code(new code-segment selector,0,EXT)); FI;
        (* idt operand to error_code is 0 because selector is used *)
   IF new code-segment descriptor is not present,
        THEN #NP(error_code(new code-segment selector,0,EXT)); FI;
        (* idt operand to error_code is 0 because selector is used *)
   IF new code segment is non-conforming with DPL < CPL
        THEN
            IF VM = 0
                THEN
                    GOTO INTER-PRIVILEGE-LEVEL-INTERRUPT;
                    (* PE = 1, VM = 0, interrupt or trap gate, nonconforming code segment,
                    DPL < CPL *)
                ELSE (* VM = 1 *)
                    IF new code-segment DPL != 0
                        THEN #GP(error_code(new code-segment selector,0,EXT));
                        (* idt operand to error_code is 0 because selector is used *)
                    GOTO INTERRUPT-FROM-VIRTUAL-8086-MODE; FI;
                    (* PE = 1, interrupt or trap gate, DPL < CPL, VM = 1 *)
            FI;
        ELSE (* PE = 1, interrupt or trap gate, DPL ≥ CPL *)
            IF VM = 1
                THEN #GP(error_code(new code-segment selector,0,EXT));
                (* idt operand to error_code is 0 because selector is used *)
            IF new code segment is conforming or new code-segment DPL = CPL
            THEN
                GOTO INTRA-PRIVILEGE-LEVEL-INTERRUPT;
            ELSE (* PE = 1, interrupt or trap gate, nonconforming code segment, DPL > CPL *)
                #GP(error_code(new code-segment selector,0,EXT));
                (* idt operand to error_code is 0 because selector is used *)
            FI;
   FI;
```

```
END;
INTER-PRIVILEGE-LEVEL-INTERRUPT:
   (* PE = 1, interrupt or trap gate, non-conforming code segment, DPL < CPL *)
   IF (IA32_EFER.LMA = 0) (* Not IA-32e mode *)
       THEN
       (* Identify stack-segment selector for new privilege level in current TSS *)
           IF current TSS is 32-bit
               THEN
                   TSSstackAddress ← (new code-segment DPL « 3) + 4;
                   IF (TSSstackAddress + 5) > current TSS limit
                       THEN #TS(error_code(current TSS selector,0,EXT)); FI;
                       (* idt operand to error_code is 0 because selector is used *)
                   NewSS ← 2 bytes loaded from (TSS base + TSSstackAddress + 4);
                   NewESP ← 4 bytes loaded from (TSS base + TSSstackAddress);
               ELSE (* current TSS is 16-bit *)
                   TSSstackAddress ← (new code-segment DPL « 2) + 2
                   IF (TSSstackAddress + 3) > current TSS limit
                       THEN #TS(error_code(current TSS selector,0,EXT)); FI;
                       (* idt operand to error_code is 0 because selector is used *)
                   NewSS ← 2 bytes loaded from (TSS base + TSSstackAddress + 2);
                   NewESP ← 2 bytes loaded from (TSS base + TSSstackAddress);
           FI;
           IF NewSS is NULL
               THEN #TS(EXT); FI;
           IF NewSS index is not within its descriptor-table limits
           or NewSS RPL != new code-segment DPL
               THEN #TS(error_code(NewSS,0,EXT)); FI;
               (* idt operand to error_code is 0 because selector is used *)
           Read new stack-segment descriptor for NewSS in GDT or LDT;
           IF new stack-segment DPL != new code-segment DPL
           or new stack-segment Type does not indicate writable data segment
               THEN #TS(error_code(NewSS,0,EXT)); FI;
               (* idt operand to error_code is 0 because selector is used *)
           IF NewSS is not present
               THEN #SS(error_code(NewSS,0,EXT)); FI;
               (* idt operand to error_code is 0 because selector is used *)
             NewSSP ← IA32_PLi_SSP (* where i = new code-segment DPL *)
       ELSE (* IA-32e mode *)
           IF IDT-gate IST = 0
               THEN TSSstackAddress ← (new code-segment DPL « 3) + 4;
               ELSE TSSstackAddress ← (IDT gate IST « 3) + 28;
           FI;
           IF (TSSstackAddress + 7) > current TSS limit
               THEN #TS(error_code(current TSS selector,0,EXT); FI;
               (* idt operand to error_code is 0 because selector is used *)
           NewRSP ← 8 bytes loaded from (current TSS base + TSSstackAddress);
           NewSS ← new code-segment DPL; (* NULL selector with RPL = new CPL *)
             IF IDT-gate IST = 0
```

```
                THEN
                    NewSSP ← IA32_PLi_SSP (* where i = new code-segment DPL *)
                ELSE
                    NewSSPAddress = IA32_INTERRUPT_SSP_TABLE_ADDR + (IDT-gate IST « 3)
                    (* Check if shadow stacks are enabled at CPL 0 *)
                    IF ShadowStackEnabled(CPL 0)
                        THEN NewSSP ← 8 bytes loaded from NewSSPAddress; FI;
            FI;
FI;
IF IDT gate is 32-bit
    THEN
        IF new stack does not have room for 24 bytes (error code pushed)
        or 20 bytes (no error code pushed)
            THEN #SS(error_code(NewSS,0,EXT)); FI;
            (* idt operand to error_code is 0 because selector is used *)
        FI
    ELSE
        IF IDT gate is 16-bit
            THEN
                IF new stack does not have room for 12 bytes (error code pushed)
                or 10 bytes (no error code pushed);
                    THEN #SS(error_code(NewSS,0,EXT)); FI;
                    (* idt operand to error_code is 0 because selector is used *)
            ELSE (* 64-bit IDT gate*)
                IF StackAddress is non-canonical
                    THEN #SS(EXT); FI; (* Error code contains NULL selector *)
    FI;
FI;
IF (IA32_EFER.LMA = 0) (* Not IA-32e mode *)
    THEN
        IF instruction pointer from IDT gate is not within new code-segment limits
            THEN #GP(EXT); FI; (* Error code contains NULL selector *)
        ESP ← NewESP;
        SS ← NewSS; (* Segment descriptor information also loaded *)
    ELSE (* IA-32e mode *)
        IF instruction pointer from IDT gate contains a non-canonical address
            THEN #GP(EXT); FI; (* Error code contains NULL selector *)
        RSP ← NewRSP & FFFFFFFFFFFFFFF0H;
        SS ← NewSS;
    FI;
IF IDT gate is 32-bit
    THEN
        CS:EIP ← Gate(CS:EIP); (* Segment descriptor information also loaded *)
    ELSE
        IF IDT gate 16-bit
            THEN
                CS:IP ← Gate(CS:IP);
                (* Segment descriptor information also loaded *)
            ELSE (* 64-bit IDT gate *)
                CS:RIP ← Gate(CS:RIP);
                (* Segment descriptor information also loaded *)
        FI;
```

```
FI;
IF IDT gate is 32-bit
    THEN
        Push(far pointer to old stack);
        (* Old SS and ESP, 3 words padded to 4 *)
        Push(EFLAGS);
        Push(far pointer to return instruction);
        (* Old CS and EIP, 3 words padded to 4 *)
        Push(ErrorCode); (* If needed, 4 bytes *)
    ELSE
        IF IDT gate 16-bit
            THEN
                Push(far pointer to old stack);
                (* Old SS and SP, 2 words *)
                Push(EFLAGS(15-0]);
                Push(far pointer to return instruction);
                (* Old CS and IP, 2 words *)
                Push(ErrorCode); (* If needed, 2 bytes *)
            ELSE (* 64-bit IDT gate *)
                Push(far pointer to old stack);
                (* Old SS and SP, each an 8-byte push *)
                Push(RFLAGS); (* 8-byte push *)
                Push(far pointer to return instruction);
                (* Old CS and RIP, each an 8-byte push *)
                Push(ErrorCode); (* If needed, 8-bytes *)
        FI;
FI;
 IF ShadowStackEnabled(CPL)
    THEN
        IF CPL = 3
            THEN IA32_PL3_SSP ←SSP; FI;
FI;
CPL ← new code-segment DPL;
CS(RPL) ← CPL;
 IF ShadowStackEnabled(CPL)
      oldSSP ← SSP
      SSP ← NewSSP
      IF SSP & 0x07 != 0
        THEN #GP(0); FI;
    Fault = 0
      Atomic Start
          SSPToken = 8 bytes locked loaded with shadow stack semantics from SSP
         IF (SSPToken AND 0x01)
                THEN fault ← 1; FI;
        IF ((EFER.LMA and CS.L) = 0 AND SSPToken[63:32] != 0)
                THEN fault ← 1; FI;
        IF ((SSPToken AND 0xFFFFFFFFFFFFFFFE) != SSP)
```

```
                    THEN fault ← 1; FI;
            IF fault = 0
                    THEN SSPToken = SSPToken OR 0x01; FI;
                Store 8 bytes of SSPToken and unlock with shadow stack semantics to SSP;
            Atomic End
            If fault = 1
              THEN #GP(0); FI;
            IF oldSS.DPL != 3
                ShadowStackPush8B(oldCS); (* Padded with 48 high-order bits of 0 *)
                ShadowStackPush8B(oldCSBASE + oldRIP); (* Padded with 32 high-order bits of 0 for 32 bit LIP*)
                ShadowStackPush8B(oldSSP);
            FI;
     FI
    IF EndbranchEnabled (CPL)
        IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH;
        IA32_S_CET.SUPPRESS = 0
     FI;

    IF IDT gate is interrupt gate
        THEN IF ← 0 (* Interrupt flag set to 0, interrupts disabled *); FI;
    TF ← 0;
    VM ← 0;
    RF ← 0;
    NT ← 0;
END;
INTERRUPT-FROM-VIRTUAL-8086-MODE:
    (* Identify stack-segment selector for privilege level 0 in current TSS *)
    IF current TSS is 32-bit
        THEN
            IF TSS limit < 9
                THEN #TS(error_code(current TSS selector,0,EXT)); FI;
                (* idt operand to error_code is 0 because selector is used *)
            NewSS ← 2 bytes loaded from (current TSS base + 8);
            NewESP ← 4 bytes loaded from (current TSS base + 4);
        ELSE (* current TSS is 16-bit *)
            IF TSS limit < 5
                THEN #TS(error_code(current TSS selector,0,EXT)); FI;
                (* idt operand to error_code is 0 because selector is used *)
            NewSS ← 2 bytes loaded from (current TSS base + 4);
            NewESP ← 2 bytes loaded from (current TSS base + 2);
    FI;
    IF NewSS is NULL
        THEN #TS(EXT); FI; (* Error code contains NULL selector *)
    IF NewSS index is not within its descriptor table limits
    or NewSS RPL != 0
        THEN #TS(error_code(NewSS,0,EXT)); FI;
        (* idt operand to error_code is 0 because selector is used *)
    Read new stack-segment descriptor for NewSS in GDT or LDT;
    IF new stack-segment DPL != 0 or stack segment does not indicate writable data segment
        THEN #TS(error_code(NewSS,0,EXT)); FI;
        (* idt operand to error_code is 0 because selector is used *)
```

```
   IF new stack segment not present
       THEN #SS(error_code(NewSS,0,EXT)); FI;
       (* idt operand to error_code is 0 because selector is used *)
    NewSSP ← IA32_PLi_SSP (* where i = new code-segment DPL *)
   IF IDT gate is 32-bit
       THEN
           IF new stack does not have room for 40 bytes (error code pushed)
           or 36 bytes (no error code pushed)
               THEN #SS(error_code(NewSS,0,EXT)); FI;
               (* idt operand to error_code is 0 because selector is used *)
           ELSE (* IDT gate is 16-bit)
               IF new stack does not have room for 20 bytes (error code pushed)
               or 18 bytes (no error code pushed)
                   THEN #SS(error_code(NewSS,0,EXT)); FI;
                   (* idt operand to error_code is 0 because selector is used *)
   FI;
   IF instruction pointer from IDT gate is not within new code-segment limits
       THEN #GP(EXT); FI; (* Error code contains NULL selector *)
   tempEFLAGS ← EFLAGS;
VM ← 0;
TF ← 0;
RF ← 0;
NT ← 0;
   IF service through interrupt gate
       THEN IF = 0; FI;
   TempSS ← SS;
TempESP ← ESP;
SS ← NewSS;
ESP ← NewESP;
(* Following pushes are 16 bits for 16-bit IDT gates and 32 bits for 32-bit IDT gates;
Segment selector pushes in 32-bit mode are padded to two words *)
Push(GS);
Push(FS);
Push(DS);
Push(ES);
Push(TempSS);
Push(TempESP);
   Push(TempEFlags);
Push(CS);
Push(EIP);
GS ← 0; (* Segment registers made NULL, invalid for use in protected mode *)
FS ← 0;
DS ← 0;
ES ← 0;
CS:IP ← Gate(CS); (* Segment descriptor information also loaded *)
IF OperandSize = 32
   THEN
       EIP ← Gate(instruction pointer);
```

```
     ELSE (* OperandSize is 16 *)
            EIP ← Gate(instruction pointer) AND 0000FFFFH;
FI;
      IF ShadowStackEnabled(CPL)
              oldSSP ← SSP
              SSP ← NewSSP
              IF SSP & 0x07 != 0
                  THEN #GP(0); FI;
            Fault = 0
              Atomic Start
                     SSPToken = 8 bytes locked loaded with shadow stack semantics from SSP
                    IF (SSPToken AND 0x01)
                                THEN fault ← 1; FI;
                   IF ((EFER.LMA and CS.L) = 0 AND SSPToken[63:32] != 0)
                                THEN fault ← 1; FI;
                   IF ((SSPToken AND 0xFFFFFFFFFFFFFFFE) != SSP)
                                THEN fault ← 1; FI;
                   IF fault = 0
                                THEN SSPToken = SSPToken OR 0x01; FI;
                        Store 8 bytes of SSPToken and unlock with shadow stack semantics to SSP;
                Atomic End
                If fault = 1
                   THEN #GP(0); FI;
                IF oldSS.DPL != 3
                        ShadowStackPush8B(oldCS); (* Padded with 48 high-order bits of 0 *)
                        ShadowStackPush8B(oldCSBASE + oldRIP); (* Padded with 32 high-order bits of 0 for 32 bit LIP*)
                        ShadowStackPush8B(oldSSP);
                FI;
      FI
     IF EndbranchEnabled (CPL)
            IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH;
            IA32_S_CET.SUPPRESS = 0
      FI;
(* Start execution of new routine in Protected Mode *)
END;
INTRA-PRIVILEGE-LEVEL-INTERRUPT:
NewSSP = SSP;
CHECK_SS_TOKEN = 0
(* PE = 1, DPL = CPL or conforming segment *)
IF IA32_EFER.LMA = 1 (* IA-32e mode *)
     IF IDT-descriptor IST != 0
            THEN
                   TSSstackAddress ← (IDT-descriptor IST « 3) + 28;
                   IF (TSSstackAddress + 7) > TSS limit
                         THEN #TS(error_code(current TSS selector,0,EXT)); FI;
                         (* idt operand to error_code is 0 because selector is used *)
                   NewRSP ← 8 bytes loaded from (current TSS base + TSSstackAddress);
                      If ShadowStackEnabled(CPL)
                             THEN
                                   NewSSPAddress = IA32_INTERRUPT_SSP_TABLE_ADDR + (IDT gate IST « 3)
                                    NewSSP ← 8 bytes loaded from NewSSPAddress
                    CHECK_SS_TOKEN = 1
                FI;
    FI;
IF 32-bit gate (* implies IA32_EFER.LMA = 0 *)
    THEN
        IF current stack does not have room for 16 bytes (error code pushed)
        or 12 bytes (no error code pushed)
            THEN #SS(EXT); FI; (* Error code contains NULL selector *)
        ELSE IF 16-bit gate (* implies IA32_EFER.LMA = 0 *)
            IF current stack does not have room for 8 bytes (error code pushed)
            or 6 bytes (no error code pushed)
                THEN #SS(EXT); FI; (* Error code contains NULL selector *)
            ELSE (* IA32_EFER.LMA = 1, 64-bit gate*)
                IF NewRSP contains a non-canonical address
                    THEN #SS(EXT); (* Error code contains NULL selector *)
    FI;
FI;
IF (IA32_EFER.LMA = 0) (* Not IA-32e mode *)
    THEN
        IF instruction pointer from IDT gate is not within new code-segment limit
            THEN #GP(EXT); FI; (* Error code contains NULL selector *)
    ELSE
        IF instruction pointer from IDT gate contains a non-canonical address
            THEN #GP(EXT); FI; (* Error code contains NULL selector *)
        RSP ← NewRSP & FFFFFFFFFFFFFFF0H;
FI;
IF IDT gate is 32-bit (* implies IA32_EFER.LMA = 0 *)
    THEN
        Push (EFLAGS);
        Push (far pointer to return instruction); (* 3 words padded to 4 *)
        CS:EIP ← Gate(CS:EIP); (* Segment descriptor information also loaded *)
        Push (ErrorCode); (* If any *)
    ELSE
        IF IDT gate is 16-bit (* implies IA32_EFER.LMA = 0 *)
            THEN
                Push (FLAGS);
                Push (far pointer to return location); (* 2 words *)
                CS:IP ← Gate(CS:IP);
                (* Segment descriptor information also loaded *)
                Push (ErrorCode); (* If any *)
            ELSE (* IA32_EFER.LMA = 1, 64-bit gate*)
                Push(far pointer to old stack);
                (* Old SS and SP, each an 8-byte push *)
                Push(RFLAGS); (* 8-byte push *)
                Push(far pointer to return instruction);
                (* Old CS and RIP, each an 8-byte push *)
                Push(ErrorCode); (* If needed, 8 bytes *)
                CS:RIP ← GATE(CS:RIP);
```

```
                    (* Segment descriptor information also loaded *)
        FI;
FI;
CS(RPL) ← CPL;
IF ShadowStackEnabled(CPL)
    IF CHECK_SS_TOKEN == 1
        THEN
            IF NewSSP & 0x07 != 0
              THEN #GP(0); FI;
             Fault = 0
              Atomic Start
                  SSPToken = 8 bytes loaded with shadow stack semantics from NewSSP
                 IF (SSPToken AND 0x01)
                      THEN fault ←1; FI;
                IF ((EFER.LMA and CS.L) = 0 AND SSPToken[63:32] != 0)
                     THEN fault ← 1; FI;
                IF ((SSPToken AND 0xFFFFFFFFFFFFFFFE) != NewSSP)
                     THEN fault ← 1; FI;
                IF fault = 0
                     THEN SSPToken = SSPToken OR 0x01; FI;
                  Store 8 bytes of SSPToken with shadow stack semantics to NewSSP;
              Atomic End
              If fault = 1
                THEN #GP(0); FI;
        FI;
    (* Align to next 8 byte boundary *)
    tempSSP = SSP;
    Shadow_stack_store 4 bytes of 0 to (SSPnewSSP – 4)
    SSP = newSSP & 0xFFFFFFFFFFFFFFF8H;
    (* push cs:lip:ssp on shadow stack *)
    ShadowStackPush8B(oldCS); (* Padded with 48 high-order bits of 0 *)
    ShadowStackPush8B(oldCSBASE + oldRIP); (* Padded with 32 high-order bits of 0 for 32 bit LIP*)
    ShadowStackPush8B(tempSSP);
FI;
IF EndbranchEnabled (CPL)
    IF CPL = 3
      THEN
            IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH
           IA32_U_CET.SUPPRESS = 0
       ELSE
            IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
           IA32_S_CET.SUPPRESS = 0
    FI;
FI;

IF IDT gate is interrupt gate
   THEN IF ← 0; FI; (* Interrupt flag set to 0; interrupts disabled *)
TF ← 0;
NT ← 0;
VM ← 0;
RF ← 0;
```

```
END;
```

Flags Affected

The EFLAGS register is pushed onto the stack. The IF, TF, NT, AC, RF, and VM flags may be cleared, depending on the mode of operation of the processor when the INT instruction is executed (see the "Operation" section). If the interrupt uses a task gate, any flags may be set or cleared, controlled by the EFLAGS image in the new task's TSS.

Protected Mode Exceptions

#GP(error_code)	If the instruction pointer in the IDT or in the interrupt-, trap-, or task gate is beyond the code segment limits.
	If the segment selector in the interrupt-, trap-, or task gate is NULL.
	If an interrupt-, trap-, or task gate, code segment, or TSS segment selector index is outside its descriptor table limits.
	If the vector selects a descriptor outside the IDT limits.
	If an IDT descriptor is not an interrupt-, trap-, or task-descriptor.
	If an interrupt is generated by the INT n, INT 3, or INTO instruction and the DPL of an interrupt-, trap-, or task-descriptor is less than the CPL.
	If the segment selector in an interrupt- or trap-gate does not point to a segment descriptor for a code segment.
	If the segment selector for a TSS has its local/global bit set for local.
	If a TSS segment descriptor specifies that the TSS is busy or not available.
	If SSP in IA32_PLi_SSP (where i is the new CPL) is not 8 byte aligned.
	If "supervisor Shadow Stack" token on new shadow stack is marked busy.
	If destination mode is 32-bit or compatibility mode, but SSP address in "supervisor shadow stack" token is beyond 4G.
	If SSP address in "supervisor shadow stack" token does not match SSP address in IA32_PLi_SSP (where i is the new CPL).
#SS(error_code)	If pushing the return address, flags, or error code onto the stack exceeds the bounds of the stack segment and no stack switch occurs.
	If the SS register is being loaded and the segment pointed to is marked not present.
	If pushing the return address, flags, error code, or stack segment pointer exceeds the bounds of the new stack segment when a stack switch occurs.
#NP(error_code)	If code segment, interrupt-, trap-, or task gate, or TSS is not present.
#TS(error_code)	If the RPL of the stack segment selector in the TSS is not equal to the DPL of the code segment being accessed by the interrupt or trap gate.
	If DPL of the stack segment descriptor pointed to by the stack segment selector in the TSS is not equal to the DPL of the code segment descriptor for the interrupt or trap gate.
	If the stack segment selector in the TSS is NULL.
	If the stack segment for the TSS is not a writable data segment.
	If segment-selector index for stack segment is outside descriptor table limits.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.
#AC(EXT)	If alignment checking is enabled, the gate DPL is 3, and a stack push is unaligned.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the interrupt vector number is outside the IDT limits.
#SS	If stack limit violation on push.
	If pushing the return address, flags, or error code onto the stack exceeds the bounds of the stack segment.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(error_code)	(For INT n, INTO, or BOUND instruction) If the IOPL is less than 3 or the DPL of the interrupt-, trap-, or task-gate descriptor is not equal to 3.
	If the instruction pointer in the IDT or in the interrupt-, trap-, or task gate is beyond the code segment limits.
	If the segment selector in the interrupt-, trap-, or task gate is NULL.
	If an interrupt-, trap-, or task gate, code segment, or TSS segment selector index is outside its descriptor table limits.
	If the vector selects a descriptor outside the IDT limits.
	If an IDT descriptor is not an interrupt-, trap-, or task-descriptor.
	If an interrupt is generated by the INT n instruction and the DPL of an interrupt-, trap-, or task-descriptor is less than the CPL.
	If the segment selector in an interrupt- or trap-gate does not point to a segment descriptor for a code segment.
	If the segment selector for a TSS has its local/global bit set for local.
#SS(error_code)	If the SS register is being loaded and the segment pointed to is marked not present.
	If pushing the return address, flags, error code, stack segment pointer, or data segments exceeds the bounds of the stack segment.
#NP(error_code)	If code segment, interrupt-, trap-, or task gate, or TSS is not present.
#TS(error_code)	If the RPL of the stack segment selector in the TSS is not equal to the DPL of the code segment being accessed by the interrupt or trap gate.
	If DPL of the stack segment descriptor for the TSS's stack segment is not equal to the DPL of the code segment descriptor for the interrupt or trap gate.
	If the stack segment selector in the TSS is NULL.
	If the stack segment for the TSS is not a writable data segment.
	If segment-selector index for stack segment is outside descriptor table limits.
#PF(fault-code)	If a page fault occurs.
#BP	If the INT 3 instruction is executed.
#OF	If the INTO instruction is executed and the OF flag is set.
#UD	If the LOCK prefix is used.
#AC(EXT)	If alignment checking is enabled, the gate DPL is 3, and a stack push is unaligned.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(error_code)	If the instruction pointer in the 64-bit interrupt gate or 64-bit trap gate is non-canonical.
	If the segment selector in the 64-bit interrupt or trap gate is NULL.
	If the vector selects a descriptor outside the IDT limits.
	If the vector points to a gate which is in non-canonical space.
	If the vector points to a descriptor which is not a 64-bit interrupt gate or 64-bit trap gate.
	If the descriptor pointed to by the gate selector is outside the descriptor table limit.
	If the descriptor pointed to by the gate selector is in non-canonical space.
	If the descriptor pointed to by the gate selector is not a code segment.
	If the descriptor pointed to by the gate selector doesn’t have the L-bit set, or has both the L-bit and D-bit set.
	If the descriptor pointed to by the gate selector has DPL > CPL.
	If SSP in IA32_PLi_SSP (where i is the new CPL) is not 8 byte aligned.
	If “supervisor shadow stack” token on new shadow stack is marked busy.
	If destination mode is 32-bit or compatibility mode, but SSP address in “supervisor shadow stack” token is beyond 4G.
	If SSP address in “supervisor shadow stack” token does not match SSP address in IA32_PLi_SSP (where i is the new CPL).
#SS(error_code)	If a push of the old EFLAGS, CS selector, EIP, or error code is in non-canonical space with no stack switch.
	If a push of the old SS selector, ESP, EFLAGS, CS selector, EIP, or error code is in non-canonical space on a stack switch (either CPL change or no-CPL with IST).
#NP(error_code)	If the 64-bit interrupt-gate, 64-bit trap-gate, or code segment is not present.
#TS(error_code)	If an attempt to load RSP from the TSS causes an access to non-canonical space.
	If the RSP from the TSS is outside descriptor table limits.
#PF(fault-code)	If a page fault occurs.
#UD	If the LOCK prefix is used.
#AC(EXT)	If alignment checking is enabled, the gate DPL is 3, and a stack push is unaligned.

4.3 JMP — Jump

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
EB *cb*	JMP *rel8*	D	Valid	Valid	Jump short, RIP = RIP + 8-bit displacement sign extended to 64-bits.
E9 *cw*	JMP *rel16*	D	N.S.	Valid	Jump near, relative, displacement relative to next instruction. Not supported in 64-bit mode.
E9 *cd*	JMP *rel32*	D	Valid	Valid	Jump near, relative, RIP = RIP + 32-bit displacement sign extended to 64-bits.
FF /4	JMP *r/m16*	M	N.S.	Valid	Jump near, absolute indirect, address = zero-extended *r/m16.* Not supported in 64-bit mode.
FF /4	JMP *r/m32*	M	N.S.	Valid	Jump near, absolute indirect, address given in *r/m32.* Not supported in 64-bit mode.
FF /4	JMP *r/m64*	M	Valid	N.E.	Jump near, absolute indirect, RIP = 64-Bit offset from register or memory.
EA *cd*	JMP *ptr16:16*	D	Inv.	Valid	Jump far, absolute, address given in operand.
EA *cp*	JMP *ptr16:32*	D	Inv.	Valid	Jump far, absolute, address given in operand.
FF /5	JMP *m16:16*	D	Valid	Valid	Jump far, absolute indirect, address given in *m16:16.*
FF /5	JMP *m16:32*	D	Valid	Valid	Jump far, absolute indirect, address given in *m16:32.*
REX.W + FF /5	JMP *m16:64*	D	Valid	N.E.	Jump far, absolute indirect, address given in *m16:64.*

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
D	Offset	NA	NA	NA
M	ModRM:r/m (r)	NA	NA	NA

Description

Transfers program control to a different point in the instruction stream without recording return information. The destination (target) operand specifies the address of the instruction being jumped to. This operand can be an immediate value, a general-purpose register, or a memory location.

This instruction can be used to execute four different types of jumps:

- Near jump—A jump to an instruction within the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intrasegment jump.
- Short jump—A near jump where the jump range is limited to –128 to +127 from the current EIP value.
- Far jump—A jump to an instruction located in a different segment than the current code segment but at the same privilege level, sometimes referred to as an intersegment jump.
- Task switch—A jump to an instruction located in a different task.

A task switch can only be executed in protected mode (see Chapter 7, in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for information on performing task switches with the JMP instruction).

Near and Short Jumps. When executing a near jump, the processor jumps to the address (within the current code segment) that is specified with the target operand. The target operand specifies either an absolute offset (that is an offset from the base of the code segment) or a relative offset (a signed displacement relative to the current value of the instruction pointer in the EIP register). A near jump to a relative offset of 8-bits (*rel8*) is referred to as a short jump. The CS register is not changed on near and short jumps.

An absolute offset is specified indirectly in a general-purpose register or a memory location (*r/m16* or *r/m32*). The operand-size attribute determines the size of the target operand (16 or 32 bits). Absolute offsets are loaded directly into the EIP register. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared, resulting in a maximum instruction pointer size of 16 bits.

A relative offset (*rel8, rel16*, or *rel32*) is generally specified as a label in assembly code, but at the machine code level, it is encoded as a signed 8-, 16-, or 32-bit immediate value. This value is added to the value in the EIP register. (Here, the EIP register contains the address of the instruction following the JMP instruction). When using relative offsets, the opcode (for short vs. near jumps) and the operand-size attribute (for near relative jumps) determines the size of the target operand (8, 16, or 32 bits).

Far Jumps in Real-Address or Virtual-8086 Mode. When executing a far jump in real-address or virtual-8086 mode, the processor jumps to the code segment and offset specified with the target operand. Here the target operand specifies an absolute far address either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). With the pointer method, the segment and address of the called procedure is encoded in the instruction, using a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address immediate. With the indirect method, the target operand specifies a memory location that contains a 4-byte (16-bit operand size) or 6-byte (32-bit operand size) far address. The far address is loaded directly into the CS and EIP registers. If the operand-size attribute is 16, the upper two bytes of the EIP register are cleared.

Far Jumps in Protected Mode. When the processor is operating in protected mode, the JMP instruction can be used to perform the following three types of far jumps.

- A far jump to a conforming or non-conforming code segment.
- A far jump through a call gate.
- A task switch.

(The JMP instruction cannot be used to perform inter-privilege-level far jumps.)

In protected mode, the processor always uses the segment selector part of the far address to access the corresponding descriptor in the GDT or LDT. The descriptor type (code segment, call gate, task gate, or TSS) and access rights determine the type of jump to be performed.

If the selected descriptor is for a code segment, a far jump to a code segment at the same privilege level is performed. (If the selected code segment is at a different privilege level and the code segment is non-conforming, a general-protection exception is generated.) A far jump to the same privilege level in protected mode is very similar to one carried out in real-address or virtual-8086 mode. The target operand specifies an absolute far address either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*). The operand-size attribute determines the size of the offset (16 or 32 bits) in the far address. The new code segment selector and its descriptor are loaded into CS register, and the offset from the instruction is loaded into the EIP register. Note that a call gate (described in the next paragraph) can also be used to perform far call to a code segment at the same privilege level. Using this mechanism provides an extra level of indirection and is the preferred method of making jumps between 16-bit and 32-bit code segments.

When executing a far jump through a call gate, the segment selector specified by the target operand identifies the call gate. (The offset part of the target operand is ignored.) The processor then jumps to the code segment specified in the call gate descriptor and begins executing the instruction at the offset specified in the call gate. No stack switch occurs. Here again, the target operand can specify the far address of the call gate either directly with a pointer (*ptr16:16* or *ptr16:32*) or indirectly with a memory location (*m16:16* or *m16:32*).

Executing a task switch with the JMP instruction is somewhat similar to executing a jump through a call gate. Here the target operand specifies the segment selector of the task gate for the task being switched to (and the offset part of the target operand is ignored). The task gate in turn points to the TSS for the task, which contains the segment selectors for the task's code and stack segments. The TSS also contains the EIP value for the next instruction that was to be executed before the task was suspended. This instruction pointer value is loaded into the EIP register so that the task begins executing again at this next instruction.

The JMP instruction can also specify the segment selector of the TSS directly, which eliminates the indirection of the task gate. See Chapter 7 in *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*, for detailed information on the mechanics of a task switch.

Note that when you execute at task switch with a JMP instruction, the nested task flag (NT) is not set in the EFLAGS register and the new TSS's previous task link field is not loaded with the old task's TSS selector. A return to the previous task can thus not be carried out by executing the IRET instruction. Switching tasks with the JMP instruction differs in this regard from the CALL instruction which does set the NT flag and save the previous task link information, allowing a return to the calling task with an IRET instruction.

In 64-Bit Mode — The instruction's operation size is fixed at 64 bits. If a selector points to a gate, then RIP equals the 64-bit displacement taken from gate; else RIP equals the zero-extended offset from the far pointer referenced in the instruction.

See the summary chart at the beginning of this section for encoding data and limits.

Operation

```
IF near jump
    IF 64-bit Mode
        THEN
            IF near relative jump
              THEN
                  tempRIP ← RIP    + DEST; (* RIP is instruction following JMP instruction*)
              ELSE (* Near absolute jump *)
                  tempRIP ← DEST;
            FI;
        ELSE
            IF near relative jump
              THEN
                  tempEIP ← EIP    + DEST; (* EIP is instruction following JMP instruction*)
```

```
                 ELSE (* Near absolute jump *)
                     tempEIP ← DEST;
                FI;
    FI;
    IF (IA32_EFER.LMA = 0 or target mode = Compatibility mode)
    and tempEIP outside code segment limit
          THEN #GP(0); FI
    IF 64-bit mode and tempRIP is not canonical
          THEN #GP(0);
    FI;
    IF OperandSize = 32
            THEN
                EIP ← tempEIP;
            ELSE
                IF OperandSize = 16
                      THEN (* OperandSize = 16 *)
                            EIP ← tempEIP AND 0000FFFFH;
                        ELSE (* OperandSize = 64)
                            RIP ← tempRIP;
                FI;
      FI;
       IF (JMP near indirect, absolute indirect)
          IF EndbranchEnabledAndNotSuppressed(CPL)
                IF CPL = 3
                      THEN
                                  IF ( no 3EH prefix OR IA32_U_CET.NO_TRACK_EN == 0 )
                                       THEN
                                         IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH
                                       FI
                      ELSE
                                  IF ( no 3EH prefix OR IA32_S_CET.NO_TRACK_EN == 0 )
                                       THEN
                                         IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
                                       FI
                FI;
          FI;
       FI;
FI;
IF far jump and (PE = 0 or (PE = 1 AND VM = 1)) (* Real-address or virtual-8086 mode *)
      THEN
            tempEIP ← DEST(Offset); (* DEST is ptr16:32 or [m16:32] *)
            IF tempEIP is beyond code segment limit
                THEN #GP(0); FI;
            CS ← DEST(segment selector); (* DEST is ptr16:32 or [m16:32] *)
            IF OperandSize = 32
                  THEN
                EIP ← tempEIP; (* DEST is ptr16:32 or [m16:32] *)
            ELSE (* OperandSize = 16 *)
                EIP ← tempEIP AND 0000FFFFH; (* Clear upper 16 bits *)
        FI;
FI;
IF far jump and (PE = 1 and VM = 0)
(* IA-32e mode or protected mode, not virtual-8086 mode *)
    THEN
        IF effective address in the CS, DS, ES, FS, GS, or SS segment is illegal
        or segment selector in target operand NULL
                THEN #GP(0); FI;
        IF segment selector index not within descriptor table limits
            THEN #GP(new selector); FI;
        Read type and access rights of segment descriptor;
        IF (EFER.LMA = 0)
            THEN
                IF segment type is not a conforming or nonconforming code
                segment, call gate, task gate, or TSS
                    THEN #GP(segment selector); FI;
            ELSE
                IF segment type is not a conforming or nonconforming code segment
                call gate
                    THEN #GP(segment selector); FI;
        FI;
        Depending on type and access rights:
            GO TO CONFORMING-CODE-SEGMENT;
            GO TO NONCONFORMING-CODE-SEGMENT;
            GO TO CALL-GATE;
            GO TO TASK-GATE;
            GO TO TASK-STATE-SEGMENT;
    ELSE
        #GP(segment selector);
FI;
CONFORMING-CODE-SEGMENT:
    IF L-Bit = 1 and D-BIT = 1 and IA32_EFER.LMA = 1
        THEN GP(new code segment selector); FI;
      IF DPL > CPL
        THEN #GP(segment selector); FI;
      IF segment not present
        THEN #NP(segment selector); FI;
    tempEIP ← DEST(Offset);
    IF OperandSize = 16
        THEN tempEIP ← tempEIP AND 0000FFFFH;
    FI;
    IF (IA32_EFER.LMA = 0 or target mode = Compatibility mode) and
    tempEIP outside code segment limit
        THEN #GP(0); FI
    IF tempEIP is non-canonical
        THEN #GP(0); FI;
    IF ShadowStackEnabled(CPL)
```

```
            IF (EFER.LMA and DEST(segment selector).L) = 0
                (* If target is legacy or compatibility mode then the SSP must be in low 4G *)
                IF (SSP & 0xFFFFFFFF00000000 != 0)
                    THEN #GP(0); FI;
        FI;
    FI;
    CS ← DEST[segment selector]; (* Segment descriptor information also loaded *)
    CS(RPL) ← CPL
    EIP ← tempEIP;
    IF EndbranchEnabled(CPL)
        IF CPL = 3
            THEN
                IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH
                IA32_U_CET.SUPPRESS = 0
            ELSE
                IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
                IA32_S_CET.SUPPRESS = 0
        FI;
    FI;
END;
NONCONFORMING-CODE-SEGMENT:
    IF L-Bit = 1 and D-BIT = 1 and IA32_EFER.LMA = 1
        THEN GP(new code segment selector); FI;
    IF (RPL > CPL) OR (DPL != CPL)
        THEN #GP(code segment selector); FI;
    IF segment not present
        THEN #NP(segment selector); FI;
    tempEIP ← DEST(Offset);
    IF OperandSize = 16
        THEN tempEIP ← tempEIP AND 0000FFFFH; FI;
    IF (IA32_EFER.LMA = 0 OR target mode = Compatibility mode)
    and tempEIP outside code segment limit
        THEN #GP(0); FI
    IF tempEIP is non-canonical THEN #GP(0); FI;
    IF ShadowStackEnabled(CPL)
            IF (EFER.LMA and DEST(segment selector).L) = 0
                (* If target is legacy or compatibility mode then the SSP must be in low 4G *)
                IF (SSP & 0xFFFFFFFF00000000 != 0)
                    THEN #GP(0); FI;
        FI;
    FI;
    CS ← DEST[segment selector]; (* Segment descriptor information also loaded *)
    CS(RPL) ← CPL;
    EIP ← tempEIP;
    IF EndbranchEnabled(CPL)
        IF CPL = 3
            THEN
```

```
                IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH
                  IA32_U_CET.SUPPRESS = 0
            ELSE
                IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
                  IA32_S_CET.SUPPRESS = 0
        FI;
    FI;
END;

CALL-GATE:
   IF call gate DPL    < CPL
   or call gate DPL    < call gate segment-selector RPL
            THEN #GP(call gate selector); FI;
   IF call gate not present
        THEN #NP(call gate selector); FI;
   IF call gate code-segment selector is NULL
        THEN #GP(0); FI;
   IF call gate code-segment selector index outside descriptor table limits
        THEN #GP(code segment selector); FI;
   Read code segment descriptor;
   IF code-segment segment descriptor does not indicate a code segment
   or code-segment segment descriptor is conforming and DPL > CPL
   or code-segment segment descriptor is non-conforming and DPL != CPL
            THEN #GP(code segment selector); FI;
   IF IA32_EFER.LMA = 1 and (code-segment descriptor is not a 64-bit code segment
   or code-segment segment descriptor has both L-Bit and D-bit set)
            THEN #GP(code segment selector); FI;
   IF code segment is not present
        THEN #NP(code-segment selector); FI;
    IF instruction pointer is not within code-segment limit
        THEN #GP(0); FI;
    tempEIP ← DEST(Offset);
    IF GateSize = 16
         THEN tempEIP ← tempEIP AND 0000FFFFH; FI;
   IF (IA32_EFER.LMA = 0 OR target mode = Compatibility mode) AND tempEIP
   outside code segment limit
        THEN #GP(0); FI
   CS ← DEST[SegmentSelector); (* Segment descriptor information also loaded *)
   CS(RPL) ← CPL;
   EIP ← tempEIP;
   IF EndbranchEnabled(CPL)
        IF CPL = 3
            THEN
                IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH;
                  IA32_U_CET.SUPPRESS = 0
            ELSE
                IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH;
                  IA32_S_CET.SUPPRESS = 0
        FI;
    FI;
END;
```

```
TASK-GATE:
    IF task gate DPL    < CPL
    or task gate DPL    < task gate segment-selector RPL
          THEN #GP(task gate selector); FI;
    IF task gate not present
          THEN #NP(gate selector); FI;
    Read the TSS segment selector in the task-gate descriptor;
    IF TSS segment selector local/global bit is set to local
    or index not within GDT limits
    or TSS descriptor specifies that the TSS is busy
          THEN #GP(TSS selector); FI;
      IF TSS not present
          THEN #NP(TSS selector); FI;
      SWITCH-TASKS to TSS;
      IF EIP not within code segment limit
          THEN #GP(0); FI;
END;
TASK-STATE-SEGMENT:
    IF TSS DPL    < CPL
    or TSS DPL    < TSS segment-selector RPL
    or TSS descriptor indicates TSS not available
          THEN #GP(TSS selector); FI;
    IF TSS is not present
          THEN #NP(TSS selector); FI;
    SWITCH-TASKS to TSS;
    IF EIP not within code segment limit
          THEN #GP(0); FI;
END;
```

Flags Affected

All flags are affected if a task switch occurs; no flags are affected if a task switch does not occur.

Protected Mode Exceptions

#GP(0)	If offset in target operand, call gate, or TSS is beyond the code segment limits.
	If the segment selector in the destination operand, call gate, task gate, or TSS is NULL.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
	If target mode is compatibility mode and SSP is not in low 4G.
#GP(selector)	If the segment selector index is outside descriptor table limits.
	If the segment descriptor pointed to by the segment selector in the -destination operand is not for a conforming-code segment, nonconforming-code segment, call gate, task gate, or task state segment.
	If the DPL for a nonconforming-code segment is not equal to the CPL

	(When not using a call gate.) If the RPL for the segment's segment selector is greater than the CPL.
	If the DPL for a conforming-code segment is greater than the CPL.
	If the DPL from a call-gate, task-gate, or TSS segment descriptor is less than the CPL or than the RPL of the call-gate, task-gate, or TSS's segment selector.
	If the segment descriptor for selector in a call gate does not indicate it is a code segment.
	If the segment descriptor for the segment selector in a task gate does not indicate an available TSS.
	If the segment selector for a TSS has its local/global bit set for local.
	If a TSS segment descriptor specifies that the TSS is busy or not available.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#NP (selector)	If the code segment being accessed is not present.
	If call gate, task gate, or TSS not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3. (Only occurs when fetching target from memory.)
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS	If a memory operand effective address is outside the SS segment limit.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	If the target operand is beyond the code segment limits.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made. (Only occurs when fetching target from memory.)
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

Same as 64-bit mode exceptions.

64-Bit Mode Exceptions

#GP(0)	If a memory address is non-canonical.
	If target offset in destination operand is non-canonical.
	If target offset in destination operand is beyond the new code segment limit.
	If the segment selector in the destination operand is NULL.
	If the code segment selector in the 64-bit gate is NULL.
	If transitioning to compatibility mode and the SSP is beyond 4G.
#GP(selector)	If the code segment or 64-bit call gate is outside descriptor table limits.
	If the code segment or 64-bit call gate overlaps non-canonical space.

	If the segment descriptor from a 64-bit call gate is in non-canonical space.
	If the segment descriptor pointed to by the segment selector in the -destination operand is not for a conforming-code segment, nonconforming-code segment, 64-bit call gate.
	If the segment descriptor pointed to by the segment selector in the -destination operand is a code segment, and has both the D-bit and the L-bit set.
	If the DPL for a nonconforming-code segment is not equal to the CPL, or the RPL for the segment's segment selector is greater than the CPL.
	If the DPL for a conforming-code segment is greater than the CPL.
	If the DPL from a 64-bit call-gate is less than the CPL or than the RPL of the 64-bit call-gate.
	If the upper type field of a 64-bit call gate is not 0x0.
	If the segment selector from a 64-bit call gate is beyond the descriptor table limits.
	If the code segment descriptor pointed to by the selector in the 64-bit gate doesn't have the L-bit set and the D-bit clear.
	If the segment descriptor for a segment selector from the 64-bit call gate does not indicate it is a code segment.
	If the code segment is non-confirming and CPL != DPL.
	If the code segment is confirming and CPL < DPL.
#NP(selector)	If a code segment or 64-bit call gate is not present.
#UD	(64-bit mode only) If a far jump is direct to an absolute address in memory.
	If the LOCK prefix is used.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
	If CPUID.01H:ECX.MONITOR[bit 3] = 0.

4.4 RET—Return from Procedure

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
C3	RET	NP	Valid	Valid	Near return to calling procedure.
CB	RET	NP	Valid	Valid	Far return to calling procedure.
C2 *iw*	RET *imm16*	I	Valid	Valid	Near return to calling procedure and pop *imm16* bytes from stack.
CA *iw*	RET *imm16*	I	Valid	Valid	Far return to calling procedure and pop *imm16* bytes from stack.

Instruction Operand Encoding

Op/ En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA
I	imm16	NA	NA	NA

Description

Transfers program control to a return address located on the top of the stack. The address is usually placed on the stack by a CALL instruction, and the return is made to the instruction that follows the CALL instruction.

The optional source operand specifies the number of stack bytes to be released after the return address is popped; the default is none. This operand can be used to release parameters from the stack that were passed to the called procedure and are no longer needed. It must be used when the CALL instruction used to switch to a new procedure uses a call gate with a non-zero word count to access the new procedure. Here, the source operand for the RET instruction must specify the same number of bytes as is specified in the word count field of the call gate.

The RET instruction can be used to execute three different types of returns.

- **Near return** — A return to a calling procedure within the current code segment (the segment currently pointed to by the CS register), sometimes referred to as an intrasegment return.
- **Far return** — A return to a calling procedure located in a different segment than the current code segment, sometimes referred to as an intersegment return.
- **Inter-privilege-level far return** — A far return to a different privilege level than that of the currently executing program or procedure.

The inter-privilege-level return type can only be executed in protected mode. See the section titled "Calling Procedures Using Call and RET" in Chapter 6 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1*, for detailed information on near, far, and inter-privilege-level returns.

When executing a near return, the processor pops the return instruction pointer (offset) from the top of the stack into the EIP register and begins program execution at the new instruction pointer. The CS register is unchanged.

When executing a far return, the processor pops the return instruction pointer from the top of the stack into the EIP register, then pops the segment selector from the top of the stack into the CS register. The processor then begins program execution in the new code segment at the new instruction pointer.

The mechanics of an inter-privilege-level far return are similar to an intersegment return, except that the processor examines the privilege levels and access rights of the code and stack segments being returned to determine if the control transfer is allowed to be made. The DS, ES, FS, and GS segment registers are cleared by the RET instruction during an inter-privilege-level return if they refer to segments that are not allowed to be accessed at the new privilege level. Since a stack switch also occurs on an inter-privilege level return, the ESP and SS registers are loaded from the stack.

If parameters are passed to the called procedure during an inter-privilege level call, the optional source operand must be used with the RET instruction to release the parameters on the return. Here, the parameters are released both from the called procedure's stack and the calling procedure's stack (that is, the stack being returned to).

In 64-bit mode, the default operation size of this instruction is the stack-address size, i.e. 64 bits. This applies to near returns, not far returns; the default operation size of far returns is 32 bits.

Operation

```
(* Near return *)
IF instruction = near return
   THEN;
      IF OperandSize = 32
         THEN
            IF top 4 bytes of stack not within stack limits
               THEN #SS(0); FI;
            EIP ← Pop();
               IF ShadowStackEnabled(CPL)
                  tempSsEIP = PopShadowStack4B();
                  IF EIP != TempSsEIP
                     THEN #CP(NEAR_RET); FI;
               FI;
         ELSE
            IF OperandSize = 64
               THEN
                  IF top 8 bytes of stack not within stack limits
                     THEN #SS(0); FI;
                  RIP ← Pop();
                     IF ShadowStackEnabled(CPL)
                        tempSsEIP = PopShadowStack8B();
                        IF RIP != tempSsEIP
                           THEN #CP(NEAR_RET); FI;
                     FI;
               ELSE (* OperandSize = 16 *)
                  IF top 2 bytes of stack not within stack limits
                     THEN #SS(0); FI;
                  tempEIP ← Pop();
                  tempEIP ← tempEIP AND 0000FFFFH;
                  IF tempEIP not within code segment limits
                     THEN #GP(0); FI;
                  EIP ← tempEIP;
                     IF ShadowStackEnabled(CPL)
                        tempSsEip = PopShadowStack4B();
```

```
                                    IF EIP != tempSsEIP
                                        THEN #CP(NEAR_RET); FI;
                                FI;
                    FI;
            FI;

    IF instruction has immediate operand
        THEN (* Release parameters from stack *)
            IF StackAddressSize = 32
                THEN
                    ESP ← ESP + SRC;
                ELSE
                    IF StackAddressSize = 64
                        THEN
                            RSP ← RSP + SRC;
                        ELSE (* StackAddressSize = 16 *)
                            SP ← SP + SRC;
                    FI;
            FI;
    FI;
FI;

(* Real-address mode or virtual-8086 mode *)
IF ((PE = 0) or (PE = 1 AND VM = 1)) and instruction = far return
    THEN
        IF OperandSize = 32
            THEN
                IF top 8 bytes of stack not within stack limits
                    THEN #SS(0); FI;
                EIP ← Pop();
                CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
            ELSE (* OperandSize = 16 *)
                IF top 4 bytes of stack not within stack limits
                    THEN #SS(0); FI;
                tempEIP ← Pop();
                tempEIP ← tempEIP AND 0000FFFFH;
                IF tempEIP not within code segment limits
                    THEN #GP(0); FI;
                EIP ← tempEIP;
                CS ← Pop(); (* 16-bit pop *)
        FI;
    IF instruction has immediate operand
        THEN (* Release parameters from stack *)
            SP ← SP + (SRC AND FFFFH);
    FI;
FI;

(* Protected mode, not virtual-8086 mode *)
IF (PE = 1 and VM = 0 and IA32_EFER.LMA = 0) and instruction = far return
    THEN
        IF OperandSize = 32
```

```
            THEN
                IF second doubleword on stack is not within stack limits
                    THEN #SS(0); FI;
            ELSE (* OperandSize = 16 *)
                IF second word on stack is not within stack limits
                    THEN #SS(0); FI;
        FI;
    IF return code segment selector is NULL
        THEN #GP(0); FI;
    IF return code segment selector addresses descriptor beyond descriptor table limit
        THEN #GP(selector); FI;
    Obtain descriptor to which return code segment selector points from descriptor table;
    IF return code segment descriptor is not a code segment
        THEN #GP(selector); FI;
    IF return code segment selector RPL    < CPL
        THEN #GP(selector); FI;
    IF return code segment descriptor is conforming
    and return code segment DPL > return code segment selector RPL
        THEN #GP(selector); FI;
    IF return code segment descriptor is non-conforming and return code
    segment DPL 1 return code segment selector RPL
        THEN #GP(selector); FI;
    IF return code segment descriptor is not present
        THEN #NP(selector); FI:
    IF return code segment selector RPL > CPL
        THEN GOTO RETURN-OUTER-PRIVILEGE-LEVEL;
        ELSE GOTO RETURN-TO-SAME-PRIVILEGE-LEVEL;
    FI;
FI;

RETURN-SAME-PRIVILEGE-LEVEL:
    IF the return instruction pointer is not within the return code segment limit
        THEN #GP(0); FI;
    IF OperandSize = 32
        THEN
            EIP ← Pop();
            CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
        ELSE (* OperandSize = 16 *)
            EIP ← Pop();
            EIP ← EIP AND 0000FFFFH;
            CS ← Pop(); (* 16-bit pop *)      FI;
    IF instruction has immediate operand
        THEN (* Release parameters from stack *)
            IF StackAddressSize = 32
                THEN
                    ESP ← ESP    + SRC;
                ELSE (* StackAddressSize = 16 *)
```

```
                    SP ← SP + SRC;
            FI;
    FI;
     IF ShadowStackEnabled(CPL)
            (* SSP must be 8 byte aligned *)
            IF SSP AND 0x7 != 0
                THEN #CP(FAR-RET/IRET); FI;
            tempSsCS = shadow_stack_load 8 bytes from SSP+16;
            tempSsLIP = shadow_stack_load 8 bytes from SSP+8;
            prevSSP = shadow_stack_load 8 bytes from SSP;
            SSP = SSP + 24;
            (* do a 64 bit-compare to check if any bits beyond bit 15 are set *)
            tempCS = CS; (* zero pad to 64 bit *)
            IF tempCS != tempSsCS
                THEN #CP(FAR-RET/IRET); FI;
            (* do a 64 bit-compare; pad CSBASE+RIP with 0 for 32 bit LIP*)
            IF CSBASE + RIP != tempSsLIP
                THEN #CP(FAR-RET/IRET); FI;
            (* prevSSP must be 4 byte aligned *)
            IF prevSSP AND 0x3 != 0
                THEN #CP(FAR-RET/IRET); FI;
            (* If returning to compatibility mode then SSP must be in low 4G *)
          IF ((EFER.LMA and CS.L) = 0 AND prevSSP[63:32] != 0)
                THEN #GP(0); FI;
            SSP ← prevSSP
     FI;
END;

RETURN-OUTER-PRIVILEGE-LEVEL:
    IF top (16 + SRC) bytes of stack are not within stack limits (OperandSize = 32)
    or top (8 + SRC) bytes of stack are not within stack limits (OperandSize = 16)
            THEN #SS(0); FI;
    Read return segment selector;
    IF stack segment selector is NULL
        THEN #GP(0); FI;
    IF return stack segment selector index is not within its descriptor table limits
        THEN #GP(selector); FI;
    Read segment descriptor pointed to by return segment selector;
    IF stack segment selector RPL != RPL of the return code segment selector
    or stack segment is not a writable data segment
    or stack segment descriptor DPL != RPL of the return code segment selector
            THEN #GP(selector); FI;
    IF stack segment not present
        THEN #SS(StackSegmentSelector); FI;
    IF the return instruction pointer is not within the return code segment limit
        THEN #GP(0); FI;
    IF OperandSize = 32
        THEN
            EIP ← Pop();
            CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded; segment descriptor loaded *)
            CS(RPL) ← CPL;
            IF instruction has immediate operand
                THEN (* Release parameters from called procedure's stack *)
                    IF StackAddressSize = 32
                        THEN
                            ESP ← ESP   + SRC;
                        ELSE (* StackAddressSize = 16 *)
                            SP ← SP   + SRC;
                    FI;
            FI;
            tempESP ← Pop();
            tempSS ← Pop(); (* 32-bit pop, high-order 16 bits discarded; seg. descriptor loaded *)
        ELSE (* OperandSize = 16 *)
            EIP ← Pop();
            EIP ← EIP AND 0000FFFFH;
            CS ← Pop(); (* 16-bit pop; segment descriptor loaded *)
            CS(RPL) ← CPL;
            IF instruction has immediate operand
                THEN (* Release parameters from called procedure's stack *)
                    IF StackAddressSize = 32
                        THEN
                            ESP ← ESP   + SRC;
                        ELSE (* StackAddressSize = 16 *)
                            SP ← SP   + SRC;
                    FI;
            FI;
            tempESP ← Pop();
            tempSS ← Pop(); (* 16-bit pop; segment descriptor loaded *)
    FI;
    IF ShadowStackEnabled(CPL)
        (* check if 8 byte aligned *)
        IF SSP AND 0x7 != 0
            THEN #CP(FAR-RET/IRET); FI;
        IF ReturnCodeSegmentSelector(RPL) !=3
            THEN
                tempSsCS = shadow_stack_load 8 bytes from SSP+16;
                tempSsLIP = shadow_stack_load 8 bytes from SSP+8;
                tempSSP = shadow_stack_load 8 bytes from SSP;
                SSP = SSP + 24;
                (* Do 64 bit compare to detect bits beyond 15 being set *)
                tempCS = CS; (* zero extended to 64 bit *)
                IF tempCS != tempSsCS
                    THEN #CP(FAR-RET/IRET); FI;
                (* Do 64 bit compare; pad CSBASE+RIP with 0 for 32 bit LA *)
                IF CSBASE + RIP != tempSsLIP
                    THEN #CP(FAR-RET/IRET); FI;
                (* check if 4 byte aligned *)
                IF tempSSP AND 0x3 != 0
```

```
                        THEN #CP(FAR-RET/IRET); FI;
    FI;
 FI;

 tempOldCPL = CPL;
CPL ← ReturnCodeSegmentSelector(RPL);
 (* update SS:ESP after CPL broadcast complete *)
ESP ← tempESP;
SS ← tempSS;
 tempOldSSP = SSP;
 IF ShadowStackEnabled(CPL)
       IF CPL = 3
            THEN tempSSP ← IA32_PL3_SSP; FI;
       IF ((EFER.LMA AND CS.L) = 0 AND tempSSP[63:32] != 0)
            THEN #GP(0); FI;
        SSP ← tempSSP
FI;
 (* Now past all faulting points; safe to free the token. The token free is done using the old SSP
   * and using a supervisor override as old CPL was a supervisor privilege level *)
 IF ShadowStackEnabled(tempOldCPL)
      Atomic Start
           SSPToken ← Load 8 bytes with shadow stack semantics and supervisor override from tempOldSSP
           invalidToken ← 0
           IF ((SSPToken AND 0x01) = 0) (* If busy bit not set then invalid token*)
                THEN invalidToken ← 1; FI;
           IF ((SSPToken AND 0xFFFFFFFFFFFFFFFE) != tempOldSSP ) (* If current SSP does not match token *)
                THEN invalidToken ← 1; FI;
           (* Valid token found; clear its busy bit *)
           IF invalidToken = 0
                THEN SSPToken ← SSPToken XOR 0x01;
           Store 8 bytes of SSPToken with shadow stack semantics and supervisor override to tempOldSSP;
       Atomic End
 FI;

FOR each of segment register (ES, FS, GS, and DS)
      DO
           IF segment register points to data or non-conforming code segment
           and CPL > segment descriptor DPL (* DPL in hidden part of segment register *)
                THEN SegmentSelector ← 0; (* Segment selector invalid *)
           FI;
      OD;

IF instruction has immediate operand
      THEN (* Release parameters from calling procedure's stack *)
           IF StackAddressSize = 32
                THEN
                     ESP ← ESP    + SRC;
                ELSE (* StackAddressSize = 16 *)
                     SP ← SP    + SRC;
           FI;
FI;
```

```
END;

(* IA-32e Mode *)
  IF (PE =1 and VM = 0 and IA32_EFER.LMA = 1) and instruction = far return
      THEN
          IF OperandSize =32
              THEN
                  IF second doubleword on stack is not within stack limits
                      THEN #SS(0); FI;
                  IF first or second doubleword on stack is not in canonical space
                      THEN #SS(0); FI;
              ELSE
                  IF OperandSize = 16
                      THEN
                          IF second word on stack is not within stack limits
                              THEN #SS(0); FI;
                          IF first or second word on stack is not in canonical space
                              THEN #SS(0); FI;
                      ELSE (* OperandSize = 64 *)
                          IF first or second quadword on stack is not in canonical space
                              THEN #SS(0); FI;
                  FI
          FI;
      IF return code segment selector is NULL
          THEN GP(0); FI;
      IF return code segment selector addresses descriptor beyond descriptor table limit
          THEN GP(selector); FI;
      IF return code segment selector addresses descriptor in non-canonical space
          THEN GP(selector); FI;
      Obtain descriptor to which return code segment selector points from descriptor table;
      IF return code segment descriptor is not a code segment
          THEN #GP(selector); FI;
      IF return code segment descriptor has L-bit = 1 and D-bit = 1
          THEN #GP(selector); FI;
      IF return code segment selector RPL    < CPL
          THEN #GP(selector); FI;
      IF return code segment descriptor is conforming
      and return code segment DPL > return code segment selector RPL
          THEN #GP(selector); FI;
      IF return code segment descriptor is non-conforming
      and return code segment DPL $^{1}$ return code segment selector RPL
          THEN #GP(selector); FI;
      IF return code segment descriptor is not present
          THEN #NP(selector); FI:
      IF return code segment selector RPL > CPL
          THEN GOTO IA-32E-MODE-RETURN-OUTER-PRIVILEGE-LEVEL;
```

```
                ELSE GOTO IA-32E-MODE-RETURN-SAME-PRIVILEGE-LEVEL;
            FI;
    FI;

IA-32E-MODE-RETURN-SAME-PRIVILEGE-LEVEL:
IF the return instruction pointer is not within the return code segment limit
    THEN #GP(0); FI;
IF the return instruction pointer is not within canonical address space
    THEN #GP(0); FI;
IF OperandSize = 32
    THEN
        EIP ← Pop();
        CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
    ELSE
        IF OperandSize = 16
            THEN
                EIP ← Pop();
                EIP ← EIP AND 0000FFFFH;
                CS ← Pop(); (* 16-bit pop *)
            ELSE (* OperandSize = 64 *)
                RIP ← Pop();
                CS ← Pop(); (* 64-bit pop, high-order 48 bits discarded *)
        FI;
FI;
IF instruction has immediate operand
    THEN (* Release parameters from stack *)
        IF StackAddressSize = 32
            THEN
                ESP ← ESP    + SRC;
            ELSE
                IF StackAddressSize = 16
                    THEN
                        SP ← SP    + SRC;
                    ELSE (* StackAddressSize = 64 *)
                        RSP ← RSP    + SRC;
                FI;
        FI;
FI;
IF ShadowStackEnabled(CPL)
        IF SSP AND 0x7 != 0 (* check if aligned to 8 bytes *)
                THEN #CP(FAR-RET/IRET); FI;
        tempSsCS = shadow_stack_load 8 bytes from SSP+16;
        tempSsLIP = shadow_stack_load 8 bytes from SSP+8;
        tempSSP = shadow_stack_load 8 bytes from SSP;
        SSP = SSP + 24;
        tempCS = CS; (* zero padded to 64 bit *)
        IF tempCS != tempSsCS (* 64 bit compare; CS zero padded to 64 bits *)
                THEN #CP(FAR-RET/IRET); FI;
        IF CSBASE + RIP != tempSsLIP (* 64 bit compare;
                THEN #CP(FAR-RET/IRET); FI;
        IF tempSSP AND 0x3 != 0 (* check if aligned to 4 bytes *)
```

```
                THEN #CP(FAR-RET/IRET); FI;
        IF ((EFER.LMA AND CS.L) = 0 AND tempSSP[63:32] != 0)
            THEN #GP(0); FI;
        SSP ← tempSSP
FI;
END;

IA-32E-MODE-RETURN-OUTER-PRIVILEGE-LEVEL:
IF top (16    + SRC) bytes of stack are not within stack limits (OperandSize = 32)
or top (8    + SRC) bytes of stack are not within stack limits (OperandSize = 16)
    THEN #SS(0); FI;
IF top (16    + SRC) bytes of stack are not in canonical address space (OperandSize =32)
or top (8    + SRC) bytes of stack are not in canonical address space (OperandSize =16)
or top (32 + SRC) bytes of stack are not in canonical address space (OperandSize = 64)
    THEN #SS(0); FI;
Read return stack segment selector;
IF stack segment selector is NULL
    THEN
        IF new CS descriptor L-bit = 0
            THEN #GP(selector);
        IF stack segment selector RPL = 3
            THEN #GP(selector);
FI;
IF return stack segment descriptor is not within descriptor table limits
        THEN #GP(selector); FI;
IF return stack segment descriptor is in non-canonical address space
        THEN #GP(selector); FI;
Read segment descriptor pointed to by return segment selector;
IF stack segment selector RPL != RPL of the return code segment selector
or stack segment is not a writable data segment
or stack segment descriptor DPL != RPL of the return code segment selector
    THEN #GP(selector); FI;
IF stack segment not present
    THEN #SS(StackSegmentSelector); FI;
IF the return instruction pointer is not within the return code segment limit
    THEN #GP(0); FI:
IF the return instruction pointer is not within canonical address space
    THEN #GP(0); FI;

IF OperandSize = 32
    THEN
        EIP ← Pop();
        CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded, segment descriptor loaded *)
        CS(RPL) ← CPL;
        IF instruction has immediate operand
            THEN (* Release parameters from called procedure's stack *)
```

```
                IF StackAddressSize = 32
                    THEN
                        ESP ← ESP    + SRC;
                    ELSE
                        IF StackAddressSize = 16
                            THEN
                                SP ← SP    + SRC;
                            ELSE (* StackAddressSize = 64 *)
                                RSP ← RSP    + SRC;
                        FI;
                FI;
        FI;
        tempESP ← Pop();
        tempSS ← Pop(); (* 32-bit pop, high-order 16 bits discarded, segment descriptor loaded *)
    ELSE
        IF OperandSize = 16
            THEN
                EIP ← Pop();
                EIP ← EIP AND 0000FFFFH;
                CS ← Pop(); (* 16-bit pop; segment descriptor loaded *)
                CS(RPL) ← CPL;
                IF instruction has immediate operand
                    THEN (* Release parameters from called procedure's stack *)
                        IF StackAddressSize = 32
                            THEN
                                ESP ← ESP    + SRC;
                            ELSE
                                IF StackAddressSize = 16
                                    THEN
                                        SP ← SP    + SRC;
                                    ELSE (* StackAddressSize = 64 *)
                                        RSP ← RSP    + SRC;
                                FI;
                        FI;
                FI;
                tempESP ← Pop();
                tempSS ← Pop(); (* 16-bit pop; segment descriptor loaded *)
            ELSE (* OperandSize = 64 *)
                RIP ← Pop();
                CS ← Pop();  (* 64-bit pop; high-order 48 bits discarded; seg. descriptor loaded *)
                CS(RPL) ← CPL;
                IF instruction has immediate operand
                    THEN (* Release parameters from called procedure's stack *)
                        RSP ← RSP    + SRC;
                FI;
                tempESP ← Pop();
                tempSS ←Pop(); (* 64-bit pop; high-order 48 bits discarded; seg. desc. loaded *)
        FI;
FI;
IF ShadowStackEnabled(CPL)
   (* check if 8 byte aligned *)
```

```
    IF SSP AND 0x7 != 0
        THEN #CP(FAR-RET/IRET); FI;
    IF ReturnCodeSegmentSelector(RPL) !=3
        THEN
            tempSsCS = shadow_stack_load 8 bytes from SSP+16;
            tempSsLIP = shadow_stack_load 8 bytes from SSP+8;
            tempSSP = shadow_stack_load 8 bytes from SSP;
            SSP = SSP + 24;
            (* Do 64 bit compare to detect bits beyond 15 being set *)
            tempCS = CS; (* zero padded to 64 bit *)
            IF tempCS != tempSsCS
                THEN #CP(FAR-RET/IRET); FI;
            (* Do 64 bit compare; pad CSBASE+RIP with 0 for 32 bit LIP *)
            IF CSBASE + RIP != tempSsLIP
                THEN #CP(FAR-RET/IRET); FI;
            (* check if 4 byte aligned *)
            IF tempSSP AND 0x3 != 0
                THEN #CP(FAR-RET/IRET); FI;
    FI;
FI;

tempOldCPL = CPL;
CPL ← ReturnCodeSegmentSelector(RPL);
(* update SS:ESP after CPL broadcast complete *)
ESP ← tempESP;
SS ← tempSS;
tempOldSSP = SSP;
IF ShadowStackEnabled(CPL)
    IF CPL = 3
        THEN tempSSP ← IA32_PL3_SSP; FI;
    IF ((EFER.LMA AND CS.L) = 0 AND tempSSP[63:32] != 0)
        THEN #GP(0); FI;
    SSP ← tempSSP
FI;
(* Now past all faulting points; safe to free the token. The token free is done using the old SSP
 * and using a supervisor override as old CPL was a supervisor privilege level *)
IF ShadowStackEnabled(tempOldCPL)
    Atomic Start
        SSPToken ← Load 8 bytes with shadow stack semantics and supervisor override from tempOldSSP
        invalidToken ← 0
        IF ((SSPToken AND 0x01) = 0) (* If busy bit not set then invalid token*)
            THEN invalidToken ← 1; FI;
        IF ((SSPToken AND 0xFFFFFFFFFFFFFFFE) != tempOldSSP) (* If current SSP does not match token *)
            THEN invalidToken ← 1; FI;
        (* Valid token found; clear its busy bit *)
        IF invalidToken = 0
            THEN SSPToken ← SSPToken XOR 0x01;
            Store 8 bytes of SSPToken with shadow stack semantics and supervisor override to tempOldSSP;
        Atomic End
FI;
FOR each of segment register (ES, FS, GS, and DS)
    DO
        IF segment register points to data or non-conforming code segment
        and CPL > segment descriptor DPL; (* DPL in hidden part of segment register *)
            THEN SegmentSelector ← 0; (* SegmentSelector invalid *)
        FI;
    OD;

IF instruction has immediate operand
    THEN (* Release parameters from calling procedure's stack *)
        IF StackAddressSize = 32
            THEN
                ESP ← ESP    + SRC;
            ELSE
                IF StackAddressSize = 16
                    THEN
                        SP ← SP    + SRC;
                    ELSE (* StackAddressSize = 64 *)
                        RSP ← RSP    + SRC;
                FI;
        FI;
FI;
END;
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If the return code or stack segment selector NULL.
	If the return instruction pointer is not within the return code segment limit.
	If returning to 32-bit or compatibility mode and the previous SSP from shadow stack (when returning to CPL <3) or from IA32_PL3_SSP (returning to CPL 3) is beyond 4G.
#GP(selector)	If the RPL of the return code segment selector is less than the CPL.
	If the return code or stack segment selector index is not within its descriptor table limits.
	If the return code segment descriptor does not indicate a code segment.
	If the return code segment is non-conforming and the segment selector's DPL is not equal to the RPL of the code segment's segment selector.
	If the return code segment is conforming and the segment selector's DPL greater than the RPL of the code segment's segment selector.
	If the stack segment is not a writable data segment.
	If the stack segment selector RPL is not equal to the RPL of the return code segment selector.
	If the stack segment descriptor DPL is not equal to the RPL of the return code segment selector.
#SS(0)	If the top bytes of stack are not within stack limits.
	If the return stack segment is not present.

#NP(selector)	If the return code segment is not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory access occurs when the CPL is 3 and alignment checking is enabled.
#CP(FAR-RET/IRET)	If the previous SSP from shadow stack (when returning to CPL <3) or from IA32_PL3_SSP (returning to CPL 3) is not 4 byte aligned.
	If return instruction pointer from stack and shadow stack do not match.

Real-Address Mode Exceptions

#GP	If the return instruction pointer is not within the return code segment limit.
#SS	If the top bytes of stack are not within stack limits.

Virtual-8086 Mode Exceptions

#GP(0)	If the return instruction pointer is not within the return code segment limit.
#SS(0)	If the top bytes of stack are not within stack limits.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory access occurs when alignment checking is enabled.

Compatibility Mode Exceptions

Same as 64-bit mode exceptions.

64-Bit Mode Exceptions

#GP(0)	If the return instruction pointer is non-canonical.
	If the return instruction pointer is not within the return code segment limit.
	If the stack segment selector is NULL going back to compatibility mode.
	If the stack segment selector is NULL going back to CPL3 64-bit mode.
	If a NULL stack segment selector RPL is not equal to CPL going back to non-CPL3 64-bit mode.
	If the return code segment selector is NULL.
	If returning to 32-bit or compatibility mode and the previous SSP from shadow stack (when returning to CPL <3) or from IA32_PL3_SSP (returning to CPL 3) is beyond 4G.
#GP(selector)	If the proposed segment descriptor for a code segment does not indicate it is a code segment.
	If the proposed new code segment descriptor has both the D-bit and L-bit set.
	If the DPL for a nonconforming-code segment is not equal to the RPL of the code segment selector.
	If CPL is greater than the RPL of the code segment selector.
	If the DPL of a conforming-code segment is greater than the return code segment selector RPL.
	If a segment selector index is outside its descriptor table limits.
	If a segment descriptor memory address is non-canonical.
	If the stack segment is not a writable data segment.
	If the stack segment descriptor DPL is not equal to the RPL of the return code segment selector.

	If the stack segment selector RPL is not equal to the RPL of the return code segment selector.
#SS(0)	If an attempt to pop a value off the stack violates the SS limit.
	If an attempt to pop a value off the stack causes a non-canonical address to be referenced.
#NP(selector)	If the return code or stack segment is not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#CP(FAR-RET/IRET)	If the previous SSP from shadow stack (when returning to CPL <3) or from IA32_PL3_SSP (returning to CPL 3) is not 4 byte aligned.
	If return instruction pointer from stack and shadow stack do not match.

4.5 SYSCALL—Fast System Call

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 05	SYSCALL	NP	Valid	Invalid	Fast call to privilege level 0 system procedures.

Instruction Operand Encoding

Op/ En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

SYSCALL invokes an OS system-call handler at privilege level 0. It does so by loading RIP from the IA32_LSTAR MSR (after saving the address of the instruction following SYSCALL into RCX). (The WRMSR instruction ensures that the IA32_LSTAR MSR always contain a canonical address.)

SYSCALL also saves RFLAGS into R11 and then masks RFLAGS using the IA32_FMASK MSR (MSR address C0000084H); specifically, the processor clears in RFLAGS every bit corresponding to a bit that is set in the IA32_FMASK MSR.

SYSCALL loads the CS and SS selectors with values derived from bits 47:32 of the IA32_STAR MSR. However, the CS and SS descriptor caches are **not** loaded from the descriptors (in GDT or LDT) referenced by those selectors. Instead, the descriptor caches are loaded with fixed values. See the Operation section for details. It is the responsibility of OS software to ensure that the descriptors (in GDT or LDT) referenced by those selector values correspond to the fixed values loaded into the descriptor caches; the SYSCALL instruction does not ensure this correspondence.

The SYSCALL instruction does not save the stack pointer (RSP). If the OS system-call handler will change the stack pointer, it is the responsibility of software to save the previous value of the stack pointer. This might be done prior to executing SYSCALL, with software restoring the stack pointer with the instruction following SYSCALL (which will be executed after SYSRET). Alternatively, the OS system-call handler may save the stack pointer and restore it before executing SYSRET.

When shadow stacks are enabled at a privilege level where the SYSCALL instruction is invoked, the SSP is saved to the IA32_PL3_SSP MSR. If shadow stacks are enabled at privilege level 0, the SSP is loaded with 0.

Operation

```
IF (CS.L != 1 ) or (IA32_EFER.LMA != 1) or (IA32_EFER.SCE != 1)
(* Not in 64-Bit Mode or SYSCALL/SYSRET not enabled in IA32_EFER *)
    THEN #UD;
FI;

RCX ← RIP;                          (* Will contain address of next instruction *)
RIP ← IA32_LSTAR;
R11 ← RFLAGS;
RFLAGS ← RFLAGS AND NOT(IA32_FMASK);

CS.Selector ← IA32_STAR[47:32] AND FFFCH          (* Operating system provides CS; RPL forced to 0 *)
```

```
(* Set rest of CS to a fixed value *)
CS.Base ← 0;                                   (* Flat segment *)
CS.Limit ← FFFFFH;                             (* With 4-KByte granularity, implies a 4-GByte limit *)
CS.Type ← 11;                                  (* Execute/read code, accessed *)
CS.S ← 1;
CS.DPL ← 0;
CS.P ← 1;
CS.L ← 1;                                  (* Entry is to 64-bit mode *)
CS.D ← 0;                                  (* Required if CS.L = 1 *)
CS.G ← 1;                                  (* 4-KByte granularity *)

IF ShadowStackEnabled(CPL)
    IA32_PL3_SSP ← SSP;   (* With shadow stacks enabled the system call is supported from Ring 3 to Ring 0 *)
                              (* OS supporting Ring 0 to Ring 0 system calls or Ring 1/2 to ring 0 system call *)
                              (* Must preserve the contents of IA32_PL3_SSP to avoid losing ring 3 state *)
FI;

CPL ← 0;

IF ShadowStackEnabled(CPL)
      SSP ← 0;
FI;
IF EndbranchEnabled(CPL)
    IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
    IA32_S_CET.SUPPRESS = 0
FI;


SS.Selector ← IA32_STAR[47:32]    + 8;                (* SS just above CS *)
(* Set rest of SS to a fixed value *)
SS.Base ← 0;                                   (* Flat segment *)
SS.Limit ← FFFFFH;                                 (* With 4-KByte granularity, implies a 4-GByte limit *)
SS.Type ← 3;                                   (* Read/write data, accessed *)
SS.S ← 1;
SS.DPL ← 0;
SS.P ← 1;
SS.B ← 1;                                  (* 32-bit stack segment *)
SS.G ← 1;                                  (* 4-KByte granularity *)
```

Flags Affected

All.

Protected Mode Exceptions

#UD	The SYSCALL instruction is not recognized in protected mode.

Real-Address Mode Exceptions

#UD	The SYSCALL instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD	The SYSCALL instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD	The SYSCALL instruction is not recognized in compatibility mode.

64-Bit Mode Exceptions

#UD	If IA32_EFER.SCE = 0.
	If the LOCK prefix is used.

4.6 SYSENTER—Fast System Call

Opcode	Instruction	Op/ En	64- Bit Mode	Compat/ Leg Mode	Description
0F 34	SYSENTER	NP	Valid	Valid	Fast call to privilege level 0 system procedures.

Instruction Operand Encoding

Op/ En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Executes a fast call to a level 0 system procedure or routine. SYSENTER is a companion instruction to SYSEXIT. The instruction is optimized to provide the maximum performance for system calls from user code running at privilege level 3 to operating system or executive procedures running at privilege level 0.

When executed in IA-32e mode, the SYSENTER instruction transitions the logical processor to 64-bit mode; otherwise, the logical processor remains in protected mode.

Prior to executing the SYSENTER instruction, software must specify the privilege level 0 code segment and code entry point, and the privilege level 0 stack segment and stack pointer by writing values to the following MSRs:

- **IA32_SYSENTER_CS** (MSR address 174H) — The lower 16 bits of this MSR are the segment selector for the privilege level 0 code segment. This value is also used to determine the segment selector of the privilege level 0 stack segment (see the Operation section). This value cannot indicate a null selector.
- **IA32_SYSENTER_EIP** (MSR address 176H) — The value of this MSR is loaded into RIP (thus, this value references the first instruction of the selected operating procedure or routine). In protected mode, only bits 31:0 are loaded.
- **IA32_SYSENTER_ESP** (MSR address 175H) — The value of this MSR is loaded into RSP (thus, this value contains the stack pointer for the privilege level 0 stack). This value cannot represent a non-canonical address. In protected mode, only bits 31:0 are loaded.

These MSRs can be read from and written to using RDMSR/WRMSR. The WRMSR instruction ensures that the IA32_SYSENTER_EIP and IA32_SYSENTER_ESP MSRs always contain canonical addresses.

While SYSENTER loads the CS and SS selectors with values derived from the IA32_SYSENTER_CS MSR, the CS and SS descriptor caches are **not** loaded from the descriptors (in GDT or LDT) referenced by those selectors. Instead, the descriptor caches are loaded with fixed values. See the Operation section for details. It is the responsibility of OS software to ensure that the descriptors (in GDT or LDT) referenced by those selector values correspond to the fixed values loaded into the descriptor caches; the SYSENTER instruction does not ensure this correspondence.

The SYSENTER instruction can be invoked from all operating modes except real-address mode.

The SYSENTER and SYSEXIT instructions are companion instructions, but they do not constitute a call/return pair. When executing a SYSENTER instruction, the processor does not save state information for the user code (e.g., the instruction pointer), and neither the SYSENTER nor the SYSEXIT instruction supports passing parameters on the stack.

To use the SYSENTER and SYSEXIT instructions as companion instructions for transitions between privilege level 3 code and privilege level 0 operating system procedures, the following conventions must be followed.

- The segment descriptors for the privilege level 0 code and stack segments and for the privilege level 3 code and stack segments must be contiguous in a descriptor table. This convention allows the processor to compute the segment selectors from the value entered in the SYSENTER_CS_MSR MSR.
- The fast system call "stub" routines executed by user code (typically in shared libraries or DLLs) must save the required return IP and processor state information if a return to the calling procedure is required. Likewise, the operating system or executive procedures called with SYSENTER instructions must have access to and use this saved return and state information when returning to the user code.

The SYSENTER and SYSEXIT instructions were introduced into the IA-32 architecture in the Pentium II processor. The availability of these instructions on a processor is indicated with the SYSENTER/SYSEXIT present (SEP) feature flag returned to the EDX register by the CPUID instruction. An operating system that qualifies the SEP flag must also qualify the processor family and model to ensure that the SYSENTER/SYSEXIT instructions are actually present. For example:

```
IF CPUID SEP bit is set
    THEN IF (Family = 6) and (Model < 3) and (Stepping < 3)
        THEN
            SYSENTER/SYSEXIT_Not_Supported; FI;
        ELSE
            SYSENTER/SYSEXIT_Supported; FI;
FI;
```

When the CPUID instruction is executed on the Pentium Pro processor (model 1), the processor returns the SEP flag as set, but does not support the SYSENTER/SYSEXIT instructions.

When shadow stacks are enabled at privilege level where SYSENTER instruction is invoked, the SSP is saved to the IA32_PL3_SSP MSR. If shadow stacks are enabled at privilege level 0, the SSP is loaded with 0.

Operation

```
IF CR0.PE = 0 OR IA32_SYSENTER_CS[15:2] = 0 THEN #GP(0); FI;

RFLAGS.VM ← 0;                                  (* Ensures protected mode execution *)
RFLAGS.IF ← 0;                              (* Mask interrupts *)
IF in IA-32e mode
    THEN
        RSP ← IA32_SYSENTER_ESP;
        RIP ← IA32_SYSENTER_EIP;
ELSE
        ESP ← IA32_SYSENTER_ESP[31:0];
        EIP ← IA32_SYSENTER_EIP[31:0];
FI;

CS.Selector ← IA32_SYSENTER_CS[15:0] AND FFFCH;
                                    (* Operating system provides CS; RPL forced to 0 *)
(* Set rest of CS to a fixed value *)
CS.Base ← 0;                        (* Flat segment *)
CS.Limit ← FFFFFH;                  (* With 4-KByte granularity, implies a 4-GByte limit *)
CS.Type ← 11;                       (* Execute/read code, accessed *)
CS.S ← 1;
CS.DPL ← 0;
CS.P ← 1;
IF in IA-32e mode
    THEN
        CS.L ← 1;                           (* Entry is to 64-bit mode *)
        CS.D ← 0;                           (* Required if CS.L = 1 *)
    ELSE
        CS.L ← 0;
        CS.D ← 1;                              (* 32-bit code segment*)
FI;
CS.G ← 1;                                      (* 4-KByte granularity *)

IF ShadowStackEnabled(CPL)
       IA32_PL3_SSP ← SSP;
FI;

CPL ← 0;

IF ShadowStackEnabled(CPL)
       SSP ← 0;
FI;
IF EndbranchEnabled(CPL)
    IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
    IA32_S_CET.SUPPRESS = 0
FI;

SS.Selector ← CS.Selector    + 8;                (* SS just above CS *)
(* Set rest of SS to a fixed value *)
SS.Base ← 0;                                     (* Flat segment *)
SS.Limit ← FFFFFH;                               (* With 4-KByte granularity, implies a 4-GByte limit *)
SS.Type ← 3;                                     (* Read/write data, accessed *)
SS.S ← 1;
SS.DPL ← 0;
SS.P ← 1;
SS.B ← 1;                                         (* 32-bit stack segment*)
SS.G ← 1;                                         (* 4-KByte granularity *)
```

Flags Affected

VM, IF (see Operation above)

Protected Mode Exceptions

#GP(0)	If IA32_SYSENTER_CS[15:2] = 0.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	The SYSENTER instruction is not recognized in real-address mode.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in protected mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

4.7 SYSEXIT—Fast Return from Fast System Call

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 35	SYSEXIT	NP	Valid	Valid	Fast return to privilege level 3 user code.
REX.W + 0F 35	SYSEXIT	NP	Valid	Valid	Fast return to 64-bit mode privilege level 3 user code.

Instruction Operand Encoding

Op/ En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Executes a fast return to privilege level 3 user code. SYSEXIT is a companion instruction to the SYSENTER instruction. The instruction is optimized to provide the maximum performance for returns from system procedures executing at protections levels 0 to user procedures executing at protection level 3. It must be executed from code executing at privilege level 0.

With a 64-bit operand size, SYSEXIT remains in 64-bit mode; otherwise, it either enters compatibility mode (if the logical processor is in IA-32e mode) or remains in protected mode (if it is not).

Prior to executing SYSEXIT, software must specify the privilege level 3 code segment and code entry point, and the privilege level 3 stack segment and stack pointer by writing values into the following MSR and general-purpose registers:

- **IA32_SYSENTER_CS** (MSR address 174H) — Contains a 32-bit value that is used to determine the segment selectors for the privilege level 3 code and stack segments (see the Operation section)
- **RDX** — The canonical address in this register is loaded into RIP (thus, this value references the first instruction to be executed in the user code). If the return is not to 64-bit mode, only bits 31:0 are loaded.
- **ECX** — The canonical address in this register is loaded into RSP (thus, this value contains the stack pointer for the privilege level 3 stack). If the return is not to 64-bit mode, only bits 31:0 are loaded.

The IA32_SYSENTER_CS MSR can be read from and written to using RDMSR and WRMSR.

While SYSEXIT loads the CS and SS selectors with values derived from the IA32_SYSENTER_CS MSR, the CS and SS descriptor caches are **not** loaded from the descriptors (in GDT or LDT) referenced by those selectors. Instead, the descriptor caches are loaded with fixed values. See the Operation section for details. It is the responsibility of OS software to ensure that the descriptors (in GDT or LDT) referenced by those selector values correspond to the fixed values loaded into the descriptor caches; the SYSEXIT instruction does not ensure this correspondence.

The SYSEXIT instruction can be invoked from all operating modes except real-address mode and virtual-8086 mode.

The SYSENTER and SYSEXIT instructions were introduced into the IA-32 architecture in the Pentium II processor. The availability of these instructions on a processor is indicated with the SYSENTER/SYSEXIT present (SEP) feature flag returned to the EDX register by the CPUID instruction. An operating system that qualifies the SEP flag must also qualify the processor family and model to ensure that the SYSENTER/SYSEXIT instructions are actually present. For example:

```
IF CPUID SEP bit is set
    THEN IF (Family = 6) and (Model < 3) and (Stepping < 3)
        THEN
            SYSENTER/SYSEXIT_Not_Supported; FI;
        ELSE
            SYSENTER/SYSEXIT_Supported; FI;
FI;
```

When the CPUID instruction is executed on the Pentium Pro processor (model 1), the processor returns the SEP flag as set, but does not support the SYSENTER/SYSEXIT instructions.

When shadow stacks are enabled at privilege level 3 the instruction loads SSP with value from IA32_PL3_SSP MSR.

Operation

```
IF IA32_SYSENTER_CS[15:2] = 0 OR CR0.PE = 0 OR CPL != 0 THEN #GP(0); FI;

IF operand size is 64-bit
    THEN        (* Return to 64-bit mode *)
        RSP ← RCX;
        RIP ← RDX;
    ELSE        (* Return to protected mode or compatibility mode *)
        RSP ← ECX;
        RIP ← EDX;
FI;

IF operand size is 64-bit                          (* Operating system provides CS; RPL forced to 3 *)
    THEN CS.Selector ← IA32_SYSENTER_CS[15:0] + 32;
    ELSE CS.Selector ← IA32_SYSENTER_CS[15:0] + 16;
FI;
CS.Selector ← CS.Selector OR 3;                    (* RPL forced to 3 *)
(* Set rest of CS to a fixed value *)
CS.Base ← 0;                                   (* Flat segment *)
CS.Limit ← FFFFFH;                             (* With 4-KByte granularity, implies a 4-GByte limit *)
CS.Type ← 11;                                  (* Execute/read code, accessed *)
CS.S ← 1;
CS.DPL ← 3;
CS.P ← 1;
IF operand size is 64-bit
    THEN        (* return to 64-bit mode *)
        CS.L ← 1;                          (* 64-bit code segment *)
        CS.D ← 0;                          (* Required if CS.L = 1 *)
    ELSE                                           (* return to protected mode or compatibility mode *)
        CS.L ← 0;
        CS.D ← 1;                          (* 32-bit code segment*)
FI;
CS.G ← 1;                                  (* 4-KByte granularity *)
CPL ← 3;
```

```
IF ShadowStackEnabled(CPL)
    SSP ← IA32_PL3_SSP;
FI;SS.Selector ← CS.Selector + 8;           (* SS just above CS *)
(* Set rest of SS to a fixed value *)
SS.Base ← 0;                                   (* Flat segment *)
SS.Limit ← FFFFFH;                             (* With 4-KByte granularity, implies a 4-GByte limit *)
SS.Type ← 3;                                   (* Read/write data, accessed *)
SS.S ← 1;
SS.DPL ← 3;
SS.P ← 1;
SS.B ← 1;                                   (* 32-bit stack segment*)
SS.G ← 1;                                   (* 4-KByte granularity *)
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If IA32_SYSENTER_CS[15:2] = 0.
	If CPL != 0.
#UD	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	The SYSEXIT instruction is not recognized in real-address mode.
#UD	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

#GP(0)	The SYSEXIT instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If IA32_SYSENTER_CS = 0.
	If CPL != 0.
	If RCX or RDX contains a non-canonical address.
#UD	If the LOCK prefix is used.

4.8 SYSRET—Return From Fast System Call

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 07	SYSRET	NP	Valid	Invalid	Return to compatibility mode from fast system call.
REX.W + 0F 07	SYSRET	NP	Valid	Invalid	Return to 64-bit mode from fast system call.

Instruction Operand Encoding

Op/ En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

SYSRET is a companion instruction to the SYSCALL instruction. It returns from an OS system-call handler to user code at privilege level 3. It does so by loading RIP from RCX and loading RFLAGS from R11.[1] With a 64-bit operand size, SYSRET remains in 64-bit mode; otherwise, it enters compatibility mode and only the low 32 bits of the registers are loaded.

SYSRET loads the CS and SS selectors with values derived from bits 63:48 of the IA32_STAR MSR. However, the CS and SS descriptor caches are **not** loaded from the descriptors (in GDT or LDT) referenced by those selectors. Instead, the descriptor caches are loaded with fixed values. See the Operation section for details. It is the responsibility of OS software to ensure that the descriptors (in GDT or LDT) referenced by those selector values correspond to the fixed values loaded into the descriptor caches; the SYSRET instruction does not ensure this correspondence.

The SYSRET instruction does not modify the stack pointer (ESP or RSP). For that reason, it is necessary for software to switch to the user stack. The OS may load the user stack pointer (if it was saved after SYSCALL) before executing SYSRET; alternatively, user code may load the stack pointer (if it was saved before SYSCALL) after receiving control from SYSRET.

If the OS loads the stack pointer before executing SYSRET, it must ensure that the handler of any interrupt or exception delivered between restoring the stack pointer and successful execution of SYSRET is not invoked with the user stack. It can do so using approaches such as the following.

- External interrupts. The OS can prevent an external interrupt from being delivered by clearing EFLAGS.IF before loading the user stack pointer.
- Nonmaskable interrupts (NMIs). The OS can ensure that the NMI handler is invoked with the correct stack by using the interrupt stack table (IST) mechanism for gate 2 (NMI) in the IDT (see Section 6.14.5, “Interrupt Stack Table,” in *Intel® 64 and IA-32 Architectures Software Developer’s Manual, Volume 3A*).

[1]Regardless of the value of R11, the RF and VM flags are always 0 in RFLAGS after execution of SYSRET. In addition, all reserved bits in RFLAGS retain the fixed values.

- General-protection exceptions (#GP). The SYSRET instruction generates #GP(0) if the value of RCX is not canonical. The OS can address this possibility using one or more of the following approaches.

— Confirming that the value of RCX is canonical before executing SYSRET.

— Using paging to ensure that the SYSCALL instruction will never save a non-canonical value into RCX.

— Using the IST mechanism for gate 13 (#GP) in the IDT.

When shadow stacks are enabled at privilege level 3 the instruction loads SSP with value from IA32_PL3_SSP MSR.

Operation

```
IF (CS.L != 1 ) or (IA32_EFER.LMA != 1) or (IA32_EFER.SCE != 1)
(* Not in 64-Bit Mode or SYSCALL/SYSRET not enabled in IA32_EFER *)
    THEN #UD; FI;
IF (CPL != 0) OR (RCX is not canonical) THEN #GP(0); FI;

IF (operand size is 64-bit)
    THEN (* Return to 64-Bit Mode *)
        RIP ← RCX;
    ELSE (* Return to Compatibility Mode *)
        RIP ← ECX;
FI;
RFLAGS ← (R11 & 3C7FD7H) | 2;                    (* Clear RF, VM, reserved bits; set bit 2 *)

IF (operand size is 64-bit)
    THEN CS.Selector ← IA32_STAR[63:48]+16;
    ELSE CS.Selector ← IA32_STAR[63:48];
FI;
CS.Selector ← CS.Selector OR 3;                  (* RPL forced to 3 *)
(* Set rest of CS to a fixed value *)
CS.Base ← 0;                                     (* Flat segment *)
CS.Limit ← FFFFFH;                               (* With 4-KByte granularity, implies a 4-GByte limit *)
CS.Type ← 11;                                    (* Execute/read code, accessed *)
CS.S ← 1;
CS.DPL ← 3;
CS.P ← 1;
IF (operand size is 64-bit)
    THEN (* Return to 64-Bit Mode *)
        CS.L ← 1;                            (* 64-bit code segment *)
        CS.D ← 0;                            (* Required if CS.L = 1 *)
    ELSE (* Return to Compatibility Mode *)
        CS.L ← 0;                            (* Compatibility mode *)
        CS.D ← 1;                            (* 32-bit code segment *)
FI;
CS.G ← 1;                                    (* 4-KByte granularity *)
CPL ← 3;
IF ShadowStackEnabled(CPL)
    SSP ← IA32_PL3_SSP;
FI;

SS.Selector ← (IA32_STAR[63:48]+8) OR 3;         (* RPL forced to 3 *)
(* Set rest of SS to a fixed value *)
SS.Base ← 0;                                     (* Flat segment *)
SS.Limit ← FFFFFH;                    (* With 4-KByte granularity, implies a 4-GByte limit *)
SS.Type ← 3;                          (* Read/write data, accessed *)
SS.S ← 1;
SS.DPL ← 3;
SS.P ← 1;
SS.B ← 1;                         (* 32-bit stack segment*)
SS.G ← 1;                         (* 4-KByte granularity *)
```

Flags Affected

All.

Protected Mode Exceptions

#UD	The SYSRET instruction is not recognized in protected mode.

Real-Address Mode Exceptions

#UD	The SYSRET instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD	The SYSRET instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD	The SYSRET instruction is not recognized in compatibility mode.

64-Bit Mode Exceptions

#UD	If IA32_EFER.SCE = 0.
	If the LOCK prefix is used.
#GP(0)	If CPL != 0.
	If RCX contains a non-canonical address.

4.9 IRET/IRETD—Interrupt Return

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
CF	IRET	NP	Valid	Valid	Interrupt return (16-bit operand size).
CF	IRETD	NP	Valid	Valid	Interrupt return (32-bit operand size).
REX.W + CF	IRETQ	NP	Valid	N.E.	Interrupt return (64-bit operand size).

Instruction Operand Encoding

Op/ En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Returns program control from an exception or interrupt handler to a program or procedure that was interrupted by an exception, an external interrupt, or a software-generated interrupt. These instructions are also used to perform a return from a nested task. (A nested task is created when a CALL instruction is used to initiate a task switch or when an interrupt or exception causes a task switch to an interrupt or exception handler.) See the section titled "Task Linking" in Chapter 7 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

IRET and IRETD are mnemonics for the same opcode. The IRETD mnemonic (interrupt return double) is intended for use when returning from an interrupt when using the 32-bit operand size; however, most assemblers use the IRET mnemonic interchangeably for both operand sizes.

In Real-Address Mode, the IRET instruction preforms a far return to the interrupted program or procedure. During this operation, the processor pops the return instruction pointer, return code segment selector, and EFLAGS image from the stack to the EIP, CS, and EFLAGS registers, respectively, and then resumes execution of the interrupted program or procedure.

In Protected Mode, the action of the IRET instruction depends on the settings of the NT (nested task) and VM flags in the EFLAGS register and the VM flag in the EFLAGS image stored on the current stack. Depending on the setting of these flags, the processor performs the following types of interrupt returns.

- Return from virtual-8086 mode.
- Return to virtual-8086 mode.
- Intra-privilege level return.
- Inter-privilege level return.
- Return from nested task (task switch).

If the NT flag (EFLAGS register) is cleared, the IRET instruction performs a far return from the interrupt procedure, without a task switch. The code segment being returned to must be equally or less privileged than the interrupt handler routine (as indicated by the RPL field of the code segment selector popped from the stack).

As with a real-address mode interrupt return, the IRET instruction pops the return instruction pointer, return code segment selector, and EFLAGS image from the stack to the EIP, CS, and EFLAGS registers, respectively, and then resumes execution of the interrupted program or procedure. If the return is to another privilege level, the IRET instruction also pops the stack pointer and SS from the stack, before resuming program execution. If the return is to virtual-8086 mode, the processor also pops the data segment registers from the stack.

If the NT flag is set, the IRET instruction performs a task switch (return) from a nested task (a task called with a CALL instruction, an interrupt, or an exception) back to the calling or interrupted task. The updated state of the task executing the IRET instruction is saved in its TSS. If the task is re-entered later, the code that follows the IRET instruction is executed.

If the NT flag is set and the processor is in IA-32e mode, the IRET instruction causes a general protection exception.

If nonmaskable interrupts (NMIs) are blocked (see Section 6.7.1, "Handling Multiple NMIs" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*), execution of the IRET instruction un-blocks NMIs. This unblocking occurs even if the instruction causes a fault. In such a case, NMIs are unmasked before the exception handler is invoked.

In 64-bit mode, the instruction's default operation size is 32 bits. Use of the REX.W prefix promotes operation to 64 bits (IRETQ). See the summary chart at the beginning of this section for encoding data and limits.

See "Changes to Instruction Behavior in VMX Non-Root Operation" in Chapter 25 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C*, for more information about the behavior of this instruction in VMX non-root operation.

Operation

```
IF PE = 0
   THEN GOTO REAL-ADDRESS-MODE;
ELSIF (IA32_EFER.LMA = 0)
   THEN
       IF (EFLAGS.VM = 1)
           THEN GOTO RETURN-FROM-VIRTUAL-8086-MODE;
           ELSE GOTO PROTECTED-MODE;
       FI;
   ELSE GOTO IA-32e-MODE;
FI;

REAL-ADDRESS-MODE;
   IF OperandSize = 32
       THEN
           EIP ← Pop();
           CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
           tempEFLAGS ← Pop();
           EFLAGS ← (tempEFLAGS AND 257FD5H) OR (EFLAGS AND 1A0000H);
       ELSE (* OperandSize = 16 *)
           EIP ← Pop(); (* 16-bit pop; clear upper 16 bits *)
           CS ← Pop(); (* 16-bit pop *)
           EFLAGS[15:0] ← Pop();
   FI;
   END;

RETURN-FROM-VIRTUAL-8086-MODE:
(* Processor is in virtual-8086 mode when IRET is executed and stays in virtual-8086 mode *)
```

```
    IF IOPL = 3 (* Virtual mode: PE = 1, VM = 1, IOPL = 3 *)
        THEN IF OperandSize = 32
            THEN
                EIP ← Pop();
                CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
                EFLAGS ← Pop();
                (* VM, IOPL, VIP and VIF EFLAG bits not modified by pop *)
                IF EIP not within CS limit
                    THEN #GP(0); FI;
            ELSE (* OperandSize = 16 *)
                EIP ← Pop(); (* 16-bit pop; clear upper 16 bits *)
                CS ← Pop(); (* 16-bit pop *)
                EFLAGS[15:0] ← Pop(); (* IOPL in EFLAGS not modified by pop *)
                IF EIP not within CS limit
                    THEN #GP(0); FI;
            FI;
        ELSE
            #GP(0); (* Trap to virtual-8086 monitor: PE = 1, VM = 1, IOPL    < 3 *)
    FI;
END;

PROTECTED-MODE:
    IF NT = 1
        THEN GOTO TASK-RETURN; (* PE = 1, VM = 0, NT = 1 *)
    FI;
    IF OperandSize = 32
        THEN
            EIP ← Pop();
            CS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
            tempEFLAGS ← Pop();
        ELSE (* OperandSize = 16 *)
            EIP ← Pop(); (* 16-bit pop; clear upper bits *)
            CS ← Pop(); (* 16-bit pop *)
            tempEFLAGS ← Pop(); (* 16-bit pop; clear upper bits *)
    FI;
    IF tempEFLAGS(VM) = 1 and CPL = 0
        THEN    GOTO RETURN-TO-VIRTUAL-8086-MODE;
        ELSE    GOTO PROTECTED-MODE-RETURN;
    FI;

TASK-RETURN: (* PE = 1, VM = 0, NT = 1 *)
    SWITCH-TASKS (without nesting) to TSS specified in link field of current TSS;
    Mark the task just abandoned as NOT BUSY;
    IF EIP is not within CS limit
        THEN #GP(0); FI;
END;

RETURN-TO-VIRTUAL-8086-MODE:
    (* Interrupted procedure was in virtual-8086 mode: PE = 1, CPL=0, VM = 1 in flag image *)
```

```
     (* If shadow stack or indirect branch tracking at CPL3 then #GP(0) *)
     IF CR4.CET AND (IA32_U_CET.ENDBR_EN OR IA32_U_CET.SHSTK_EN)
         THEN #GP(0); FI;
     shadowStackEnabled = ShadowStackEnabled(CPL)
    EFLAGS ← tempEFLAGS;
    ESP ← Pop();
    SS ← Pop(); (* Pop 2 words; throw away high-order word *)
    ES ← Pop(); (* Pop 2 words; throw away high-order word *)
    DS ← Pop(); (* Pop 2 words; throw away high-order word *)
    FS ← Pop(); (* Pop 2 words; throw away high-order word *)
    GS ← Pop(); (* Pop 2 words; throw away high-order word *)
        IF shadowStackEnabled
          (* check if 8 byte aligned *)
          IF SSP AND 0x7 != 0
              THEN #CP(FAR-RET/IRET); FI;
    FI;

     CPL ← 3;
    (* Resume execution in Virtual-8086 mode *)
     tempOldSSP = SSP;
     (* Now past all faulting points; safe to free the token. The token free is done using the old SSP
       * and using a supervisor override as old CPL was a supervisor privilege level *)
    IF shadowStackEnabled
         Atomic Start
             SSPToken ← Load 8 bytes with shadow stack semantics with supervisor override from tempOldSSP
             invalidToken ← 0
            IF ((SSPToken AND 0x01) = 0) (* If busy bit not set then invalid token*)
                THEN invalidToken ←1; FI;
            IF ((SSPToken AND 0xFFFFFFFFFFFFFFFE) != tempOldSSP) (* If current SSP does not match token *)
                THEN invalidToken ←1; FI;
             (* Valid token found; clear its busy bit *)
            IF invalidToken = 0
                THEN SSPToken ← SSPToken XOR 0x01;
             Store 8 bytes of SSPToken with shadow stack semantics with supervisor override to tempOldSSP;
        Atomic End
    FI;

END;

PROTECTED-MODE-RETURN: (* PE = 1 *)
    IF CS(RPL) > CPL
        THEN GOTO RETURN-OUTER-PRIVILEGE-LEVEL;
        ELSE GOTO RETURN-TO-SAME-PRIVILEGE-LEVEL; FI;
END;

RETURN-TO-OUTER-PRIVILEGE-LEVEL:
        EFLAGS (CF, PF, AF, ZF, SF, TF, DF, OF, NT) ← tempEFLAGS;
    IF OperandSize = 32
        THEN EFLAGS(RF, AC, ID) ← tempEFLAGS; FI;
    IF CPL <= IOPL
        THEN EFLAGS(IF) ← tempEFLAGS; FI;
    IF CPL = 0
        THEN
              EFLAGS(IOPL) ← tempEFLAGS;
              IF OperandSize = 32
                THEN EFLAGS(VM, VIF, VIP) ← tempEFLAGS; FI;
              IF OperandSize = 64
                THEN EFLAGS(VIF, VIP) ← tempEFLAGS; FI;
    FI;
     IF OperandSize = 32
          THEN
            tempESP ← Pop();
            tempSS ← Pop(); (* 32-bit pop, high-order 16 bits discarded; segment descriptor loaded *)
          ELSE
               IF OperandSize = 16
                    THEN
                        tempESP ← Pop();
                        tempSS ← Pop(); (* 16-bit pop; segment descriptor loaded *)
                    ELSE (* OperandSize = 64 *)
                       tempRSP ← Pop();
                      tempSS ←Pop(); (* 64-bit pop; high-order 48 bits discarded; seg. desc. loaded *)
               FI;
     FI;

    IF ShadowStackEnabled(CPL)
         (* check if 8 byte aligned *)
         IF SSP AND 0x7 != 0
              THEN #CP(FAR-RET/IRET); FI;
        IF CS(RPL) != 3
            THEN
                tempSsCS = shadow_stack_load 8 bytes from SSP+16;
                tempSsLIP = shadow_stack_load 8 bytes from SSP+8;
                tempSSP = shadow_stack_load 8 bytes from SSP;
                  SSP = SSP + 24;
                 (* Do 64 bit compare to detect bits beyond 15 being set *)
                  tempCS = CS; (* zero padded to 64 bit *)
                 IF tempCS != tempSsCS
                      THEN #CP(FAR-RET/IRET); FI;
                 (* Do 64 bit compare; pad CSBASE+RIP with 0 for 32 bit LIP *)
                 IF CSBASE + RIP != tempSsEIP
                         THEN #CP(FAR-RET/IRET); FI;
                    (* check if 4 byte aligned *)
                IF tempSSP AND 0x3 != 0
                         THEN #CP(FAR-RET/IRET); FI;
          FI;
    FI;
```

```
tempOldCPL = CPL;
CPL ← CS(RPL);
 (* update SS and RSP after CPL broadcast *)
 IF OperandSize = 64
         THEN
              RSP ← tempRSP;
              SS ← tempSS;
         ELSE
              ESP ← tempESP;
              SS ← tempSS;
         FI;
 IF new mode != 64-Bit Mode
      THEN
           IF EIP is not within CS limit
                THEN #GP(0); FI;
      ELSE (* new mode = 64-bit mode *)
           IF RIP is non-canonical
                     THEN #GP(0); FI;
FI;
 tempOldSSP = SSP;

IF ShadowStackEnabled(CPL)
        IF CPL = 3
              THEN tempSSP ← IA32_PL3_SSP; FI;
      IF ((EFER.LMA AND CS.L) = 0 AND tempSSP[63:32] != 0)
             THEN #GP(0); FI;
      SSP ← tempSSP
FI;
 (* Now past all faulting points; safe to free the token. The token free is done using the old SSP
   * and using a supervisor override as old CPL was a supervisor privilege level *)
IF ShadowStackEnabled(tempOldCPL)
       Atomic Start
              SSPToken ← Load 8 bytes with shadow stack semantics with supervisor override from tempOldSSP
              invalidToken ← 0
             IF ((SSPToken AND 0x01) = 0) (* If busy bit not set then invalid token*)
                 THEN invalidToken ←1; FI;
             IF ((SSPToken AND 0xFFFFFFFFFFFFFFFE) != tempOldSSP) (* If current SSP does not match token *)
                 THEN invalidToken ←1; FI;
              (* Valid token found; clear its busy bit *)
             IF invalidToken = 0
                 THEN SSPToken ← SSPToken XOR 0x01;
              Store 8 bytes of SSPToken with shadow stack semantics with supervisor override to tempOldSSP;
      Atomic End
FI;

FOR each SegReg in (ES, FS, GS, and DS)
      DO
```

```
            tempDesc ← descriptor cache for SegReg (* hidden part of segment register *)
            IF tempDesc(DPL) < CPL AND tempDesc(Type) is data or non-conforming code
                THEN (* Segment register invalid *)
                    SegReg ← NULL;
            FI;
    OD;
END;

RETURN-TO-SAME-PRIVILEGE-LEVEL: (* PE = 1, RPL = CPL *)
    IF new mode != 64-Bit Mode
        THEN
            IF EIP is not within CS limit
                THEN #GP(0); FI;
        ELSE (* new mode = 64-bit mode *)
            IF RIP is non-canonical
                    THEN #GP(0); FI;
    FI;
    EFLAGS (CF, PF, AF, ZF, SF, TF, DF, OF, NT) ← tempEFLAGS;
    IF OperandSize = 32 or OperandSize = 64
        THEN EFLAGS(RF, AC, ID) ← tempEFLAGS; FI;
    IF CPL <= IOPL
        THEN EFLAGS(IF) ← tempEFLAGS; FI;
    IF CPL = 0
         THEN (* VM = 0 in flags image *)
             EFLAGS(IOPL) ← tempEFLAGS;
             IF OperandSize = 32 or OperandSize = 64
                 THEN EFLAGS(VIF, VIP) ← tempEFLAGS; FI;
    FI;
    IF ShadowStackEnabled(CPL)
          IF SSP AND 0x7 != 0 (* check if aligned to 8 bytes *)
               THEN #CP(FAR-RET/IRET); FI;
          tempSsCS = shadow_stack_load 8 bytes from SSP+16;
          tempSsLIP = shadow_stack_load 8 bytes from SSP+8;
          tempSSP = shadow_stack_load 8 bytes from SSP;
          SSP = SSP + 24;
          tempCS = CS; (* zero padded to 64 bit *)
          IF tempCS != tempSsCS (* 64 bit compare; CS zero padded to 64 bits *)
               THEN #CP(FAR-RET/IRET); FI;
          IF CSBASE + RIP != tempSsLIP (* 64 bit compare; CSBASE+RIP zero padded to 64 bit for 32 bit LIP *)
               THEN #CP(FAR-RET/IRET); FI;
          IF tempSSP AND 0x3 != 0 (* check if aligned to 4 bytes *)
               THEN #CP(FAR-RET/IRET); FI;
          IF ((EFER.LMA AND CS.L) = 0 AND tempSSP[63:32] != 0)
               THEN #GP(0); FI;
    FI;

    IF ShadowStackEnabled(CPL)
          IF IA32_EFER.LMA = 1
               (* In IA-32e-mode the IRET may be switching stacks if the interrupt/exception was delivered
               * through an IDT with a non-zero IST *)
```

```
                Atomic Start
                    SSPToken ← Load 8 bytes with shadow stack semantics from SSP
                    invalidToken ← 0
                   IF ((SSPToken AND 0x01) = 0) (* If busy bit not set then invalid token*)
                        THEN invalidToken ← 1; FI;
                  IF ((SSPToken AND 0xFFFFFFFFFFFFFFFE) != SSP) (* If current SSP does not match token *)
                        THEN invalidToken ← 1; FI;
                    (* In IA-32e mode for same CPL IRET there is always a stack switch. The below check verifies
                       If the stack switch was to self stack and if so we don't try to free the token on this shadow
                       stack. If the tempSSP was not to same stack then there was a stack switch so do attempt
                       to free the token *)
                    If tempSSP == SSP
                        THEN invalidToken ← 1; FI;
                    (* Valid token found; clear its busy bit *)
                   IF invalidToken = 0
                      THEN SSPToken ← SSPToken XOR 0x01;
                    Store 8 bytes of SSPToken with shadow stack semantics to SSP;
               Atomic End
            FI;
            SSP ← tempSSP
     FI;
    FOR each of segment register (ES, FS, GS, and DS)
        DO
            IF segment register points to data or non-conforming code segment
            and CPL > segment descriptor DPL (* DPL in hidden part of segment register *)
                THEN SegmentSelector ← 0; (* Segment selector invalid *)
            FI;
        OD;
END;

IA-32e-MODE:
    IF NT = 1
        THEN #GP(0);
    ELSE IF OperandSize = 32
        THEN
            EIP ← Pop();
            CS ← Pop();
            tempEFLAGS ← Pop();
        ELSE IF OperandSize = 16
            THEN
                EIP ← Pop(); (* 16-bit pop; clear upper bits *)
                CS ← Pop(); (* 16-bit pop *)
                tempEFLAGS ← Pop(); (* 16-bit pop; clear upper bits *)
            FI;
        ELSE (* OperandSize = 64 *)
            THEN
                    RIP ← Pop();
                    CS ← Pop(); (* 64-bit pop, high-order 48 bits discarded *)
                    tempRFLAGS ← Pop();
    FI;
    IF tempCS.RPL > CPL
        THEN GOTO RETURN-OUTER-PRIVILEGE-LEVEL;
        ELSE
            IF instruction began in 64-Bit Mode
                THEN
                    IF OperandSize = 32
                        THEN
                            ESP ← Pop();
                            SS ← Pop(); (* 32-bit pop, high-order 16 bits discarded *)
                    ELSE IF OperandSize = 16
                        THEN
                            ESP ← Pop(); (* 16-bit pop; clear upper bits *)
                            SS ← Pop(); (* 16-bit pop *)
                        ELSE (* OperandSize = 64 *)
                            RSP ← Pop();
                            SS ← Pop(); (* 64-bit pop, high-order 48 bits discarded *)
                    FI;
            FI;
            GOTO RETURN-TO-SAME-PRIVILEGE-LEVEL; FI;
END;
```

Flags Affected

All the flags and fields in the EFLAGS register are potentially modified, depending on the mode of operation of the processor. If performing a return from a nested task to a previous task, the EFLAGS register will be modified according to the EFLAGS image stored in the previous task's TSS.

Protected Mode Exceptions

#GP(0)	If the return code or stack segment selector is NULL.
	If the return instruction pointer is not within the return code segment limit.
#GP(selector)	If a segment selector index is outside its descriptor table limits.
	If the return code segment selector RPL is less than the CPL.
	If the DPL of a conforming-code segment is greater than the return code segment selector RPL.
	If the DPL for a nonconforming-code segment is not equal to the RPL of the code segment selector.
	If the stack segment descriptor DPL is not equal to the RPL of the return code segment selector.
	If the stack segment is not a writable data segment.
	If the stack segment selector RPL is not equal to the RPL of the return code segment selector.
	If the segment descriptor for a code segment does not indicate it is a code segment.
	If the segment selector for a TSS has its local/global bit set for local.
	If a TSS segment descriptor specifies that the TSS is not busy.
	If a TSS segment descriptor specifies that the TSS is not available.
#SS(0)	If the top bytes of stack are not within stack limits.
#NP(selector)	If the return code or stack segment is not present.

#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference occurs when the CPL is 3 and alignment checking is enabled.
#UD	If the LOCK prefix is used.
#CP(FAR-RET/IRET)	If the previous SSP from shadow stack (when returning to CPL <3) or from IA32_PL3_SSP (returning to CPL 3) is not 4 byte aligned.
	If returning to 32-bit or compatibility mode and the previous SSP from shadow stack (when returning to CPL <3) or from IA32_PL3_SSP (returning to CPL 3) is beyond 4G.
	If return instruction pointer from stack and shadow stack do not match.

Real-Address Mode Exceptions

#GP	If the return instruction pointer is not within the return code segment limit.
#SS	If the top bytes of stack are not within stack limits.

Virtual-8086 Mode Exceptions

#GP(0)	If the return instruction pointer is not within the return code segment limit.
	IF IOPL not equal to 3.
#PF(fault-code)	If a page fault occurs.
#SS(0)	If the top bytes of stack are not within stack limits.
#AC(0)	If an unaligned memory reference occurs and alignment checking is enabled.
#UD	If the LOCK prefix is used.

Compatibility Mode Exceptions

#GP(0)	If EFLAGS.NT[bit 14] = 1.

Other exceptions same as in Protected Mode.

64-Bit Mode Exceptions

#GP(0)	If EFLAGS.NT[bit 14] = 1.
	If the return code segment selector is NULL.
	If the stack segment selector is NULL going back to compatibility mode.
	If the stack segment selector is NULL going back to CPL3 64-bit mode.
	If a NULL stack segment selector RPL is not equal to CPL going back to non-CPL3 64-bit mode.
	If the return instruction pointer is not within the return code segment limit.
	If the return instruction pointer is non-canonical.
#GP(Selector)	If a segment selector index is outside its descriptor table limits.
	If a segment descriptor memory address is non-canonical.
	If the segment descriptor for a code segment does not indicate it is a code segment.
	If the proposed new code segment descriptor has both the D-bit and L-bit set.
	If the DPL for a nonconforming-code segment is not equal to the RPL of the code segment selector.
	If CPL is greater than the RPL of the code segment selector.
	If the DPL of a conforming-code segment is greater than the return code segment selector RPL.
	If the stack segment is not a writable data segment.

	If the stack segment descriptor DPL is not equal to the RPL of the return code segment selector.
	If the stack segment selector RPL is not equal to the RPL of the return code segment selector.
#SS(0)	If an attempt to pop a value off the stack violates the SS limit.
	If an attempt to pop a value off the stack causes a non-canonical address to be referenced.
#NP(selector)	If the return code or stack segment is not present.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If an unaligned memory reference occurs when the CPL is 3 and alignment checking is enabled.
#UD	If the LOCK prefix is used.
#CP(FAR-RET/IRET)	If the previous SSP from shadow stack (when returning to CPL <3) or from IA32_PL3_SSP (returning to CPL 3) is not 4 byte aligned.
	If returning to 32-bit or compatibility mode and the previous SSP from shadow stack (when returning to CPL <3) or from IA32_PL3_SSP (returning to CPL 3) is beyond 4G.
	If return instruction pointer from stack and shadow stack do not match.

5 Task Management Interactions with CET

5.1 32-bit Task-State Segment (TSS)

When shadow stack is enabled, the SSP to be established when the task is dispatched is contained in the TSS.

If shadow stack is enabled, then the 4 bytes SSP of the task is located at offset 104 in the 32 bit TSS and is used by the processor to establish the TSS when a task switch occurs to task associated with this TSS. Note that the processor does not write the SSP of the task initiating the task switch to the TSS of that task, and the SSP of the previous task is pushed on to the shadow stack of the new task.

The SSP of the task should have a token formatted like the supervisor shadow stack token at the address pointed to by the task SSP. This token will be verified and made busy when switching to that shadow stack using a CALL/JMP instruction, and made free when switching out of that task using an IRET.

5.2 Task Switching

The processor transfers execution to another task in one of four cases.

- The current program, task, or procedure executes a JMP or CALL instruction to a TSS descriptor in the GDT.
- The current program, task, or procedure executes a JMP or CALL instruction to a task-gate descriptor in the GDT or the current LDT.
- An interrupt or exception vector points to a task-gate descriptor in the IDT.
- The current task executes an IRET when the NT flag in the EFLAGS register is set.

JMP, CALL, and IRET instructions, as well as interrupts and exceptions, are all mechanisms for redirecting a program. The referencing of a TSS descriptor or a task gate (when calling or jumping to a task) or the state of the NT flag (when executing an IRET instruction) determines whether a task switch occurs.

The processor performs the following operations when switching to a new task.

1. Obtains the TSS segment selector for the new task as the operand of the JMP or CALL instruction, from a task gate, or from the previous task link field (for a task switch initiated with an IRET instruction).
2. Checks that the current (old) task is allowed to switch to the new task. Data-access privilege rules apply to JMP and CALL instructions. The CPL of the current (old) task and the RPL of the segment selector for the new task must be less than or equal to the DPL of the TSS descriptor or task gate being referenced. Exceptions, interrupts (except for interrupts generated by the INT n instruction), and the IRET instruction are permitted to switch tasks regardless of the DPL of the destination task-gate or TSS descriptor. For interrupts generated by the INT n instruction, the DPL is checked.
3. Checks that the TSS descriptor of the new task is marked present and has a valid limit (greater than or equal to 67H). If task switch was initiated by IRET and shadow stacks are enabled at the current CPL, then the SSP must be aligned to 8 bytes else a #TS(current task TSS) fault is generated. If CR4.CET is 1 then the TSS must be a 32 bit TSS and the limit of the new task's TSS must be greater than or equal to 107 bytes, else a #TS(new task TSS) fault is generated.
4. Checks that the new task is available (call, jump, exception, or interrupt) or busy (IRET return).
5. Checks that the current (old) TSS, new TSS, and all segment descriptors used in the task switch are paged into system memory.
6. Saves the state of the current (old) task in the current task's TSS. The processor finds the base address of the current TSS in the task register and then copies the states of the following registers into the current TSS: all the general-purpose registers, segment selectors from the segment registers, the temporarily saved image of the EFLAGS register, and the instruction pointer register (EIP).

7. Loads the task register with the segment selector and descriptor for the new task's TSS.

8. The processor performs following shadow stack actions:

```
Read CS of new task from new task TSS
Read EFLAGS of new task from new task TSS
IF EFLAGS.VM = 1
   THEN
      new task CPL = 3;
   ELSE
      new task CPL = CS.RPL;
FI;
pushCsLipSsp = 0
If task switch was initiated by CALL instruction, exception or interrupt
   If shadow stack enabled at current CPL
      If new task CPL < CPL and current task CPL = 3
           THEN
                IA32_PL3_SSP = SSP    (* user -> supervisor *)
           ELSE
                pushCsLipSsp = 1 (* no privilege change; supv->supv; supv->user *)
                tempSSP = SSP
                tempSsLIP =CSBASE + EIP
                tempSsCS = CS
      FI;
   FI
FI
verifyCsLIP = 0
If task switch was initiated by IRET
   IF shadow stacks enabled at current CPL
      IF (CPL of new Task = CPL of current Task) OR
        (CPL of new Task < 3 AND CPL of current Task < 3) OR
        (CPL or new Task < 3 AND CPL of current task = 3)
           (* no privilege change or supervisor -> supervisor or user -> supervisor IRET *)
           tempSsCS = ShadowStackPop8B()
           tempSsLIP = ShadowStackPop8B()
           tempSSP = ShadowStackPop8B()
           verifyCsLIP = 1
      FI
      // Clear busy flag on current shadow stack
      Atomic Start
           SSPToken ← Load 8 bytes with shadow stack semantics from SSP
           invalidToken ← 0
           IF ((SSPToken AND 0x01) = 0) (* If busy bit not set then invalid token*)
                THEN invalidToken ← 1; FI;
           IF SSP & 0x07 != 0 (* if SSP not aligned to 8 bytes then invalid token *)
              THEN invalidToken ← 1; FI;
          IF ((SSPToken AND 0xFFFFFFFFFFFFFFFE) != SSP) (* If current SSP does not match token *)
              THEN invalidToken ← 1; FI;
           (* Valid token found; clear its busy bit *)
          IF invalidToken = 0
                THEN SSPToken ← SSPToken XOR 0x01; FI;
            Store 8 bytes of SSPToken with shadow stack semantics to SSP;
      Atomic End
        SSP = 0
    FI
FI
```

9. The TSS state is loaded into the processor. This includes the LDTR register, the PDBR (control register CR3), the EFLAGS register, the EIP register, the general-purpose registers, and the segment selectors. A fault during the load of this state may corrupt architectural state. (If paging is not enabled, a PDBR value is read from the new task's TSS, but it is not loaded into CR3.).
10. If the task switch was initiated with a JMP or IRET instruction, the processor clears the busy (B) flag in the current (old) task's TSS descriptor; if initiated with a CALL instruction, an exception, or an interrupt: the busy (B) flag is left set.
11. If the task switch was initiated with an IRET instruction, the processor clears the NT flag in a temporarily saved image of the EFLAGS register; if initiated with a CALL or JMP instruction, an exception, or an interrupt, the NT flag is left unchanged in the saved EFLAGS image.
12. If the task switch was initiated with a CALL instruction, an exception, or an interrupt, the processor will set the NT flag in the EFLAGS loaded from the new task. If initiated with an IRET instruction or JMP instruction, the NT flag will reflect the state of NT in the EFLAGS loaded from the new task.
13. If the task switch was initiated with a CALL instruction, JMP instruction, an exception, or an interrupt, the processor sets the busy (B) flag in the new task's TSS descriptor; if initiated with an IRET instruction, the busy (B) flag is left set.
14. The descriptors associated with the segment selectors are loaded and qualified. Any errors associated with this loading and qualification occur in the context of the new task and may corrupt architectural state.
15. The processor performs following shadow stack actions:

```
IF shadow stack enabled at current CPL OR indirect branch tracking at current CPL
   THEN
      IF EFLAGS.VM = 1
                    THEN #TSS(new-Task-TSS);FI;
   FI;
IF shadow stack enabled at current CPL
    IF task switch initiated by CALL instruction, JMP instruction, interrupt or exception (* switch stack *)
      new_SSP := Load the 4 byte from offset 104 in the TSS
      // Verify new SSP to be legal
      IF new_SSP & 0x07 != 0
         THEN #TSS(New-Task-TSS); FI;
      Fault = 0
      Atomic Start
          SSPToken = 8 bytes loaded with shadow stack semantics from new_SSP
             IF (SSPToken AND 0x01)
                 THEN fault ← 1; FI;
              IF ((EFER.LMA and CS.L) = 0 AND SSPToken[63:32] != 0)
                 THEN fault ← 1; FI;
           IF ((SSPToken AND 0xFFFFFFFFFFFFFFFE) != new_SSP)
               THEN fault ← 1; FI;
         IF fault = 0
                 THEN SSPToken = SSPToken OR 0x01; FI;
           Store 8 bytes of SSPToken with shadow stack semantics to new_SSP;
       Atomic End
       IF fault = 1
           THEN GP(0#TSS(New-Task-TSS); FI;
       SSP = new_SSP
      IF pushCsLipSsp = 1 (* call, int, exception from user->user or supervisor->supervisor or supv -> user *)
          Push tempSsCS, tempSsLip, tempSsSSP on shadow stack using 8B pushes
```

```
            FI
        FI
FI
IF task switch initiated by IRET
        IF verifyCsLIP = 1
                (* do 64 bit comparisons; CS zero padded to 64 bit; CSBASE+EIP zero padded to 64 bit *)
              If tempSsCS and tempSsLIP do not match CS and CSBASE+EIP
                      THEN #CP(FAR-RET/IRET); FI;
          FI
          IF ShadowStackEnabled(CPL)
                THEN
                     IF (verifyCsLIP == 0) tempSSP = IA32_PL3_SSP;
                       IF tempSSP & 0x03 != 0 THEN #CP(FAR-RET/IRET) // verify aligned to 4 bytes
                       IF tempSSP[63:32] != 0 THEN # CP(FAR-RET/IRET)
                     SSP = tempSSP
            FI
FI
IF EndbranchEnabled(CPL)
        IF task switch initiated by CALL instruction, JMP instruction, interrupt or exception
               IF CPL = 3
                   THEN
                          IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH
                            IA32_U_CET.SUPPRESS = 0
                   ELSE
                          IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
                            IA32_S_CET.SUPPRESS = 0
            FI;
       FI;
FI;
```

16. Begins executing the new task. (To an exception handler, the first instruction of the new task appears not to have been executed.)

NOTES

If all checks and saves have been carried out successfully, the processor commits to the task switch. If an unrecoverable error occurs in steps 1 through 8, the processor does not complete the task switch and insures that the processor is returned to its state prior to the execution of the instruction that initiated the task switch.

If an unrecoverable error occurs in step 9, architectural state may be corrupted, but an attempt will be made to handle the error in the prior execution environment. If an unrecoverable error occurs after the commit point (in step 13), the processor completes the task switch (without performing additional access and segment availability checks) and generates the appropriate exception prior to beginning execution of the new task.

If exceptions occur after the commit point, the exception handler must finish the task switch itself before allowing the processor to begin executing the new task. See Chapter 6, "Interrupt 10—Invalid TSS Exception (#TS)," for more information about the effect of exceptions on a task when they occur after the commit point of a task switch.

The state of the currently executing task is always saved when a successful task switch occurs. If the task is resumed, execution starts with the instruction pointed to by the saved EIP value, and the registers are restored to the values they held when the task was suspended.

When switching tasks, the privilege level of the new task does not inherit its privilege level from the suspended task. The new task begins executing at the privilege level specified in the CPL field of the CS register, which is loaded from the TSS. Because tasks are isolated by their separate address spaces and TSSs and because privilege rules control access to a TSS, software does not need to perform explicit privilege checks on a task switch.

The following table, Table 1 Exception Conditions Checked During a Task Switch shows the exception conditions that the processor checks for when switching tasks. It also shows the exception that is generated for each check if an error is detected and the segment that the error code references. (The order of the checks in the table is the order used in the P6 family processors. The exact order is model specific and may be different for other IA-32 processors.) Exception handlers designed to handle these exceptions may be subject to recursive calls if they attempt to reload the segment selector that generated the exception. The cause of the exception (or the first of multiple causes) should be fixed before reloading the selector.

Table 1 Exception Conditions Checked During a Task Switch

Condition Checked	Exception[1]	Error Code Reference[2]
Segment selector for a TSS descriptor references the GDT and is within the limits of the table.	#GP #TS (for IRET)	New Task's TSS
TSS descriptor is present in memory.	#NP	New Task's TSS
TSS descriptor is not busy (for task switch initiated by a call, interrupt, or exception).	#GP (for JMP, CALL, INT)	Task's back-link TSS
TSS descriptor is not busy (for task switch initiated by an IRET instruction).	#TS (for IRET)	New Task's TSS
TSS segment limit greater than or equal to 104 (for 32-bit TSS) or 44 (for 16-bit TSS).	#TS	New Task's TSS
TSS segment limit greater than or equal to 108 (for 32-bit TSS) if CR4.CET = 1.	#TS	New Task's TSS
If shadow stack enabled and SSP not aligned to 8 bytes (for task switch initiated by an IRET instruction).	#TS	Current task TSS
Registers are loaded from the values in the TSS.		
LDT segment selector of new task is valid [3].	#TS	New Task's LDT
Code segment DPL matches segment selector RPL.	#TS	New Code Segment
SS segment selector is valid [2].	#TS	New Stack Segment
Stack segment is present in memory.	#SS	New Stack Segment
Stack segment DPL matches CPL.	#TS	New stack segment

LDT of new task is present in memory.	#TS	New Task's LDT
CS segment selector is valid [3].	#TS	New Code Segment
Code segment is present in memory.	#NP	New Code Segment
Stack segment DPL matches selector RPL.	#TS	New Stack Segment
DS, ES, FS, and GS segment selectors are valid [3].	#TS	New Data Segment
DS, ES, FS, and GS segments are readable.	#TS	New Data Segment
DS, ES, FS, and GS segments are present in memory.	#NP	New Data Segment
DS, ES, FS, and GS segment DPL greater than or equal to CPL (unless these are conforming segments).	#TS	New Data Segment
Shadow Stack Pointer in of task not aligned to 8 bytes (for task switch initiated by a call, interrupt, or exception).	#TS	New Task's TSS
If EFLAGS.VM=1 and shadow stacks are enabled.	#TS	New Task's TSS
Shadow Stack Token verification failures (for task switch initiated by a call, interrupt, jump, or exception): - Busy bit already set. - L bit in token does not match (EFER.LMA & CS.L), i.e. not 0. - Address in Shadow stack token does not match address SSP value from TSS.	#TS	New Task's TSS
If task switch initiated by IRET, CS and LIP stored on old task shadow stack does not match CS and LIP of new task.	#CP	FAR-RET/IRET
If task switch initiated by IRET and SSP of new task loaded from shadow stack of old task (if new task CPL is < 3) OR the SSP from IA32_PL3_SSP (if new task CPL = 3) fails the following checks: - Not aligned to 4 bytes. - Is beyond 4G.	#CP	FAR-RET/IRET

NOTES:

1. #NP is segment-not-present exception, #GP is general-protection exception, #TS is invalid-TSS exception, and #SS is stack-fault exception.
2. The error code contains an index to the segment descriptor referenced in this column.
3. A segment selector is valid if it is in a compatible type of table (GDT or LDT), occupies an address within the table's segment limit, and refers to a compatible type of descriptor (for example, a segment selector in the CS register only is valid when it points to a code-segment descriptor).

6 Shadow Stack Management Instructions

Shadow stack management instructions allow the program and run-time to perform operations like recovering from control protection faults, shadow stack switching, etc. The following instructions are provided.

- **INCSSP** – This instruction is used to increment the shadow stack pointer by the number of frames specified by an 8 bit instruction operand, 'n'. SSP increments by n*4 bytes in 32-bit/compatibility mode and by n*4 or n*8 bytes in 64-bit mode.
- **RDSSP** – instruction used to read the contents of the SSP register into a GPR.
- **SAVEPREVSSP** – this instruction uses a "previous ssp" token at the top of the current shadow stack to find the address to save a "shadow stack restore" token on the previous shadow stack. The "shadow stack restore" token contains the SSP at the time of invoking the RSTORSSP instruction that created the "previous ssp" token along with the mode of the machine. The format of this token is as follows.
  - Bit 63:2 – SSP at the time of invoking the RSTORSSP instruction.
  - Bit 1 – Must be 1.
  - Bit 0 – L flag; if 1, indicates this token was created in 64-bit mode.
- **RSTORSSP** – this instruction is used to restore a shadow stack context previously saved on the shadow stack as a "shadow stack restore" token. This instruction loads the "shadow stack restore" token from the memory operand specified in the instruction, pointing to a valid "shadow stack restore" token. This instruction replaces the "shadow stack restore" token with a "previous ssp" token and establishes the SSP to point to the memory operand of this instruction. Thus following completion of this instruction the "previous ssp" token is at the top of the shadow stack. The format of the "shadow stack restore" token is as follows.
  - Bit 63:2 – SSP at the time of creating this restore point.
  - Bit 1 – Must be 0.
- Bit 0 – L flag; if 1, indicates this token was created in 64-bit mode.
- **WRSS** – This instruction does a write to the shadow stack. This instruction is associated with a control to disable this instruction. WRSS can only write to user shadow stack when invoked at CPL 3 and supervisor shadow stacks when invoked at CPL != 3.
- **WRUSS –** This instruction is similar to WRSS but is a privileged instruction. It can only write to user shadow stacks.
- **SETSSBSY** – This instruction verifies the "supervisor shadow stack" token pointed to by the IA32_PL0_SSP MSR, and if the token is a valid token sets it busy and sets the SSP to the value of the IA32_PL0_SSP MSR.
- **CLRSSBSY** – This instruction takes a memory operand to a "supervisor shadow stack" token, and if the token is a valid token clears its busy bit.

6.1 INCSSP—Increment Shadow Stack Pointer

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F3 0F AE /05	INCSSPD r32	R	Valid	Valid	Increment SSP by 4 * r32[7:0].
F3 REX.W 0F AE /05	INCSSPQ r64	R	Valid	N.E.	Increment SSP by 8 * r64[7:0].

Instruction Operand Encoding

Op/ En	Operand 1	Operand 2	Operand 3	Operand 4
R	ModRM:r/m(r)	NA	NA	NA

Description

This instruction can be used to increment the current shadow stack pointer by operand size of the instruction times the unsigned 8-bit value specified by bits 7:0 in the source operand. The instruction performs a pop and discard of the first and last element on the shadow stack in the range specified by the unsigned 8-bit value in bits 7:0 of the source operand.

Operation

```
IF CPL = 3
    IF (CR4.CET & IA32_U_CET.SH_STK_EN) = 0
        THEN #UD; FI;
ELSE
    IF (CR4.CET & IA32_S_CET.SH_STK_EN) = 0
        THEN #UD; FI;
ENDIF

IF (operand size is 64-bit)
  THEN
      TMP1 = (R64[7:0] == 0) ? 1 : R64[7:0]
        TMP = ShadowStackLoad8B(SSP)
        TMP = ShadowStackLoad8B(SSP + TMP1 * 8 - 8)
      SSP ← SSP + R64[7:0] * 8;
  ELSE
        TMP1 = (R32[7:0] == 0) ? 1 : R32[7:0]
        TMP = ShadowStackLoad4B(SSP)
        TMP = ShadowStackLoad4B(SSP + TMP1 * 4 - 4)
      SSP ← SSP + R32[7:0] * 4;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#UD	If the LOCK prefix is used.
	If CR4.CET = 0.
	IF CPL = 3 and IA32_U_CET.SH_STK_EN = 0.
	IF CPL < 3 and IA32_S_CET.SH_STK_EN = 0.
#PF(fault-code)	If a page fault occurs.

Real-Address Mode Exceptions

#UD	The INCSSP instruction is not recognized in real-address mode.

Virtual-8086 Mode Exceptions

#UD	The INCSSP instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD	If the LOCK prefix is used.
	If CR4.CET = 0.
	IF CPL = 3 and IA32_U_CET.SH_STK_EN = 0.
	IF CPL < 3 and IA32_S_CET.SH_STK_EN = 0.
#PF(fault-code)	If a page fault occurs.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used.
	If CR4.CET = 0.
	IF CPL = 3 and IA32_U_CET.SH_STK_EN = 0.
	IF CPL < 3 and IA32_S_CET.SH_STK_EN = 0.
#PF(fault-code)	If a page fault occurs.

6.2 RDSSP—Read Shadow Stack Pointer

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F3 0F 1E /1 (mod=11)	RDSSPD	R32	Valid	Valid	Read low 32 bits of SSP.
F3 REX.W 0F 1E /1 (mod=11)	RDSSPQ	R64	Valid	N.E.	Read SSP.

Instruction Operand Encoding

Op/ En	Operand 1	Operand 2	Operand 3	Operand 4
R32	ModRM:r/m(w)	NA	NA	NA
R64	ModRM:r/m(w)	NA	NA	NA

Description

Reads the current shadow stack pointer to the register destination. Note this opcode is a NOP when CET is not enabled.

Operation

```
IF CPL = 3
    IF CR4.CET & IA32_U_CET.SH_STK_EN
        IF (operand size is 64 bit)
            THEN
                Dest ← SSP;
            ELSE
                Dest ← SSP[31:0];
        FI;
    ENDIF
ELSE
    IF CR4.CET & IA32_S_CET.SH_STK_EN
        IF (operand size is 64 bit)
            THEN
                Dest ← SSP;
            ELSE
                Dest ← SSP[31:0];
        FI;
    ENDIF
ENDIF
```

Flags Affected

None.

Protected Mode Exceptions

None

Real-Address Mode Exceptions

None.

Virtual-8086 Mode Exceptions

None.

Compatibility Mode Exceptions

None.

64-Bit Mode Exceptions

None.

6.3 SAVEPREVSSP —Save Previous Shadow Stack Pointer

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F3 0F 01 EA (mod=11, /5, RM=010)	SAVEPREV SSP	NP	Valid	Valid	Save previous shadow stack pointer context.

Instruction Operand Encoding

Op/ En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Push the previous SSP and state of (EFER.LMA & CS.L) to the previous shadow stack after aligning to next 8 byte boundary. The previous SSP is obtained from the "previous SSP" token at top of the current shadow stack.

Operation

```
IF CPL = 3
    IF (CR4.CET & IA32_U_CET.SH_STK_EN) = 0
        THEN #UD; FI;
ELSE
    IF (CR4.CET & IA32_S_CET.SH_STK_EN) = 0
        THEN #UD; FI;
ENDIF


IF SSP not aligned to 8 bytes
    THEN #GP(0); FI;
(* Pop the "previous-ssp" token from current shadow stack *)
previous_ssp_token = ShadowStackPop8B(SSP)

(* If the CF flag indicates there was a alignment hole on current shadow stack then pop that alignment hole *)
(* Note that the alignment hole must be zero and can be present only when in legacy/compatibility mode *)
IF RFLAGS.CF == 1 AND (EFER.LMA AND CS.L)
    #GP(0)
ENDIF
IF RFLAGS.CF == 1
    must_be_zero = ShadowStackPop4B(SSP)
    IF must_be_zero != 0 THEN #GP(0)
ENDIF

(* Previous SSP token must have the bit 1 set *)
IF ((previous_ssp_token & 0x02) == 0)
  THEN #GP(0); (* bit 1 was 0 *)

IF ((EFER.LMA AND CS.L) = 0 AND previous_ssp_token [63:32] != 0)
  THEN #GP(0); FI; (* If compatibility/legacy mode and SSP not in 4G *)
```

```
(* Save Prev SSP from previous_ssp_token to the old shadow stack at next 8 byte aligned address *)
old_SSP = previous_ssp_token & ~0x03
temp ← (old_SSP | (EFER.LMA & CS.L));
Shadow_stack_store 4 bytes of 0 to (old_SSP – 4)
old_SSP ← old_SSP & ~0x07;
Shadow_stack_store 8 bytes of temp to (old_SSP – 8)
```

Flags Affected

None.

Protected Mode Exceptions

#UD	If the LOCK prefix is used.
	If CR4.CET = 0.
	IF CPL = 3 and IA32_U_CET.SH_STK_EN = 0.
	IF CPL < 3 and IA32_S_CET.SH_STK_EN = 0.

#GP(0)	If SSP not 8 byte aligned. If alignment hole on shadow stack is not 0. If bit 1 of the “previous ssp” token not set to 1. If in 32-bit/compatibility mode and SSP recorded in “previous ssp” token is beyond 4G.
#PF(fault-code)	If a page fault occurs.

Real-Address Mode Exceptions

#UD	The SAVEPREVSSP instruction is not recognized in virtual-8086 mode.

Virtual-8086 Mode Exceptions

#UD	The SAVEPREVSSP instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD	If the LOCK prefix is used.
	If CR4.CET = 0.
	IF CPL = 3 and IA32_U_CET.SH_STK_EN = 0.
	IF CPL < 3 and IA32_S_CET.SH_STK_EN = 0.

#GP(0)	If SSP not 8 byte aligned If alignment hole on shadow stack is not 0 If bit 1 of the “previous ssp” token not set to 1 If in 32-bit/compatibility mode and SSP recorded in “previous ssp” token is beyond 4G.
#PF(fault-code)	If a page fault occurs.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used.
	If CR4.CET = 0.
	IF CPL = 3 and IA32_U_CET.SH_STK_EN = 0.
	IF CPL < 3 and IA32_S_CET.SH_STK_EN = 0.
#GP(0)	If SSP not 8 byte aligned.
	If carry flag set.
	If bit 1 of the “previous ssp” token not set to 1.
#PF(fault-code)	If a page fault occurs.

6.4 RSTORSSP — Restore saved Shadow Stack Pointer

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F3 0F 01 /5 (mod!=11, /5, memory only)	RSTORSSP	M64	Valid	Valid	Restore SSP.

Instruction Operand Encoding

Op/ En	Operand 1	Operand 2	Operand 3	Operand 4
M64	ModRM:r/m (r, w)	NA	NA	NA

Description

Restore SSP from the "shadow stack restore" token previously created on shadow stack by SAVEPREVSSP and create a "previous ssp" token on the restored shadow stack to allow saving the previous SSP on previous shadow stack.

Operation

```
IF CPL = 3
    IF (CR4.CET & IA32_U_CET.SH_STK_EN) = 0
        THEN #UD; FI;
ELSE
    IF (CR4.CET & IA32_S_CET.SH_STK_EN) = 0
        THEN #UD; FI;
ENDIF

SSP_LA = Linear_Address(mem operand)
IF SSP_LA not aligned to 8 bytes
    THEN #GP(0); FI;

previous_ssp_token = SSP | (EFER.LMA AND CS.L) | 0x02

Atomic Start
SSP_Tmp = Locked shadow_Stack_Load with store intent 8 bytes from SSP_LA
Fault = 0

IF ((SSP_Tmp & 0x03) != (EFER.LMA & CS.L))
    THEN fault = 1; FI; (* If L flag in token does not match EFER.LMA & CS.L or bit 1 is not 0 *)

IF ((EFER.LMA AND CS.L) = 0 AND SSP_Tmp[63:32] != 0)
    THEN fault = 1; FI; (* If compatibility/legacy mode and SSP not in 4G *)

TMP = SSP_Tmp & ~0x01
TMP = (TMP – 8)
TMP = TMP & ~0x07
IF TMP != SSP_LA
        THEN fault = 1; FI; (* If address in token does not match the requested top of stack *)

TMP = (fault == 0) ? previous_ssp_token : SSP_Tmp
Shadow_stack_store 8 bytes of TMP to SSP_LA and release lock
Atomic End

IF fault == 1
        THEN #CP(rstorssp); FI;

SSP = SSP_LA

// Set the CF if the SSP in the restore token was 4 byte aligned i.e. there is an alignment hole
RFLAGS.CF = (SSP_Tmp & 0x04) ? 1 : 0;
RFLAGS.ZF,PF,AF,OF,SF ← 0;
```

Flags Affected

CF is set to indicate if the shadow stack pointer in the restore token was 4 byte aligned else is cleared. ZF, PF, AF, OF and SF are cleared.

Protected Mode Exceptions

#UD	If the LOCK prefix is used.
	If CR4.CET = 0.
	IF CPL = 3 and IA32_U_CET.SH_STK_EN = 0.
	IF CPL < 3 and IA32_S_CET.SH_STK_EN = 0.
#GP(0)	If linear address of memory operand not 8 byte aligned.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If destination is located in a non-writeable segment
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#CP(rstorssp)	If L bit in token does not match (EFER.LMA & CS.L).
	If address in token does not match linear address of memory operand.
	If in 32-bit or compatibility mode and the address in token is not below 4G.
#PF(fault-code)	If a page fault occurs.

Real-Address Mode Exceptions

#UD	The RSTORSSP instruction is not recognized in virtual-8086 mode.

Virtual-8086 Mode Exceptions

#UD	The RSTORSSP instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD	If the LOCK prefix is used.
	If CR4.CET = 0.
	If CPL = 3 and IA32_U_CET.SH_STK_EN = 0.
	If CPL < 3 and IA32_S_CET.SH_STK_EN = 0.

#GP(0)	Same as Protected mode exceptions.
#SS(0)	Same as Protected mode exceptions.
#CP(rstorssp)	Same as Protected mode exceptions.
#PF(fault-code)	If a page fault occurs.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used.
	If CR4.CET = 0.
	If CPL = 3 and IA32_U_CET.SH_STK_EN = 0.
	If CPL < 3 and IA32_S_CET.SH_STK_EN = 0.
#GP(0)	If linear address of memory operand not 8 byte aligned.
	If a memory address is in a non-canonical form.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#CP(rstorssp)	If L bit in token does not match (EFER.LMA & CS.L).
	If address in token does not match linear address of memory operand.
#PF(fault-code)	If a page fault occurs.

6.5 WRSS — Write to shadow stack

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
0F 38 F6	WRSSD	MR	Valid	Valid	Write 4 bytes to shadow stack.
REX.W 0F 38 F6	WRSSQ	MR	Valid	N.E.	Write 8 bytes to shadow stack.

Instruction Operand Encoding

Op/ En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Write bytes in register source to the shadow stack.

Operation

```
IF CPL = 3
    IF (CR4.CET & IA32_U_CET.SH_STK_EN) = 0
        THEN #UD; FI;
    IF (IA32_U_CET.WR_SHSTK_EN) = 0
        THEN #UD; FI;
ELSE
    IF (CR4.CET & IA32_S_CET.SH_STK_EN) = 0
        THEN #UD; FI;
    IF (IA32_S_CET.WR_SHSTK_EN) = 0
        THEN #UD; FI;
ENDIF
DEST_LA = Linear_Address(mem operand)
IF (operand size is 64 bit)
    THEN
        (* Destination not 8B aligned *)
        IF DEST_LA[2:0]
            THEN GP(0); FI;
        Shadow_stack_store 8 bytes of SRC to DEST_LA;
    ELSE
        (* Destination not 4B aligned *)
        IF DEST_LA[1:0]
            THEN GP(0); FI;
        Shadow_stack_store 4 bytes of SRC[31:0] to DEST_LA;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

#UD	If the LOCK prefix is used.
	If CR4.CET = 0.
	IF CPL = 3 and IA32_U_CET.SH_STK_EN = 0.
	IF CPL < 3 and IA32_S_CET.SH_STK_EN = 0.
	IF CPL = 3 and IA32_U_CET.WR_SHSTK_EN = 0.
	IF CPL < 3 and IA32_S_CET.WR_SHSTK_EN = 0.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If destination is located in a non-writeable segment.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
	If linear address of destination is not 4 byte aligned.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs if destination is not a user shadow stack when CPL3 and not a supervisor shadow stack when CPL < 3.
	Other terminal and non-terminal faults.

Real-Address Mode Exceptions

#UD	The WRSS instruction is not recognized in virtual-8086 mode.

Virtual-8086 Mode Exceptions

#UD	The WRSS instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD	If the LOCK prefix is used.
	If CR4.CET = 0.
	IF CPL = 3 and IA32_U_CET.SH_STK_EN = 0.
	IF CPL < 3 and IA32_S_CET.SH_STK_EN = 0.
	IF CPL = 3 and IA32_U_CET.WR_SHSTK_EN = 0.
	IF CPL < 3 and IA32_S_CET.WR_SHSTK_EN = 0.
#PF(fault-code)	If a page fault occurs if destination is not a user shadow stack when CPL3 and not a supervisor shadow stack when CPL < 3.
	Other terminal and non-terminal faults.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used.
	If CR4.CET = 0.
	IF CPL = 3 and IA32_U_CET.SH_STK_EN = 0.
	IF CPL < 3 and IA32_S_CET.SH_STK_EN = 0.
	IF CPL = 3 and IA32_U_CET.WR_SHSTK_EN = 0.
	IF CPL < 3 and IA32_S_CET.WR_SHSTK_EN = 0.
#GP(0)	If a memory address is in a non-canonical form.

	If linear address of destination is not 4 byte aligned.
#PF(fault-code)	If a page fault occurs if destination is not a user shadow stack when CPL3 and not a supervisor shadow stack when CPL < 3.
	Other terminal and non-terminal faults.

6.6 WRUSS — Write to User Shadow Stack

Opcode	Instruction	Op/ En	64- Bit Mode	Compat/ Leg Mode	Description
66 0F 38 F5	WRUSSD	MR	Valid	Valid	Write 4 bytes to shadow stack.
66 REX.W 0F 38 F5	WRUSSQ	MR	Valid	N.E.	Write 8 bytes to shadow stack.

Instruction Operand Encoding

Op/ En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA

Description

Write bytes in register source to a user shadow stack page. This instruction does the store the user shadow stack page. The shadow stack store done by this instruction is with user-access intent and thus paging access control checks will be treated as a user-mode shadow stack store.

Operation

```
IF CR4.CET = 0
    THEN #UD; FI;
IF CPL > 0
    THEN #GP(0); FI;
DEST_LA = Linear_Address(mem operand)
Setup mode to perform next shadow stack store with user-access intent
IF (operand size is 64 bit)
    THEN
        (* Destination not 8B aligned *)
        IF DEST_LA[2:0]
            THEN GP(0); FI;
        Shadow_stack_store 8 bytes of SRC to DEST_LA with user-access intent;
    ELSE
        (* Destination not 4B aligned *)
        IF DEST_LA[1:0]
            THEN GP(0); FI;
        Shadow_stack_store 4 bytes of SRC[31:0] to DEST_LA with user-access intent;
FI;
Clear mode previously setup to do user-access intent shadow stack store
```

Flags Affected

None.

Protected Mode Exceptions

#UD	If the LOCK prefix is used.
	If CR4.CET = 0.
#GP(0)	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If destination is located in a non-writeable segment.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
	If linear address of destination is not 4 byte aligned.
	If CPL is not 0.
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs if destination is not a user shadow stack.
	Other terminal and non-terminal faults.

Real-Address Mode Exceptions

#UD	The WRUSS instruction is not recognized in virtual-8086 mode.

Virtual-8086 Mode Exceptions

#UD	The WRUSS instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD	If the LOCK prefix is used.
	If CR4.CET = 0.
#GP(0)	If a memory address is in a non-canonical form.
	If linear address of destination is not 4 byte aligned.
	If CPL is not 0.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code)	If a page fault occurs if destination is not a user shadow stack.
	Other terminal and non-terminal faults.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used.
	If CR4.CET = 0.
#GP(0)	If a memory address is in a non-canonical form.
	If linear address of destination is not 4 byte aligned.
	If CPL is not 0.
#PF(fault-code)	If a page fault occurs if destination is not a user shadow stack.
	Other terminal and non-terminal faults.

6.7 SETSSBSY — Mark Shadow Stack Busy

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F3 0F 01 E8	SETSSBSY	NP	Valid	Valid	Mark shadow stack pointed by IA32_PL0_SSP as busy.

Instruction Operand Encoding

Op/ En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Mark shadow stack pointed to by IA32_PL0_SSP as busy and load SSP with value from MSR IA32_PL0_SSP.

Operation

```
IF (CR4.CET = 0)
    THEN #UD; FI;
IF (IA32_S_CET.SH_STK_EN = 0)
    THEN #UD; FI;
IF CPL > 0
    THEN GP(0); FI;

SSP_LA = IA32_PL0_SSP
If SSP_LA not aligned to 8 bytes
    THEN #GP(0); FI;

Fault = 0
Tmp = Locked shadow_Stack_Load with store intent 8 bytes from SSP_LA
If (Tmp & 0x01)
    THEN fault = 1; FI; (* Fault if busy bit already set *)

IF ((EFER.LMA AND CS.L) = 0 AND Tmp[63:32] != 0)
    THEN fault = 1; FI; (* In legacy mode/compatibility mode the address in token must be in low 4G *)

IF (Tmp & ~0x01)    != SSP_LA
    THEN fault = 1; FI; (* The SSP address in token must match the address specified *)

Tmp = (fault == 1) ? Tmp : (Tmp | 0x01); (* If fault is 0 then set the busy bit in the token *)
Shadow_stack_store 8 bytes of Tmp to SSP_LA and release lock

If (fault == 1)
    THEN # CP(SETSSBSY); FI;    (* If invalid token then fault *)
```

SSP = SSP_LA

Flags Affected

None.

Protected Mode Exceptions

#UD	If the LOCK prefix is used.
	If CR4.CET = 0.
	IF IA32_S_CET.SH_STK_EN = 0.
#GP(0)	If IA32_PL0_SSP not aligned to 8 bytes.
	If CPL is not 0.
#CP(setssbsy)	If busy bit in token set.
	If in 32-bit or compatibility mode, and the address in token is not below 4G.
#PF(fault-code)	If a page fault occurs.

Real-Address Mode Exceptions

#UD	The SETSSBSY instruction is not recognized in virtual-8086 mode.

Virtual-8086 Mode Exceptions

#UD	The SETSSBSY instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD	If the LOCK prefix is used.
	If CR4.CET = 0.
	IF IA32_S_CET.SH_STK_EN = 0.
#GP(0)	Same as protected mode exceptions.
#CP(setssbsy)	Same as protected mode exceptions.
#PF(fault-code)	If a page fault occurs.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used.
	If CR4.CET = 0.
	IF IA32_S_CET.SH_STK_EN = 0.
#GP(0)	If IA32_PL0_SSP not aligned to 8 bytes.
	If CPL is not 0.
#CP(setssbsy)	If busy bit in token set
	If in 32-bit or compatibility mode, and the address in token is not below 4G.
#PF(fault-code)	If a page fault occurs.

6.8 CLRSSBSY — Clear Shadow Stack Busy Flag

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F3 0F AE /6	CLRSSBSY	M64	Valid	Valid	Mark shadow stack pointed by m64 as not busy.

Instruction Operand Encoding

Op/ En	Operand 1	Operand 2	Operand 3	Operand 4
M64	ModRM:r/m (r, w)	NA	NA	NA

Description

Mark shadow stack pointed to by memory operand as not busy. Subsequent to marking the shadow stack as not busy the SSP is loaded with value 0.

Operation

```
IF (CR4.CET = 0)
    THEN #UD; FI;

IF (IA32_S_CET.SH_STK_EN = 0)
    THEN #UD; FI;

IF CPL > 0
    THEN GP(0); FI;

SSP_LA = Linear_Address(mem operand)
IF SSP_LA not aligned to 8 bytes
    THEN #GP(0); FI;
Invalid_token = 0
Tmp = Locked shadow_Stack_Load with store intent 8 bytes from SSP_LA
IF (Tmp & 0x01) != 1
    THEN invalid_token = 1; FI; (* if busy bit not set then token is invalid *)
IF (Tmp & ~0x01) != SSP_LA
    THEN invalid_token = 1; FI; (* The SSP address in token must match the SSP_LA *)
Tmp = (invalid_token == 1) ? Tmp : (Tmp & !0x01); (* If valid then clear the busy bit *)
Shadow_stack_store 8 bytes of Tmp to SSP_LA and release lock

(* Set the CF if invalid token was detected *)
RFLAGS.CF = (invalid_token == 1) ? 1 : 0;
RFLAGS.ZF,PF,AF,OF,SF ← 0;

SSP = 0
```

Flags Affected

CF is set if an invalid token was detected else is cleared. ZF, PF, AF, OF and SF are cleared.

Protected Mode Exceptions

#UD	If the LOCK prefix is used.
	If CR4.CET = 0.
	IF IA32_S_CET.SH_STK_EN = 0.
#GP(0)	If memory operand linear address not aligned to 8 bytes.
	If a memory operand effective address is outside the CS, DS, ES, FS, or GS segment limit.
	If destination is located in a non-writeable segment.
	If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.
	If CPL is not 0
#SS(0)	If a memory operand effective address is outside the SS segment limit.
#PF(fault-code)	If a page fault occurs.

Real-Address Mode Exceptions

#UD	The CLRSSBSY instruction is not recognized in virtual-8086 mode.

Virtual-8086 Mode Exceptions

#UD	The CLRSSBSY instruction is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

#UD	If the LOCK prefix is used.
	If CR4.CET = 0.
	IF IA32_S_CET.SH_STK_EN = 0.
#GP(0)	Same as protected mode exceptions.
#PF(fault-code)	If a page fault occurs.

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used.
	If CR4.CET = 0.
	IF IA32_S_CET.SH_STK_EN = 0.
#GP(0)	If memory operand linear address not aligned to 8 bytes.
	If CPL is not 0.
	If the memory address is in a non-canonical form.
	If token is invalid.
#SS(0)	If a memory address referencing the SS segment is in a non-canonical form.
#PF(fault-code)	If a page fault occurs.

7 Control Transfer Terminating Instructions

7.1 ENDBR64 — Terminate an Indirect Branch in 64-bit Mode

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F3 0F 1E FA	ENDBR64	NP	Valid	Valid	Terminate indirect branch in 64 bit mode.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Terminate an indirect branch in 64 bit mode.

Operation

```
IF EndbranchEnabled(CPL) & EFER.LMA = 1 & CS.L = 1
    IF CPL = 3
        THEN
            IA32_U_CET.TRACKER = IDLE
              IA32_U_CET.SUPPRESS = 0
        ELSE
            IA32_S_CET.TRACKER = IDLE
              IA32_S_CET.SUPPRESS = 0
    FI
FI;
```

Flags Affected

None.

Exceptions

None.

7.2 ENDBR32 — Terminate an Indirect Branch in 32-bit and Compatibility Mode

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F3 0F 1E FB	ENDBR32	NP	Valid	Valid	Terminate indirect branch in 32 bit and compatibility mode.

Instruction Operand Encoding

Op/ En	Operand 1	Operand 2	Operand 3	Operand 4
NP	NA	NA	NA	NA

Description

Terminate an indirect branch in 32 bit and compatibility mode.

Operation

```
IF EndbranchEnabled(CPL) & (EFER.LMA = 0 | (EFER.LMA=1 & CS.L = 0)
   IF CPL = 3
       THEN
           IA32_U_CET.TRACKER = IDLE
             IA32_U_CET.SUPPRESS = 0
       ELSE
           IA32_S_CET.TRACKER = IDLE
             IA32_S_CET.SUPPRESS = 0
   FI
FI;
```

Flags Affected

None.

Exceptions

None.

8 Control Protection Exception, Enumeration, Enables and Extended State Management

8.1 Control Protection Exception

Interrupt 21 — Control Protection Exception (#CP)

Exception Class Fault.

Description

Indicates a control flow transfer attempt violated the control flow enforcement technology constraints. This exception is a contributory class exception.

Exception Error Code

Yes (special format). The processor provides the control protection exception handler with following information through the error code on the stack.

- NEAR-RET (value 1) – indicates the #CP was caused by a near RET instruction.
- FAR-RET/IRET (value 2) – indicates the #CP was caused by a FAR RET or IRET instruction.
- ENDBRANCH (value 3) – indicates the #CP was due to missing ENDBRANCH at target of an indirect call or jump instruction.
- RSTORSSP (value 4) – indicates the #CP was caused by a token check failure in RSTORSSP instruction.
- SETSSBSY (value 5) – indicates #CP was caused by a token check failure in SETSSBSY instruction.

Bit 15 (ENCL) of the error code, if set to 1, indicates the #CP occurred during enclave execution.

Saved Instruction Pointer

Saved contents of CS and EIP registers point to the instruction that generated the exception.

Program State Change

A program-state change does not accompany the control protection fault, because the exception occurs before the faulting instruction is executed

8.2 Feature Enumeration

CET shadow stacks feature flag - if CPUID.(EAX=7, ECX=0):ECX.CET_SS[bit 7] is 1, the processor supports CET shadow stack features, including the MSR described in section 9.5.

CET indirect branch tracking feature flag - if CPUID.(EAX=7, ECX=0):EDX.CET_IBT[bit 20] is 1, the processor supports CET indirect branch tracking, including the MSR described in section 9.5.

8.3 Master Enable

CR4.CET bit (bit 23) is as master enable for CET.

8.4 CET MSRs

- IA32_U_CET (0x6A0) – The bits 1:0 are defined if CPUID.(EAX=7, ECX=0):ECX.CET_SS is 1. The bits 5:2 and bits 63:10 are defined if CPUID.(EAX=7, ECX=0):EDX.CET_IBT is 1.
    - Bit 0 - SH_STK_EN – when set to 1, enable shadow stacks at CPL3.
    - Bit 1 - WR_SHSTK_EN – when set to 1, enables the WRSS{D,Q}W instructions.
    - Bit 2 - ENDBR_EN – when set to 1, enables indirect branch tracking.
    - Bit 3 - LEG_IW_EN – Enable legacy compatibility treatment for indirect branch tracking.

  - Bit 4 – NO_TRACK_EN – when set to 1, enables use of no-track prefix for indirect branch tracking.
  - Bit 5 – SUPPRESS_DIS – when set to 1, disables suppression of CET indirect branch tracking on legacy compatibility.
  - Bit 9:6 – RSVD – must be 0.
  - Bit 10 – SUPPRESS – when set to 1, indirect branch tracking is suppressed. This bit can be written to 1 only if TRACKER is written as IDLE.
  - Bit 11 - TRACKER – Value of the indirect branch tracking state machine - Values: IDLE (0), WAIT_FOR_ENDBRANCH(1).
  - Bit 63:12 - EB_LEG_BITMAP_BASE - linear address of a bitmap in memory indicating valid pages as target of CALL/JMP_indirect that do not land on ENDBRANCH when CET is enabled and not suppressed. Valid when ENDBR_EN is 1. Must be machine canonical when written on parts that support 64 bit mode. On parts that do not support 64 bit mode, the bits 63:32 are reserved and must be 0. This value is extended by 12 bits at the low end to form the base address (this automatically aligns the address on a 4-Kbyte boundary).
- IA32_S_CET (0x6A2) – similar format as IA32_U_CET – configures supervisor mode CET.

The following MSRs are defined if CPUID.(EAX=7, ECX=0):ECX.CET_SS is 1.

- IA32_PL3_SSP (0x6A7) – linear address of the user mode top of shadow stack pointer to be loaded into SSP on next supervisor to user mode transition. Must be machine canonical when written and the address must be aligned to 4 bytes, i.e. bits 1:0 are reserved. On parts that do not support 64 bit mode, the bits 63:32 are reserved and must be 0.
- IA32_PL2_SSP (0x6A6) - linear address of the user mode top of shadow stack pointer to be loaded into SSP on next transition to CPL 2. Must be machine canonical when written on parts that support 64 bit mode and the address must be aligned to 4 bytes, i.e. bits 1:0 are reserved. On parts that do not support 64 bit mode, the bits 63:32 are reserved and must be 0.
- IA32_PL1_SSP (0x6A5) - linear address of the user mode top of shadow stack pointer to be loaded into SSP on next transition to CPL 1. Must be machine canonical when written on parts that support 64 bit mode and the address must be aligned to 4 bytes, i.e. bits 1:0 are reserved. On parts that do not support 64 bit mode, the bits 63:32 are reserved and must be 0.
- IA32_PL0_SSP (0x6A4) – linear address of the user mode top of shadow stack pointer to be loaded into SSP on next transition to CPL 0. Must be machine canonical when written on parts that support 64 bit mode and the address must be aligned to 4 bytes, i.e. bits 1:0 are reserved. On parts that do not support 64 bit mode, the bits 63:32 are reserved and must be 0.
- IA32_INTERRUPT_SSP_TABLE_ADDR (0x6A8) – linear address of the table of pointers to shadow stacks to be switched to when initiating a stack switch in 64 bit mode through IST mechanism. Must be machine canonical when written on parts that support 64 bit mode. On parts that do not support 64 bit mode, this MSR is not present.

8.5 CET Extended State Management

CET defines two set of state that can be saved and restored with XSAVES/XRSTORS. The user space CET state save/restore is controlled by the IA32_XSS.CET_U[bit 11] and the supervisor space CET state save/restore is controlled by IA32_XSS.CET_S[bit 12].

XSAVE feature set support for CET is enumerated by the sub-leaf functions CPUID.(EAX=0DH, ECX=1), CPUID.(EAX = 0DH, ECX = 11), CPUID.(EAX = 0DH, ECX = 12).

- Processor Extended State Enumeration Sub-leaf (EAX = 0DH, ECX = 1) returns:
  - EBX
    - If CET_U bit (bit 11) set in IA32_XSS then reports additional 16 bytes to save CET user state.
    - If CET_S bit (bit 12) set in IA32_XSS then reports additional 24 bytes to save CET supervisor state.

  - ECX
    - IA32_XSS[CET_U] bit (bit 11) is supported if 1.
    - IA32_XSS[CET_S] bit (bit 12) is supported if 1.
- Processor Extended State Enumeration Sub-leaf (EAX = 0DH, ECX = 11) returns:
  - EAX – 16 bytes
  - EBX – 0
  - ECX – 1 (supervisory state)
  - EDX – 0
- Processor Extended State Enumeration Sub-leaf (EAX = 0DH, ECX = 12) returns:
  - EAX – 24 bytes
  - EBX – 0
  - ECX – 1 (supervisory state)
  - EDX – 0

The CET_U state buffer is as follows.

- Offset 0: IA32_U_CET
- Offset 8: IA32_PL3_SSP

The CET_S state buffer is as follows.

- Offset 0: IA32_PL0_SSP
- Offset 8 : IA32_PL1_SSP
- Offset 16: IA32_PL2_SSP

XRSTORS on CET state will do reserved bit and canonicality checks on the state in similar manner as done by the WRMSR to these state elements.

9 Shadow Stack, Paging and EPT

This section describes interactions between the shadow-stack feature and memory management as controlled by paging and EPT.

The shadow-stack feature defines numerous operations that may access a shadow stack as part of new instructions or of CET-defined changes to existing control-flow operations.
While these shadow-stack accesses use linear addresses, as do ordinary data accesses, the processor distinguishes them from ordinary data accesses. Specifically, the paging and EPT features enforce access rights differently for shadow-stack accesses. In part, this is done by identifying certain pages as shadow-stack pages.

Like ordinary data accesses, each shadow-stack access is defined (for paging and EPT) as being either a user access or a supervisor access. In general, a shadow-stack access is a user access if CPL = 3 and a supervisor access if CPL < 3. The WRUSS instruction is an exception: although it can be executed only if CPL = 0, the processor treats its shadow-stack accesses as user accesses.

This section describes in the impact on paging and EPT when shadow stacks are enabled by setting CR4.CET. The processor does not allow CR4.CET to be set if CR0.WP = 0 (similarly, it does not allow CR0.WP to be cleared while CR4.CET = 1). As a result, this section does not account for the treatment of shadow-stack pages when CR0.WP = 0.

When paging is disabled (CR0.PG=0), the shadow stack accesses are allowed to complete always.

Section 9.1 how the existing paging architecture is extended to identify certain pages as shadow-stack pages. Section 9.2 explains how paging enforces access rights for shadow-stack pages.

9.1 Shadow-Stack Pages as Defined by Paging

In its translation of a linear address, paging defines the properties of the address's page based on the paging-structure entries used to translate the address. For example, a user-mode page is one in which the U/S flag (bit 2) is 1 in each of the paging-structure entries used to translate an address on the page.
When CR4.CET = 1, paging identifies certain pages as shadow-stack pages. A page is a shadow-stack page if following are all true of its translation:

- The R/W flag (bit 1) is 1 in every paging-structure entry controlling the translation except the last entry (the entry that maps the page).
- The R/W flag is 0 in the paging-structure entry maps the page.
- The dirty flag (bit 6) is 1 in the paging-structure entry maps the page.

Normally, whenever there is a write to a page, the processor sets the dirty flag in the paging-structure entry that maps the page. As long as CR0.WP = 1, no processor that supports CET will ever set the dirty flag in a paging-structure entry in which the R/W flag is 0. This ensures that software has full control over which pages are shadow-stack pages.

Because the R/W flag is 0 in the paging-structure entry that maps a shadow-stack page, ordinary data writes are not allowed to shadow-stack pages. A shadow-stack page may be either a user-mode page or a supervisor-mode page, depending on the values of the U/S flags in the paging-structure entries that translate the page. Ordinary data reads from shadow-stack page may be allowed subject to ordinary paging access rights.

9.2 Shadow-Stack Access Rights as Enforced by Paging (Outside an Enclave)

As noted earlier, the shadow-stack feature defines certain memory accesses as shadow-stack accesses. Outside an enclave, paging enforces access rights on shadow-stack accesses as follows:

- A shadow-stack access is not allowed to a page that is not a shadow-stack page as defined in Section 9.1.
- A user-mode shadow-stack access is not allowed to a supervisor-mode page. (In general, user-mode accesses are not allowed to supervisor-mode pages.) Recall that the shadow-stack accesses made by the WRUSS instruction are user-mode accesses, even though that instruction requires CPL = 0.
- A supervisor-mode shadow-stack access is not allowed to a user-mode page. (This may be enforced more generally if CR4.SMAP = 1; it is enforced for shadow-stack accesses regardless of the value of CR4.SMAP.)

- The fact that the R/W flag is 0 in the paging-structure entry maps a shadow-stack page (which must be the case) does not prevent shadow-stack writes to that page.
- Shadow-stack accesses are not allowed to a page for which protection keys disable access.
- Shadow-stack write accesses are not allowed to a page for which protection keys disable write access.

A shadow-stack access causes a page fault (#PF) when any of the above conditions disallow the access. (A page fault also occurs if there is a no translation for the access's linear address due a paging-structure entry that is not present or that sets a reserved bit.)

Any page fault caused by a shadow-stack access sets the SS flag (bit 6) in the error code associated with the page fault. Other bits in the error code are set normally.

The paging interactions described in this section apply only when software is not in an enclave. Section 9.3 describes the treatment of shadow-stack accesses in an enclave.

9.3 Shadow-Stack Accesses in an Enclave

When software is in an enclave, shadow-stack accesses are treated differently from how they are outside an enclave (Section 9.2). The following items provide details:

- Shadow-stack accesses from inside an enclave to a linear address outside that enclave (outside its ELRANGE) result in a #GP(0) exception. This exception has priority higher than linear-address translation through paging.
- Shadow-stack accesses from inside an enclave are subject to the same access rights as ordinary data accesses (including protection keys). In particular, they are not limited to shadow-stack pages as defined in Section 9.1; shadow-stack write accesses are not allowed to page for which the R/W flag is 0 in the paging-structure entry maps the page

Even though they are subject to the same access rights as ordinary data accesses, a page fault caused by a shadow-stack access from inside an enclave sets the SS flag (bit 6) in the error code associated with the page fault. (Other bits in the error code are set normally.)

A shadow-stack access allowed by paging is subject new enclave access controls specific to shadow-stack accesses. Violations of these access controls cause SGX-induced page faults. See Section 13.1 for more details.

9.4 Basic EPT Control of Shadow-Stack Accesses

Extend page tables (EPT) is a feature by which a virtual-machine monitor (VMM) can specify the translation of the guest-physical addresses, which are the output of ordinary paging as configured by supervisor software in a virtual machine (VM). Similar to ordinary paging, EPT is defined with a hierarchy of paging structures, and it specifies, for each translated guest-physical address, a physical address and access rights.
The basic EPT mechanism does not treat shadow-stack accesses differently from ordinary data accesses. Shadow-stack read (write) accesses are treated the same as ordinary read (write) accesses. Section 9.5 introduces a new EPT feature distinguishes shadow-stack accesses from ordinary data accesses.

A violation of access control established by EPT is called an EPT violation. EPT violations are generally delivered as VM exits. EPT violations save information about the access and the violation in the exit qualification in the current VMCS.

When the shadow-stack feature is enabled, a new bit is defined in the exit qualification used by EPT violations. Specifically, bit 13 (which had been reserved) is set to 1 by any EPT violation resulting from a shadow-stack access. Other bits in the exit qualification are set normally.

9.5 Supervisor Shadow-Stack Control

The definition of shadow-stack pages by paging (Section 9.1) ensures the integrity of those pages by protecting them from ordinary data writes. While this protects shadow-stack use by user code, supervisor shadow-stack pages might be compromised by malicious supervisor software that reconfigures the paging structures temporarily to allow writes to a shadow-stack page.

A VMM can prevent such attacks by using EPT to prevent writes to supervisor shadow-stack pages of a VM; such protection will apply regardless the access rights established by ordinary paging. There are two challenges to protecting supervisor shadow-stack pages in this way: (1) the VMM must know which guest-physical addresses hold the VM's supervisor shadow-stack pages; and (2) shadow-stack writes to these pages should be allowed while ordinary data writes should not.

The first challenge can be addresses through paravirtualization. The VMM must define a mechanism by which supervisor software in a VM can identify to the VMM the guest-physical addresses of supervisor shadow-stack pages. Techniques for accomplishing this are outside the scope of this document.
The second challenge is addressed by a new EPT feature called supervisor shadow-stack control, a feature designed to prevent such attacks on supervisor software operating in a virtual machine. With this feature, a virtual-machine monitor (VMM), operating outside the virtual machine, can identify supervisor shadow-stack pages using extended page table (EPT).

Supervisor shadow-stack control is enabled by setting a bit in the extended-page-table pointer (EPTP) field in the VMCS. Specifically, supervisor shadow-stack control is enabled if bit 7 of EPTP (which had been reserved) is 1.

When supervisor shadow-stack control is enabled, the processor identifies a supervisor shadow-stack pages using bit 60 of the EPT paging-structure entry maps the page; this bit is called the supervisor shadow-stack (SSS) bit. (The processor ignores bit 60 of other EPT paging-structure entries. When supervisor shadow-stack control is not enabled, the processor ignores bit 60 of all EPT paging-structure entries.)

Section 9.5.1 explains how EPT defines supervisor shadow-stack pages, while Section 9.5.2 details how EPT controls supervisor shadow-stack accesses.

9.5.1 Supervisor Shadow-Stack Pages as Defined by EPT

In its translation of a guest-physical address, EPT defines the properties of the address's page based on the EPT paging-structure entries used to translate the address. For example, a page is writeable if the W bit (bit 1) is 1 in each of the EPT paging-structure entries used to translate an address on the page.

When supervisor shadow-stack control is enabled, EPT identifies certain pages as supervisor shadow-stack pages. A page is a supervisor shadow-stack page if following are all true of its translation:

- The R bit (bit 0) is 1 in every EPT paging-structure entry controlling the translation.
- The W bit (bit 1) is 1 in every EPT paging-structure entry controlling the translation except the last entry (the entry that maps the page).
- The SSS bit (bit 60) is 1 in the EPT paging-structure entry maps the page.

9.5.2 Supervisor Shadow-Stack Access Rights as Enforced by EPT

As noted earlier, certain memory accesses as shadow-stack accesses. As explained earlier, each such access is either a user access or a supervisor access. EPT applies no special treatment to user shadow-stack accesses (see Section 9.4; this is true also for supervisor shadow-stack accesses when supervisor shadow-stack control is not enabled). When supervisor shadow-stack control is enabled, EPT paging enforces access rights on supervisor shadow-stack accesses as follows:

- A supervisor shadow-stack access is not allowed to a page that is not a supervisor shadow-stack page as defined in Section 9.5.1.
- The fact that the W bit is 0 in the EPT paging-structure entry maps a supervisor shadow-stack page (which is allowed) does not prevent supervisor shadow-stack writes to that page.

A supervisor shadow-stack access causes an EPT violation if the first condition above disallows the access. (An EPT violation also occurs if there is a no translation for the access's guest-physical address due an EPT paging-structure entry that is not present or that sets a reserved bit.)
(Because the W bit may be 0 in the EPT paging-structure entry that maps a supervisor shadow-stack page, ordinary data writes may be disallowed to supervisor shadow-stack pages while still allowing supervisor shadow-stack writes. It is not possible to disallow ordinary data reads from supervisor shadow-stack pages while still allowing shadow-stack accesses.)

When supervisor shadow-stack control is enabled, a new bit is defined in the exit qualification used by EPT violations. Specifically, an EPT violation resulting from an access to a guest-physical address sets bit 14 of the exit qualification (which had been reserved) as follows:

- If there is a no translation for the guest-physical address (due an EPT paging-structure entry that is not present or that sets a reserved bit), bit 14 is cleared to 0.
- Otherwise, bit 14 contains the value of the SSS bit (bit 6) in the EPT paging-structure entry mapping the page containing the guest-physical address.

The items above apply to all EPT violations that occur when supervisor shadow-stack control is enabled (and not only to those resulting from supervisor shadow-stack accesses).

10 VMX Interactions

This section describes the interactions of CET with VM-exits and VM-entries to/from the executive monitor and the SMM-transfer monitor. For interactions with SMM when the dual-monitor treatment is not activated. A VMM emulating control transfer instructions or events (e.g., indirect call, indirect jmp, task switch, etc.) for a CET enabled guest must emulate the corresponding CET state changes.

10.1 VMCS Guest State Area Extensions

To support CET, the VMCS Guest-state area is extended to add following new state elements.

Field	Encod-ing	Size (bits)	
VMX_GUEST_IA32_S_CET	0x6828	Natu-ral	Guest IA32_S_CET MSR. This field is defined if CPUID.(EAX=7, ECX=0):ECX.CET_SS is 1 or if CPUID.(EAX=7, ECX=0):EDX.CET_IBT is 1
VMX_GUEST_SSP	0x682A	Natu-ral	Guest Shadow Stack Pointer (SSP). This field is defined if CPUID.(EAX=7, ECX=0):ECX.CET_SS is 1
VMX_GUEST_IA32_INTER-RUPT_SSP_TABLE_ADDR	0x682C	Natu-ral	Guest IA32_INTER-RUPT_SSP_TABLE_ADDR MSR. . This field is defined if CPUID.(EAX=7, ECX=0):ECX.CET_SS is 1

10.2 VMCS Host State Area Extensions

To support CET, the VMCS Host-state area is extended to add following new state elements.

Field	Encod-ing	Size (bits)	
VMX_HOST_IA32_S_CET	0x6C18	Natu-ral	Host IA32_S_CET MSR. This field is defined if CPUID.(EAX=7, ECX=0):ECX.CET_SS is 1 or if CPUID.(EAX=7, ECX=0):EDX.CET_IBT is 1
VMX_HOST_SSP	0x6C1A	Natu-ral	Host Shadow Stack Pointer (SSP). This field is defined if CPUID.(EAX=7, ECX=0):ECX.CET_SS is 1
VMX_HOST_IA32_INTER-RUPT_SSP_TABLE_ADDR	0x6C1C	Natu-ral	Host IA32_INTERRUPT_SSP_TA-BLE_ADDR MSR. This field is de-fined if CPUID.(EAX=7, ECX=0):ECX.CET_SS is 1

10.3 VMCS VM-Exit Controls Extensions

The VM-Exit controls are extended with a new exit control as follows.

Bit Posi-tion(s)	Name	Description
28	Load Host CET state.	This control determines if CET state in the VMCS host state area is loaded on VM exit.

10.4 VMCS VM-Entry Controls Extensions

The VM-Entry controls are extended with a new exit control as follows.

Bit Posi-tion(s)	Name	Description
20	Load Guest CET state.	This control determines if CET state in the VMCS guest state area is loaded on VM entry.

10.5 EPTP

The EPTP field of VMCS is extended as follows:

Bit Posi-tion(s)	Name	Description
7	Enable supervisor shadow stack control.	Enable supervisor shadow stack control bit in EPT.

When enabled, the SSS bit in the EPT PTE provides read/write permission to supervisor (U/S=0) shadow stack accesses. User shadow stack accesses test EPT read/write permissions normally.

10.6 VM Exit

On processors that support CET, the VM exit saves the state of IA32_S_CET, SSP and IA32_INTERRUPT_SSP_TABLE_ADDR MSR to the VMCS guest-state area unconditionally.

If "Load host CET state" VM-exit control is 1, the CET state is restored from the VMCS host-state area as follows.

- IA32_S_CET MSR is loaded from the IA32_S_CET field. Bits that are reserved in the MSR are maintained with their reserved values. If host address space size is 1 then each of the 63:N of the EB_LEG_BITMAP_BASE field of this MSR is set to the value of the N-1 bit (where N is the linear-address bits) else bits 63:32 are set to 0. If the TRACKER is set to WAIT_FOR_ENDBRANCH and SUPPRESS is 1 then then SUPPRESS is cleared to 0.
- SSP is loaded from the HOST_SSP field. If host address space size is 1 then each of the 63:N is set to the value of the N-1 bit (where N is the linear-address bits) else bits 63:32 are set to 0.
- IA32_INTERRUPT_SSP_TABLE_ADDR MSR is loaded from the IA32_INTERRUPT_SSP_TABLE_ADDR field. If host address space size is 1, each of the bits 63:N is set to the value of the N-1 bit (where N is the supported number of linear-address bits).

To context switch guest CET state the VMM uses XSAVES/XRSTORS instructions to save/restore the guest CET state. The VMM can then use the “Load guest CET state” control to reload the supervisor mode CET state of the guest as saved in the VMCS.

VM exit may abort if the VMCS is updated following VM entry to set host state CR0.WP=0 and CR4.CET=1.

10.7 VM Entry

Following early VM entry checks are performed and failures leads to a VM entry failure with RFLAGS.ZF set to 1 and VM instruction error field set to 7 indicating “VM entry with invalid control fields”.

- On processors that do not support CET, setting the “load host CET state” exit control or “load guest CET state” entry control must be 0.
- On processors that do not support CET, “Enable supervisor shadow stack control” control bit in EPTP must be 0 if EPT is enabled.

If the “Load host CET state” VM-exit control is 1, then the host state area checks are extended as follows. Failure of these checks leads to a VM entry failure with RFLAGS.ZF set to 1 and the VM-instruction error field set to 8 indicating “VM entry with invalid host state fields”.

- IA32_S_CET bits 9:6 must be 0. If host address space size is 0 then bits 63:32 must be 0 else EB_LEG_BIT-MAP_BASE field of this MSR must contain a canonical address. Both tracker and suppress bits must not be both set to 1.
- If host address space size is 0 then bits 63:32 of HOST_SSP must be 0 else HOST_SSP must contain a canonical address.
- IA32_INTERRUPT_SSP_TABLE_ADDR fields must contain a canonical address.

If “Load Guest CET State” VM-entry control is 1, the guest state area checks are extended as follows and failure of these checks to a failed VM entry VM exit with reason set to “Bad guest state”.

- IA32_S_CET bits 9:6 must be 0. Linear address in bits 63:12 must be canonical. Both tracker and suppress bits must not be both set to 1.
- The GUEST_SSP fields must have Bits 63:32 must be 0 if the “IA-32e mode guest” VM-entry control is 0 or if the L bit (bit 13) in the access rights field for CS is 0. If the processor supports N < 64 linear-address bits, bits 63:N must be identical if the “IA-32e mode guest” VM-entry control is 1 and the L bit in the access-rights field for CS is 1.
- IA32_INTERRUPT_SSP_TABLE_ADDR fields must contain a canonical address.

Additionally VM entry checks the disallowed configuration of CR0.WP and CR4.CET as follows.

- If host CR0.WP=0 and host CR4.CET=1 then VM entry fails with RFLAGS.ZF set to 1 and VM-instruction error field set to 8 indicating “VM entry with invalid host state”.
- If guest CR0.WP=0 and guest CR4.CET=1 then VM entry leads to failed VM entry VM exit with reason set to “Bad guest state”.

The VM-entry interrupt information field consistency checks are extended to allow #CP to be delivered with an error code. The #CP exception is delivered as a contributory exception. The bit 11 of the VM-entry interruption-information field determines whether delivery pushes an error code on the guest stack. If the valid bit (bit 31) of the VM-entry interruption-information field is 1, the field's deliver-error-code bit (bit 11) shall be 1 if and only if (1) either (a) the "unrestricted guest" VMexecution control is 0; or (b) bit 0 (corresponding to CR0.PE) is set in the CR0 field in the guest-state area; (2) the interruption type is hardware exception; and (3) the vector indicates an exception that would normally deliver an error code (8 = #DF; 10 = TS; 11 = #NP; 12 = #SS; 13 = #GP; 14 = #PF; 17 = #AC; or 21=#CP) and IA32_VMX_BASIC MSR bit 56 is enumerated as 0. On parts that enumerate IA32_VMX_BASIC MSR bit as 1, any exception vector, including #CP, can be delivered with or without an error code if the other consistency checks are satisfied.

Subsequent to these checks the IA32_S_CET, SSP and IA32_INTERRUPT_SSP_TABLE_ADDR MSR are loaded from corresponding guest-state VMCS fields.

10.8 IA32_VMX_EPT_VPID_CAP

Bit 23 of this MSR enumerates support for setting “Enable Shadow Stack Control” (bit 7) in EPTP.

11 SMM Interactions

This section describes the interactions of CET with SMIs and RSM when the dual-monitor treatment is not activated.

11.1 SMRAM State Save Map

The SMRAM state save map is extended as follows:

Offset (Added to SMBASE + 8000H)	MSR Address (on processors that support internal state save)	Register	Writeable?
0xFEC8	C26H	SSP	Yes

11.2 SMI Handler Execution Environment

Processors that support CET shadow stacks, save the SSP registers to the SMRAM state save area. The CR4.CET is cleared to 0 on SMI. Thus the initial execution environment of the SMI handler has CET disabled and all of the CET state still in the machine. An SMM that uses CET is required to save and restore the CET state in the processor.

On an SMM VM exit caused by a VMCALL that activates the dual-monitor treatment, the current VMCS is the one established by the executive monitor and does not contain the VM-exit controls and host state required to initialize the STM. This VM exit thus initializes the CR4 state to a fixed value or value loaded from content of MSEG header. The CR4.CET is cleared on this SMM VM exit caused by a VMCALL that activates the dual-monitor treatment.

11.3 RSM

The RSM on processors that support CET shadow stacks loads the SSP value from the SMRAM state save area. On processors that support Intel 64 architecture, if the SSP value is not canonical then forces it to be canonical by sign extending it.

RSM will go to shut down if attempting to restore CR0.WP to 0 and CR4.CET to 1.

12 TXT Interactions

GETSEC[ENTERACCS] and GETSEC[SENTER] clear CR4.CET, and it is not restored when these instructions complete.

GETSEC[EXITAC] will cause #GP(0) fault if CR4.CET is set.

13 SGX Interactions

This section discussions extensions to the SGX architecture to support CET.

13.1 CET in Enclaves Model

The basic model for CET support in an enclave is that the enclave has its private configuration for CET and does not share it with the enclosing applications CET configuration. On entry into the enclave the CET state of the enclosing application is saved into scratchpad registers inside the processor and the CET state of the enclave is established. On an asynchronous exit the enclave CET state is saved into the enclave state save area frame. On exit from the enclave the CET state of the enclosing application is re-established from the scratchpad registers.

A new page type – PT_SS_FIRST – is used to denote pages in enclave that can be used as first page of a shadow stack.

A new page type – PT_SS_REST – is used to denote pages in enclave that can be used as non-first page of a shadow stack.

Having two page types allows OS software to ensure that dynamic thread creation within an enclave does not cause one enclave thread to point its shadow stack pointer to the shadow stack of another thread. This allows an operating system to ensure there is a guard page between any two shadow stacks.

A page denoted as PT_SS_FIRST and PT_SS_REST will be legal target for shadow_stack_load, shadow_stack_store and regular load operations. Regular stores will be disallowed to such pages. A PT_SS_FIRST/PT_SS_REST page must be writeable in the IA page tables and in EPT.

When in enclave mode, shadow_stack_load and shadow_stack_store operations must be to addresses in the enclave ELRANGE. This prevents an enclave from operating on shadow stacks of the enclosing application.

CET extends EAUG to enable CET EPC page allocation. In these case, the caller must provide a SECINFO structure that specifies the page parameters. Shadow page permission must be R/W. Regular R/W pages may continue to be allocated by providing a SECINFO pointer value of 0. Regular R/W pages may also be allocated by providing a SECINFO structure that specifies the page parameters. The EDMM support for CET pages will be enumerated with CET enumeration and thus any part that supports CET and EDMM will also support the EDMM extensions for CET. The EAUG instruction creates a "shadow stack restore" token at offset 0xFF8 on a PT_SS_FIRST page. This allows a dynamically created shadow stack to be restored using the RSTORSSP instruction. EADD and EAUG prevent creating a PT_SS_FIRST or PT_SS_REST page as the first page or last page in ELRANGE to avoid SSP value of 0 or SSP value of (0xFFFFFFF8(32 bit mode)/0xFFFFFFFF_FFFFFFF8(64 bit mode)) being misconfigured to be a valid shadow stack page in the enclave.

EADD instruction requires that the PT_SS_REST page be all zero. The EADD instruction requires that a PT_SS_FIRST page be all zero except the 8 bytes at offset 0xFF8 on that page that must have a "shadow stack restore" token. This "shadow stack restore" token must have a linear address which is the linear address of the PT_SS_FIRST page + 4096. As an enclave could be loaded at varying linear addresses, the enclave builder is should not extend the measurement of the PT_SS_FIRST pages into the measurement registers. On first entry on to the enclave using a TCS, the enclave software can use the RSTORSSP instruction to restore its SSP. Subsequent to performing a RSTORSSP the enclave software can use the INCSSP instruction to pop the "previous ssp" token that is created by the RSTORSSP instruction at the top of the restored shadow stack.

On an enclave entry, the SSP will be initialized to the value in a new TCS field called PREVSSP. The PREVSSP is written with the value of SSP on enclave exit and is loaded into SSP at enclave entry. When a TCS page is added using EADD or accepted using EACCEPT, the processor requires the PREVSSP field to be initialized to 0.

13.2 Operations Not Supported on Shadow Stack Pages

Following operations are not allowed on pages of type PT_SS_FIRST and PT_SS_REST:

- EACCEPTCOPY
- EMODPR
- EMODPE

13.3 Indirect Branch Tracking – Legacy Compatibility Treatment

The legacy code page bitmap is tested using the page offset within the ELRANGE instead of the absolute linear address of the address where ENDBRANCH was missed – see detailed algorithm in section 3.6.

14 Enclave Access Control and Data Structures

14.1 Overview of Enclave Execution Environment

When an enclave is created, it has a range of linear addresses to which the processor applies enhanced access control. This range is called the ELRANGE (see Section 14.3). When an enclave generates a memory access, the existing IA32 segmentation and paging architecture are applied. Additionally, linear addresses inside the ELRANGE must map to an EPC page otherwise when an enclave attempts to access that linear address a fault is generated.

The EPC pages need not be physically contiguous. System software allocates EPC pages to various enclaves. Enclaves must abide by OS/VMM imposed segmentation and paging policies. OS/VMM-managed page tables and extended page tables provide address translation for the enclave pages. Hardware requires that these pages are properly mapped to EPC (any failure generates an exception).

Enclave entry must happen through specific enclave instructions:

- ENCLU[EENTER], ENCLU[ERESUME].

Enclave exit must happen through specific enclave instructions or events:

- ENCLU[EEXIT], Asynchronous Enclave Exit (AEX).

Attempts to execute, read, or write to linear addresses mapped to EPC pages when not inside an enclave will result in the processor altering the access to preserve the confidentiality and integrity of the enclave. The exact behavior may be different between implementations. As an example a read of an enclave page may result in the return of all one's or return of ciphertext of the cache line. Writing to an enclave page may result in a dropped write or a machine check at a later time. The processor will provide the protections as described in Section 14.4 and Section 14.5 on such accesses.

14.2 Terminology

A memory access to the ELRANGE and initiated by an instruction executed by an enclave is called a Direct Enclave Access (Direct EA).

Memory accesses initiated by certain Intel® SGX instruction leaf functions such as ECREATE, EADD, EDBGRD, EDBGWR, ELDU/ELDB, EWB, EREMOVE, EENTER, and ERESUME to EPC pages are called Indirect Enclave Accesses (Indirect EA). Table 2 lists additional details of the indirect EA of SGX1 and SGX2 extensions.

Direct EAs and Indirect EAs together are called Enclave Accesses (EAs).

Any memory access that is not an Enclave Access is called a non-enclave access.

14.3 Access-control Requirements

Enclave accesses have the following access-control attributes:

- All memory accesses must conform to segmentation and paging protection mechanisms.
- Code fetches from inside an enclave to a linear address outside that enclave result in a #GP(0) exception.
- Shadow-stack-load, shadow-stack-store or shadow-stack-store-intent from inside an enclave to a linear address outside that enclave results in a #GP(0) exception.
- Non-enclave accesses to EPC memory result in undefined behavior. EPC memory is protected as described in Section 14.4 and Section 14.5 on such accesses.
- EPC pages of page types PT_REG, PT_TCS and PT_TRIM must be mapped to ELRANGE at the linear address specified when the EPC page was allocated to the enclave using ENCLS[EADD] or ENCLS[EAUG] leaf functions. Enclave accesses through other linear address result in a #PF with the PFEC.SGX bit set.
- Direct EAs to any EPC pages must conform to the currently defined security attributes for that EPC page in the EPCM. These attributes may be defined at enclave creation time (EADD) or when the enclave sets them using SGX2 instructions. The failure of these checks results in a #PF with the PFEC.SGX bit set.
  - Target page must belong to the currently executing enclave.
  - Data may be written to an EPC page if the EPCM allow write access.

— Data may be read from an EPC page if the EPCM allow read access.
— Instruction fetches from an EPC page are allowed if the EPCM allows execute access.
— Shadow-stack-load from an EPC page and shadow-stack-store to an EPC page are allowed only if the page type is PT_SS_FIRST or PT_SS_REST.
— Data writes that are not shadow-stack-store are not allowed if the EPCM page type is PT_SS_FIRST or PT_SS_REST.
— Target page must not have a restricted page type[2] (PT_SECS, PT_TCS, PT_VA, PT_SS_FIRST, PT_SS_REST or PT_TRIM). The PT_SS_FIRST and PT_SS_REST pages are not restricted page types when CPUID.(EAX=07H, ECX=00h):ECX[CET] is 1.
— The EPC page must not be BLOCKED.
— The EPC page must not be PENDING.
— The EPC page must not be MODIFIED.

14.4 Segment-based Access Control

Intel SGX architecture does not modify the segment checks performed by a logical processor. All memory accesses arising from a logical processor in protected mode (including enclave access) are subject to segmentation checks with the applicable segment register.

To ensure that outside entities do not modify the enclave's logical-to-linear address translation in an unexpected fashion, ENCLU[EENTER] and ENCLU[ERESUME] check that CS, DS, ES, and SS, if usable (i.e., not null), have segment base value of zero. A non-zero segment base value for these registers results in a #GP(0).

On enclave entry either via EENTER or ERESUME, the processor saves the contents of the external FS and GS registers, and loads these registers with values stored in the TCS at build time to enable the enclave’s use of these registers for accessing the thread-local storage inside the enclave. On EEXIT and AEX, the contents at time of entry are restored. On AEX, the values of FS and GS are saved in the SSA frame. On ERESUME, FS and GS are restored from the SSA frame. The details of these operations can be found in the descriptions of EENTER, ERESUME, EEXIT, and AEX flows.

14.5 Page-based Access Control

14.5.1 Access-control for Accesses that Originate from non-SGX Instructions

Intel SGX builds on the processor's paging mechanism to provide page-granular access-control for enclave pages. Enclave pages are designed to be accessible only from inside the currently executing enclave if they belong to that enclave. In addition, enclave accesses must conform to the access control requirements described in Section 14.3, or through certain Intel SGX instructions. Attempts to execute, read, or write to linear addresses mapped to EPC pages when not inside an enclave will result in the processor altering the access to preserve the confidentiality and integrity of the enclave. The exact behavior may be different between implementations.

14.5.2 Memory Accesses that Split across ELRANGE

Memory data accesses are allowed to split across ELRANGE (i.e., a part of the access is inside ELRANGE and a part of the access is outside ELRANGE) while the processor is inside an enclave. If an access splits across ELRANGE, the processor splits the access into two sub-accesses (one inside ELRANGE and the other outside ELRANGE), and each access is evaluated. A code-fetch access that splits across ELRANGE results in a #GP due to the portion that lies outside of the ELRANGE.

[2]EPCM may allow write, read or execute access only for pages with page type PT_REG.

14.5.3 Implicit vs. Explicit Accesses

Memory accesses originating from Intel SGX instruction leaf functions are categorized as either explicit accesses or implicit accesses. Table 2 lists the implicit and explicit memory accesses made by Intel SGX leaf functions.

14.5.3.1 Explicit Accesses

Accesses to memory locations provided as explicit operands to Intel SGX instruction leaf functions, or their linked data structures are called explicit accesses.

Explicit accesses are always made using logical addresses. These accesses are subject to segmentation, paging, extended paging, and APIC-virtualization checks, and trigger any faults/exit associated with these checks when the access is made.

The interaction of explicit memory accesses with data breakpoints is leaf-function-specific.

14.5.3.2 Implicit Accesses

Accesses to data structures whose physical addresses are cached by the processor are called implicit accesses. These addresses are not passed as operands of the instruction but are implied by use of the instruction.

These accesses do not trigger any access-control faults/exits or data breakpoints. Table 2 lists memory objects that Intel SGX instruction leaf functions access either by explicit access or implicit access. The addresses of explicit access objects are passed via register operands with the second through fourth column of Table 2 matching implicitly encoded registers RBX, RCX, RDX.

Physical addresses used in different implicit accesses are cached via different instructions and for different durations. The physical address of SECS associated with each EPC page is cached at the time the page is added to the enclave via ENCLS[EADD] or ENCLS[EAUG], or when the page is loaded to EPC via ENCLS[ELDB] or ENCLS[ELDU]. This binding is severed when the corresponding page is removed from the EPC via ENCLS[EREMOVE] or ENCLS[EWB]. Physical addresses of TCS and SSA pages are cached at the time of most-recent enclave entry. Exit from an enclave (ENCLU[EEXIT] or AEX) flushes this caching. Details of Asynchronous Enclave Exit is described in Section 15.

The physical addresses that are cached for use by implicit accesses are derived from logical (or linear) addresses after checks such as segmentation, paging, EPT, and APIC virtualization checks. These checks may trigger exceptions or VM exits. Note, however, that such exception or VM exits may not occur after a physical address is cached and used for an implicit access.

Table 2 List of Implicit and Explicit Memory Access by Intel® SGX Enclave Instructions

Instr. Leaf	Enum.	Explicit 1	Explicit 2	Explicit 3	Implicit
EACCEPT	SGX2	SECINFO	EPCPAGE		SECS
EACCEPTCOPY	SGX2	SECINFO	EPCPAGE (Src)	EPCPAGE (Dst)	
EADD	SGX1	PAGEINFO and linked structures	EPCPAGE		
EAUG	SGX2	PAGEINFO and linked structures	EPCPAGE		SECS
EBLOCK	SGX1	EPCPAGE			SECS
ECREATE	SGX1	PAGEINFO and linked structures	EPCPAGE		
EDBGRD	SGX1	EPCADDR	Destination		SECS
EDBGWR	SGX1	EPCADDR	Source		SECS
EDECVIRTCHILD	OVERSUB	EPCPAGE	SECS		
EENTER	SGX1	TCS and linked SSA			SECS
EEXIT	SGX1				SECS, TCS

EEXTEND	SGX1	SECS	EPCPAGE		
EGETKEY	SGX1	KEYREQUEST	KEY		SECS
EINCVIRTCHILD	OVERSUB	EPCPAGE	SECS		
EINIT	SGX1	SIGSTRUCT	SECS	EINITTOKEN	
ELDB/ELDU	SGX1	PAGEINFO and linked structures, PCMD	EPCPAGE	VAPAGE	
ELDBC/ELDUC	OVERSUB	PAGEINFO and linked structures	EPCPAGE	VAPAGE	
EMODPE	SGX2	SECINFO	EPCPAGE		
EMODPR	SGX2	SECINFO	EPCPAGE		SECS
EMODT	SGX2	SECINFO	EPCPAGE		SECS
EPA	SGX1	EPCADDR			
ERDINFO	OVERSUB	RDINFO	EPCPAGE		
EREMOVE	SGX1	EPCPAGE			SECS
EREPORT	SGX1	TARGETINFO	REPORTDATA	OUTPUTDATA	SECS
ERESUME	SGX1	TCS and linked SSA			SECS
ESETCONTEXT	OVERSUB		SECS	ContextValue	
ETRACK	SGX1	EPCPAGE			
ETRACKC	OVERSUB		EPCPAGE		
EWB	SGX1	PAGEINFO and linked structures, PCMD	EPCPAGE	VAPAGE	SECS
Asynchronous Enclave Exit*					SECS, TCS, SSA
*Details of Asynchronous Enclave Exit (AEX) is described in Section 15.4					

14.6 Intel® SGX Data Structures Overview

Enclave operation is managed via a collection of data structures. Many of the top-level data structures contain sub-structures. The top-level data structures relate to parameters that may be used in enclave setup/maintenance, by Intel SGX instructions, or AEX event. The top-level data structures are:

- SGX Enclave Control Structure (SECS)
- Thread Control Structure (TCS)
- State Save Area (SSA)
- Page Information (PAGEINFO)
- Security Information (SECINFO)

- Paging Crypto MetaData (PCMD)
- Enclave Signature Structure (SIGSTRUCT)
- EINIT Token Structure (EINITTOKEN)
- Report Structure (REPORT)
- Report Target Info (TARGETINFO)
- Key Request (KEYREQUEST)
- Version Array (VA)
- Enclave Page Cache Map (EPCM)
- Read Info (RDINFO)

Details of the top-level data structures and associated sub-structures are listed in Section 14.7 through Section 14.18.

14.7 SGX Enclave Control Structure (SECS)

The SECS data structure requires 4K-Bytes alignment.

Table 3 Layout of SGX Enclave Control Structure (SECS)

Field	OFFSET (Bytes)	Size (Bytes)	Description
SIZE	0	8	Size of enclave in bytes; must be power of 2.
BASEADDR	8	8	Enclave Base Linear Address must be naturally aligned to size.
SSAFRAMESIZE	16	4	Size of one SSA frame in pages, including XSAVE, pad, GPR, and MISC (if CPUID.(EAX=12H, ECX=0):.EBX != 0).
MISCSELECT	20	4	Bit vector specifying which extended features are saved to the MISC region (see Section 14.7.2) of the SSA frame when an AEX occurs.
CET_LEG_BITMAP_OFFSET	24	8	Page aligned offset of legacy code page bitmap from enclave base. Software is expected to program this offset such that the entire bitmap resides in the ELRANGE when legacy compatibility mode for indirect branch tracking is enabled. However this is not enforced by the hardware. This field exists when CPUID.(EAX=7, ECX=0):EDX.CET_IBT[bit 20] is enumerated as 1 else it is reserved.
CET_ATTRIBUTES	32	1	CET feature attributes of the enclave, see Table 6. This field exists when CPUID.(EAX=12,ECX=1):EAX[6] is enumerated as 1 else it is reserved.
RESERVED	33	15	
ATTRIBUTES	48	16	Attributes of the Enclave, see Table 4.
MRENCLAVE	64	32	Measurement Register of enclave build process. See SIGSTRUCT for format.
RESERVED	96	32	
MRSIGNER	128	32	Measurement Register extended with the public key that verified the enclave. See SIGSTRUCT for format.
RESERVED	160	32	

CONFIGID	192	64	Post EINIT configuration identity.
ISVPRODID	256	2	Product ID of enclave.
ISVSVN	258	2	Security version number (SVN) of the enclave.
CONFIGSVN	260	2	Post EINIT configuration security version number (SVN).
RESERVED	260	3834	The RESERVED field consists of the following: ▪ EID: An 8 byte Enclave Identifier. Its location is implementation specific. ▪ PAD: A 352 bytes padding pattern from the Signature (used for key derivation strings). Its location is implementation specific. ▪ VIRTCHILDCNT: An 8 byte Count of virtual children that have been paged out by a VMM. Its location is implementation specific. ▪ ENCLAVECONTEXT: An 8 byte Enclave context pointer. Its location is implementation specific. ▪ ISVFAMILYID: A 16 byte value assigned to identify the family of products the enclave belongs to. ▪ ISVEXTPRODID: A 16 byte value assigned to identify the product identity of the enclave. ▪ The remaining 3226 bytes are reserved area. The entire 3836 byte field must be cleared prior to executing ECREATE.

14.7.1 ATTRIBUTES

The ATTRIBUTES data structure is comprised of bit-granular fields that are used in the SECS, the REPORT and the KEYREQUEST structures. CPUID.(EAX=12H, ECX=1) enumerates a bitmap of permitted 1-setting of bits in ATTRIBUTES.

Table 4 Layout of ATTRIBUTES Structure

Field	Bit Position	Description
INIT	0	This bit indicates if the enclave has been initialized by EINIT. It must be cleared when loaded as part of ECREATE. For EREPORT instruction, TARGET_INFO.ATTRIBUTES[ENIT] must always be 1 to match the state after EINIT has initialized the enclave.
DEBUG	1	If 1, the enclave permit debugger to read and write enclave data using EDBGRD and EDBGWR.
MODE64BIT	2	Enclave runs in 64-bit mode.
RESERVED	3	Must be Zero.
PROVISIONKEY	4	Provisioning Key is available from EGETKEY.
EINITTOKEN_KEY	5	EINIT token key is available from EGETKEY.
CET	6	Enable CET attributes. When CPUID.(EAX=12H, ECX=1):EAX[6] is 0 this bit is reserved and must be 0.
KSS	7	Key Separation and Sharing Enabled.
RESERVED	63:8	Must be zero.

XFRM	127:64	XSAVE Feature Request Mask.

14.7.2 SECS.MISCSELECT Field

CPUID.(EAX=12H, ECX=0):EBX[31:0] enumerates which extended information that the processor can save into the MISC region of SSA when an AEX occurs. An enclave writer can specify via SIGSTRUCT how to set the SECS.MISCSELECT field. The bit vector of MISCSELECT selects which extended information is to be saved in the MISC region of the SSA frame when an AEX is generated. The bit vector definition of extended information is listed in Table 5.

If CPUID.(EAX=12H, ECX=0):EBX[31:0] = 0, SECS.MISCSELECT field must be all zeros.

The SECS.MISCSELECT field determines the size of MISC region of the SSA frame.

Table 5 Bit Vector Layout of MISCSELECT Field of Extended Information

Field	Bit Position	Description
EXINFO	0	Report information about page fault and general protection exception that occurred inside an enclave.
CPINFO	1	Report information about control protection exception that occurred inside an enclave. When CPUID.(EAX=12H, ECX=0):EBX[1] is 0, this bit is reserved.
Reserved	31:2	Reserved (0).

14.7.3 SECS.CET_ATTRIBUTES Field

This field can be used by the enclave writer to enable various CET attributes in an enclave. This field exists when CPUID.(EAX=12,ECX=1):EAX[6] is enumerated as 1. Bits 1:0 are defined when CPUID.(EAX=7, ECX=0):ECX.CET_SS is 1 and bits 5:2 are defined when CPUID.(EAX=7, ECX=0):EDX.CET_IBT is 1.

Table 6 Bit Vector Layout of CET_ATTRIBUTES Field of Extended Information

Field	Bit Position	Description
SH_STK_EN	0	When set to 1 enable shadow stacks.
WR_SHSTK_EN	1	When set to 1 enables the WRSS{D,Q}W instructions.
ENDBR_EN	2	When set to 1 enables indirect branch tracking.
LEG_IW_EN	3	Enable legacy compatibility treatment for indirect branch tracking.
NO_TRACK_EN	4	When set to 1 enables use of no-track prefix for indirect branch tracking.
SUPPRESS_DIS	5	When set to 1 disables suppression of CET indirect branch tracking on legacy compatibility.
Reserved	7:6	Reserved (0).

14.8 Thread Control Structure (TCS)

Each executing thread in the enclave is associated with a Thread Control Structure. It requires 4K-Bytes alignment.

Table 7 Layout of Thread Control Structure (TCS)

Field	OFFSET (Bytes)	Size (Bytes)	Description
STAGE	0	8	Enclave execution state of the thread controlled by this TCS. A value of 0 indicates that this TCS is available for enclave entry. A value of 1 indicates that a processer is currently executing an enclave in the context of this TCS.
FLAGS	8	8	The thread's execution flags.
OSSA	16	8	Offset of the base of the State Save Area stack, relative to the enclave base. Must be page aligned.
CSSA	24	4	Current slot index of an SSA frame, cleared by EADD and EACCEPT.
NSSA	28	4	Number of available slots for SSA frames.
OENTRY	32	8	Offset in enclave to which control is transferred on EENTER relative to the base of the enclave.
AEP	40	8	The value of the Asynchronous Exit Pointer that was saved at EENTER time.
OFSBASGX	48	8	Offset to add to the base address of the enclave for producing the base address of FS segment inside the enclave. Must be page aligned.
OGSBASGX	56	8	Offset to add to the base address of the enclave for producing the base address of GS segment inside the enclave. Must be page aligned.
FSLIMIT	64	4	Size to become the new FS limit in 32-bit mode.
GSLIMIT	68	4	Size to become the new GS limit in 32-bit mode.
OCETSSA	72	8	When CPUID.(EAX=12H, ECX=1):EAX[6] is 1 is 1, this field provides then Offset of CET state save area from enclave base. When CPUID.(EAX=12H, ECX=1):EAX[6] is 1 is 0, this field is reserved and must be 0.
PREVSSP	80	8	When CPUID.(EAX=07H, ECX=00h):ECX[CET_SS] is 1, this field records the SSP at the time of AEX or EEXIT; used to setup SSP on entry. When CPUID.(EAX=07H, ECX=00h):ECX[CET_SS] is 0, this field is reserved and must be 0
RESERVED	88	4024	Must be zero.

14.8.1 TCS.FLAGS

Table 8 Layout of TCS.FLAGS Field

Field	Bit Position	Description
DBGOPTIN	0	If set, allows debugging features (single-stepping, breakpoints, etc.) to be enabled and active while executing in the enclave on this TCS. Hardware clears this bit on EADD. A debugger may later modify it if the enclave's ATTRIBUTES.DEBUG is set.
RESERVED	63:1	

14.8.2 State Save Area Offset (OSSA)

The OSSA points to a stack of State Save Area (SSA) frames (see Section 14.9) used to save the processor state when an interrupt or exception occurs while executing in the enclave.

14.8.3 Current State Save Area Frame (CSSA)

CSSA is the index of the current SSA frame that will be used by the processor to determine where to save the processor state on an interrupt or exception that occurs while executing in the enclave. It is an index into the array of frames addressed by OSSA. CSSA is incremented on an AEX and decremented on an ERESUME.

14.8.4 Number of State Save Area Frames (NSSA)

NSSA specifies the number of SSA frames available for this TCS. There must be at least one available SSA frame when EENTER-ing the enclave or the EENTER will fail.

14.9 State Save Area (SSA) Frame

When an AEX occurs while running in an enclave, the architectural state is saved in the thread's current SSA frame, which is pointed to by TCS.CSSA. An SSA frame must be page aligned, and contains the following regions:

- The XSAVE region starts at the base of the SSA frame, this region contains extended feature register state in an XSAVE/FXSAVE-compatible non-compacted format.
- A Pad region: software may choose to maintain a pad region separating the XSAVE region and the MISC region. Software choose the size of the pad region according to the sizes of the MISC and GPRSGX regions.
- The GPRSGX region. The GPRSGX region is the last region of an SSA frame (see Table 9). This is used to hold the processor general purpose registers (RAX … R15), the RIP, the outside RSP and RBP, RFLAGS and the AEX information.
- The MISC region (If CPUIDEAX=12H, ECX=0):EBX[31:0] != 0). The MISC region is adjacent to the GRPSGX region, and may contain zero or more components of extended information that would be saved when an AEX occurs. If the MISC region is absent, the region between the GPRSGX and XSAVE regions is the pad region that software can use. If the MISC region is present, the region between the MISC and XSAVE regions is the pad region that software can use.

Table 9 Top-to-Bottom Layout of an SSA Frame

Region	Offset (Byte)	Size (Bytes)	Description
XSAVE	0	Calculate using CPUID leaf 0DH information	The size of XSAVE region in SSA is derived from the enclave's support of the collection of processor extended states that would be managed by XSAVE. The enablement of those processor extended state components in conjunction with CPUID leaf 0DH information determines the XSAVE region size in SSA.
Pad	End of XSAVE region	Chosen by enclave writer	Ensure the end of GPRSGX region is aligned to the end of a 4KB page.
MISC	base of GPRSGX – sizeof(MISC)	Calculate from highest set bit of SECS.MISCSELECT	See Section 0.
GPRSGX	SSAFRAMESIZE – 176	176	See Table 10 Layout of GPRSG for layout of the GPRSGX region.

14.9.1 GPRSGX Region

The layout of the GPRSGX region is shown in Table 10.

Table 10 Layout of GPRSGX Portion of the State Save Area

Field	OFFSET (Bytes)	Size (Bytes)	Description
RAX	0	8	
RCX	8	8	
RDX	16	8	
RBX	24	8	
RSP	32	8	
RBP	40	8	
RSI	48	8	
RDI	56	8	
R8	64	8	
R9	72	8	
R10	80	8	
R11	88	8	
R12	96	8	

R13	104	8	
R14	112	8	
R15	120	8	
RFLAGS	128	8	Flag register.
RIP	136	8	Instruction pointer.
URSP	144	8	Non-Enclave (outside) stack pointer. Saved by EENTER, restored on AEX.
URBP	152	8	Non-Enclave (outside) RBP pointer. Saved by EENTER, restored on AEX.
EXITINFO	160	4	Contains information about exceptions that cause AEXs, which might be needed by enclave software (see Section 14.9.1.1).
RESERVED	164	4	
FSBASE	168	8	FS BASE.
GSBASE	176	8	GS BASE.

14.9.1.1 EXITINFO

EXITINFO contains the information used to report exit reasons to software inside the enclave. It is a 4 byte field laid out as in Table 11. The VALID bit is set only for the exceptions conditions which are reported inside an enclave. See Table 12 for which exceptions are reported inside the enclave. If the exception condition is not one reported inside the enclave then VECTOR and EXIT_TYPE are cleared.

Table 11 Layout of EXITINFO Field

Field	Bit Position	Description
VECTOR	7:0	Exception number of exceptions reported inside enclave.
EXIT_TYPE	10:8	011b: Hardware exceptions. 110b: Software exceptions. Other values: Reserved.
RESERVED	30:11	Reserved as zero.
VALID	31	0: unsupported exceptions. 1: Supported exceptions. Includes two categories: • Unconditionally supported exceptions: #DE, #DB, #BP, #BR, #UD, #MF, #AC, #XM. • Conditionally supported exception: — #PF, #GP if SECS.MISCSELECT.EXINFO = 1. — #CP if SECS.MISCSELECT.CPINFO=1.

14.9.1.2 VECTOR Field Definition

Table 12 contains the VECTOR field. This field contains information about some exceptions which occur inside the enclave. These vector values are the same as the values that would be used when vectoring into regular exception handlers. All values not shown are not reported inside an enclave.

Table 12 Exception Vectors

Name	Vector #	Description
#DE	0	Divider exception.
#DB	1	Debug exception.
#BP	3	Breakpoint exception.
#BR	5	Bound range exceeded exception.
#UD	6	Invalid opcode exception.
#GP	13	General protection exception. Only reported if SECS.MISCSELECT.EXINFO = 1.
#PF	14	Page fault exception. Only reported if SECS.MISCSELECT.EXINFO = 1.
#MF	16	x87 FPU floating-point error.
#AC	17	Alignment check exceptions.
#XM	19	SIMD floating-point exceptions.
#CP	21	Control protection exception. Only reported if SECS.MISCSELECT.CPINFO=1

14.9.2 MISC Region

The layout of the MISC region is shown in Table 13. The number of components that the processor supports in the MISC region corresponds to the bits of CPUID.(EAX=12H, ECX=0):EBX[31:0] set to 1. Each set bit in CPUID.(EAX=12H, ECX=0):EBX[31:0] has a defined size for the corresponding component, as shown in Table 13. Enclave writers needs to do the following:

- Decide which MISC region components will be supported for the enclave.
- Allocate an SSA frame large enough to hold the components chosen above.
- Instruct each enclave builder software to set the appropriate bits in SECS.MISCSELECT.

The first component, EXINFO, starts next to the GPRSGX region. Additional components in the MISC region grow in ascending order within the MISC region towards the XSAVE region.

The size of the MISC region is calculated as follows:

- If CPUID.(EAX=12H, ECX=0):EBX[31:0] = 0, MISC region is not supported.
- If CPUID.(EAX=12H, ECX=0):EBX[31:0] != 0, the size of MISC region is derived from sum of the highest bit set in SECS.MISCSELECT and the size of the MISC component corresponding to that bit. Offset and size information of currently defined MISC components are listed in Table 13. For example, if the highest bit set in SECS.MISCSELECT is bit 0, the MISC region offset is OFFSET(GPRSGX)-16 and size is 16 bytes.
- The processor saves a MISC component i in the MISC region if and only if SECS.MISCSELECT[i] is 1.

Table 13 Layout of MISC region of the State Save Area

MISC Components	OFFSET (Bytes)	Size (Bytes)	Description
EXINFO	Offset(GPRSGX) – 16	16	If CPUID.(EAX=12H, ECX=0):EBX[0] = 1, exception information on #GP or #PF that occurred inside an enclave can be written to the EXINFO structure if specified by SECS.MISCSELECT[0] = 1. If CPUID.(EAX=12H, ECX=0):EBX[1] = 1, exception information on #CP that occurred inside an enclave can be written to the EXINFO structure if specified by SECS.MISCSELECT[1] = 1.
Future Extension	Below EXINFO	TBD	Reserved. (Zero size if CPUID.(EAX=12H, ECX=0):EBX[31:1] =0).

14.9.2.1 EXINFO Structure

Table 14 contains the layout of the EXINFO structure that provides additional information.

Table 14 Layout of EXINFO Structure

Field	OFFSET (Bytes)	Size (Bytes)	Description
MADDR	0	8	If #PF: contains the page fault linear address that caused a page fault. If #GP: the field is cleared. If #CP: the field is cleared.
ERRCD	8	4	Exception error code for either #GP or #PF.
RESERVED	12	4	

14.9.2.2 Page Fault Error Code

Table 15 contains page fault error code that may be reported in EXINFO.ERRCD.

Table 15 Page Fault Error Code

Name	Bit Position	Description
P	0	Same as non-SGX page fault exception P flag.
W/R	1	Same as non-SGX page fault exception W/R flag.
U/S[3]	2	Always set to 1 (user mode reference).
RSVD	3	Same as non-SGX page fault exception RSVD flag.
I/D	4	Same as non-SGX page fault exception I/D flag.
PK	5	Protection Key induced fault.
RSVD	14:6	Reserved.
SGX	15	EPCM induced fault.

[3]Page faults incident to enclave mode that report U/S=0 are not reported in EXINFO.

RSVD	31:5	Reserved.

14.10 CET State Save Area Frame

The CET state save area consists of an array of CET state save frames. The number of CET state save frames is equal to the TCS.NSSA. The current CET SSA frame is indicated by TCS.CSSA. The offset of the CET state save area is specified by TCS.OCETSSA.

Field	Offset (bytes)	Size (Bytes)	Description
SSP	0	8	Shadow Stack Pointer. This field is reserved when CPUID.(EAX=7, ECX=0):ECX[CET_SS] is 0.
IB_TRACK_STATE	8	8	Indirect branch tracker state: Bit 0: SUPPRESS – suppressed(1), tracking(0) Bit 1: TRACKER - IDLE (0), WAIT_FOR_ENDBRANCH (1) Bit 63:2 – Reserved This field is reserved when CPUID.(EAX=7, ECX=0):EDX[CET_IBT] is 0.

14.11 Page Information (PAGEINFO)

PAGEINFO is an architectural data structure that is used as a parameter to the EPC-management instructions. It requires 32-Byte alignment.

Table 16 Layout of PAGEINFO Data Structure

Field	OFFSET (Bytes)	Size (Bytes)	Description
LINADDR	0	8	Enclave linear address.
SRCPGE	8	8	Effective address of the page where contents are located.
SECINFO/PCMD	16	8	Effective address of the SECINFO or PCMD (for ELDU, ELDB, EWB) structure for the page.
SECS	24	8	Effective address of EPC slot that currently contains the SECS.

14.12 Security Information (SECINFO)

The SECINFO data structure holds meta-data about an enclave page.

Table 17 Layout of SECINFO Data Structure

Field	OFFSET (Bytes)	Size (Bytes)	Description
FLAGS	0	8	Flags describing the state of the enclave page.
RESERVED	8	56	Must be zero.

14.12.1 SECINFO.FLAGS

The SECINFO.FLAGS are a set of fields describing the properties of an enclave page.

Table 18 Layout of SECINFO.FLAGS Field

Field	Bit Position	Description
R	0	If 1 indicates that the page can be read from inside the enclave; otherwise the page cannot be read from inside the enclave.
W	1	If 1 indicates that the page can be written from inside the enclave; otherwise the page cannot be written from inside the enclave.
X	2	If 1 indicates that the page can be executed from inside the enclave; otherwise the page cannot be executed from inside the enclave.
PENDING	3	If 1 indicates that the page is in the PENDING state; otherwise the page is not in the PENDING state.
MODIFIED	4	If 1 indicates that the page is in the MODIFIED state; otherwise the page is not in the MODIFIED state.
PR	5	If 1 indicates that a permission restriction operation on the page is in progress, otherwise a permission restriction operation is not in progress.
RESERVED	7:6	Must be zero.
PAGE_TYPE	15:8	The type of page that the SECINFO is associated with.
RESERVED	63:16	Must be zero.

14.12.2 PAGE_TYPE Field Definition

The SECINFO flags and EPC flags contain bits indicating the type of page.

Table 19 Supported PAGE_TYPE

TYPE	Value	Description
PT_SECS	0	Page is an SECS.
PT_TCS	1	Page is a TCS.
PT_REG	2	Page is a regular page.
PT_VA	3	Page is a Version Array.
PT_TRIM	4	Page is in trimmed state.
PT_SS_FIRST	5	When CPUID.(EAX=12H, ECX=1):EAX[6] is 1, Page is first page of a shadow stack. When CPUID.(EAX=12H, ECX=1):EAX[6] is 0, this value is reserved.
PT_SS_REST	6	When CPUID.(EAX=12H, ECX=1):EAX[6] is 1, Page is not first page of a shadow stack. When CPUID.(EAX=12H, ECX=1):EAX[6] is 0, this value is reserved.
	All other	Reserved.

14.13 Paging Crypto MetaData (PCMD)

The PCMD structure is used to keep track of crypto meta-data associated with a paged-out page. Combined with PAGEINFO, it provides enough information for the processor to verify, decrypt, and reload a paged-out EPC page. The size of the PCMD structure (128 bytes) is architectural.

EWB calculates the Message Authentication Code (MAC) value and writes out the PCMD. ELDB/U reads the fields and checks the MAC.

The format of PCMD is as follows:

Table 20 Layout of PCMD Data Structure

Field	OFFSET (Bytes)	Size (Bytes)	Description
SECINFO	0	64	Flags describing the state of the enclave page; R/W by software.
ENCLAVEID	64	8	Enclave Identifier used to establish a cryptographic binding between paged-out page and the enclave.
RESERVED	72	40	Must be zero.
MAC	112	16	Message Authentication Code for the page, page meta-data and reserved field.

14.14 Enclave Signature Structure (SIGSTRUCT)

SIGSTRUCT is a structure created and signed by the enclave developer that contains information about the enclave. SIGSTRUCT is processed by the EINIT leaf function to verify that the enclave was properly built.

SIGSTRUCT includes ENCLAVEHASH as SHA256 digest, as defined in FIPS PUB 180-4. The digests are byte strings of length 32. Each of the 8 HASH dwords is stored in little-endian order.

SIGSTRUCT includes four 3072-bit integers (MODULUS, SIGNATURE, Q1, Q2). Each such integer is represented as a byte strings of length 384, with the most significant byte at the position “offset + 383”, and the least significant byte at position “offset”.

The (3072-bit integer) SIGNATURE should be an RSA signature, where: a) the RSA modulus (MODULUS) is a 3072-bit integer; b) the public exponent is set to 3; c) the signing procedure uses the EMSA-PKCS1-v1.5 format with DER encoding of the “DigestInfo” value as specified in of PKCS#1 v2.1/RFC 3447.

The 3072-bit integers Q1 and Q2 are defined by:

q1 = floor(Signature^2 / Modulus);

q2 = floor((Signature^3 - q1 * Signature * Modulus) / Modulus);

SIGSTRUCT must be page aligned

In column 5 of Table 21, ‘Y’ indicates that this field should be included in the signature generated by the developer.

Table 21 Layout of Enclave Signature Structure (SIGSTRUCT)

Field	OFFSET (Bytes)	Size (Bytes)	Description	Signed
HEADER	0	16	Must be byte stream 06000000E10000000000010000000000H	Y
VENDOR	16	4	Intel Enclave: 00008086H Non-Intel Enclave: 00000000H	Y
DATE	20	4	Build date is yyyymmdd in hex: yyyy=4 digit year, mm=1-12, dd=1-31	Y
HEADER2	24	16	Must be byte stream 01010000600000006000000001000000H	Y
SWDEFINED	40	4	Available for software use.	Y
RESERVED	44	84	Must be zero.	Y
MODULUS	128	384	Module Public Key (keylength=3072 bits).	N
EXPONENT	512	4	RSA Exponent = 3.	N
SIGNATURE	516	384	Signature over Header and Body.	N
MISCSELECT*	900	4	Bit vector specifying Extended SSA frame feature set to be used.	Y
MISCMASK*	904	4	Bit vector mask of MISCSELECT to enforce.	Y
CET_ATTRIBUTES	908	1	When CPUID.(EAX=12H, ECX=1):EAX[6] is 1, this field provides Enclave CET attributes that must be set. When CPUID.(EAX=12H, ECX=1):EAX[6] is 0, this field is reserved and must be 0.	Y
CET_ATTRIBUTES_MASK	909	1	When CPUID.(EAX=12H, ECX=1):EAX[6] is 1, this field provides Mask of CET attributes to enforce. When CPUID.(EAX=12H, ECX=1):EAX[6] is 0, this field is reserved and must be 0.	Y
RESERVED	910	2	Must be zero.	Y
ISVFAMILYID	912	16	ISV assigned Product Family ID.	Y
ATTRIBUTES	928	16	Enclave Attributes that must be set.	Y
ATTRIBUTEMASK	944	16	Mask of Attributes to enforce.	Y
ENCLAVEHASH	960	32	MRENCLAVE of enclave this structure applies to.	Y
RESERVED	992	16	Must be zero.	Y
ISVEXTPRODID	1008	16	ISV assigned extended Product ID.	Y
ISVPRODID	1024	2	ISV assigned Product ID.	Y
ISVSVN	1026	2	ISV assigned SVN (security version number).	Y
RESERVED	1028	12	Must be zero.	N

Q1	1040	384	Q1 value for RSA Signature Verification.	N
Q2	1424	384	Q2 value for RSA Signature Verification.	N
* If CPUID.(EAX=12H, ECX=0):EBX[31:0] = 0, MISCSELECT must be 0. If CPUID.(EAX=12H, ECX=0):EBX[31:0] !=0, enclave writers must specify MISCSELECT such that each cleared bit in MISCMASK must also specify the corresponding bit as 0 in MISCSELECT.				

14.15 EINIT Token Structure (EINITTOKEN)

The EINIT token is used by EINIT to verify that the enclave is permitted to launch. EINIT token is generated by an enclave in possession of the EINITTOKEN key (the Launch Enclave).

EINIT token must be 512-Byte aligned.

Table 22 Layout of EINIT Token (EINITTOKEN)

Field	OFFSET (Bytes)	Size (Bytes)	MACed	Description
Valid	0	4	Y	Bit 0: 1: Valid; 0: Invalid. All other bits reserved.
RESERVED	4	44	Y	Must be zero.
ATTRIBUTES	48	16	Y	ATTRIBUTES of the Enclave.
MRENCLAVE	64	32	Y	MRENCLAVE of the Enclave.
RESERVED	96	32	Y	Reserved.
MRSIGNER	128	32	Y	MRSIGNER of the Enclave.
RESERVED	160	32	Y	Reserved.
CPUSVNLE	192	16	N	Launch Enclave's CPUSVN.
ISVPRODIDLE	208	02	N	Launch Enclave's ISVPRODID.
ISVSVNLE	210	02	N	Launch Enclave's ISVSVN.
CET_MASKED_ATTRIBUTES_LE	212	1	N	When CPUID.(EAX=12H, ECX=1):EAX[6] is 1, this field provides Launch enclaves masked CET attributes. This should be set to LE's CET_ATTRIBUTES masked with CET_ATTTRIBUTES_MASK of the LE's KEYREQUEST. When CPUID.(EAX=12H, ECX=1):EAX[6] is 0, this field is reserved.
RESERVED	213	23	N	Reserved.
MASKEDMISCSELECTLE	236	4		Launch Enclave's MASKEDMISCSELECT: set by the LE to the resolved MISCSELECT value, used by EGETKEY (after applying KEYREQUEST's masking).
MASKEDATTRIBUTESLE	240	16	N	Launch Enclave's MASKEDATTRIBUTES: This should be set to the

				LE's ATTRIBUTES masked with ATTRIBUTEMASK of the LE's KEYRE-QUEST.
KEYID	256	32	N	Value for key wear-out protection.
MAC	288	16	N	Message Authentication Code on EINITTOKEN using EINITTO-KEN_KEY.

14.16 Report (REPORT)

The REPORT structure is the output of the EREPORT instruction, and must be 512-Byte aligned.

Table 23 Layout of REPORT

Field	OFFSET (Bytes)	Size (Bytes)	Description
CPUSVN	0	16	The security version number of the processor.
MISCSELECT	16	4	Bit vector specifying which extended features are saved to the MISC region of the SSA frame when an AEX occurs.
CET_ATTRIBUTES	20	1	When CPUID.(EAX=12H, ECX=1):EAX[6] is 1, this field reports the CET_ATTRIBUTES of the Enclave. When CPUID.(EAX=12H, ECX=1):EAX[6] is 0, this field is reserved and must be 0.
RESERVED	21	11	Zero.
ISVEXTNPRODID	32	16	The value of SECS.ISVEXTPRODID.
ATTRIBUTES	48	16	ATTRIBUTES of the Enclave. See Section 14.7.1.
MRENCLAVE	64	32	The value of SECS.MRENCLAVE.
RESERVED	96	32	Zero.
MRSIGNER	128	32	The value of SECS.MRSIGNER.
RESERVED	160	32	Zero.
CONFIGID	192	64	Value provided by SW to identify enclave's post EINIT configuration.
ISVPRODID	256	2	Product ID of enclave.
ISVSVN	258	2	Security version number (SVN) of the enclave.
CONFIGSVN	260	2	Value provided by SW to indicate expected SVN of enclave's post EINIT configuration.
RESERVED	262	42	Zero.
ISVFAMILYID	304	16	The value of SECS.ISVFAMILYID.
REPORTDATA	320	64	Data provided by the user and protected by the REPORT's MAC.
KEYID	384	32	Value for key wear-out protection.
MAC	416	16	Message Authentication Code on the report using report key.

14.16.1 REPORTDATA

REPORTDATA is a 64-Byte data structure that is provided by the enclave and included in the REPORT. It can be used to securely pass information from the enclave to the target enclave.

14.17 Report Target Info (TARGETINFO)

This structure is an input parameter to the EREPORT leaf function. The address of TARGETINFO is specified as an effective address in RBX. It is used to identify the target enclave which will be able to cryptographically verify the REPORT structure returned by EREPORT. TARGETINFO must be 512-Byte aligned.

Table 24 Layout of TARGETINFO Data Structure

Field	OFFSET (Bytes)	Size (Bytes)	Description
MEASUREMENT	0	32	The MRENCLAVE of the target enclave.
ATTRIBUTES	32	16	The ATTRIBUTES field of the target enclave.
CET_ATTRIBUTES	48	1	When CPUID.(EAX=12H, ECX=1):EAX[6] is 1, then this field provides the CET_ATTRIBUTES field of the target enclave. When CPUID.(EAX=12H, ECX=1):EAX[6] is 0, this field is reserved.
RESERVED	49	1	Must be zero.
CONFIGSVN	50	2	CONFIGSVN of the target enclave.
MISCSELECT	52	4	The MISCSELECT of the target enclave.
RESERVED	56	8	Must be zero.
CONFIGID	64	64	CONFIGID of target enclave.
RESERVED	128	384	Must be zero.

14.18 Key Request (KEYREQUEST)

This structure is an input parameter to the EGETKEY leaf function. It is passed in as an effective address in RBX and must be 512-Byte aligned. It is used for selecting the appropriate key and any additional parameters required in the derivation of that key.

Table 25 Layout of KEYREQUEST Data Structure

Field	OFFSET (Bytes)	Size (Bytes)	Description
KEYNAME	0	2	Identifies the Key Required.
KEYPOLICY	2	2	Identifies which inputs are required to be used in the key derivation.
ISVSVN	4	2	The ISV security version number that will be used in the key derivation.
CET_ATTRIBUTES_MASK	6	1	When CPUID.(EAX=12H, ECX=1):EAX[6] is 1, this field provides a mask defining which CET_ATTRIBUTES bits will be included in key derivation. When CPUID.(EAX=12H, ECX=1):EAX[6] is 0, then this field is reserved and must be 0.
RESERVED	7	1	Must be zero.
CPUSVN	8	16	The security version number of the processor used in the key derivation.
ATTRIBUTEMASK	24	16	A mask defining which ATTRIBUTES bits will be included in key derivation.
KEYID	40	32	Value for key wear-out protection.
MISCMASK	72	4	A mask defining which MISCSELECT bits will be included in key derivation.
CONFIGSVN	76	2	Identifies which enclave Configuration's Security Version should be used in key derivation.
RESERVED	78	434	

14.18.1 KEY REQUEST KeyNames

Table 26 Supported KEYName Values

Key Name	Value	Description
EINITTOKEN_KEY	0	EINIT_TOKEN key
PROVISION_KEY	1	Provisioning Key
PROVISION_SEAL_KEY	2	Provisioning Seal Key
REPORT_KEY	3	Report Key
SEAL_KEY	4	Seal Key
	All other	Reserved

14.18.2 Key Request Policy Structure

Table 27 Layout of KEYPOLICY Field

Field	Bit Position	Description
MRENCLAVE	0	If 1, derive key using the enclave's MRENCLAVE measurement register.
MRSIGNER	1	If 1, derive key using the enclave's MRSIGNER measurement register.
NOISVPRODID	2	If 1, derive key WITHOUT using the enclave' ISVPRODID value.
CONFIGID	3	If 1, derive key using the enclave's CONFIGID value.
ISVFAMILYID	4	If 1, derive key using the enclave ISVFAMILYID value.
ISVEXTPRODID	5	If 1, derive key using enclave's ISVEXTPRODID value.
RESERVED	15:6	Must be zero.

14.19 Version Array (VA)

In order to securely store the versions of evicted EPC pages, Intel SGX defines a special EPC page type called a Version Array (VA). Each VA page contains 512 slots, each of which can contain an 8-byte version number for a page evicted from the EPC. When an EPC page is evicted, software chooses an empty slot in a VA page; this slot receives the unique version number of the page being evicted. When the EPC page is reloaded, there must be a VA slot that must hold the version of the page. If the page is successfully reloaded, the version in the VA slot is cleared.

VA pages can be evicted, just like any other EPC page. When evicting a VA page, a version slot in some other VA page must be used to hold the version for the VA being evicted. A Version Array Page must be 4K-Bytes aligned.

Table 28 Layout of Version Array Data Structure

Field	OFFSET (Bytes)	Size (Bytes)	Description
Slot 0	0	8	Version Slot 0
Slot 1	8	8	Version Slot 1
...			
Slot 511	4088	8	Version Slot 511

14.20 Enclave Page Cache Map (EPCM)

EPCM is a secure structure used by the processor to track the contents of the EPC. The EPCM holds exactly one entry for each page that is currently loaded into the EPC. EPCM is not accessible by software, and the layout of EPCM fields is implementation specific.

Table 29 Content of an Enclave Page Cache Map Entry

Field	Description
VALID	Indicates whether the EPCM entry is valid.
R	Read access; indicates whether enclave accesses for reads are allowed from the EPC page referenced by this entry.
W	Write access; indicates whether enclave accesses for writes are allowed to the EPC page referenced by this entry.
X	Execute access; indicates whether enclave accesses for instruction fetches are allowed from the EPC page referenced by this entry.
PT	EPCM page type (PT_SECS, PT_TCS, PT_REG, PT_VA, PT_TRIM, PT_SS_FIRST, PT_SS_REST).
ENCLAVESECS	SECS identifier of the enclave to which the EPC page belongs.
ENCLAVEADDRESS	Linear enclave address of the EPC page.
BLOCKED	Indicates whether the EPC page is in the blocked state.
PENDING	Indicates whether the EPC page is in the pending state.
MODIFIED	Indicates whether the EPC page is in the modified state.
PR	Indicates whether the EPC page is in a permission restriction state.

14.21 Read Info (RDINFO)

The RDINFO structure contains status information about an EPC page. It must be aligned to 32-Bytes.

Table 30 Layout of RDINFO Structure

Field	OFFSET (Bytes)	Size (Bytes)	Description
STATUS	0	8	Page status information.
FLAGS	8	8	EPCM state of the page.
ENCLAVECONTEXT	16	8	Context pointer describing the page’s parent location.

14.21.1 RDINFO Status Structure

Table 31 Layout of RDINFO STATUS Structure

Field	Bit Position	Description
CHILDPRESENT	0	Indicates that the page has one or more child pages present (always zero for non-SECS pages). In VMX non-root operation includes the presence of virtual children.
VIRTCHLDPRESENT	1	Indicates that the page has one or more virtual child pages present (always zero for non-SECS pages). In VMX non-root operation this value is always zero.
RESERVED	63:2	

14.21.2 RDINFO Flags Structure

Table 32 Layout of RDINFO FLAGS Structure

Field	Bit Position	Description
R	0	Read access; indicates whether enclave accesses for reads are allowed from the EPC page referenced by this entry.
W	1	Write access; indicates whether enclave accesses for writes are allowed to the EPC page referenced by this entry.
X	2	Execute access; indicates whether enclave accesses for instruction fetches are allowed from the EPC page referenced by this entry.
PENDING	3	Indicates whether the EPC page is in the pending state.
MODIFIED	4	Indicates whether the EPC page is in the modified state.
PR	5	Indicates whether the EPC page is in a permission restriction state.
RESERVED	7:6	
PAGE_TYPE	15:8	Indicates the page type of the EPC page.
RESERVED	62:16	
BLOCKED	63	Indicates whether the EPC page is in the blocked state.

15 Enclave Exiting Events

Certain events, such as exceptions and interrupts, incident to (but asynchronous with) enclave execution may cause control to transition outside of enclave mode. (Most of these also cause a change of privilege level.) To protect the integrity and security of the enclave, the processor will exit the enclave (and enclave mode) before invoking the handler for such an event. For that reason, such events are called **enclave-exiting events** (EEE); EEEs include external interrupts, non-maskable interrupts, system-management interrupts, exceptions, and VM exits.

The process of leaving an enclave in response to an EEE is called an **asynchronous enclave exit** (AEX). To protect the secrecy of the enclave, an AEX saves the state of certain registers within enclave memory and then loads those registers with fixed values called **synthetic state**.

15.1 Compatible Switch to the Exiting Stack of AEX

AEXs load registers with a pre-determined synthetic state. These registers may be later pushed onto the appropriate stack in a form as defined by the enclave-exiting event. To allow enclave execution to resume after the invoking handler has processed the enclave exiting event, the asynchronous enclave exit loads the address of trampoline code outside of the enclave into RIP. This trampoline code eventually returns to the enclave by means of an ENCLU(ERESUME) leaf function. Prior to exiting the enclave the RSP and RBP registers are restored to their values prior to enclave entry.

The stack to be used is chosen using the same rules as for non-SGX mode:

- If there is a privilege level change, the stack will be the one associated with the new ring.
- If there is no privilege level change, the current application stack is used.
- If the IA-32e IST mechanism is used, the exit stack is chosen using that method.

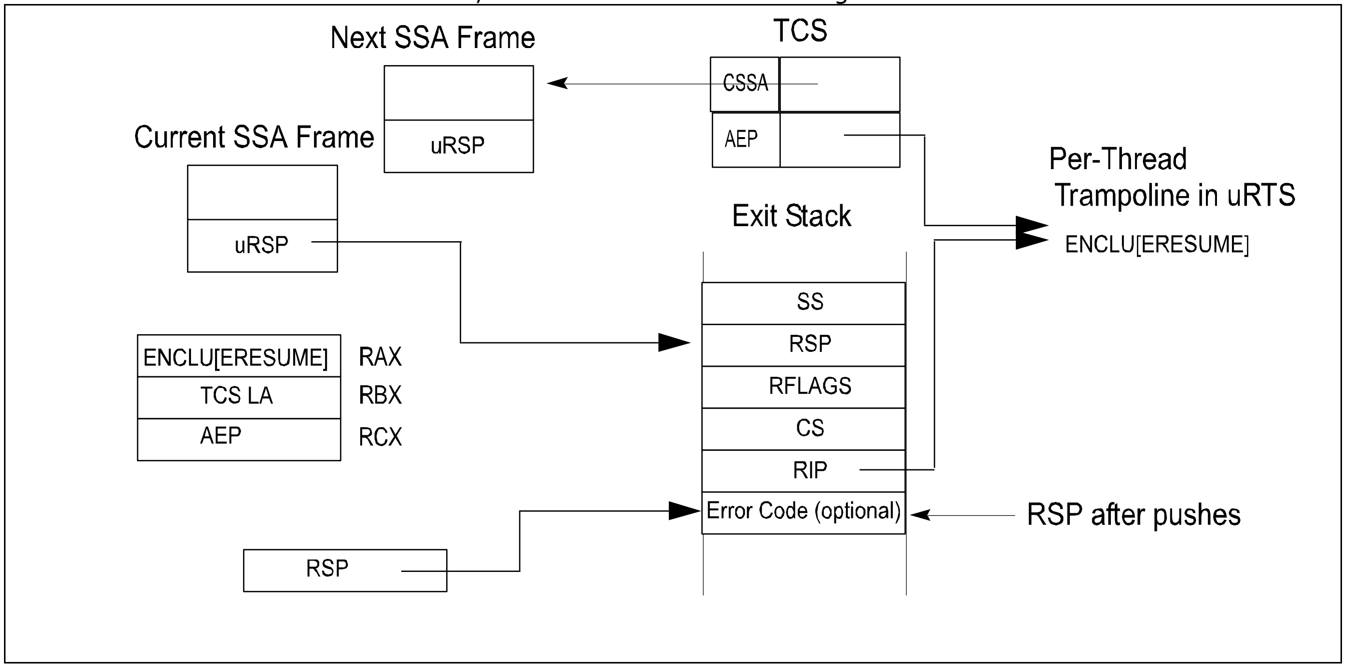

Figure 6 Exit Stack Just After Interrupt with Stack Switch

In all cases, the choice of exit stack and the information pushed onto it is consistent with non-SGX operation. Figure 6 shows the Application and Exiting Stacks after an exit with a stack switch. An exit without a stack switch uses the Application Stack. The ERESUME leaf index value is placed into RAX, the TCS pointer is placed in RBX and the AEP (see below) is placed into RCX to facilitate resuming the enclave after the exit.

Upon an AEX, the AEP (Asynchronous Exit Pointer) is loaded into the RIP. The AEP points to a trampoline code sequence which includes the ERESUME instruction that is later used to reenter the enclave.

The following bits of RFLAGS are cleared before RFLAGS is pushed onto the exit stack: CF, PF, AF, ZF, SF, OF, RF. The remaining bits are left unchanged.

15.2 State Saving by AEX

The State Save Area holds the processor state at the time of an AEX. To allow handling events within the enclave and re-entering it after an AEX, the SSA can be a stack of multiple SSA frames as illustrated in Figure 7.

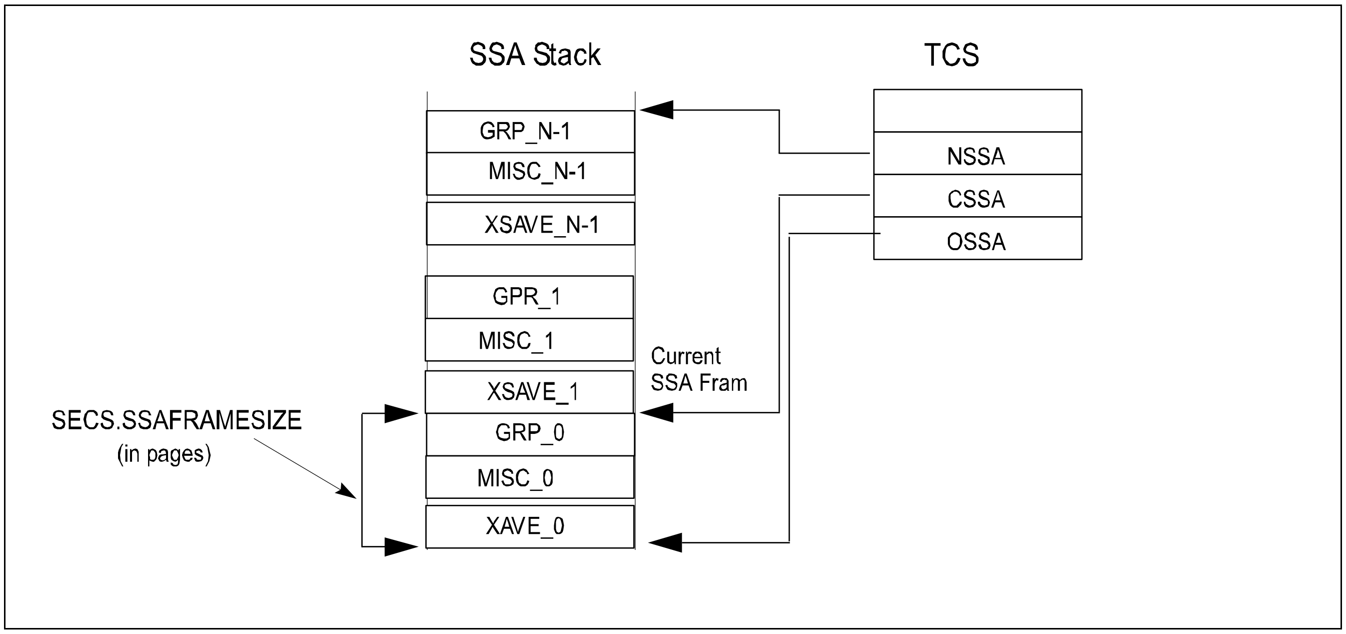

Figure 7 The SSA Stack

The location of the SSA frames to be used is controlled by the following variables in the TCS and the SECS:

- Size of a frame in the State Save Area (SECS.SSAFRAMESIZE): This defines the number of 4-KByte pages in a single frame in the State Save Area. The SSA frame size must be large enough to hold the GPR state, the XSAVE state, and the MISC state.
- Base address of the enclave (SECS.BASEADDR): This defines the enclave's base linear address from which the offset to the base of the SSA stack is calculated.
- Number of State Save Area Slots (TCS.NSSA): This defines the total number of slots (frames) in the State Save Area stack.
- Current State Save Area Slot (TCS.CSSA): This defines the slot to use on the next exit.
- State Save Area Offset (TCS.OSSA): This defines the offset of the base address of a set of State Save Area slots from the enclave’s base address.

When an AEX occurs, hardware selects the SSA frame to use by examining TCS.CSSA. Processor state is saved into the SSA frame (see Section 15.4) and loaded with a synthetic state (as described in Section 15.3.1) to avoid leaking secrets, RSP and RBP are restored to their values prior to enclave entry, and TCS.CSSA is incremented. As will be described later, if an exception takes the last slot, it will not be possible to reenter the enclave to handle the exception from within the enclave. A subsequent ERESUME restores the processor state from the current SSA frame and frees the SSA frame.

The format of the XSAVE section of SSA is identical to the format used by the XSAVE/XRSTOR instructions. On EENTER, CSSA must be less than NSSA, ensuring that there is at least one State Save Area slot available for exits. If there is no free SSA frame when executing EENTER, the entry will fail.

15.3 Synthetic State on Asynchronous Enclave Exit

15.3.1 Processor Synthetic State on Asynchronous Enclave Exit

Table 33 shows the synthetic state loaded on AEX. The values shown are the lower 32 bits when the processor is in 32 bit mode and 64 bits when the processor is in 64 bit mode.

Table 33 GPR, x87 Synthetic States on Asynchronous Enclave Exit

Register	Value
RAX	3 (ENCLU[3] is ERESUME).
RBX	Pointer to TCS of interrupted enclave thread.
RCX	AEP of interrupted enclave thread.
RDX, RSI, RDI	0.
RSP	Restored from SSA.uRSP.
RBP	Restored from SSA.uRBP.
R8-R15	0 in 64-bit mode; unchanged in 32-bit mode.
RIP	AEP of interrupted enclave thread.
RFLAGS	CF, PF, AF, ZF, SF, OF, RF bits are cleared. All other bits are left unchanged.
x87/SSE State	Unless otherwise listed here, all x87 and SSE state are set to the INIT state. The INIT state is the state that would be loaded by the XRSTOR instruction with bits 1:0 both set in the requested feature bit-mask (RFBM), and both clear in XSTATE_BV the XSAVE header.
FCW	On #MF exception: set to 037EH. On all other exits: set to 037FH.
FSW	On #MF exception: set to 8081H. On all other exits: set to 0H.
MXCSR	On #XM exception: set to 1F01H. On all other exits: set to 1FB0H.
CR2	If the event that caused the AEX is a #PF, and the #PF does not directly cause a VM exit, then the low 12 bits are cleared. If the #PF leads directly to a VM exit, CR2 is not updated (usual IA behavior). Note: The low 12 bits are not cleared if a #PF is encountered during the delivery of the EEE that caused the AEX. This is because the #PF was not the EEE.
FS, GS	Restored to values as of most recent EENTER/ERESUME.

15.3.2 Synthetic State for Extended Features

When CR4.OSXSAVE = 1, extended features (those controlled by XCR0[63:2]) are set to their respective INIT states when this corresponding bit of SECS.XFRM is set. The INIT state is the state that would be loaded by the XRSTOR instruction had the instruction mask and the XSTATE_BV field of the XSAVE header each contained the value XFRM. (When the AEX occurs in 32-bit mode, those features that do not exist in 32-bit mode are unchanged.)

15.3.3 Synthetic State for MISC Features

State represented by SECS.MISCSELECT might also be overridden by synthetic state after it has been saved into the SSA. State represented by MISCSELECT[0] is not overridden but if the exiting event is a page fault then lower 12 bits of CR2 are cleared.

15.4 AEX Flow

On Enclave Exiting Events (interrupts, exceptions, VM exits or SMIs), the processor state is securely saved inside the enclave, a synthetic state is loaded and the enclave is exited. The EEE then proceeds in the usual exit-defined fashion. The following sections describes the details of an AEX:

1. The exact processor state saved into the current SSA frame depends on whether the enclave is a 32-bit or a 64-bit enclave. In 32-bit mode (IA32_EFER.LMA = 0 || CS.L = 0), the low 32 bits of the legacy registers (EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI, EIP and EFLAGS) are stored. The upper 32 bits of the legacy registers and the 64-bit registers (R8 … R15) are not stored.

   In 64-bit mode (IA32_EFER.LMA = 1 && CS.L = 1), all 64 bits of the general processor registers (RAX, RBX, RCX, RDX, RSP, RBP, RSI, RDI, R8 … R15, RIP and RFLAGS) are stored.

   The state of those extended features specified by SECS.ATTRIBUTES.XFRM are stored into the XSAVE area of the current SSA frame. The layout of the x87 and XMM portions (the 1st 512 bytes) depends on the current values of IA32_EFER.LMA and CS.L:

   If IA32_EFER.LMA = 0 || CS.L = 0, the same format (32-bit) that XSAVE/FXSAVE uses with these values.

   If IA32_EFER.LMA = 1 && CS.L = 1, the same format (64-bit) that XSAVE/FXSAVE uses with these values when REX.W = 1.

   The cause of the AEX is saved in the EXITINFO field.

   The state of those miscellaneous features specified by SECS.MISCSELECT are stored into the MISC area of the current SSA frame.

   If CET was enabled in the enclave then the CET state of the enclave is saved in the CET state save area. If shadow stacks were enabled in the enclave then the SSP is also saved into the TCS.PREVSSP field.

2. Synthetic state is created for a number of processor registers to present an opaque view of the enclave state. Table 33 shows the values for GPRs, x87, SSE, FS, GS, Debug and performance monitoring on AEX. The synthetic state for other extended features (those controlled by XCR0[62:2]) is set to their respective INIT states when their corresponding bit of SECS.ATTRIBUTES.XFRM is set. The INIT state is that state as defined by the behavior of the XRSTOR instruction when HEADER.XSTATE_BV[n] is 0. Synthetic state of those miscellaneous features specified by SECS.MISCSELECT depends on the miscellaneous feature. There is no synthetic state required for the miscellaneous state controlled by SECS.MISCSELECT[0].
3. Any code and data breakpoints that were suppressed at the time of enclave entry are unsuppressed when exiting the enclave.
4. RFLAGS.TF is set to the value that it had at the time of the most recent enclave entry (except for the situation that the entry was opt-in for debug). In the SSA, RFLAGS.TF is set to 0.
5. RFLAGS.RF is set to 0 in the synthetic state. In the SSA, the value saved is the same as what would have been saved on stack in the non-SGX case (architectural value of RF). Thus, AEXs due to interrupts, traps, and code breakpoints save RF unmodified into SSA, while AEXs due to other faults save RF as 1 in the SSA.

   If the event causing AEX happened on intermediate iteration of a REP-prefixed instruction, then RF=1 is saved on SSA, irrespective of its priority.

6. Any performance monitoring activity (including PEBS) or profiling activity (LBR, Tracing using Intel PT) on the exiting thread that was suppressed due to the enclave entry on that thread is unsuppressed. Any counting that had been demoted from AnyThread counting to MyThread counting (on one logical processor) is promoted back to AnyThread counting.
6. The CET state of the enclosing application is restored to the state at the time of the most recent enclave entry and if CET indirect branch tracking was enabled then the indirect branch tracker is unsuppressed and moved to WAIT_FOR_ENDBRANCH state.

15.4.1 AEX Operational Detail

Temp Variables in AEX Operational Flow

Name	Type	Size (bits)	Description
TMP_RIP	Effective Address	32/64	Address of instruction at which to resume execution on ERESUME.
TMP_MODE64	binary	1	((IA32_EFER.LMA = 1) && (CS.L = 1)).
TMP_BRANCH_RECORD	LBR Record	2x64	From/To address to be pushed onto LBR stack.

The pseudo code in this section describes the internal operations that are executed when an AEX occurs in enclave mode. These operations occur just before the normal interrupt or exception processing occurs.

```
(* Save RIP for later use *)
TMP_RIP = Linear Address of Resume RIP
(* Is the processor in 64-bit mode? *)
TMP_MODE64 ← ((IA32_EFER.LMA = 1) && (CS.L = 1));

(* Save all registers, When saving EFLAGS, the TF bit is set to 0 and
   the RF bit is set to what would have been saved on stack in the non-SGX case *)

 IF (TMP_MODE64 = 0)
   THEN
        Save EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI, EFLAGS, EIP into the current SSA frame using
CR_GPR_PA; (* see Table 38 for list of CREGs used to describe internal operation within Intel SGX *)
        SSA.RFLAGS.TF ← 0;
   ELSE     (* TMP_MODE64 = 1 *)
        Save RAX, RBX, RCX, RDX, RSP, RBP, RSI, RDI, R8-R15, RFLAGS, RIP into the current SSA frame using
CR_GPR_PA;
        SSA.RFLAGS.TF ← 0;
FI;
Save FS and GS BASE into SSA using CR_GPR_PA;

(* store XSAVE state into the current SSA frame's XSAVE area using the physical addresses
   that were determined and cached at enclave entry time with CR_XSAVE_PAGE_i. *)
For each XSAVE state i defined by (SECS.ATTRIBUTES.XFRM[i] = 1, destination address cached in
CR_XSAVE_PAGE_i)
    SSA.XSAVE.i ← XSAVE_STATE_i;

(* Clear bytes 8 to 23 of XSAVE_HEADER, i.e. the next 16 bytes after XHEADER_BV *)
```

```
CR_XSAVE_PAGE_0.XHEADER_BV[191:64] ← 0;

(* Clear bits in XHEADER_BV[63:0] that are not enabled in ATTRIBUTES.XFRM *)

CR_XSAVE_PAGE_0.XHEADER_BV[63:0] ←
   CR_XSAVE_PAGE_0.XHEADER_BV[63:0] & SECS(CR_ACTIVE_SECS).ATTRIBUTES.XFRM;
   Apply synthetic state to GPRs, RFLAGS, extended features, etc.

(* Restore the RSP and RBP from the current SSA frame's GPR area using the physical address
   that was determined and cached at enclave entry time with CR_GPR_PA. *)
RSP ← CR_GPR_PA.URSP;
RBP ← CR_GPR_PA.URBP;

(* Restore the FS and GS *)
FS.selector ← CR_SAVE_FS.selector;
FS.base ← CR_SAVE_FS.base;
FS.limit ← CR_SAVE_FS.limit;
FS.access_rights ← CR_SAVE_FS.access_rights;
GS.selector ← CR_SAVE_GS.selector;
GS.base ← CR_SAVE_GS.base;
GS.limit ← CR_SAVE_GS.limit;
GS.access_rights ← CR_SAVE_GS.access_rights;

(* Examine exception code and update enclave internal states*)
exception_code ← Exception or interrupt vector;

(* Indicate the exit reason in SSA *)
IF (exception_code = (#DE OR #DB OR #BP OR #BR OR #UD OR #MF OR #AC OR #XM ))
   THEN
      CR_GPR_PA.EXITINFO.VECTOR ← exception_code;
      IF (exception code = #BP)
         THEN CR_GPR_PA.EXITINFO.EXIT_TYPE ← 6;
         ELSE CR_GPR_PA.EXITINFO.EXIT_TYPE ← 3;
      FI;
      CR_GPR_PA.EXITINFO.VALID ← 1;
   ELSE IF (exception_code is #PF or #GP )
      THEN
      (* Check SECS.MISCSELECT using CR_ACTIVE_SECS *)
      IF (SECS.MISCSELECT[0] is set)
         THEN
         CR_GPR_PA.EXITINFO.VECTOR ← exception_code;
         CR_GPR_PA.EXITINFO.EXIT_TYPE ← 3;
         IF (exception_code is #PF)
            THEN
               SSA.MISC.EXINFO. MADDR ← CR2;
               SSA.MISC.EXINFO.ERRCD ← PFEC;
               SSA.MISC.EXINFO.RESERVED ← 0;
         ELSE
            SSA.MISC.EXINFO. MADDR ← 0;
```

```
                SSA.MISC.EXINFO.ERRCD ← GPEC;
                SSA.MISC.EXINFO.RESERVED ← 0;
            FI;
            CR_GPR_PA.EXITINFO.VALID ← 1;
  ELSE IF (exception code is #CP)
     THEN
         IF (SECS.MISCSELECT[1] is set)
            THEN
                 CR_GPR_PA.EXITINFO.VECTOR ←  exception_code;
                 CR_GPR_PA.EXITINFO.EXIT_TYPE ←  3;
                 CR_GPR_PA.EXITINFO.VALID ←  1;
                 SSA.MISC.EXINFO. MADDR ←  0;
                 SSA.MISC.EXINFO.ERRCD ←  CPEC;
                 SSA.MISC.EXINFO.RESERVED ←  0;
             FI;
     FI;
   ELSE
       CR_GPR_PA.EXITINFO.VECTOR ← 0;
       CR_GPR_PA.EXITINFO.EXIT_TYPE ← 0
       CR_GPR_PA.REASON.VALID ← 0;
FI;

(* Execution will resume at the AEP *)
RIP ← CR_TCS_PA.AEP;

(* Set EAX to the ERESUME leaf index *)
EAX ← 3;

(* Put the TCS LA into RBX for later use by ERESUME *)
RBX ← CR_TCS_LA;

(* Put the AEP into RCX for later use by ERESUME *)
RCX ← CR_TCS_PA.AEP;

(* Increment the SSA frame # *)
CR_TCS_PA.CSSA ← CR_TCS_PA.CSSA + 1;

(* Restore XCR0 if needed *)
IF (CR4.OSXSAVE = 1)
   THEN   XCR0 ← CR_SAVE_XCR0; FI;

Un-suppress all code breakpoints that are outside ELRANGE
IF (CPUID.(EAX=12H, ECX=1):EAX[6]= 1)
   THEN
       IF (CR4.CET == 1 AND IA32_U_CET.SH_STK_EN == 1)
          THEN
              CR_CET_SAVE_AREA_PA.SSP ← SSP;
              CR_TCS_PA.PREVSSP ← SSP;
       FI;

       IF (CR4.CET == 1 AND IA32_U_CET.ENDBR_EN == 1)
          THEN
              CR_CET_SAVE_AREA_PA.TRACKER ← IA32_U_CET.TRACKER;
              CR_CET_SAVE_AREA_PA.SUPPRESS ← IA32_U_CET.SUPPRESS
       FI;
```

```
        (* restore enclosing applications CET state *)
        IA32_U_CET ← CR_SAVE_IA32_U_CET;

        IF (CPUID.(EAX=7, ECX=0):ECX[CET_SS])
             SSP ← CR_SAVE_SSP; FI;

        (* If indirect branch tracking enabled for enclosing application *)
        (* then move the tracker to wait_for_endbranch *)
        IF (CR4.CET == 1 AND IA32_U_CET.ENDBR_EN == 1)
            THEN
                IA32_U_CET.TRACKER ← WAIT_FOR_ENDBRANCH;
                IA32_U_CET.SUPPRESS ← 0;
        FI;
    FI;

(* Update the thread context to show not in enclave mode *)
CR_ENCLAVE_MODE ← 0;

(* Assure consistent translations. *)
Flush linear context including TLBs and paging-structure caches

IF (CR_DBGOPTIN = 0)
    THEN
        Un-suppress all breakpoints that overlap ELRANGE
        (* Clear suppressed breakpoint matches *)
        Restore suppressed breakpoint matches
        (* Restore TF *)
        RFLAGS.TF ← CR_SAVE_TF;
        Un-suppress monitor trap flag;
        Un-suppress branch recording facilities;
        Un-suppress all suppressed performance monitoring activity;
        Promote any sibling-thread counters that were demoted from AnyThread to MyThread during enclave en-
try back to AnyThread;
FI;

IF the "monitor trap flag" VM-execution control is 1
    THEN Pend MTF VM Exit at the end of exit; FI;

(* Clear low 12 bits of CR2 on #PF *)
IF (Exception code is #PF)
    THEN CR2 ← CR2 & ~0xFFF; FI;

(* end_of_flow *)
(* Execution continues with normal event processing. *)
```

16 SGX Instruction References

This chapter describes the supervisor and user level instructions provided by Intel® Software Guard Extensions (Intel® SGX). In general, various functionality is encoded as leaf functions within the ENCLS (supervisor), ENCLU (user), and the ENCLV (virtualization operation) instruction mnemonics. Different leaf functions are encoded by specifying an input value in the EAX register of the respective instruction mnemonic.

16.1 Intel® SGX Instruction Syntax and Operation

ENCLS, ENCLU and ENCLV instruction mnemonics for all leaf functions are covered in this section.

For all instructions, the value of CS.D is ignored; addresses and operands are 64 bits in 64-bit mode and are otherwise 32 bits. Aside from EAX specifying the leaf number as input, each instruction leaf may require all or some subset of the RBX/RCX/RDX as input parameters. Some leaf functions may return data or status information in one or more of the general purpose registers.

16.1.1 ENCLS Register Usage Summary

Table 34 summarizes the implicit register usage of supervisor mode enclave instructions.

Table 34 Register Usage of Privileged Enclave Instruction Leaf Functions

Instr. Leaf	EAX	RBX	RCX	RDX
ECREATE	00H (In)	PAGEINFO (In, EA)	EPCPAGE (In, EA)	
EADD	01H (In)	PAGEINFO (In, EA)	EPCPAGE (In, EA)	
EINIT	02H (In)	SIGSTRUCT (In, EA)	SECS (In, EA)	EINITTOKEN (In, EA)
EREMOVE	03H (In)		EPCPAGE (In, EA)	
EDBGRD	04H (In)	Result Data (Out)	EPCPAGE (In, EA)	
EDBGWR	05H (In)	Source Data (In)	EPCPAGE (In, EA)	
EEXTEND	06H (In)	SECS (In, EA)	EPCPAGE (In, EA)	
ELDB	07H (In)	PAGEINFO (In, EA)	EPCPAGE (In, EA)	VERSION (In, EA)
ELDU	08H (In)	PAGEINFO (In, EA)	EPCPAGE (In, EA)	VERSION (In, EA)
EBLOCK	09H (In)		EPCPAGE (In, EA)	
EPA	0AH (In)	PT_VA (In)	EPCPAGE (In, EA)	
EWB	0BH (In)	PAGEINFO (In, EA)	EPCPAGE (In, EA)	VERSION (In, EA)
ETRACK	0CH (In)		EPCPAGE (In, EA)	
EAUG	0DH (In)	PAGEINFO (In, EA)	EPCPAGE (In, EA)	
EMODPR	0EH (In)	SECINFO (In, EA)	EPCPAGE (In, EA)	
EMODT	0FH (In)	SECINFO (In, EA)	EPCPAGE (In, EA)	
ERDINFO	010H (In)	RDINFO (In, EA*)	EPCPAGE (In, EA)	
ETRACKC	011H (In)		EPCPAGE (In, EA)	
ELDBC	012H (In)	PAGEINFO (In, EA*)	EPCPAGE (In, EA)	VERSION (In, EA)

ELDUC	013H (In)	PAGEINFO (In, EA*)	EPCPAGE (In, EA)	VERSION (In, EA)
EA: Effective Address				

16.1.2 ENCLU Register Usage Summary

Table 35 summarizes the implicit register usage of user mode enclave instructions.

Table 35 Register Usage of Unprivileged Enclave Instruction Leaf Functions

Instr. Leaf	EAX	RBX	RCX	RDX
EREPORT	00H (In)	TARGETINFO (In, EA)	REPORTDATA (In, EA)	OUTPUTDATA (In, EA)
EGETKEY	01H (In)	KEYREQUEST (In, EA)	KEY (In, EA)	
EENTER	02H (In)	TCS (In, EA)	AEP (In, EA)	
	RBX.CSSA (Out)		Return (Out, EA)	
ERESUME	03H (In)	TCS (In, EA)	AEP (In, EA)	
EEXIT	04H (In)	Target (In, EA)	Current AEP (Out)	
EACCEPT	05H (In)	SECINFO (In, EA)	EPCPAGE (In, EA)	
EMODPE	06H (In)	SECINFO (In, EA)	EPCPAGE (In, EA)	
EACCEPTCOPY	07H (In)	SECINFO (In, EA)	EPCPAGE (In, EA)	EPCPAGE (In, EA)
EA: Effective Address				

16.1.3 ENCLV Register Usage Summary

Table 36 summarizes the implicit register usage of virtualization operation enclave instructions.

Table 36 Register Usage of Virtualization Operation Enclave Instruction Leaf Functions

Instr. Leaf	EAX	RBX	RCX	RDX
EDECVIRTCHILD	00H (In)	EPCPAGE (In, EA)	SECS (In, EA)	
EINCVIRTCHILD	01H (In)	EPCPAGE (In, EA)	SECS (In, EA)	
ESETCONTEXT	02H (In)		EPCPAGE (In, EA)	Context Value (In, EA)
EA: Effective Address				

16.1.4 Information and Error Codes

Information and error codes are reported by various instruction leaf functions to show an abnormal termination of the instruction or provide information which may be useful to the developer. Table 37 shows the various codes and the instruction which generated the code. Details of the meaning of the code is provided in the individual instruction.

Table 37 Error or Information Codes for Intel® SGX Instructions

Name	Value	Returned By
No Error	0	
SGX_INVALID_SIG_STRUCT	1	EINIT
SGX_INVALID_ATTRIBUTE	2	EINIT, EGETKEY
SGX_BLSTATE	3	EBLOCK
SGX_INVALID_MEASUREMENT	4	EINIT
SGX_NOTBLOCKABLE	5	EBLOCK
SGX_PG_INVLD	6	EBLOCK, ERDINFO, ETRACKC
SGX_EPC_PAGE_CONFLICT	7	EBLOCK, EMODPR, EMODT, ERDINFO , EDECVIRTCHILD, EINCVIRTCHILD, ELDBC, ELDUC, ESETCONTEXT, ETRACKC
SGX_INVALID_SIGNATURE	8	EINIT
SGX_MAC_COMPARE_FAIL	9	ELDB, ELDU, ELDBC, ELDUC
SGX_PAGE_NOT_BLOCKED	10	EWB
SGX_NOT_TRACKED	11	EWB, EACCEPT
SGX_VA_SLOT_OCCUPIED	12	EWB
SGX_CHILD_PRESENT	13	EWB, EREMOVE
SGX_ENCLAVE_ACT	14	EREMOVE
SGX_ENTRYEPOCH_LOCKED	15	EBLOCK
SGX_INVALID_EINITTOKEN	16	EINIT
SGX_PREV_TRK_INCMPL	17	ETRACK, ETRACKC
SGX_PG_IS_SECS	18	EBLOCK
SGX_PAGE_ATTRIBUTES_MISMATCH	19	EACCEPT, EACCEPTCOPY
SGX_PAGE_NOT_MODIFIABLE	20	EMODPR, EMODT
SGX_PAGE_NOT_DEBUGGABLE	21	EDBGRD, EDBGWR
SGX_INVALID_COUNTER	25	EDECVIRTCHILD
SGX_PG_NONEPC	26	ERDINFO

SGX_TRACK_NOT_REQUIRED	27	ETRACKC
SGX_INVALID_CPUSVN	32	EINIT, EGETKEY
SGX_INVALID_ISVSVN	64	EGETKEY
SGX_UNMASKED_EVENT	128	EINIT
SGX_INVALID_KEYNAME	256	EGETKEY

16.1.5 Internal CREGs

The CREGs as shown in Table 38 are hardware specific registers used in this document to indicate values kept by the processor. These values are used while executing in enclave mode or while executing an Intel SGX instruction. These registers are not software visible and are implementation specific. The values in Table 38 appear at various places in the pseudo-code of this document. They are used to enhance understanding of the operations.

Table 38 List of Internal CREG

Name	Size (Bits)	Scope
CR_ENCLAVE_MODE	1	LP
CR_DBGOPTIN	1	LP
CR_TCS_LA	64	LP
CR_TCS_PA	64	LP
CR_ACTIVE_SECS	64	LP
CR_ELRANGE	128	LP
CR_SAVE_TF	1	LP
CR_SAVE_FS	64	LP
CR_GPR_PA	64	LP
CR_XSAVE_PAGE_n	64	LP
CR_SAVE_DR7	64	LP
CR_SAVE_PERF_GLOBAL_CTRL	64	LP
CR_SAVE_DEBUGCTL	64	LP
CR_SAVE_PEBS_ENABLE	64	LP
CR_CPUSVN	128	PACKAGE
CR_SGXOWNEREPOCH	128	PACKAGE
CR_SAVE_XCR0	64	LP

CR_SGX_ATTRIBUTES_MASK	128	LP
CR_PAGING_VERSION	64	PACKAGE
CR_VERSION_THRESHOLD	64	PACKAGE
CR_NEXT_EID	64	PACKAGE
CR_BASE_PK	128	PACKAGE
CR_SEAL_FUSES	128	PACKAGE
CR_CET_SAVE_AREA_PA	64	LP
CR_ENCLAVE_SS_TOKEN_PA	64	LP
CR_SAVE_IA32_U_CET	64	LP
CR_SAVE_SSP	64	LP

16.1.6 Concurrent Operation Restrictions

Under certain conditions, Intel SGX disallows certain leaf functions from operating concurrently. Listed below are some examples of concurrency that are not allowed.

- For example, Intel SGX disallows the following leafs to concurrently operate on the same EPC page.
  - ECREATE, EADD, and EREMOVE are not allowed to operate on the same EPC page concurrently with themselves.
  - EADD, EEXTEND, and EINIT leaves are not allowed to operate on the same SECS concurrently.
- Intel SGX disallows the EREMOVE leaf from removing pages from an enclave that is in use.
- Intel SGX disallows entry (EENTER and ERESUME) to an enclave while a page from that enclave is being removed.

When disallowed operation is detected, a leaf function may do one of the following:

- Return an SGX_EPC_PAGE_CONFLICT error code in RAX.
- Cause a #GP(0) exception.

To prevent such exceptions, software must serialize leaf functions or prevent these leaf functions from accessing the same EPC page.

16.1.6.1 Concurrency Tables of Intel® SGX Instructions

The tables below detail the concurrent operation restrictions of all SGX leaf functions. For each leaf function, the table has a separate line for each of the EPC pages the leaf function accesses.

For each such EPC page, the base concurrency requirements are detailed as follows:

- **Exclusive Access** means that no other leaf function that requires either shared or exclusive access to the same EPC page may be executed concurrently. For example, EADD requires an exclusive access to the target page it accesses.
- **Shared Access** means that no other leaf function that requires an exclusive access to the same EPC page may be executed concurrently. Other leaf functions that require shared access may run concurrently. For example, EADD requires a shared access to the SECS page it accesses.
- **Concurrent Access** means that any other leaf function that requires any access to the same EPC page may be executed concurrently. For example, EGETKEY has no concurrency requirements for the KEYREQUEST page.

In addition to the base concurrency requirements, additional concurrency requirements are listed, which apply only to specific sets of leaf functions. For example, there are additional requirements that apply for EADD, EXTEND and EINIT. EADD and EEXTEND can't execute concurrently on the same SECS page.

The tables also detail the leaf function's behavior when a conflict happens, i.e., a concurrency requirement is not met. In this case, the leaf function may return an SGX_EPC_PAGE_CONFLICT error code in RAX, or it may cause an exception. In addition, the tables detail those conflicts where a VM Exit may be triggered, and list the Exit Qualification code that is provided in such cases.

Table 39 Base Concurrency Restrictions

Leaf	Parameter		Base Concurrency Restrictions		
			Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EACCEPT	Target	[DS:RCX]	Shared	#GP	
	SECINFO	[DS:RBX]	Concurrent		
EACCEPTCOPY	Target	[DS:RCX]	Concurrent		
	Source	[DS:RDX]	Concurrent		
	SECINFO	[DS:RBX]	Concurrent		
EADD	Target	[DS:RCX]	Exclusive	#GP	EPC_PAGE_CONFLICT_EXCEPTION
	SECS	[DS:RBX]PAGEINFO.SECS	Shared	#GP	
EAUG	Target	[DS:RCX]	Exclusive	#GP	EPC_PAGE_CONFLICT_EXCEPTION
	SECS	[DS:RBX]PAGEINFO.SECS	Shared	#GP	
EBLOCK	Target	[DS:RCX]	Shared	SGX_EPC_PAGE_CONFLICT	
ECREATE	SECS	[DS:RCX]	Exclusive	#GP	EPC_PAGE_CONFLICT_EXCEPTION
EDBGRD	Target	[DS:RCX]	Shared	#GP	
EDBGWR	Target	[DS:RCX]	Shared	#GP	
EDECVIRTCHILD	Target	[DS:RBX]	Shared	SGX_EPC_PAGE_CONFLICT	
	SECS	[DS:RCX]	Concurrent		
EENTERTCS	SECS	[DS:RBX]	Shared	#GP	
EEXIT			Concurrent		
EEXTEND	Target	[DS:RCX]	Shared	#GP	

	SECS	[DS:RBX]	Concurrent		
EGETKEY	KEYREQUEST	[DS:RBX]	Concurrent		
	OUTPUTDATA	[DS:RCX]	Concurrent		
EINCVIRTCHILD	Target	[DS:RBX]	Shared	SGX_EPC_PAGE _CONFLICT	
	SECS	[DS:RCX]	Concurrent		
EINIT	SECS	[DS:RCX]	Shared	#GP	
ELDB/ELDU	Target	[DS:RCX]	Exclusive	#GP	EPC_PAGE_CONFLICT_EXCEPTION
	VA	[DS:RDX]	Shared	#GP	
	SECS	[DS:RBX]PAGEINFO. SECS	Shared	#GP	
EDLBC/ELDUC	Target	[DS:RCX]	Exclusive	SGX_EPC_PAGE _CONFLICT	EPC_PAGE_CONFLICT_ERROR
	VA	[DS:RDX]	Shared	SGX_EPC_PAGE _CONFLICT	
	SECS	[DS:RBX]PAGEINFO. SECS	Shared	SGX_EPC_PAGE _CONFLICT	
EMODPE	Target	[DS:RCX]	Concurrent		
	SECINFO	[DS:RBX]	Concurrent		
EMODPR	Target	[DS:RCX]	Shared	#GP	
EMODT	Target	[DS:RCX]	Exclusive	SGX_EPC_PAGE _CONFLICT	EPC_PAGE_CONFLICT_ERROR
EPA	VA	[DS:RCX]	Exclusive	#GP	EPC_PAGE_CONFLICT_EXCEPTION
ERDINFO	Target	[DS:RCX]	Shared	SGX_EPC_PAGE _CONFLICT	
EREMOVE	Target	[DS:RCX]	Exclusive	#GP	EPC_PAGE_CONFLICT_EXCEPTION
EREPORT	TARGETINFO	[DS:RBX]	Concurrent		
	REPORTDATA	[DS:RCX]	Concurrent		
	OUTPUTDATA	[DS:RDX]	Concurrent		
ERESUME	TCS	[DS:RBX]	Shared	#GP	
ESETCONTEXT	SECS	[DS:RCX]	Shared	SGX_EPC_PAGE _CONFLICT	
ETRACK	SECS	[DS:RCX]	Shared	#GP	
ETRACKC	Target	[DS:RCX]	Shared	SGX_EPC_PAGE _CONFLICT	

	SECS	Implicit	Concurrent		
EWB	Source	[DS:RCX]	Exclusive	#GP	EPC_PAGE_CONFLICT_EXCEPTION
	VA	[DS:RDX]	Shared	#GP	

Table 40 Additional Concurrency Restrictions

Leaf	Parameter		Additional Concurrency Restrictions					
			vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
			Access	On Conflict	Access	On Conflict	Access	On Conflict
EACCEPT	Target	[DS:RCX]	Exclusive	#GP	Concurrent		Concurrent	
	SECINFO	[DS:RBX]	Concurrent		Concurrent		Concurrent	
EACCEPTCOPY	Target	[DS:RCX]	Exclusive	#GP	Concurrent		Concurrent	
	Source	[DS:RDX]	Concurrent		Concurrent		Concurrent	
	SECINFO	[DS:RBX]	Concurrent		Concurrent		Concurrent	
EADD	Target	[DS:RCX]	Concurrent		Concurrent		Concurrent	
	SECS	[DS:RBX]PAGEINFO. SECS	Concurrent		Exclusive	#GP	Concurrent	
EAUG	Target	[DS:RCX]	Concurrent		Concurrent		Concurrent	
	SECS	[DS:RBX]PAGEINFO. SECS	Concurrent		Concurrent		Concurrent	
EBLOCK	Target	[DS:RCX]	Concurrent		Concurrent		Concurrent	
ECREATE	SECS	[DS:RCX]	Concurrent		Concurrent		Concurrent	
EDBGRD	Target	[DS:RCX]	Concurrent		Concurrent		Concurrent	
EDBGWR	Target	[DS:RCX]	Concurrent		Concurrent		Concurrent	
EDECVIRTCHILD	Target	[DS:RBX]	Concurrent		Concurrent		Concurrent	
	SECS	[DS:RCX]	Concurrent		Concurrent		Concurrent	
EENTERTCS	SECS	[DS:RBX]	Concurrent		Concurrent		Concurrent	
EEXIT			Concurrent		Concurrent		Concurrent	

EEXTEND	Target	[DS:RCX]	Concurrent		Concurrent		Concurrent	
	SECS	[DS:RBX]	Concurrent		Exclusive	#GP	Concurrent	
EGETKEY	KEYREQUEST	[DS:RBX]	Concurrent		Concurrent		Concurrent	
	OUTPUTDATA	[DS:RCX]	Concurrent		Concurrent		Concurrent	
EINCVIRTCHILD	Target	[DS:RBX]	Concurrent		Concurrent		Concurrent	
	SECS	[DS:RCX]	Concurrent		Concurrent		Concurrent	
EINIT	SECS	[DS:RCX]	Concurrent		Exclusive	#GP	Concurrent	
ELDB/ELDU	Target	[DS:RCX]	Concurrent		Concurrent		Concurrent	
	VA	[DS:RDX]	Concurrent		Concurrent		Concurrent	
	SECS	[DS:RBX]PAGEINFO. SECS	Concurrent		Concurrent		Concurrent	
EDLBC/ELDUC	Target	[DS:RCX]	Concurrent		Concurrent		Concurrent	
	VA	[DS:RDX]	Concurrent		Concurrent		Concurrent	
	SECS	[DS:RBX]PAGEINFO. SECS	Concurrent		Concurrent		Concurrent	
EMODPE	Target	[DS:RCX]	Exclusive	#GP	Concurrent		Concurrent	
	SECINFO	[DS:RBX]	Concurrent		Concurrent		Concurrent	
EMODPR	Target	[DS:RCX]	Exclusive	SGX_EPC_ PAGE_CO NFLICT	Concurrent		Concurrent	
EMODT	Target	[DS:RCX]	Exclusive	SGX_EPC_ PAGE_CO NFLICT	Concurrent		Concurrent	
EPA	VA	[DS:RCX]	Concurrent		Concurrent		Concurrent	
ERDINFO	Target	[DS:RCX]	Concurrent		Concurrent		Concurrent	
EREMOVE	Target	[DS:RCX]	Concurrent		Concurrent		Concurrent	
EREPORT	TARGETINFO	[DS:RBX]	Concurrent		Concurrent		Concurrent	
	REPORTDATA	[DS:RCX]	Concurrent		Concurrent		Concurrent	
	OUTPUTDATA	[DS:RDX]	Concurrent		Concurrent		Concurrent	
ERESUME	TCS	[DS:RBX]	Concurrent		Concurrent		Concurrent	
ESETCONTEXT	SECS	[DS:RCX]	Concurrent		Concurrent		Concurrent	

ETRACK	SECS	[DS:RCX]	Concurrent		Concurrent		Exclusive	SGX_EPC_PAGE_CONFLICT[4]
ETRACKC	Target	[DS:RCX]	Concurrent		Concurrent		Concurrent	
	SECS	Implicit	Concurrent		Concurrent		Exclusive	SGX_EPC_PAGE_CONFLICT[1]
EWB	Source	[DS:RCX]	Concurrent		Concurrent		Concurrent	
	VA	[DS:RDX]	Concurrent		Concurrent		Concurrent	

16.2 Intel® SGX Instruction Reference

[4]SGX_CONFLICT VM Exit Qualification =TRACKING_RESOURCE_CONFLICT.

ENCLS—Execute an Enclave System Function of Specified Leaf Number

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 01 CF ENCLS	NP	V/V	NA	This instruction is used to execute privileged Intel SGX leaf functions that are used for managing and debugging the enclaves.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Implicit Register Operands
NP	NA	NA	NA	See Section 16.3

Description

The ENCLS instruction invokes the specified privileged Intel SGX leaf function for managing and debugging enclaves. Software specifies the leaf function by setting the appropriate value in the register EAX as input. The registers RBX, RCX, and RDX have leaf-specific purpose, and may act as input, as output, or may be unused. In 64-bit mode, the instruction ignores upper 32 bits of the RAX register.

The ENCLS instruction produces an invalid-opcode exception (#UD) if CR0.PE = 0 or RFLAGS.VM = 1, or if it is executed in system-management mode (SMM). Additionally, any attempt to execute the instruction when CPL > 0 results in #UD. The instruction produces a general-protection exception (#GP) if CR0.PG = 0 or if an attempt is made to invoke an undefined leaf function.

In VMX non-root operation, execution of ENCLS may cause a VM exit if the "enable ENCLS exiting" VM-execution control is 1. In this case, execution of individual leaf functions of ENCLS is governed by the ENCLS-exiting bitmap field in the VMCS. Each bit in that field corresponds to the index of an ENCLS leaf function (as provided in EAX).

Software in VMX root operation can thus intercept the invocation of various ENCLS leaf functions in VMX non-root operation by setting the "enable ENCLS exiting" VM-execution control and setting the corresponding bits in the ENCLS-exiting bitmap.

Addresses and operands are 32 bits outside 64-bit mode (IA32_EFER.LMA = 0 || CS.L = 0) and are 64 bits in 64-bit mode (IA32_EFER.LMA = 1 || CS.L = 1). CS.D value has no impact on address calculation. The DS segment is used to create linear addresses.

Segment override prefixes and address-size override prefixes are ignored, and is the REX prefix in 64-bit mode.

Operation

```
IF TSX_ACTIVE
    THEN GOTO TSX_ABORT_PROCESSING; FI;

IF CR0.PE = 0 or RFLAGS.VM = 1 or in SMM or CPUID.SGX_LEAF.0:EAX.SE1 = 0
    THEN #UD; FI;

IF (CPL > 0)
    THEN #UD; FI;

IF in VMX non-root operation and the "enable ENCLS exiting" VM-execution control is 1
    THEN
        IF EAX < 63 and ENCLS_exiting_bitmap[EAX] = 1 or EAX> 62 and ENCLS_exiting_bitmap[63] = 1
            THEN VM exit;
        FI;
```

```
FI;
IF IA32_FEATURE_CONTROL.LOCK = 0 or IA32_FEATURE_CONTROL.SGX_ENABLE = 0
    THEN #GP(0); FI;

IF (EAX is an invalid leaf number)
    THEN #GP(0); FI;

IF CR0.PG = 0
    THEN #GP(0); FI;

(* DS must not be an expanded down segment *)
IF not in 64-bit mode and DS.Type is expand-down data
    THEN #GP(0); FI;

Jump to leaf specific flow
```

Flags Affected

See individual leaf functions

Protected Mode Exceptions

#UD	If any of the LOCK/OSIZE/REP/VEX prefix is used.
	If current privilege level is not 0.
	If CPUID.(EAX=12H,ECX=0):EAX.SGX1 [bit 0] = 0.
	If logical processor is in SMM.
#GP(0)	If IA32_FEATURE_CONTROL.LOCK = 0.
	If IA32_FEATURE_CONTROL.SGX_ENABLE = 0.
	If input value in EAX encodes an unsupported leaf.
	If data segment expand down.
	If CR0.PG=0.

Real-Address Mode Exceptions

#UD	ENCLS is not recognized in real mode.

Virtual-8086 Mode Exceptions

#UD	ENCLS is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD	If any of the LOCK/OSIZE/REP/VEX prefix is used.
	If current privilege level is not 0.
	If CPUID.(EAX=12H,ECX=0):EAX.SGX1 [bit 0] = 0.
	If logical processor is in SMM.
#GP(0)	If IA32_FEATURE_CONTROL.LOCK = 0.
	If IA32_FEATURE_CONTROL.SGX_ENABLE = 0.
	If input value in EAX encodes an unsupported leaf.

ENCLU—Execute an Enclave User Function of Specified Leaf Number

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 01 D7 ENCLU	NP	V/V	NA	This instruction is used to execute non-privileged Intel SGX leaf functions.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Implicit Register Operands
NP	NA	NA	NA	See Section 16.4

Description

The ENCLU instruction invokes the specified non-privileged Intel SGX leaf functions. Software specifies the leaf function by setting the appropriate value in the register EAX as input. The registers RBX, RCX, and RDX have leaf-specific purpose, and may act as input, as output, or may be unused. In 64-bit mode, the instruction ignores upper 32 bits of the RAX register.

The ENCLU instruction produces an invalid-opcode exception (#UD) if CR0.PE = 0 or RFLAGS.VM = 1, or if it is executed in system-management mode (SMM). Additionally, any attempt to execute this instruction when CPL < 3 results in #UD. The instruction produces a general-protection exception (#GP) if either CR0.PG or CR0.NE is 0, or if an attempt is made to invoke an undefined leaf function. The ENCLU instruction produces a device not available exception (#NM) if CR0.TS = 1.

Addresses and operands are 32 bits outside 64-bit mode (IA32_EFER.LMA = 0 or CS.L = 0) and are 64 bits in 64-bit mode (IA32_EFER.LMA = 1 and CS.L = 1). CS.D value has no impact on address calculation. The DS segment is used to create linear addresses.

Segment override prefixes and address-size override prefixes are ignored, as is the REX prefix in 64-bit mode.

Operation

```
IN_64BIT_MODE← 0;
IF TSX_ACTIVE
    THEN GOTO TSX_ABORT_PROCESSING; FI;

(* If enclosing app has CET indirect branch tracking enabled then  if it is not ERESUME leaf cause a #CP fault  *)
(* If the ERESUME is not successful it will leave tracker in WAIT_FOR_ENDBRANCH  *)
TRACKER = (CPL == 3) ? IA32_U_CET.TRACKER : IA32_S_CET.TRACKER
IF EndbranchEnabledAndNotSuppressed(CPL) and  TRACKER = WAIT_FOR_ENDBRANCH and
    (EAX != ERESUME or CR0.TS or (in SMM) or (CPUID.SGX_LEAF.0:EAX.SE1 = 0) or (CPL < 3))
        THEN
            Handle_CET State machine violation – see section 3.6
        FI;

IF CR0.PE= 0 or RFLAGS.VM = 1 or in SMM or CPUID.SGX_LEAF.0:EAX.SE1 = 0
    THEN #UD; FI;

IF CR0.TS = 1
```

```
    THEN #NM; FI;

IF CPL < 3
    THEN #UD; FI;

IF IA32_FEATURE_CONTROL.LOCK = 0 or IA32_FEATURE_CONTROL.SGX_ENABLE = 0
    THEN #GP(0); FI;

IF EAX is invalid leaf number
    THEN #GP(0); FI;

IF CR0.PG = 0 or CR0.NE = 0
    THEN #GP(0); FI;

IN_64BIT_MODE ← IA32_EFER.LMA AND CS.L ? 1 : 0;
(* Check not in 16-bit mode and DS is not a 16-bit segment *)
IF not in 64-bit mode and (CS.D = 0 or DS.B = 0)
    THEN #GP(0); FI;

IF CR_ENCLAVE_MODE = 1 and (EAX = 2 or EAX = 3) (* EENTER or ERESUME *)
    THEN #GP(0); FI;

IF CR_ENCLAVE_MODE = 0 and (EAX = 0 or EAX = 1 or EAX = 4 or EAX = 5 or            EAX = 6 or EAX = 7)
(* EREPORT, EGETKEY, EEXIT, EACCEPT, EMODPE, or EACCEPTCOPY *)
    THEN #GP(0); FI;

Jump to leaf specific flow
```

Flags Affected

See individual leaf functions

Protected Mode Exceptions

#UD	If any of the LOCK/OSIZE/REP/VEX prefix is used.
	If current privilege level is not 3.
	If CPUID.(EAX=12H,ECX=0):EAX.SGX1 [bit 0] = 0.
	If logical processor is in SMM.
#GP(0)	If IA32_FEATURE_CONTROL.LOCK = 0.
	If IA32_FEATURE_CONTROL.SGX_ENABLE = 0.
	If input value in EAX encodes an unsupported leaf.
	If input value in EAX encodes EENTER/ERESUME and ENCLAVE_MODE = 1.
	If input value in EAX encodes EGETKEY/EREPORT/EEXIT/EACCEPT/EACCEPTCOPY/EMODPE and ENCLAVE_MODE = 0.
	If operating in 16-bit mode.
	If data segment is in 16-bit mode.
	If CR0.PG = 0 or CR0.NE= 0.
#NM	If CR0.TS = 1.

Real-Address Mode Exceptions

#UD	ENCLS is not recognized in real mode.

Virtual-8086 Mode Exceptions

#UD	ENCLS is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD	If any of the LOCK/OSIZE/REP/VEX prefix is used.
	If current privilege level is not 3.
	If CPUID.(EAX=12H,ECX=0):EAX.SGX1 [bit 0] = 0.
	If logical processor is in SMM.
#GP(0)	If IA32_FEATURE_CONTROL.LOCK = 0.
	If IA32_FEATURE_CONTROL.SGX_ENABLE = 0.
	If input value in EAX encodes an unsupported leaf.
	If input value in EAX encodes EENTER/ERESUME and ENCLAVE_MODE = 1.
	If input value in EAX encodes EGETKEY/EREPORT/EEXIT/EACCEPT/EACCEPTCOPY/EMODPE and ENCLAVE_MODE = 0.
	If CR0.NE= 0.
#NM	If CR0.TS = 1.

ENCLV—Execute an Enclave VMM Function of Specified Leaf Number

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 01 C0 ENCLV	NP	V/V	NA	This instruction is used to execute privileged SGX leaf functions that are reserved for VMM use. They are used for managing the enclaves.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Implicit Register Operands
NP	NA	NA	NA	See Section 16.3

Description

The ENCLV instruction invokes the virtualization SGX leaf functions for managing enclaves in a virtualized environment. Software specifies the leaf function by setting the appropriate value in the register EAX as input. The registers RBX, RCX, and RDX have leaf-specific purpose, and may act as input, as output, or may be unused. In non 64-bit mode, the instruction ignores upper 32 bits of the RAX register.

The ENCLV instruction produces an invalid-opcode exception (#UD) if CR0.PE = 0 or RFLAGS.VM = 1, if it is executed in system-management mode (SMM), or not in VMX operation. Additionally, any attempt to execute the instruction when CPL > 0 results in #UD. The instruction produces a general-protection exception (#GP) if CR0.PG = 0 or if an attempt is made to invoke an undefined leaf function.

Software in VMX root mode of operation can enable execution of the ENCLV instruction in VMX non-root mode by setting enable ENCLV execution control in the VMCS. If enable ENCLV execution control in the VMCS is clear, execution of the ENCLV instruction in VMX non-root mode results in #UD.

When execution of ENCLV instruction in VMX non-root mode is enabled, software in VMX root operation can intercept the invocation of various ENCLV leaf functions in VMX non-root operation by setting the corresponding bits in the ENCLV-exiting bitmap.

Addresses and operands are 32 bits in 32-bit mode (IA32_EFER.LMA == 0 || CS.L == 0) and are 64 bits in 64-bit mode (IA32_EFER.LMA == 1 && CS.L == 1). CS.D value has no impact on address calculation.

Segment override prefixes and address-size override prefixes are ignored, as is the REX prefix in 64-bit mode.

Operation

```
IF TSX_ACTIVE
    THEN GOTO TSX_ABORT_PROCESSING; FI;

IF CR0.PE = 0 or RFLAGS.VM = 1 or in SMM or CPUID.SGX_LEAF.0:EAX.OSS = 0
    THEN #UD; FI;

IF in VMX non-root operation and IA32_EFER.LMA = 1 and CS.L = 1
    THEN #UD; FI;

IF (CPL > 0)
    THEN #UD; FI;

IF in VMX non-root operation
     IF “enable ENCLV exiting“ VM-execution control is 1
          THEN
```

```
                IF EAX < 63 and ENCLV_exiting_bitmap[EAX] = 1 or EAX> 62 and ENCLV_exiting_bitmap[63] = 1
                        THEN VM exit;
                FI;
        ELSE
            #UD; FI;
FI;

IF IA32_FEATURE_CONTROL.LOCK = 0 or IA32_FEATURE_CONTROL.SGX_ENABLE = 0
    THEN #GP(0); FI;

IF (EAX is an invalid leaf number)
    THEN #GP(0); FI;

IF CR0.PG = 0
    THEN #GP(0); FI;

(* DS must not be an expanded down segment *)
IF not in 64-bit mode and DS.Type is expand-down data
    THEN #GP(0); FI;

Jump to leaf specific flow
```

Flags Affected

See individual leaf functions.

Protected Mode Exceptions

#UD	If any of the LOCK/OSIZE/REP/VEX prefix is used.
	If current privilege level is not 0.
	If CPUID.(EAX=12H,ECX=0):EAX.OSS [bit 5] = 0.
	If logical processor is in SMM.
#GP(0)	If IA32_FEATURE_CONTROL.LOCK = 0.
	If IA32_FEATURE_CONTROL.SGX_ENABLE = 0.
	If input value in EAX encodes an unsupported leaf.
	If data segment expand down.
	If CR0.PG=0.

Real-Address Mode Exceptions

#UD	ENCLV is not recognized in real mode.

Virtual-8086 Mode Exceptions

#UD	ENCLV is not recognized in virtual-8086 mode.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#UD	If any of the LOCK/OSIZE/REP/VEX prefix is used.
	If current privilege level is not 0.
	If CPUID.(EAX=12H,ECX=0):EAX.OSS [bit 5] = 0.
	If logical processor is in SMM.
#GP(0)	If IA32_FEATURE_CONTROL.LOCK = 0.
	If IA32_FEATURE_CONTROL.SGX_ENABLE = 0.
	If input value in EAX encodes an unsupported leaf.

16.3 Intel® SGX System Leaf Function Reference

Leaf functions available with the ENCLS instruction mnemonic are covered in this section. In general, each instruction leaf requires EAX to specify the leaf function index and/or additional implicit registers specifying leaf-specific input parameters. An instruction operand encoding table provides details of each implicit register usage and associated input/output semantics.

In many cases, an input parameter specifies an effective address associated with a memory object inside or outside the EPC, the memory addressing semantics of these memory objects are also summarized in a separate table.

EADD—Add a Page to an Uninitialized Enclave

Opcode/ Instruction	**Op/En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
EAX = 01H ENCLS[EADD]	IR	V/V	SGX1	This leaf function adds a page to an uninitialized enclave.

Instruction Operand Encoding

Op/En	EAX	RBX	RCX
IR	EADD (In)	Address of a PAGEINFO (In)	Address of the destination EPC page (In)

Description

This leaf function copies a source page from non-enclave memory into the EPC, associates the EPC page with an SECS page residing in the EPC, and stores the linear address and security attributes in EPCM. As part of the association, the enclave offset and the security attributes are measured and extended into the SECS.MRENCLAVE. This instruction can only be executed when current privilege level is 0.

RBX contains the effective address of a PAGEINFO structure while RCX contains the effective address of an EPC page. The table below provides additional information on the memory parameter of EADD leaf function.

EADD Memory Parameter Semantics

PAGEINFO	PAGEINFO.SECS	PAGEINFO.SRCPGE	PAGEINFO.SECINFO	EPCPAGE
Read access permitted by Non Enclave	Read/Write access permitted by Enclave	Read access permitted by Non Enclave	Read access permitted by Non Enclave	Write access permitted by Enclave

EADD Faulting Conditions

The instruction faults if any of the following:

The operands are not properly aligned.	Unsupported security attributes are set.
Refers to an invalid SECS.	Reference is made to an SECS that is locked by another thread.
The EPC page is locked by another thread.	RCX does not contain an effective address of an EPC page.
The EPC page is already valid.	If security attributes specifies a TCS and the source page specifies unsupported TCS values or fields.
The SECS has been initialized.	The specified enclave offset is outside of the enclave address space.

Concurrency Restrictions

Base Concurrency Restrictions of EADD

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EADD	Target [DS:RCX]	Exclusive	#GP	EPC_PAGE_CONFLICT_EXCEPTION
	SECS [DS:RBX]PAGEINFO.SECS	Shared	#GP	

Additional Concurrency Restrictions of EADD

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EADD	Target [DS:RCX]	Concurrent		Concurrent		Concurrent	
	SECS [DS:RBX]PAGE-INFO.SECS	Concurrent		Exclusive	#GP	Concurrent	

Operation

Temp Variables in EADD Operational Flow

Name	Type	Size (bits)	Description
TMP_SRCPGE	Effective Address	32/64	Effective address of the source page.
TMP_SECS	Effective Address	32/64	Effective address of the SECS destination page.
TMP_SECINFO	Effective Address	32/64	Effective address of an SECINFO structure which contains security attributes of the page to be added.
SCRATCH_SECINFO	SECINFO	512	Scratch storage for holding the contents of DS:TMP_SECINFO.
TMP_LINADDR	Unsigned Integer	64	Holds the linear address to be stored in the EPCM and used to calculate TMP_ENCLAVEOFFSET.
TMP_ENCLAVEOFFSET	Enclave Offset	64	The page displacement from the enclave base address.
TMPUPDATEFIELD	SHA256 Buffer	512	Buffer used to hold data being added to TMP_SECS.MRENCLAVE.

```
IF (DS:RBX is not 32Byte Aligned)
    THEN #GP(0); FI;

IF (DS:RCX is not 4KByte Aligned)
    THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;

TMP_SRCPGE ← DS:RBX.SRCPGE;
TMP_SECS ← DS:RBX.SECS;
TMP_SECINFO ← DS:RBX.SECINFO;
TMP_LINADDR ← DS:RBX.LINADDR;

IF (DS:TMP_SRCPGE is not 4KByte aligned or DS:TMP_SECS is not 4KByte aligned or
    DS:TMP_SECINFO is not 64Byte aligned or TMP_LINADDR is not 4KByte aligned)
    THEN #GP(0); FI;

IF (DS:TMP_SECS does not resolve within an EPC)
    THEN #PF(DS:TMP_SECS); FI;

SCRATCH_SECINFO ← DS:TMP_SECINFO;

(* Check for misconfigured SECINFO flags*)
IF (SCRATCH_SECINFO reserved fields are not zero or
    ! (SCRATCH_SECINFO.FLAGS.PT is PT_REG or SCRATCH_SECINFO.FLAGS.PT is PT_TCS or
      (SCRATCH_SECINFO.FLAGS.PT is PT_SS_FIRST and CPUID.(EAX=12H, ECX=1):EAX[6] = 1) or
      (SCRATCH_SECINFO.FLAGS.PT is PT_SS_REST and CPUID.(EAX=12H, ECX=1):EAX[6] = 1)))
    THEN #GP(0); FI;
```

```
(* If PT_SS_FIRST/PT_SS_REST page types are requested then CR4.CET must be 1 *)
IF ( (SCRATCH_SECINFO.FLAGS.PT is PT_SS_FIRST OR
      SCRATCH_SECINFO.FLAGS.PT is PT_SS_REST) AND CR4.CET == 0)
     THEN #GP(0); FI;

(* Check the EPC page for concurrency *)
IF (EPC page is not available for EADD)
   THEN
      IF (<<VMX non-root operation>> AND <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>>)
         THEN
            VMCS.Exit_reason ← SGX_CONFLICT;
            VMCS.Exit_qualification.code ← EPC_PAGE_CONFLICT_EXCEPTION;
            VMCS.Exit_qualification.error ← 0;
            VMCS.Guest-physical_address ← << translation of DS:RCX produced by paging >>;
            VMCS.Guest-linear_address ← DS:RCX;
               Deliver VMEXIT;
         ELSE
            #GP(0);
      FI;
FI;

IF (EPCM(DS:RCX).VALID ≠ 0)
   THEN #PF(DS:RCX); FI;

(* Check the SECS for concurrency *)
IF (SECS is not available for EADD)
   THEN #GP(0); FI;

IF (EPCM(DS:TMP_SECS).VALID = 0 or EPCM(DS:TMP_SECS).PT ≠ PT_SECS)
   THEN #PF(DS:TMP_SECS); FI;

(* Copy 4KBytes from source page to EPC page*)
DS:RCX[32767:0] ← DS:TMP_SRCPGE[32767:0];

CASE (SCRATCH_SECINFO.FLAGS.PT)

   PT_TCS:
      IF (DS:RCX.RESERVED ≠ 0) #GP(0); FI;
      IF ( (DS:TMP_SECS.ATTRIBUTES.MODE64BIT = 0) and
         ((DS:TCS.FSLIMIT & 0FFFH ≠ 0FFFH) or (DS:TCS.GSLIMIT & 0FFFH ≠ 0FFFH) )) #GP(0); FI;
       (* Ensure TCS.PREVSSP is zero *)
       IF (CPUID.(EAX=07H, ECX=00h):ECX[CET_SS] = 1) and (DS:RCX.PREVSSP != 0) #GP(0); FI;

      BREAK;
   PT_REG:
      IF (SCRATCH_SECINFO.FLAGS.W = 1 and SCRATCH_SECINFO.FLAGS.R = 0) #GP(0); FI;
      BREAK;
   PT_SS_FIRST:
   PT_SS_REST:
      (* SS pages cannot created on first or last page of ELRANGE *)
      IF ( TMP_LINADDR = DS:TMP_SECS.BASEADDR or TMP_LINADDR = (DS:TMP_SECS.BASEADDR + DS:TMP_SECS.SIZE - 0x1000) )
          THEN #GP(0); FI;
     IF ( DS:RCX[4087:0] != 0 ) #GP(0); FI;
     IF (SCRATCH_SECINFO.FLAGS.PT == PT_SS_FIRST)
         THEN
             (* Check that valid RSTORSSP token exists *)
             IF ( DS:RCX[4095:4088] != ((TMP_LINADDR + 0x1000) | DS:TMP_SECS.ATTRIBUTES.MODE64BIT) ) #GP(0); FI;
             (* Check the 8 bytes are zero *)
             IF ( DS:RCX[4095:4088] != 0 ) #GP(0); FI;
      FI;
      IF (SCRATCH_SECINFO.FLAGS.W = 0 OR SCRATCH_SECINFO.FLAGS.R = 0 OR
          SCRATCH_SECINFO.FLAGS.X = 1) #GP(0); FI;
         BREAK;

ESAC;

(* Check the enclave offset is within the enclave linear address space *)
IF (TMP_LINADDR < DS:TMP_SECS.BASEADDR or TMP_LINADDR ≥ DS:TMP_SECS.BASEADDR + DS:TMP_SECS.SIZE)
   THEN #GP(0); FI;

(* Check concurrency of measurement resource*)
IF (Measurement being updated)
   THEN #GP(0); FI;

(* Check if the enclave to which the page will be added is already in Initialized state *)
IF (DS:TMP_SECS already initialized)
   THEN #GP(0); FI;

(* For TCS pages, force EPCM.rwx bits to 0 and no debug access *)
IF (SCRATCH_SECINFO.FLAGS.PT = PT_TCS)
   THEN
      SCRATCH_SECINFO.FLAGS.R ← 0;
      SCRATCH_SECINFO.FLAGS.W ← 0;
      SCRATCH_SECINFO.FLAGS.X ← 0;
      (DS:RCX).FLAGS.DBGOPTIN ← 0; // force TCS.FLAGS.DBGOPTIN off
      DS:RCX.CSSA ← 0;
      DS:RCX.AEP ← 0;
      DS:RCX.STATE ← 0;
FI;

(* Add enclave offset and security attributes to MRENCLAVE *)
TMP_ENCLAVEOFFSET ← TMP_LINADDR - DS:TMP_SECS.BASEADDR;
TMPUPDATEFIELD[63:0] ← 0000000044444145H; // "EADD"
TMPUPDATEFIELD[127:64] ← TMP_ENCLAVEOFFSET;
TMPUPDATEFIELD[511:128] ← SCRATCH_SECINFO[375:0]; // 48 bytes
DS:TMP_SECS.MRENCLAVE ← SHA256UPDATE(DS:TMP_SECS.MRENCLAVE, TMPUPDATEFIELD)
INC enclave's MRENCLAVE update counter;

(* Add enclave offset and security attributes to MRENCLAVE *)
EPCM(DS:RCX).R ← SCRATCH_SECINFO.FLAGS.R;
EPCM(DS:RCX).W ← SCRATCH_SECINFO.FLAGS.W;
EPCM(DS:RCX).X ← SCRATCH_SECINFO.FLAGS.X;
EPCM(DS:RCX).PT ← SCRATCH_SECINFO.FLAGS.PT;
EPCM(DS:RCX).ENCLAVEADDRESS ← TMP_LINADDR;

(* associate the EPCPAGE with the SECS by storing the SECS identifier of DS:TMP_SECS *)
Update EPCM(DS:RCX) SECS identifier to reference DS:TMP_SECS identifier;

(* Set EPCM entry fields *)
EPCM(DS:RCX).BLOCKED ← 0;
EPCM(DS:RCX).PENDING ← 0;
EPCM(DS:RCX).MODIFIED ← 0;
EPCM(DS:RCX).VALID ← 1;
```

Flags Affected

None

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the DS segment limit.
	If a memory operand is not properly aligned.
	If an enclave memory operand is outside of the EPC.
	If an enclave memory operand is the wrong type.
	If a memory operand is locked.
	If the enclave is initialized.
	If the enclave's MRENCLAVE is locked.
	If the TCS page reserved bits are set.
	If the TCS page PREVSSP field is not zero.
	If the PT_SS_REST or PT_SS_REST page is first or last page in enclave.
	If the PT_SS_FIRST or PT_SS_REST page not initialized correctly.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If the EPC page is valid.

64-Bit Mode Exceptions

#GP(0)	If a memory operand is non-canonical form.
	If a memory operand is not properly aligned.
	If an enclave memory operand is outside of the EPC.
	If an enclave memory operand is the wrong type.
	If a memory operand is locked.
	If the enclave is initialized.
	If the enclave's MRENCLAVE is locked.
	If the TCS page reserved bits are set.
	If the TCS page PREVSSP field is not zero.
	If the PT_SS_REST or PT_SS_REST page is first or last page in enclave.
	If the PT_SS_FIRST or PT_SS_REST page not initialized correctly.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If the EPC page is valid.

EAUG—Add a Page to an Initialized Enclave

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 0DH ENCLS[EAUG]	IR	V/V	SGX2	This leaf function adds a page to an initialized enclave.

Instruction Operand Encoding

Op/En	EAX	RBX	RCX
IR	EAUG (In)	Address of a SECINFO (In)	Address of the destination EPC page (In)

Description

This leaf function zeroes a page of EPC memory, associates the EPC page with an SECS page residing in the EPC, and stores the linear address and security attributes in the EPCM. As part of the association, the security attributes are configured to prevent access to the EPC page until a corresponding invocation of the EACCEPT leaf or EACCEPTCOPY leaf confirms the addition of the new page into the enclave. This instruction can only be executed when current privilege level is 0.

RBX contains the effective address of a PAGEINFO structure while RCX contains the effective address of an EPC page. The table below provides additional information on the memory parameter of the EAUG leaf function.

EAUG Memory Parameter Semantics

PAGEINFO	PAGEINFO.SECS	PAGEINFO.SRCPGE	PAGEINFO.SECINFO	EPCPAGE
Read access permitted by Non Enclave	Read/Write access permitted by Enclave	Must be zero	Read access permitted by Non Enclave	Write access permitted by Enclave

EAUG Faulting Conditions

The instruction faults if any of the following:

The operands are not properly aligned.	Unsupported security attributes are set.
Refers to an invalid SECS.	Reference is made to an SECS that is locked by another thread.
The EPC page is locked by another thread.	RCX does not contain an effective address of an EPC page.
The EPC page is already valid.	The specified enclave offset is outside of the enclave address space.
The SECS has been initialized.	

Concurrency Restrictions

Base Concurrency Restrictions of EAUG

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EAUG	Target [DS:RCX]	Exclusive	#GP	EPC_PAGE_CONFLICT_EXCEPTION
	SECS [DS:RBX]PAGEINFO.SECS	Shared	#GP	

Additional Concurrency Restrictions of EAUG

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EAUG	Target [DS:RCX]	Concurrent		Concurrent		Concurrent	
	SECS [DS:RBX]PAGE-INFO.SECS	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in EAUG Operational Flow

Name	Type	Size (bits)	Description
TMP_SECS	Effective Address	32/64	Effective address of the SECS destination page.
TMP_SECINFO	Effective Address	32/64	Effective address of an SECINFO structure which contains security attributes of the page to be added.
SCRATCH_SECINFO	SECINFO	512	Scratch storage for holding the contents of DS:TMP_SECINFO.
TMP_LINADDR	Unsigned Integer	64	Holds the linear address to be stored in the EPCM and used to calculate TMP_ENCLAVEOFFSET.

```
IF (DS:RBX is not 32Byte Aligned)
    THEN #GP(0); FI;

IF (DS:RCX is not 4KByte Aligned)
    THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;

TMP_SECS ← DS:RBX.SECS;
TMP_SECINFO ← DS:RBX.SECINFO;
IF (DS:RBX.SECINFO is not 0)
    THEN
        IF (DS:TMP_SECINFO is not 64B aligned)
             THEN #GP(0); FI;
    FI;

TMP_LINADDR ← DS:RBX.LINADDR;

IF ( DS:TMP_SECS is not 4KByte aligned or TMP_LINADDR is not 4KByte aligned )
    THEN #GP(0); FI;

IF ( (DS:RBX.SRCPAGE is not 0) ~~or (DS:RBX.SECINFO is not 0)~~ )
    THEN #GP(0); FI;

IF (DS:TMP_SECS does not resolve within an EPC)
    THEN #PF(DS:TMP_SECS); FI;

(* Check the EPC page for concurrency *)
IF (EPC page in use)
    THEN #GP(0); FI;

IF (EPCM(DS:RCX).VALID ≠ 0)
    THEN #PF(DS:RCX); FI;
```

```
(*    copy SECINFO contents into a scratch SECINFO *)
IF (DS:RBX.SECINFO is 0)
    THEN
        (* allocate and initialize a new scratch secinfo structure *)
        SCRATCH_SECINFO.PT ← PT_REG;
        SCRATCH_SECINFO.R ← 1;
        SCRATCH_SECINFO.W ←  1;
        SCRATCH_SECINFO.X ← 0;
        << zero out remaining fields of SCRATCH_SECINFO >>
    ELSE
        (* copy SECINFO contents into scratch secinfo *)
        SCRATCH_SECINFO ← DS:TMP_SECINFO;
        (* check SECINFO flags for misconfiguration *)
        (* reserved flags must be zero *)
        (* SECINFO.FLAGS.PT must either be PT_SS_FIRST, or PT_SS_REST *)
        IF ( (SCRATCH_SECINFO reserved fields are not 0) OR
            (SCRATCH_SECINFO.PT is not PT_SS_FIRST, or PT_SS_REST) OR
            ( (SCRATCH_SECINFO.FLAGS.R is 0) OR    (SCRATCH_SECINFO.FLAGS.W is 0) OR (SCRATCH_SECINFO.FLAGS.X is 1) ) )
                THEN #GP(0); FI;
    FI;

(* Check if PT_SS_FIRST/PT_SS_REST page types are requested then CR4.CET must be 1 *)
IF (    (SCRATCH_SECINFO.PT is PT_SS_FIRST OR SCRATCH_SECINFO.PT is PT_SS_REST) AND CR4.CET == 0 )
       THEN #GP(0); FI;

(* Check the SECS for concurrency *)
IF (SECS is not available for EAUG)
    THEN #GP(0); FI;

IF (EPCM(DS:TMP_SECS).VALID = 0 or EPCM(DS:TMP_SECS).PT ≠ PT_SECS)
    THEN #PF(DS:TMP_SECS); FI;

(* Check if the enclave to which the page will be added is in the Initialized state *)
IF (DS:TMP_SECS is not initialized)
    THEN #GP(0); FI;

(* Check the enclave offset is within the enclave linear address space *)
IF ( (TMP_LINADDR < DS:TMP_SECS.BASEADDR) or (TMP_LINADDR ≥ DS:TMP_SECS.BASEADDR + DS:TMP_SECS.SIZE) )
    THEN #GP(0); FI;

IF (    (SCRATCH_SECINFO.PT is PT_SS_FIRST OR SCRATCH_SECINFO.PT is PT_SS_REST) )
       THEN
           (* SS pages cannot created on first or last page of ELRANGE *)
           IF ( TMP_LINADDR == DS:TMP_SECS.BASEADDR OR
               TMP_LINADDR == (DS:TMP_SECS.BASEADDR + DS:TMP_SECS.SIZE - 0x1000) )
               THEN
```

```
                #GP(0); FI;
    FI;

(* Clear the content of EPC page*)
DS:RCX[32767:0] ← 0;

(* Set EPCM security attributes *)
~~EPCM(DS:RCX).R ← 1;~~
~~EPCM(DS:RCX).W ← 1;~~
~~EPCM(DS:RCX).X ← 0;~~
~~EPCM(DS:RCX).PT ← PT_REG;~~
EPCM(DS:RCX).R ← SCRATCH_SECINFO.FLAGS.R;
EPCM(DS:RCX).W ← SCRATCH_SECINFO.FLAGS.W;
EPCM(DS:RCX).X ← SCRATCH_SECINFO.FLAGS.X;
EPCM(DS:RCX).PT ← SCRATCH_SECINFO.FLAGS.PT;
EPCM(DS:RCX).ENCLAVEADDRESS ← TMP_LINADDR;
EPCM(DS:RCX).BLOCKED ← 0;
EPCM(DS:RCX).PENDING ← 1;
EPCM(DS:RCX).MODIFIED ← 0;
EPCM(DS:RCX).PR ← 0;

(* associate the EPCPAGE with the SECS by storing the SECS identifier of DS:TMP_SECS *)
Update EPCM(DS:RCX) SECS identifier to reference DS:TMP_SECS identifier;

(* Set EPCM valid fields *)
EPCM(DS:RCX).VALID ← 1;
```

Flags Affected

None

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the DS segment limit.
	If a memory operand is not properly aligned.
	If a memory operand is locked.
	If the enclave is not initialized.
#PF(error code)	If a page fault occurs in accessing memory operands.

64-Bit Mode Exceptions

#GP(0)	If a memory operand is non-canonical form.
	If a memory operand is not properly aligned.
	If a memory operand is locked.
	If the enclave is not initialized.
#PF(error code)	If a page fault occurs in accessing memory operands.

EBLOCK—Mark a page in EPC as Blocked

Opcode/ Instruction	**Op/En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
EAX = 09H ENCLS[EBLOCK]	IR	V/V	SGX1	This leaf function marks a page in the EPC as blocked.

Instruction Operand Encoding

Op/En	EAX		RCX
IR	EBLOCK (In)	Return error code (Out)	Effective address of the EPC page (In)

Description

This leaf function causes an EPC page to be marked as BLOCKED. This instruction can only be executed when current privilege level is 0.

The content of RCX is an effective address of an EPC page. The DS segment is used to create linear address. Segment override is not supported.

An error code is returned in RAX.

The table below provides additional information on the memory parameter of EBLOCK leaf function.

EBLOCK Memory Parameter Semantics

EPCPAGE
Read/Write access permitted by Enclave

The error codes are:

EBLOCK Return Value in RAX

Error Code	Description
No Error	EBLOCK successful.
SGX_BLKSTATE	Page already blocked. This value is used to indicate to a VMM that the page was already in BLOCKED state as a result of EBLOCK and thus will need to be restored to this state when it is eventually reloaded (using ELDB).
SGX_ENTRYEPOCH_LOCKED	SECS locked for Entry Epoch update. This value indicates that an ETRACK is currently executing on the SECS. The EBLOCK should be reattempted.
SGX_NOTBLOCKABLE	Page type is not one which can be blocked.
SGX_PG_INVLD	Page is not valid and cannot be blocked.
SGX_EPC_PAGE_CONFLICT	Page is being written by EADD, EAUG, ECREATE, ELDU/B, EMODT, or EWB.

Concurrency Restrictions

Base Concurrency Restrictions of EBLOCK

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EBLOCK	Target [DS:RCX]	Shared	SGX_EPC_PAGE_ CONFLICT	

Additional Concurrency Restrictions of EBLOCK

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EBLOCK	Target [DS:RCX]	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in EBLOCK Operational Flow

Name	Type	Size (Bits)	Description
TMP_BLKSTATE	Integer	64	Page is already blocked.

```
IF (DS:RCX is not 4KByte Aligned)
    THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;

RFLAGS.ZF,CF,PF,AF,OF,SF ← 0;
RAX← 0;

(* Check the EPC page for concurrency*)
IF (EPC page in use)
    THEN
        RFLAGS.ZF ← 1;
        RAX← SGX_EPC_PAGE_CONFLICT;
        GOTO DONE;
FI;

IF (EPCM(DS:RCX). VALID = 0)
    THEN
        RFLAGS.ZF ← 1;
        RAX← SGX_PG_INVLD;
        GOTO DONE;
FI;

IF ( (EPCM(DS:RCX).PT ≠ PT_REG) and (EPCM(DS:RCX).PT ≠ PT_TCS) and (EPCM(DS:RCX).PT ≠ PT_TRIM)
     and EPCM(DS:RCX).PT ≠ PT_SS_FIRST) and (EPCM(DS:RCX).PT ≠ PT_SS_REST))
    THEN
        RFLAGS.CF ← 1;
        IF (EPCM(DS:RCX).PT = PT_SECS)
            THEN RAX← SGX_PG_IS_SECS;
            ELSE RAX← SGX_NOTBLOCKABLE;
        FI;
        GOTO DONE;
FI;

(* Check if the page is already blocked and report blocked state *)
TMP_BLKSTATE ← EPCM(DS:RCX).BLOCKED;
```

```
(* at this point, the page must be valid and PT_TCS or PT_REG or PT_TRIM*)
IF (TMP_BLKSTATE = 1)
    THEN
        RFLAGS.CF ← 1;
        RAX← SGX_BLKSTATE;
    ELSE
        EPCM(DS:RCX).BLOCKED ← 1
FI;
DONE:
```

Flags Affected

Sets ZF if SECS is in use or invalid, otherwise cleared. Sets CF if page is BLOCKED or not blockable, otherwise cleared. Clears PF, AF, OF, SF.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the DS segment limit.
	If a memory operand is not properly aligned.
	If the specified EPC resource is in use.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If a memory operand is not an EPC page.

64-Bit Mode Exceptions

#GP(0)	If a memory operand is non-canonical form.
	If a memory operand is not properly aligned.
	If the specified EPC resource is in use.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If a memory operand is not an EPC page.

ECREATE—Create an SECS page in the Enclave Page Cache

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 00H ENCLS[ECREATE]	IR	V/V	SGX1	This leaf function begins an enclave build by creating an SECS page in EPC.

Instruction Operand Encoding

Op/En	EAX	RBX	RCX
IR	ECREATE (In)	Address of a PAGEINFO (In)	Address of the destination SECS page (In)

Description

ENCLS[ECREATE] is the first instruction executed in the enclave build process. ECREATE copies an SECS structure outside the EPC into an SECS page inside the EPC. The internal structure of SECS is not accessible to software.

ECREATE will set up fields in the protected SECS and mark the page as valid inside the EPC. ECREATE initializes or checks unused fields.

Software sets the following fields in the source structure: SECS:BASEADDR, SECS:SIZE in bytes, ATTRIBUTES, CONFIGID and CONFIGSVN. SECS:BASEADDR must be naturally aligned on an SECS.SIZE boundary. SECS.SIZE must be at least 2 pages (8192).

The source operand RBX contains an effective address of a PAGEINFO structure. PAGEINFO contains an effective address of a source SECS and an effective address of an SECINFO. The SECS field in PAGEINFO is not used.

The RCX register is the effective address of the destination SECS. It is an address of an empty slot in the EPC. The SECS structure must be page aligned. SECINFO flags must specify the page as an SECS page.

ECREATE Memory Parameter Semantics

PAGEINFO	PAGEINFO.SRCPGE	PAGEINFO.SECINFO	EPCPAGE
Read access permitted by Non Enclave	Read access permitted by Non Enclave	Read access permitted by Non Enclave	Write access permitted by Enclave

ECREATE will fault if the SECS target page is in use; already valid; outside the EPC. It will also fault if addresses are not aligned; unused PAGEINFO fields are not zero.

If the amount of space needed to store the SSA frame is greater than the amount specified in SECS.SSAFRAMESIZE, a #GP(0) results. The amount of space needed for an SSA frame is computed based on DS:TMP_SECS.ATTRIBUTES.XFRM size.

Concurrency Restrictions

Base Concurrency Restrictions of ECREATE

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
ECREATE	SECS [DS:RCX]	Exclusive	#GP	EPC_PAGE_CONFLICT_EXCEPTION

Additional Concurrency Restrictions of ECREATE

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
ECREATE	SECS [DS:RCX]	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in ECREATE Operational Flow

Name	Type	Size (Bits)	Description
TMP_SRCPGE	Effective Address	32/64	Effective address of the SECS source page.
TMP_SECS	Effective Address	32/64	Effective address of the SECS destination page.
TMP_SECINFO	Effective Address	32/64	Effective address of an SECINFO structure which contains security attributes of the SECS page to be added.
TMP_XSIZE	SSA Size	64	The size calculation of SSA frame.
TMP_MISC_SIZE	MISC Field Size	64	Size of the selected MISC field components.
TMPUPDATEFIELD	SHA256 Buffer	512	Buffer used to hold data being added to TMP_SECS.MRENCLAVE.

```
IF (DS:RBX is not 32Byte Aligned)
    THEN #GP(0); FI;

IF (DS:RCX is not 4KByte Aligned)
    THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;

TMP_SRCPGE ← DS:RBX.SRCPGE;
TMP_SECINFO ← DS:RBX.SECINFO;

IF (DS:TMP_SRCPGE is not 4KByte aligned or   DS:TMP_SECINFO is not 64Byte aligned)
    THEN #GP(0); FI;

IF (DS:RBX.LINADDR ! = 0 or DS:RBX.SECS ≠ 0)
    THEN #GP(0); FI;

(* Check for misconfigured SECINFO flags*)
IF (DS:TMP_SECINFO reserved fields are not zero or      DS:TMP_SECINFO.FLAGS.PT ≠ PT_SECS)
    THEN #GP(0); FI;

TMP_SECS ← RCX;

IF (EPC entry in use)
    THEN
        IF (<<VMX non-root operation>> AND <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>>)
            THEN
                VMCS.Exit_reason ← SGX_CONFLICT;
                VMCS.Exit_qualification.code ← EPC_PAGE_CONFLICT_EXCEPTION;
                VMCS.Exit_qualification.error ← 0;
                VMCS.Guest-physical_address ←
                    << translation of DS:TMP_SECS produced by paging >>;
                VMCS.Guest-linear_address ← DS:TMP_SECS;
                    Deliver VMEXIT;
            ELSE
                #GP(0);
        FI;
FI;

IF (EPC entry in use)
    THEN #GP(0); FI;

IF (EPCM(DS:RCX).VALID = 1)
    THEN #PF(DS:RCX); FI;

(* Copy 4KBytes from source page to EPC page*)
DS:RCX[32767:0] ← DS:TMP_SRCPGE[32767:0];
```

```
(* Check lower 2 bits of XFRM are set *)
IF ( ( DS:TMP_SECS.ATTRIBUTES.XFRM BitwiseAND 03H) ≠ 03H)
    THEN #GP(0); FI;

IF (XFRM is illegal)
    THEN #GP(0); FI;

(* Check legality of CET_ATTRIBUTES *)
IF ((DS:TMP_SECS.ATTRIBUTES.CET = 0 and DS:TMP_SECS.CET_ATTRIBUTES ≠ 0) ||
        (DS:TMP_SECS.ATTRIBUTES.CET = 0 and DS:TMP_SECS.CET_LEG_BITMAP_OFFSET ≠ 0) ||
        (CPUID.(EAX=7, ECX=0):EDX[CET_IBT] = 0 and DS:TMP_SECS.CET_LEG_BITMAP_OFFSET ≠ 0) ||
        (CPUID.(EAX=7, ECX=0):EDX[CET_IBT] = 0 and DS:TMP_SECS.CET_ATTRIBUTES[5:2] ≠ 0) ||
        (CPUID.(EAX=7, ECX=0):ECX[CET_SS] = 0 and DS:TMP_SECS.CET_ATTRIBUTES[1:0] ≠ 0) ||
        (DS:TMP_SECS.ATTRIBUTES.MODE64BIT = 1 and
          (DS:TMP_SECS.BASEADDR + DS:TMP_SECS.CET_LEG_BITMAP_OFFSET) not canonical) ||
        (DS:TMP_SECS.ATTRIBUTES.MODE64BIT = 0 and
          (DS:TMP_SECS.BASEADDR + DS:TMP_SECS.CET_LEG_BITMAP_OFFSET) & 0xFFFFFFFF00000000) ||
        (DS:TMP_SECS.CET_ATTRIBUTES.reserved fields not 0) or
        (DS:TMP_SECS.CET_LEG_BITMAP_OFFSET) is not page aligned))
        THEN
            #GP(0);
        FI;

(* Make sure that the SECS does not have any unsupported MISCSELECT options*)
IF ( !(CPUID.(EAX=12H, ECX=0):EBX[31:0] & DS:TMP_SECS.MISCSELECT[31:0]) )
    THEN
        EPCM(DS:TMP_SECS).EntryLock.Release();
        #GP(0);
FI;

( * Compute size of MISC area *)
TMP_MISC_SIZE ← compute_misc_region_size();

(* Compute the size required to save state of the enclave on async exit *)
TMP_XSIZE ← compute_xsave_size(DS:TMP_SECS.ATTRIBUTES.XFRM) + GPR_SIZE + TMP_MISC_SIZE;

(* Ensure that the declared area is large enough to hold XSAVE and GPR stat *)
IF ( DS:TMP_SECS.SSAFRAMESIZE*4096 < TMP_XSIZE)
    THEN #GP(0); FI;

IF ( (DS:TMP_SECS.ATTRIBUTES.MODE64BIT = 1) and (DS:TMP_SECS.BASEADDR is not canonical) )
    THEN #GP(0); FI;

IF ( (DS:TMP_SECS.ATTRIBUTES.MODE64BIT = 0) and (DS:TMP_SECS.BASEADDR and 0FFFFFFFF00000000H) )
    THEN #GP(0); FI;

IF ( (DS:TMP_SECS.ATTRIBUTES.MODE64BIT = 0) and (DS:TMP_SECS.SIZE ≥ 2 ^ (CPUID.(EAX=12H, ECX=0):.EDX[7:0]) ) )
    THEN #GP(0); FI;

IF ( (DS:TMP_SECS.ATTRIBUTES.MODE64BIT = 1) and (DS:TMP_SECS.SIZE ≥ 2 ^ (CPUID.(EAX=12H, ECX=0):.EDX[15:8]) ) )
    THEN #GP(0); FI;
```

```
(* Enclave size must be at least 8192 bytes and must be power of 2 in bytes*)
IF (DS:TMP_SECS.SIZE < 8192 or popcnt(DS:TMP_SECS.SIZE) > 1)
    THEN #GP(0); FI;

(* Ensure base address of an enclave is aligned on size*)
IF ( ( DS:TMP_SECS.BASEADDR and (DS:TMP_SECS.SIZE-1) ) )
    THEN #GP(0); FI;

(* Ensure the SECS does not have any unsupported attributes*)
IF ( DS:TMP_SECS.ATTRIBUTES and (~CR_SGX_ATTRIBUTES_MASK) )
    THEN #GP(0); FI;

IF ( DS:TMP_SECS reserved fields are not zero)
    THEN #GP(0); FI;

(* Verify that CONFIGID/CONFIGSVN are not set with attribute *)
IF ( ((DS:TMP_SECS.CONFIGID ≠ 0) or (DS:TMP_SECS.CONFIGSVN ≠0)) AND (DS:TMP_SECS.ATTRIBUTES.KSS == 0 ))
    THEN #GP(0); FI;

Clear DS:TMP_SECS to Uninitialized;
DS:TMP_SECS.MRENCLAVE ← SHA256INITIALIZE(DS:TMP_SECS.MRENCLAVE);
DS:TMP_SECS.ISVSVN ← 0;
DS:TMP_SECS.ISVPRODID ← 0;

(* Initialize hash updates etc.*)
Initialize enclave's MRENCLAVE update counter;

(* Add "ECREATE" string and SECS fields to MRENCLAVE *)
TMPUPDATEFIELD[63:0] ← 0045544145524345H; // "ECREATE"
TMPUPDATEFIELD[95:64] ← DS:TMP_SECS.SSAFRAMESIZE;
TMPUPDATEFIELD[159:96] ← DS:TMP_SECS.SIZE;
IF (CPUID.(EAX=7, ECX=0):EDX[CET_IBT] = 1)
    THEN
        TMPUPDATEFIELD[223:160] ← DS:TMP_SECS.CET_LEG_BITMAP_OFFSET;
    ELSE
        TMPUPDATEFIELD[223:160] ← 0;
    FI;
TMPUPDATEFIELD[511:224] ← 0;
DS:TMP_SECS.MRENCLAVE ← SHA256UPDATE(DS:TMP_SECS.MRENCLAVE, TMPUPDATEFIELD)
INC enclave's MRENCLAVE update counter;

(* Set EID *)
DS:TMP_SECS.EID ← LockedXAdd(CR_NEXT_EID, 1);

(* Initialize the virtual child count to zero *)
DS:TMP_SECS.VIRTCHILDCNT ← 0;

(* Load ENCLAVECONTEXT with Address out of paging of SECS *)
<< store translation of DS:RCX produced by paging in SECS(DS:RCX).ENCLAVECONTEXT >>
```

```
(* Set the EPCM entry, first create SECS identifier and store the identifier in EPCM *)
EPCM(DS:TMP_SECS).PT ← PT_SECS;
EPCM(DS:TMP_SECS).ENCLAVEADDRESS ← 0;
EPCM(DS:TMP_SECS).R ← 0;
EPCM(DS:TMP_SECS).W ← 0;
EPCM(DS:TMP_SECS).X ← 0;

(* Set EPCM entry fields *)
EPCM(DS:RCX).BLOCKED ← 0;
EPCM(DS:RCX).PENDING ← 0;
EPCM(DS:RCX).MODIFIED ← 0;
EPCM(DS:RCX).PR ← 0;
EPCM(DS:RCX).VALID ← 1;
```

Flags Affected

None

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the DS segment limit.
	If a memory operand is not properly aligned.
	If the reserved fields are not zero.
	If PAGEINFO.SECS is not zero.
	If PAGEINFO.LINADDR is not zero.
	If the SECS destination is locked.
	If SECS.SSAFRAMESIZE is insufficient.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If the SECS destination is outside the EPC.

64-Bit Mode Exceptions

#GP(0)	If a memory address is non-canonical form.
	If a memory operand is not properly aligned.
	If the reserved fields are not zero.
	If PAGEINFO.SECS is not zero.
	If PAGEINFO.LINADDR is not zero.
	If the SECS destination is locked.
	If SECS.SSAFRAMESIZE is insufficient.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If the SECS destination is outside the EPC.

EDBGRD—Read From a Debug Enclave

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 04H ENCLS[EDBGRD]	IR	V/V	SGX1	This leaf function reads a dword/quadword from a debug en-clave.

Instruction Operand Encoding

Op/En	EAX	RBX	RCX
IR	EDBGRD (In)	Data read from a debug enclave (Out)	Address of source memory in the EPC (In)

Description

This leaf function copies a quadword/doubleword from an EPC page belonging to a debug enclave into the RBX register. Eight bytes are read in 64-bit mode, four bytes are read in non-64-bit modes. The size of data read cannot be overridden.

The effective address of the source location inside the EPC is provided in the register RCX.

EDBGRD Memory Parameter Semantics

EPCQW
Read access permitted by Enclave

The error codes are:

EDBGRD Return Value in RAX

Error Code	Description
No Error	EDBGRD successful.
SGX_PAGE_NOT_DEBUGGABLE	The EPC page cannot be accessed because it is in the PENDING or MODIFIED state.

EDBGRD Faulting Conditions

The instruction faults if any of the following:

RCX points into a page that is an SECS.	RCX does not resolve to a naturally aligned linear address.
RCX points to a page that does not belong to an enclave that is in debug mode.	RCX points to a location inside a TCS that is beyond the architectural size of the TCS (SGX_TCS_LIMIT).
An operand causing any segment violation.	May page fault.
CPL > 0.	

This instruction ignores the EPCM RWX attributes on the enclave page. Consequently, violation of EPCM RWX attributes via EDBGRD does not result in a #GP.

Concurrency Restrictions

Base Concurrency Restrictions of EDBGRD

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EDBGRD	Target [DS:RCX]	Shared	#GP	

Additional Concurrency Restrictions of EDBGRD

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EDBGRD	Target [DS:RCX]	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in EDBGRD Operational Flow

Name	Type	Size (Bits)	Description
TMP_MODE64	Binary	1	((IA32_EFER.LMA = 1) && (CS.L = 1))
TMP_SECS		64	Physical address of SECS of the enclave to which source operand belongs.

```
TMP_MODE64 ← ((IA32_EFER.LMA = 1) && (CS.L = 1));

IF ( (TMP_MODE64 = 1) and (DS:RCX is not 8Byte Aligned) )
    THEN #GP(0); FI;

IF ( (TMP_MODE64 = 0) and (DS:RCX is not 4Byte Aligned) )
    THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;

(* make sure no other Intel SGX instruction is accessing EPCM *)
IF (Other EPCM modifying instructions executing)
    THEN #GP(0); FI;

IF (EPCM(DS:RCX). VALID = 0)
    THEN #PF(DS:RCX); FI;

(* make sure that DS:RCX (SOURCE) is pointing to a PT_REG or PT_TCS or PT_VA or PT_SS_FIRST or PT_SS_REST *)
IF ( (EPCM(DS:RCX).PT ≠ PT_REG) and (EPCM(DS:RCX).PT ≠ PT_TCS) and (EPCM(DS:RCX).PT ≠ PT_VA)
      and (EPCM(DS:RCX).PT ≠ PT_SS_FIRST) and (EPCM(DS:RCX).PT ≠ PT_SS_REST))
    THEN #PF(DS:RCX); FI;

(* make sure that DS:RCX points to an accessible EPC page *)
IF (EPCM(DS:RCX).PENDING is not 0 or (EPCM(DS:RCX).MODIFIED is not 0) )
    THEN
        RFLAGS.ZF ← 1;
        RAX ← SGX_PAGE_NOT_DEBUGGABLE;
        GOTO DONE;
FI;

(* If source is a TCS, then make sure that the offset into the page is not beyond the TCS size*)
IF ( ( EPCM(DS:RCX). PT = PT_TCS) and ((DS:RCX) & FFFH ≥ SGX_TCS_LIMIT) )
    THEN #GP(0); FI;

(* make sure the enclave owning the PT_REG or PT_TCS page allow debug *)
```

```
IF ( (EPCM(DS:RCX).PT = PT_REG) or (EPCM(DS:RCX).PT = PT_TCS) )
   THEN
      TMP_SECS ← GET_SECS_ADDRESS;
      IF (TMP_SECS.ATTRIBUTES.DEBUG = 0)
         THEN #GP(0); FI;
      IF ( (TMP_MODE64 = 1) )
         THEN RBX[63:0] ← (DS:RCX)[63:0];
         ELSE EBX[31:0] ← (DS:RCX)[31:0];
      FI;
   ELSE
      TMP_64BIT_VAL[63:0] ← (DS:RCX)[63:0] & (~07H); // Read contents from VA slot
      IF (TMP_MODE64 = 1)
         THEN
            IF (TMP_64BIT_VAL ≠ 0H)
               THEN RBX[63:0] ← 0FFFFFFFFFFFFFFFFH;
               ELSE RBX[63:0] ← 0H;
            FI;
         ELSE
            IF (TMP_64BIT_VAL ≠ 0H)
               THEN EBX[31:0] ← 0FFFFFFFFH;
               ELSE EBX[31:0] ← 0H;
            FI;
FI;

(* clear EAX and ZF to indicate successful completion *)
RAX ← 0;
RFLAGS.ZF ← 0;

DONE:
(* clear flags *)
RFLAGS.CF,PF,AF,OF,SF ← 0;
```

Flags Affected

None

Protected Mode Exceptions

#GP(0)	If the address in RCS violates DS limit or access rights.
	If DS segment is unusable.
	If RCX points to a memory location not 4Byte-aligned.
	If the address in RCX points to a page belonging to a non-debug enclave.
	If the address in RCX points to a page which is not PT_TCS, PT_REG or PT_VA.
	If the address in RCX points to a location inside TCS that is beyond SGX_TCS_LIMIT.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If the address in RCX points to a non-EPC page.
	If the address in RCX points to an invalid EPC page.

64-Bit Mode Exceptions

#GP(0)	If RCX is non-canonical form.
	If RCX points to a memory location not 8Byte-aligned.
	If the address in RCX points to a page belonging to a non-debug enclave.
	If the address in RCX points to a page which is not PT_TCS, PT_REG or PT_VA.
	If the address in RCX points to a location inside TCS that is beyond SGX_TCS_LIMIT.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If the address in RCX points to a non-EPC page.
	If the address in RCX points to an invalid EPC page.

EDBGWR—Write to a Debug Enclave

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 05H ENCLS[EDBGWR]	IR	V/V	SGX1	This leaf function writes a dword/quadword to a debug enclave.

Instruction Operand Encoding

Op/En	EAX	RBX	RCX
IR	EDBGWR (In)	Data to be written to a debug enclave (In)	Address of Target memory in the EPC (In)

Description

This leaf function copies the content in EBX/RBX to an EPC page belonging to a debug enclave. Eight bytes are written in 64-bit mode, four bytes are written in non-64-bit modes. The size of data cannot be overridden.

The effective address of the source location inside the EPC is provided in the register RCX.

EDBGWR Memory Parameter Semantics Conditions

EPCQW
Write access permitted by Enclave

EDBGWR Faulting Conditions

The instruction faults if any of the following:

RCX points into a page that is an SECS.	RCX does not resolve to a naturally aligned linear address.
RCX points to a page that does not belong to an enclave that is in debug mode.	RCX points to a location inside a TCS that is not the FLAGS word.
An operand causing any segment violation.	May page fault.
CPL > 0.	

The error codes are:

EDBGWR Return Value in RAX

Error Code	Description
No Error	EDBGWR successful.
SGX_PAGE_NOT_DEBUGGABLE	The EPC page cannot be accessed because it is in the PENDING or MODIFIED state.

This instruction ignores the EPCM RWX attributes on the enclave page. Consequently, violation of EPCM RWX attributes via EDBGRD does not result in a #GP.

Concurrency Restrictions

Base Concurrency Restrictions of EDBGWR

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EDBGWR	Target [DS:RCX]	Shared	#GP	

Additional Concurrency Restrictions of EDBGWR

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EDBGWR	Target [DS:RCX]	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in EDBGWR Operational Flow

Name	Type	Size (Bits)	Description
TMP_MODE64	Binary	1	((IA32_EFER.LMA = 1) && (CS.L = 1)).
TMP_SECS		64	Physical address of SECS of the enclave to which source operand belongs.

```
TMP_MODE64 ← ((IA32_EFER.LMA = 1) && (CS.L = 1));

IF ( (TMP_MODE64 = 1) and (DS:RCX is not 8Byte Aligned) )
    THEN #GP(0); FI;

IF ( (TMP_MODE64 = 0) and (DS:RCX is not 4Byte Aligned) )
    THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;

(* make sure no other Intel SGX instruction is accessing EPCM *)
IF (Other EPCM modifying instructions executing)
    THEN #GP(0); FI;

IF (EPCM(DS:RCX). VALID = 0)
    THEN #PF(DS:RCX); FI;

(* make sure that DS:RCX (DST) is pointing to a PT_REG or PT_TCS or PT_SS_FIRST or PT_SS_REST *)
IF ( (EPCM(DS:RCX).PT ≠ PT_REG) and (EPCM(DS:RCX).PT ≠ PT_TCS)
     and (EPCM(DS:RCX).PT ≠ PT_SS_FIRST) and (EPCM(DS:RCX).PT ≠ PT_SS_REST))
    THEN #PF(DS:RCX); FI;

(* make sure that DS:RCX points to an accessible EPC page *)
IF ( (EPCM(DS:RCX).PENDING is not 0) or (EPCM(DS:RCS).MODIFIED is not 0) )
    THEN
        RFLAGS.ZF ← 1;
        RAX ← SGX_PAGE_NOT_DEBUGGABLE;
        GOTO DONE;
FI;

(* If destination is a TCS, then make sure that the offset into the page can only point to the FLAGS field*)
IF ( ( EPCM(DS:RCX). PT = PT_TCS) and ((DS:RCX) & FF8H ≠ offset_of_FLAGS & 0FF8H) )
    THEN #GP(0); FI;

(* Locate the SECS for the enclave to which the DS:RCX page belongs *)
TMP_SECS ← GET_SECS_PHYS_ADDRESS(EPCM(DS:RCX).ENCLAVESECS);

(* make sure the enclave owning the PT_REG or PT_TCS page allow debug *)
IF (TMP_SECS.ATTRIBUTES.DEBUG = 0)
```

```
    THEN #GP(0); FI;

IF ( (TMP_MODE64 = 1) )
    THEN (DS:RCX)[63:0] ← RBX[63:0];
    ELSE (DS:RCX)[31:0] ← EBX[31:0];
FI;

(* clear EAX and ZF to indicate successful completion *)
RAX ← 0;
RFLAGS.ZF ← 0;

DONE:
(* clear flags *)
RFLAGS.CF,PF,AF,OF,SF ← 0
```

Flags Affected

None

Protected Mode Exceptions

#GP(0)	If the address in RCS violates DS limit or access rights.
	If DS segment is unusable.
	If RCX points to a memory location not 4Byte-aligned.
	If the address in RCX points to a page belonging to a non-debug enclave.
	If the address in RCX points to a page which is not PT_TCS or PT_REG.
	If the address in RCX points to a location inside TCS that is not the FLAGS word.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If the address in RCX points to a non-EPC page.
	If the address in RCX points to an invalid EPC page.

64-Bit Mode Exceptions

#GP(0)	If RCX is non-canonical form.
	If RCX points to a memory location not 8Byte-aligned.
	If the address in RCX points to a page belonging to a non-debug enclave.
	If the address in RCX points to a page which is not PT_TCS or PT_REG.
	If the address in RCX points to a location inside TCS that is not the FLAGS word.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If the address in RCX points to a non-EPC page.
	If the address in RCX points to an invalid EPC page.

EEXTEND—Extend Uninitialized Enclave Measurement by 256 Bytes

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 06H ENCLS[EEXTEND]	IR	V/V	SGX1	This leaf function measures 256 bytes of an uninitialized en-clave page.

Instruction Operand Encoding

Op/En	EAX	EBX	RCX
IR	EEXTEND (In)	Effective address of the SECS of the data chunk (In)	Effective address of a 256-byte chunk in the EPC (In)

Description

This leaf function updates the MRENCLAVE measurement register of an SECS with the measurement of an EX-TEND string compromising of “EEXTEND” || ENCLAVEOFFSET || PADDING || 256 bytes of the enclave page. This instruction can only be executed when current privilege level is 0 and the enclave is uninitialized.

RBX contains the effective address of the SECS of the region to be measured. The address must be the same as the one used to add the page into the enclave.

RCX contains the effective address of the 256 byte region of an EPC page to be measured. The DS segment is used to create linear addresses. Segment override is not supported.

EEXTEND Memory Parameter Semantics

EPC[RCX]
Read access by Enclave

EEXTEND Faulting Conditions

The instruction faults if any of the following:

RBX points to an address not 4KBytes aligned.	RBX does not resolve to an SECS.
RBX does not point to an SECS page.	RBX does not point to the SECS page of the data chunk.
RCX points to an address not 256B aligned.	RCX points to an unused page or a SECS.
RCX does not resolve in an EPC page.	If SECS is locked.
If the SECS is already initialized.	May page fault.
CPL > 0.	

Concurrency Restrictions

Base Concurrency Restrictions of EEXTEND

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EEXTEND	Target [DS:RCX]	Shared	#GP	
	SECS [DS:RBX]	Concurrent		

Additional Concurrency Restrictions of EEXTEND

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EEXTEND	Target [DS:RCX]	Concurrent		Concurrent		Concurrent	
	SECS [DS:RBX]	Concurrent		Exclusive	#GP	Concurrent	

Operation

Temp Variables in EEXTEND Operational Flow

Name	Type	Size (Bits)	Description
TMP_SECS		64	Physical address of SECS of the enclave to which source operand belongs.
TMP_ENCLAVEOFFSET	Enclave Offset	64	The page displacement from the enclave base address.
TMPUPDATEFIELD	SHA256 Buffer	512	Buffer used to hold data being added to TMP_SECS.MRENCLAVE.

```
TMP_MODE64 ← ((IA32_EFER.LMA = 1) && (CS.L = 1));

IF (DS:RBX is not 4096 Byte Aligned)
    THEN #GP(0); FI;

IF (DS:RBX does resolve to an EPC page)
    THEN #PF(DS:RBX); FI;

IF (DS:RCX is not 256Byte Aligned)
    THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;

(* make sure no other Intel SGX instruction is accessing EPCM *)
IF (Other instructions accessing EPCM)
    THEN #GP(0); FI;

IF (EPCM(DS:RCX). VALID = 0)
    THEN #PF(DS:RCX); FI;

(* make sure that DS:RCX (DST) is pointing to a PT_REG or PT_TCS or PT_SS_FIRST or PT_SS_REST *)
IF ( (EPCM(DS:RCX).PT ≠ PT_REG) and (EPCM(DS:RCX).PT ≠ PT_TCS)
     and (EPCM(DS:RCX).PT ≠ PT_SS_FIRST) and (EPCM(DS:RCX).PT ≠ PT_SS_REST)    )
    THEN #PF(DS:RCX); FI;

TMP_SECS ← Get_SECS_ADDRESS();

IF (DS:RBX does not resolve to TMP_SECS)
    THEN #GP(0); FI;

(* make sure no other instruction is accessing MRENCLAVE or ATTRIBUTES.INIT *)
IF ( (Other instruction accessing MRENCLAVE) or (Other instructions checking or updating the initialized state of the SECS))
    THEN #GP(0); FI;

(* Calculate enclave offset *)
TMP_ENCLAVEOFFSET ←   EPCM(DS:RCX).ENCLAVEADDRESS - TMP_SECS.BASEADDR;
TMP_ENCLAVEOFFSET ←   TMP_ENCLAVEOFFSET + (DS:RCX & 0FFFH)
```

```
(* Add EEXTEND message and offset to MRENCLAVE *)
TMPUPDATEFIELD[63:0] ← 00444E4554584545H; // "EEXTEND"
TMPUPDATEFIELD[127:64] ← TMP_ENCLAVEOFFSET;
TMPUPDATEFIELD[511:128] ← 0; // 48 bytes
TMP_SECS.MRENCLAVE ← SHA256UPDATE(TMP_SECS.MRENCLAVE, TMPUPDATEFIELD)
INC enclave's MRENCLAVE update counter;

(*Add 256 bytes to MRENCLAVE, 64 byte at a time *)
TMP_SECS.MRENCLAVE ← SHA256UPDATE(TMP_SECS.MRENCLAVE, DS:RCX[511:0] );
TMP_SECS.MRENCLAVE ← SHA256UPDATE(TMP_SECS.MRENCLAVE, DS:RCX[1023: 512] );
TMP_SECS.MRENCLAVE ← SHA256UPDATE(TMP_SECS.MRENCLAVE, DS:RCX[1535: 1024] );
TMP_SECS.MRENCLAVE ← SHA256UPDATE(TMP_SECS.MRENCLAVE, DS:RCX[2047: 1536] );
INC enclave's MRENCLAVE update counter by 4;
```

Flags Affected

None

Protected Mode Exceptions

#GP(0)	If the address in RBX is outside the DS segment limit.
	If RBX points to an SECS page which is not the SECS of the data chunk.
	If the address in RCX is outside the DS segment limit.
	If RCX points to a memory location not 256Byte-aligned.
	If another instruction is accessing MRENCLAVE.
	If another instruction is checking or updating the SECS.
	If the enclave is already initialized.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If the address in RBX points to a non-EPC page.
	If the address in RCX points to a page which is not PT_TCS or PT_REG.
	If the address in RCX points to a non-EPC page.
	If the address in RCX points to an invalid EPC page.

64-Bit Mode Exceptions

#GP(0)	If RBX is non-canonical form.
	If RBX points to an SECS page which is not the SECS of the data chunk.
	If RCX is non-canonical form.
	If RCX points to a memory location not 256 Byte-aligned.
	If another instruction is accessing MRENCLAVE.
	If another instruction is checking or updating the SECS.
	If the enclave is already initialized.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If the address in RBX points to a non-EPC page.
	If the address in RCX points to a page which is not PT_TCS or PT_REG.
	If the address in RCX points to a non-EPC page.
	If the address in RCX points to an invalid EPC page.

EINIT—Initialize an Enclave for Execution

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 02H ENCLS[EINIT]	IR	V/V	SGX1	This leaf function initializes the enclave and makes it ready to execute enclave code.

Instruction Operand Encoding

Op/En	EAX		RBX	RCX	RDX
IR	EINIT (In)	Error code (Out)	Address of SIGSTRUCT (In)	Address of SECS (In)	Address of EINITTOKEN (In)

Description

This leaf function is the final instruction executed in the enclave build process. After EINIT, the MRENCLAVE measurement is complete, and the enclave is ready to start user code execution using the EENTER instruction.

EINIT takes the effective address of a SIGSTRUCT and EINITTOKEN. The SIGSTRUCT describes the enclave including MRENCLAVE, ATTRIBUTES, ISVSVN, a 3072 bit RSA key, and a signature using the included key. SIGSTRUCT must be populated with two values, q1 and q2. These are calculated using the formulas shown below:

q1 = floor(Signature^2 / Modulus);

q2 = floor((Signature^3 - q1 * Signature * Modulus) / Modulus);

The EINITTOKEN contains the MRENCLAVE, MRSIGNER, and ATTRIBUTES. These values must match the corresponding values in the SECS. If the EINITTOKEN was created with a debug launch key, the enclave must be in debug mode as well.

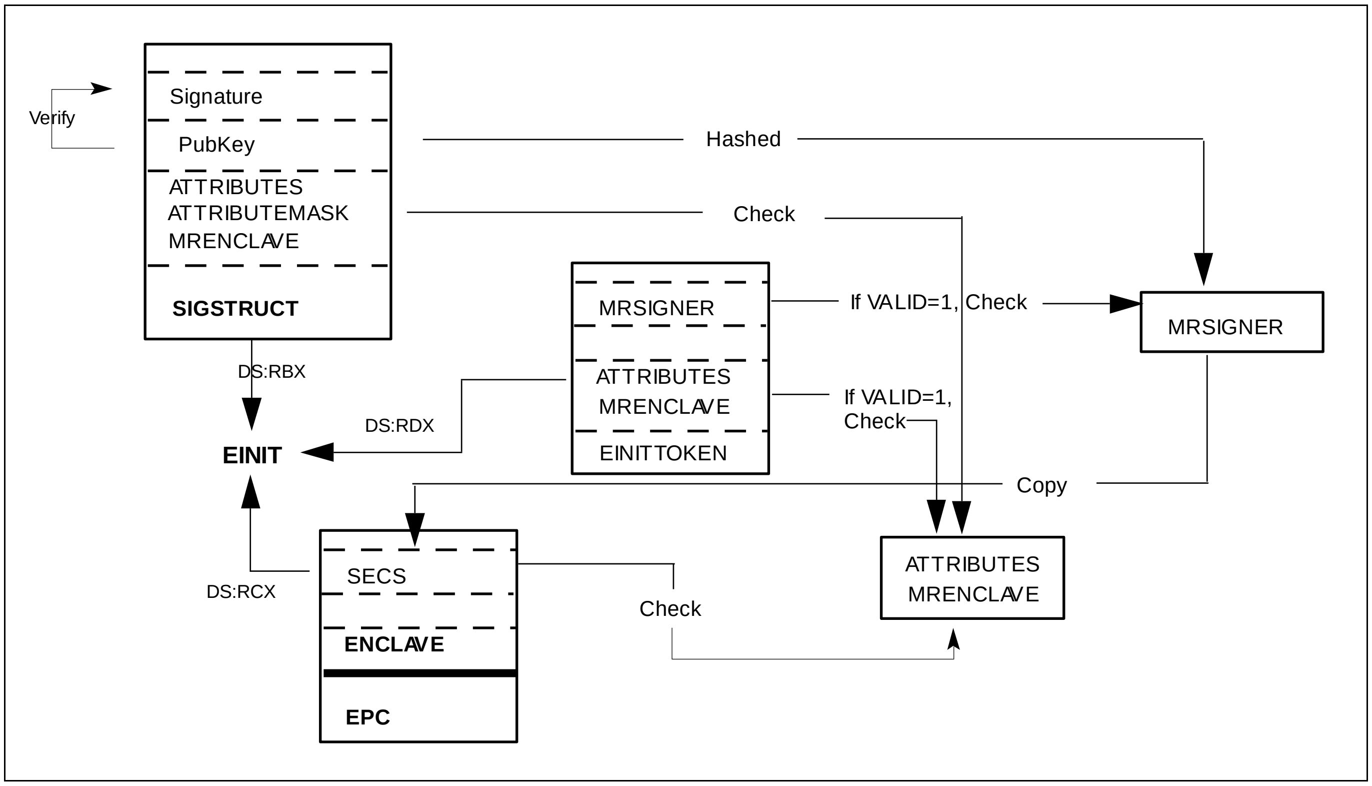

Figure 8 Relationships Between SECS, SIGSTRUCT and EINITTOKEN

EINIT Memory Parameter Semantics

SIGSTRUCT	SECS	EINITTOKEN
Access by non-Enclave	Read/Write access by Enclave	Access by non-Enclave

EINIT performs the following steps:

Validates that SIGSTRUCT is signed using the enclosed public key.

Checks that the completed computation of SECS.MRENCLAVE equals SIGSTRUCT.HASHENCLAVE.

Checks that no reserved bits are set to 1 in SIGSTRUCT.ATTRIBUTES and no reserved bits in SIGSTRUCT.ATTRIBUTESMASK are set to 0.

Checks that no controlled ATTRIBUTES bits are set in SIGSTRUCT.ATTRIBUTES unless the SHA256 digest of SIGSTRUCT.MODULUS equals IA32_SGX_LEPUBKEYHASH.

Checks that SIGSTRUCT.ATTRIBUTES equals the result of logically and-ing SIGSTRUCT.ATTRIBUTEMASK with SECS.ATTRIBUTES.

If EINITTOKEN.VALID is 0, checks that the SHA256 digest of SIGSTRUCT.MODULUS equals IA32_SGX_LEPUBKEYHASH.

If EINITTOKEN.VALID is 1, checks the validity of EINITTOKEN.

If EINITTOKEN.VALID is 1, checks that EINITTOKEN.MRENCLAVE equals SECS.MRENCLAVE.

If EINITTOKEN.VALID is 1 and EINITTOKEN.ATTRIBUTES.DEBUG is 1, SECS.ATTRIBUTES.DEBUG must be 1.

Commits SECS.MRENCLAVE, and sets SECS.MRSIGNER, SECS.ISVSVN, and SECS.ISVPRODID based on SIGSTRUCT.

Update the SECS as Initialized.

Periodically, EINIT polls for certain asynchronous events. If such an event is detected, it completes with failure code (ZF=1 and RAX = SGX_UNMASKED_EVENT), and RIP is incremented to point to the next instruction. These events includes external interrupts, non-maskable interrupts, system-management interrupts, machine checks, INIT signals, and the VMX-preemption timer. EINIT does not fail if the pending event is inhibited (e.g., external interrupts could be inhibited due to blocking by MOV SS blocking or by STI).

The following bits in RFLAGS are cleared: CF, PF, AF, OF, and SF. When the instruction completes with an error, RFLAGS.ZF is set to 1, and the corresponding error bit is set in RAX. If no error occurs, RFLAGS.ZF is cleared and RAX is set to 0.

The error codes are:

EINIT Return Value in RAX

Error Code	Description
No Error	EINIT successful.
SGX_INVALID_SIG_STRUCT	If SIGSTRUCT contained an invalid value.
SGX_INVALID_ATTRIBUTE	If SIGSTRUCT contains an unauthorized attributes mask.
SGX_INVALID_MEASUREMENT	If SIGSTRUCT contains an incorrect measurement. If EINITTOKEN contains an incorrect measurement.
SGX_INVALID_SIGNATURE	If signature does not validate with enclosed public key.
SGX_INVALID_LICENSE	If license is invalid.
SGX_INVALID_CPUSVN	If license SVN is unsupported.
SGX_UNMASKED_EVENT	If an unmasked event is received before the instruction completes its operation.

Concurrency Restrictions

Base Concurrency Restrictions of EINIT

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EINIT	SECS [DS:RCX]	Shared	#GP	

Additional Concurrency Restrictions of ENIT

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EINIT	SECS [DS:RCX]	Concurrent		Exclusive	#GP	Concurrent	

Operation

Temp Variables in EINIT Operational Flow

Name	Type	Size	Description
TMP_SIG	SIGSTRUCT	1808Bytes	Temp space for SIGSTRUCT.
TMP_TOKEN	EINITTOKEN	304Bytes	Temp space for EINITTOKEN.
TMP_MRENCLAVE		32Bytes	Temp space for calculating MRENCLAVE.
TMP_MRSIGNER		32Bytes	Temp space for calculating MRSIGNER.
CONTROLLED_ATTRIBUTES	ATTRIBUTES	16Bytes	Constant mask of all ATTRIBUTE bits that can only be set for authorized enclaves.
TMP_KEYDEPENDENCIES	Buffer	224Bytes	Temp space for key derivation.
TMP_EINITTOKENKEY		16Bytes	Temp space for the derived EINITTOKEN Key.
TMP_SIG_PADDING	PKCS Padding Buffer	352Bytes	The value of the top 352 bytes from the computation of Signature^3 modulo MRSIGNER.

```
(* make sure SIGSTRUCT and SECS are aligned *)
IF ( (DS:RBX is not 4KByte Aligned) or (DS:RCX is not 4KByte Aligned) )
    THEN #GP(0); FI;

(* make sure the EINITTOKEN is aligned *)
IF (DS:RDX is not 512Byte Aligned)
    THEN #GP(0); FI;

(* make sure the SECS is inside the EPC *)
```

```
IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;

TMP_SIG[14463:0] ← DS:RBX[14463:0]; // 1808 bytes
TMP_TOKEN[2423:0] ← DS:RDX[2423:0]; // 304 bytes

(* Verify SIGSTRUCT Header. *)
IF ( (TMP_SIG.HEADER ≠ 06000000E10000000000010000000000h) or
    ((TMP_SIG.VENDOR ≠ 0) and (TMP_SIG.VENDOR ≠ 00008086h) ) or
    (TMP_SIG HEADER2 ≠ 01010000600000006000000001000000h) or
    (TMP_SIG.EXPONENT     ≠ 00000003h) or (Reserved space is not 0's) )
    THEN
        RFLAGS.ZF ← 1;
        RAX ← SGX_INVALID_SIG_STRUCT;
        GOTO EXIT;
FI;

(* Open "Event Window" Check for Interrupts. Verify signature using embedded public key, q1, and q2. Save upper 352 bytes of the
PKCS1.5 encoded message into the TMP_SIG_PADDING*)
IF (interrupt was pending) THEN
    RFLAGS.ZF ← 1;
    RAX ← SGX_UNMASKED_EVENT;
    GOTO EXIT;
FI
IF (signature failed to verify) THEN
    RFLAGS.ZF ← 1;
    RAX ← SGX_INVALID_SIGNATURE;
    GOTO EXIT;
FI;
(*Close "Event Window" *)

(* make sure no other Intel SGX instruction is modifying SECS*)
IF (Other instructions modifying SECS)
    THEN #GP(0); FI;

IF ( (EPCM(DS:RCX). VALID = 0) or (EPCM(DS:RCX).PT ≠ PT_SECS) )
    THEN #PF(DS:RCX); FI;

(* Verify ISVFAMILYID is not used on an enclave with KSS disabled *)
IF ((TMP_SIG.ISVFAMILYID != 0) AND (DS:RCX.ATTRIBUTES.KSS == 0))
    THEN
         RFLAGS.ZF ← 1;
         RAX ← SGX_INVALID_SIG_STRUCT;
         GOTO EXIT;
FI;

(* make sure no other instruction is accessing MRENCLAVE or ATTRIBUTES.INIT *)
IF ( (Other instruction modifying MRENCLAVE) or (Other instructions modifying the SECS's Initialized state))
    THEN #GP(0); FI;

(* Calculate finalized version of MRENCLAVE *)
(* SHA256 algorithm requires one last update that compresses the length of the hashed message into the output SHA256 digest *)
TMP_ENCLAVE ←   SHA256FINAL( (DS:RCX).MRENCLAVE, enclave's MRENCLAVE update count *512);
```

```
(* Verify MRENCLAVE from SIGSTRUCT *)
IF (TMP_SIG.ENCLAVEHASH ≠ TMP_MRENCLAVE)
    RFLAGS.ZF ← 1;
    RAX ← SGX_INVALID_MEASUREMENT;
    GOTO EXIT;
FI;

TMP_MRSIGNER ← SHA256(TMP_SIG.MODULUS)

(* if controlled ATTRIBUTES are set, SIGSTRUCT must be signed using an authorized key *)
CONTROLLED_ATTRIBUTES ← 0000000000000020H;
IF ( ( (DS:RCX.ATTRIBUTES & CONTROLLED_ATTRIBUTES) ≠ 0) and (TMP_MRSIGNER ≠ IA32_SGXLEPUBKEYHASH) )
    RFLAGS.ZF ← 1;
    RAX ← SGX_INVALID_ATTRIBUTE;
    GOTO EXIT;
FI;

(* Verify SIGSTRUCT.ATTRIBUTE requirements are met *)
IF ( (DS:RCX.ATTRIBUTES & TMP_SIG.ATTRIBUTEMASK) ≠ (TMP_SIG.ATTRIBUTE & TMP_SIG.ATTRIBUTEMASK) )
    RFLAGS.ZF ← 1;
    RAX ← SGX_INVALID_ATTRIBUTE;
    GOTO EXIT;
FI;

( *Verify SIGSTRUCT.MISCSELECT requirements are met *)
IF ( (DS:RCX.MISCSELECT & TMP_SIG.MISCMASK) ≠ (TMP_SIG.MISCSELECT & TMP_SIG.MISCMASK) )
    THEN
        RFLAGS.ZF ← 1;
        RAX ← SGX_INVALID_ATTRIBUTE;
    GOTO EXIT
FI;

IF (CPUID.(EAX=12H, ECX=1):EAX[6] = 1)
    IF ( DS:RCX.CET_ATTRIBUTES & TMP_SIG.CET_ATTRIBUTES_MASK ≠ TMP_SIG.CET_ATTRIBUTES & TMP_SIG.CET_ATTRIB-
    UTES_MASK )
        THEN
            RFLAGS.ZF ← 1;
            RAX ← SGX_INVALID_ATTRIBUTE;
            GOTO EXIT
        FI;
FI;

(* if EINITTOKEN.VALID[0] is 0, verify the enclave is signed by an authorized key *)
IF (TMP_TOKEN.VALID[0] = 0)
    IF (TMP_MRSIGNER ≠ IA32_SGXLEPUBKEYHASH)
        RFLAGS.ZF ← 1;
        RAX ← SGX_INVALID_EINITTOKEN;
        GOTO EXIT;
    FI;
    GOTO COMMIT;
FI;

(* Debug Launch Enclave cannot launch Production Enclaves *)
IF ( (DS:RDX.MASKEDATTRIBUTESLE.DEBUG = 1) and (DS:RCX.ATTRIBUTES.DEBUG = 0) )
    RFLAGS.ZF ← 1;
    RAX ← SGX_INVALID_EINITTOKEN;
    GOTO EXIT;
FI;

(* Check reserve space in EINIT token includes reserved regions and upper bits in valid field *)
IF (TMP_TOKEN reserved space is not clear)
    RFLAGS.ZF ← 1;
    RAX ← SGX_INVALID_EINITTOKEN;
    GOTO EXIT;
FI;

(* EINIT token must be ≤ CR_CPUSVN *)
IF (TMP_TOKEN.CPUSVN > CR_CPUSVN)
    RFLAGS.ZF ← 1;
    RAX ← SGX_INVALID_CPUSVN;
    GOTO EXIT;
FI;

(* Derive Launch key used to calculate EINITTOKEN.MAC *)
HARDCODED_PKCS1_5_PADDING[15:0] ← 0100H;
HARDCODED_PKCS1_5_PADDING[2655:16] ← SignExtend330Byte(-1); // 330 bytes of 0FFH
HARDCODED_PKCS1_5_PADDING[2815:2656] ← 2004000501020403650148866009060D30313000H;

TMP_KEYDEPENDENCIES.KEYNAME ← EINITTOKEN_KEY;
TMP_KEYDEPENDENCIES.ISVFAMILYID ← 0;
TMP_KEYDEPENDENCIES.ISVEXTPRODID ← 0;
TMP_KEYDEPENDENCIES.ISVPRODID ← TMP_TOKEN.ISVPRODIDLE;
TMP_KEYDEPENDENCIES.ISVSVN ← TMP_TOKEN.ISVSVN;
TMP_KEYDEPENDENCIES.SGXOWNEREPOCH ← CR_SGXOWNEREPOCH;
TMP_KEYDEPENDENCIES.ATTRIBUTES ← TMP_TOKEN.MASKEDATTRIBUTESLE;
TMP_KEYDEPENDENCIES.ATTRIBUTESMASK ← 0;
TMP_KEYDEPENDENCIES.MRENCLAVE ← 0;
TMP_KEYDEPENDENCIES.MRSIGNER ← IA32_SGXLEPUBKEYHASH;
TMP_KEYDEPENDENCIES.KEYID ← TMP_TOKEN.KEYID;
TMP_KEYDEPENDENCIES.SEAL_KEY_FUSES ← CR_SEAL_FUSES;
TMP_KEYDEPENDENCIES.CPUSVN ← TMP_TOKEN.CPUSVN;
TMP_KEYDEPENDENCIES.MISCSELECT ← TMP_TOKEN.MASKEDMISCSELECTLE;
TMP_KEYDEPENDENCIES.MISCMASK ← 0;
TMP_KEYDEPENDENCIES.PADDING ← HARDCODED_PKCS1_5_PADDING;
TMP_KEYDEPENDENCIES.KEYPOLICY ← 0;
TMP_KEYDEPENDENCIES.CONFIGID ← 0;
TMP_KEYDEPENDENCIES.CONFIGSVN ← 0;
IF (CPUID.(EAX=12H, ECX=1):EAX[6] = 1))
        TMP_KEYDEPENDENCIES.CET_ATTRIBUTES ← TMP_TOKEN.CET_MASKED_ATTRIBUTES_ LE;
        TMP_KEYDEPENDENCIES.CET_ATTRIBUTES_MASK ← 0;
FI;
```

```
(* Calculate the derived key*)
TMP_EINITTOKENKEY ← derivekey(TMP_KEYDEPENDENCIES);

(* Verify EINITTOKEN was generated using this CPU's Launch key and that it has not been modified since issuing by the Launch En-
clave. Only 192 bytes of EINITTOKEN are CMACed *)
IF (TMP_TOKEN.MAC ≠ CMAC(TMP_EINITTOKENKEY, TMP_TOKEN[1535:0] ) )
    RFLAGS.ZF ← 1;
    RAX ← SGX_INVALID_EINITTOKEN;
    GOTO EXIT;
FI;

(* Verify EINITTOKEN (RDX) is for this enclave *)
IF ( (TMP_TOKEN.MRENCLAVE ≠ TMP_MRENCLAVE) or (TMP_TOKEN.MRSIGNER ≠ TMP_MRSIGNER) )
    RFLAGS.ZF ← 1;
    RAX ← SGX_INVALID_MEASUREMENT;
    GOTO EXIT;
FI;

(* Verify ATTRIBUTES in EINITTOKEN are the same as the enclave's *)
IF (TMP_TOKEN.ATTRIBUTES ≠ DS:RCX.ATTRIBUTES)
    RFLAGS.ZF ← 1;
    RAX ← SGX_INVALID_EINIT_ATTRIBUTE;
    GOTO EXIT;
FI;

COMMIT:
(* Commit changes to the SECS; Set ISVPRODID, ISVSVN, MRSIGNER, INIT ATTRIBUTE fields in SECS (RCX) *)
DS:RCX.MRENCLAVE ← TMP_MRENCLAVE;
(* MRSIGNER stores a SHA256 in little endian implemented natively on x86 *)
DS:RCX.MRSIGNER ← TMP_MRSIGNER;
DS:RCX.ISVEXTPRODID ← TMP_SIG.ISVEXTPRODID;
DS:RCX.ISVPRODID ← TMP_SIG.ISVPRODID;
DS:RCX.ISVSVN ← TMP_SIG.ISVSVN;
DS:RCX.ISVFAMILYID ← TMP_SIG.ISVFAMILYID;
DS:RCX.PADDING ← TMP_SIG_PADDING;

(* Mark the SECS as initialized *)
Update DS:RCX to initialized;

(* Set RAX and ZF for success*)
    RFLAGS.ZF ← 0;
    RAX ← 0;
EXIT:
RFLAGS.CF,PF,AF,OF,SF ← 0;
```

Flags Affected

ZF is cleared if successful, otherwise ZF is set and RAX contains the error code. CF, PF, AF, OF, SF are cleared.

Protected Mode Exceptions

#GP(0)	If a memory operand is not properly aligned.
	If another instruction is modifying the SECS.
	If the enclave is already initialized.
	If the SECS.MRENCLAVE is in use.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If RCX does not resolve in an EPC page.
	If the memory address is not a valid, uninitialized SECS.

64-Bit Mode Exceptions

#GP(0)	If a memory operand is not properly aligned.
	If another instruction is modifying the SECS.
	If the enclave is already initialized.
	If the SECS.MRENCLAVE is in use.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If RCX does not resolve in an EPC page.
	If the memory address is not a valid, uninitialized SECS.

ELDB/ELDU/ELDBC/ELBUC—Load an EPC Page and Mark its State

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 07H ENCLS[ELDB]	IR	V/V	SGX1	This leaf function loads, verifies an EPC page and marks the page as blocked.
EAX = 08H ENCLS[ELDU]	IR	V/V	SGX1	This leaf function loads, verifies an EPC page and marks the page as unblocked.
EAX = 12H ENCLS[ELDBC]	IR	V/V	EAX[5]	This leaf function behaves lie ELDB but with improved conflict handling for oversubscription.
EAX = 13H ENCLS[ELDBC]	IR	V/V	EAX[5]	This leaf function behaves like ELDU but with improved conflict handling for oversubscription.

Instruction Operand Encoding

Op/En	EAX		RBX	RCX	RDX
IR	ELDB/ELDU (In)	Return error code (Out)	Address of the PAGEINFO (In)	Address of the EPC page (In)	Address of the version-array slot (In)

Description

This leaf function copies a page from regular main memory to the EPC. As part of the copying process, the page is cryptographically authenticated and decrypted. This instruction can only be executed when current privilege level is 0.

The ELDB leaf function sets the BLOCK bit in the EPCM entry for the destination page in the EPC after copying. The ELDU leaf function clears the BLOCK bit in the EPCM entry for the destination page in the EPC after copying.

RBX contains the effective address of a PAGEINFO structure; RCX contains the effective address of the destination EPC page; RDX holds the effective address of the version array slot that holds the version of the page.

The ELDBC/ELDUC leafs are very similar to ELDB and ELDU. They provide an error code on the concurrency conflict for any of the pages which need to acquire a lock. These include the destination, SECS, and VA slot.

The table below provides additional information on the memory parameter of ELDB/ELDU leaf functions.

ELDB/ELDU/ELDBC/ELBUC Memory Parameter Semantics

PAGEINFO	PAGEINFO.SRCPGE	PAGEINFO.PCMD	PAGEINFO.SECS	EPCPAGE	Version-Array Slot
Non-enclave read access	Non-enclave read access	Non-enclave read access	Enclave read/write access	Read/Write access permitted by Enclave	Read/Write access permitted by Enclave

The error codes are:

ELDB/ELDU/ELDBC/ELBUC Return Value in RAX

Error Code	Description
No Error	ELDB/ELDU successful.
SGX_MAC_COMPARE_FAIL	If the MAC check fails.

Concurrency Restrictions

Base Concurrency Restrictions of ELDB/ELDU/ELDBC/ELBUC

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
ELDB/ELDU/	Target [DS:RCX]	Exclusive	#GP	EPC_PAGE_CONFLICT_EXCEPTION
	VA [DS:RDX]	Shared	#GP	
	SECS [DS:RBX]PAGEINFO.SECS	Shared	#GP	
ELDBC/ELBUC	Target [DS:RCX]	Exclusive	SGX_EPC_PAGE_ CONFLICT	EPC_PAGE_CONFLICT_ERROR
	VA [DS:RDX]	Shared	SGX_EPC_PAGE_ CONFLICT	
	SECS [DS:RBX]PAGEINFO.SECS	Shared	SGX_EPC_PAGE_ CONFLICT	

Additional Concurrency Restrictions of ELDB/ELDU/ELDBC/ELBUC

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
ELDB/ELDU/	Target [DS:RCX]	Concurrent		Concurrent		Concurrent	
	VA [DS:RDX]	Concurrent		Concurrent		Concurrent	
	SECS [DS:RBX]PAGEINFO.SECS	Concurrent		Concurrent		Concurrent	
ELDBC/ELBUC	Target [DS:RCX]	Concurrent		Concurrent		Concurrent	
	VA [DS:RDX]	Concurrent		Concurrent		Concurrent	
	SECS [DS:RBX]PAGEINFO.SECS	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in ELDB/ELDU/ELDBC/ELBUC Operational Flow

Name	Type	Size (Bits)	Description
TMP_SRCPGE	Memory page	4KBytes	
TMP_SECS	Memory page	4KBytes	
TMP_PCMD	PCMD	128 Bytes	
TMP_HEADER	MACHEADER	128 Bytes	
TMP_VER	UINT64	64	
TMP_MAC	UINT128	128	
TMP_PK	UINT128	128	Page encryption/MAC key.
SCRATCH_PCMD	PCMD	128 Bytes	

```
(* Check PAGEINFO and EPCPAGE alignment *)
IF ( (DS:RBX is not 32Byte Aligned) or (DS:RCX is not 4KByte Aligned) )
    THEN #GP(0); FI;
```

```
IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;

(* Check VASLOT alignment *)
IF (DS:RDX is not 8Byte aligned)
    THEN #GP(0); FI;

IF (DS:RDX does not resolve within an EPC)
    THEN #PF(DS:RDX); FI;

TMP_SRCPGE ← DS:RBX.SRCPGE;
TMP_SECS ← DS:RBX.SECS;
TMP_PCMD ← DS:RBX.PCMD;

(* Check alignment of PAGEINFO (RBX) linked parameters. Note: PCMD pointer is overlaid on top of PAGEINFO.SECINFO field *)
IF ( (DS:TMP_PCMD is not 128Byte aligned) or (DS:TMP_SRCPGE is not 4KByte aligned) )
    THEN #GP(0); FI;

(* Check concurrency of EPC by other Intel SGX instructions *)
IF (other instructions accessing EPC)
    THEN
        IF ((EAX==07h) OR (EAX==08h))     (* ELDB/ELDU *)
            THEN
                    IF (<<VMX non-root operation>> AND
                    <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>>)
                    THEN
                        VMCS.Exit_reason ← SGX_CONFLICT;
                        VMCS.Exit_qualification.code ← EPC_PAGE_CONFLICT_EXCEPTION;
                        VMCS.Exit_qualification.error ← 0;
                        VMCS.Guest-physical_address ←
                                << translation of DS:RCX produced by paging >>;
                        VMCS.Guest-linear_address ← DS:RCX;
                        Deliver VMEXIT;
                    ELSE
                        #GP(0);
                FI;
            ELSE (* ELDBC/ELDUC *)
                    IF (<<VMX non-root operation>> AND
                    <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>>)
                    THEN
                        VMCS.Exit_reason ← SGX_CONFLICT;
                        VMCS.Exit_qualification.code ← EPC_PAGE_CONFLICT_ERROR;
                        VMCS.Exit_qualification.error ← SGX_EPC_PAGE_CONFLICT;
                        VMCS.Guest-physical_address ←
                                << translation of DS:RCX produced by paging >>;
                        VMCS.Guest-linear_address ← DS:RCX;
                        Deliver VMEXIT;
                    ELSE
                            RFLAGS.ZF ← 1;
                        RFLAGS.CF ← 0;
                            RAX ← SGX_EPC_PAGE_CONFLICT;
                            GOTO ERROR_EXIT;
                FI;
        FI;
FI;

(* Check concurrency of EPC and VASLOT by other Intel SGX instructions *)
IF (Other instructions modifying VA slot)
    THEN
        IF ((EAX==07h) OR (EAX==08h))     (* ELDB/ELDU *)
         #GP(0);
        FI;
     ELSE (* ELDBC/ELDUC *)
         RFLAGS.ZF ← 1;
         RFLAGS.CF ← 0;
         RAX ← SGX_EPC_PAGE_CONFLICT;
         GOTO ERROR_EXIT;
FI;

(* Verify EPCM attributes of EPC page, VA, and SECS *)
IF (EPCM(DS:RCX).VALID = 1)
    THEN #PF(DS:RCX); FI;

IF ( (EPCM(DS:RDX & ~0FFFH).VALID = 0) or (EPCM(DS:RDX & ~0FFFH).PT ≠ PT_VA) )
    THEN #PF(DS:RDX); FI;

(* Copy PCMD into scratch buffer *)
SCRATCH_PCMD[1023: 0]← DS:TMP_PCMD[1023:0];

(* Zero out TMP_HEADER*)
TMP_HEADER[sizeof(TMP_HEADER)-1: 0]← 0;

TMP_HEADER.SECINFO ← SCRATCH_PCMD.SECINFO;
TMP_HEADER.RSVD ← SCRATCH_PCMD.RSVD;
TMP_HEADER.LINADDR ← DS:RBX.LINADDR;

(* Verify various attributes of SECS parameter *)
IF ( (TMP_HEADER.SECINFO.FLAGS.PT = PT_REG) or (TMP_HEADER.SECINFO.FLAGS.PT = PT_TCS) or
     (TMP_HEADER.SECINFO.FLAGS.PT = PT_TRIM) or
      (TMP_HEADER.SECINFO.FLAGS.PT = PT_SS_FIRST and CPUID.(EAX=12H, ECX=1):EAX[6] = 1)) or
      (TMP_HEADER.SECINFO.FLAGS.PT = PT_SS_REST and CPUID.(EAX=12H, ECX=1):EAX[6] = 1)))
    THEN
        IF ( DS:TMP_SECS is not 4KByte aligned)
            THEN #GP(0) FI;
        IF (DS:TMP_SECS does not resolve within an EPC)
            THEN #PF(DS:TMP_SECS) FI;
        IF ( Other instructions modifying SECS)
            THEN
                IF ((EAX==07h) OR (EAX==08h))     (* ELDB/ELDU *)
                        #GP(0);
                FI;
            ELSE (* ELDBC/ELDUC *)
```

```
                    RFLAGS.ZF ← 1;
                        RFLAGS.CF ← 0;
                        RAX ← SGX_EPC_PAGE_CONFLICT;
                        GOTO ERROR_EXIT;
            FI;
FI;

IF ( (TMP_HEADER.SECINFO.FLAGS.PT = PT_REG) or (TMP_HEADER.SECINFO.FLAGS.PT = PT_TCS) or
     (TMP_HEADER.SECINFO.FLAGS.PT = PT_TRIM) or
      (TMP_HEADER.SECINFO.FLAGS.PT = PT_SS_FIRST and CPUID.(EAX=12H, ECX=1):EAX[6] = 1)) or
      (TMP_HEADER.SECINFO.FLAGS.PT = PT_SS_REST and CPUID.(EAX=12H, ECX=1):EAX[6] = 1)))
    THEN
        TMP_HEADER.EID ← DS:TMP_SECS.EID;
    ELSE
        (* These pages do not have any parent, and hence no EID binding *)
        TMP_HEADER.EID ← 0;
FI;

(* Copy 4KBytes SRCPGE to secure location *)
DS:RCX[32767: 0]← DS:TMP_SRCPGE[32767: 0];
TMP_VER ← DS:RDX[63:0];

(* Decrypt and MAC page. AES_GCM_DEC has 2 outputs, {plain text, MAC} *)
(* Parameters for AES_GCM_DEC {Key, Counter, ..} *)
{DS:RCX, TMP_MAC} ← AES_GCM_DEC(CR_BASE_PK, TMP_VER << 32, TMP_HEADER, 128, DS:RCX, 4096);

IF ( (TMP_MAC ≠ DS:TMP_PCMD.MAC) )
    THEN
        RFLAGS.ZF ← 1;
        RAX← SGX_MAC_COMPARE_FAIL;
        GOTO ERROR_EXIT;
FI;

(* Check version before committing *)
IF (DS:RDX ≠ 0)
    THEN #GP(0);
    ELSE
        DS:RDX← TMP_VER;
FI;

(* Commit EPCM changes *)
EPCM(DS:RCX).PT ← TMP_HEADER.SECINFO.FLAGS.PT;
EPCM(DS:RCX).RWX ← TMP_HEADER.SECINFO.FLAGS.RWX;
EPCM(DS:RCX).PENDING ← TMP_HEADER.SECINFO.FLAGS.PENDING;
EPCM(DS:RCX).MODIFIED ← TMP_HEADER.SECINFO.FLAGS.MODIFIED;
EPCM(DS:RCX).PR ← TMP_HEADER.SECINFO.FLAGS.PR;
EPCM(DS:RCX).ENCLAVEADDRESS ← TMP_HEADER.LINADDR;

IF ( ((EAX = 07H) or (EAX = 12H)) and (TMP_HEADER.SECINFO.FLAGS.PT is NOT PT_SECS or PT_VA))
    THEN
        EPCM(DS:RCX).BLOCKED ← 1;
    ELSE
        EPCM(DS:RCX).BLOCKED ← 0;
```

```
FI;

IF (TMP_HEADER.SECINFO.FLAGS.PT is PT_SECS)
        << store translation of DS:RCX produced by paging in SECS(DS:RCX).ENCLAVECONTEXT >>
FI;

EPCM(DS:RCX). VALID ← 1;

RAX← 0;
RFLAGS.ZF ← 0;

ERROR_EXIT:
RFLAGS.CF,PF,AF,OF,SF ← 0;
```

Flags Affected

Sets ZF if unsuccessful, otherwise cleared and RAX returns error code. Clears CF, PF, AF, OF, SF.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the DS segment limit.
	If a memory operand is not properly aligned.
	If the instruction’s EPC resource is in use by others.
	If the instruction fails to verify MAC.
	If the version-array slot is in use.
	If the parameters fail consistency checks.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If a memory operand expected to be in EPC does not resolve to an EPC page.
	If one of the EPC memory operands has incorrect page type.
	If the destination EPC page is already valid.

64-Bit Mode Exceptions

#GP(0)	If a memory operand is non-canonical form.
	If a memory operand is not properly aligned.
	If the instruction’s EPC resource is in use by others.
	If the instruction fails to verify MAC.
	If the version-array slot is in use.
	If the parameters fail consistency checks.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If a memory operand expected to be in EPC does not resolve to an EPC page.
	If one of the EPC memory operands has incorrect page type.
	If the destination EPC page is already valid.

EMODPR—Restrict the Permissions of an EPC Page

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 0EH ENCLS[EMODPR]	IR	V/V	SGX2	This leaf function restricts the access rights associated with a EPC page in an initialized enclave.

Instruction Operand Encoding

Op/En	EAX		RBX	RCX
IR	EMODPR (In)	Return Error Code (Out)	Address of a SECINFO (In)	Address of the destination EPC page (In)

Description

This leaf function restricts the access rights associated with an EPC page in an initialized enclave. THE RWX bits of the SECINFO parameter are treated as a permissions mask; supplying a value that does not restrict the page permissions will have no effect. This instruction can only be executed when current privilege level is 0.

RBX contains the effective address of a SECINFO structure while RCX contains the effective address of an EPC page. The table below provides additional information on the memory parameter of the EMODPR leaf function.

EMODPR Memory Parameter Semantics

SECINFO	EPCPAGE
Read access permitted by Non Enclave	Read/Write access permitted by Enclave

EMODPR Faulting Conditions

The instruction faults if any of the following:

The operands are not properly aligned.	If unsupported security attributes are set.
The Enclave is not initialized.	SECS is locked by another thread.
The EPC page is locked by another thread.	RCX does not contain an effective address of an EPC page in the running enclave.
The EPC page is not valid.	

The error codes are:

EMODPR Return Value in RAX

Error Code	Description
No Error	EMODPR successful.
SGX_PAGE_NOT_MODIFIABLE	The EPC page cannot be modified because it is in the PENDING or MODIFIED state.
SGX_EPC_PAGE_CONFLICT	Page is being written by EADD, EAUG, ECREATE, ELDU/B, EMODT, or EWB.

Concurrency Restrictions

Base Concurrency Restrictions of EMODPR

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EMODPR	Target [DS:RCX]	Shared	#GP	

Additional Concurrency Restrictions of EMODPR

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EMODPR	Target [DS:RCX]	Exclusive	SGX_EPC_PAGE_CONFLICT	Concurrent		Concurrent	

Operation

Temp Variables in EMODPR Operational Flow

Name	Type	Size (bits)	Description
TMP_SECS	Effective Address	32/64	Physical address of SECS to which EPC operand belongs.
SCRATCH_SECINFO	SECINFO	512	Scratch storage for holding the contents of DS:RBX.

```
IF (DS:RBX is not 64Byte Aligned)
    THEN #GP(0); FI;

IF (DS:RCX is not 4KByte Aligned)
    THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;

SCRATCH_SECINFO ← DS:RBX;

(* Check for misconfigured SECINFO flags*)
IF ( (SCRATCH_SECINFO reserved fields are not zero ) or
    (SCRATCH_SECINFO.FLAGS.R is 0 and SCRATCH_SECINFO.FLAGS.W is not 0) )
    THEN #GP(0); FI;

(* Check concurrency with SGX1 or SGX2 instructions on the EPC page *)
IF (SGX1 or other SGX2 instructions accessing EPC page)
    THEN #GP(0); FI;

IF (EPCM(DS:RCX).VALID is 0 )
    THEN #PF(DS:RCX); FI;

(* Check the EPC page for concurrency *)
IF (EPC page in use by another SGX2 instruction)
    THEN
        RFLAGS.ZF ← 1;
        RAX ← SGX_EPC_PAGE_CONFLICT;
        GOTO DONE;
FI;

IF (EPCM(DS:RCX).PENDING is not 0 or (EPCM(DS:RCX).MODIFIED is not 0) )
    THEN
        RFLAGS.ZF ← 1;
        RAX ← SGX_PAGE_NOT_MODIFIABLE;
        GOTO DONE;
FI;

IF (EPCM(DS:RCX).PT is not PT_REG)
    THEN #PF(DS:RCX); FI;
```

```
TMP_SECS ← GET_SECS_ADDRESS

IF (TMP_SECS.ATTRIBUTES.INIT = 0)
    THEN #GP(0); FI;

(* Set the PR bit to indicate that permission restriction is in progress *)
EPCM(DS:RCX).PR ← 1;

(* Update EPCM permissions *)
EPCM(DS:RCX).R ← EPCM(DS:RCX).R & SCRATCH_SECINFO.FLAGS.R;
EPCM(DS:RCX).W ← EPCM(DS:RCX).W & SCRATCH_SECINFO.FLAGS.W;
EPCM(DS:RCX).X ← EPCM(DS:RCX).X & SCRATCH_SECINFO.FLAGS.X;

RFLAGS.ZF ← 0;
RAX ← 0;

DONE:
RFLAGS.CF,PF,AF,OF,SF ← 0;
```

Flags Affected

Sets ZF if page is not modifiable or if other SGX2 instructions are executing concurrently, otherwise cleared. Clears CF, PF, AF, OF, SF.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the DS segment limit.
	If a memory operand is not properly aligned.
	If a memory operand is locked.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If a memory operand is not an EPC page.

64-Bit Mode Exceptions

#GP(0)	If a memory operand is non-canonical form.
	If a memory operand is not properly aligned.
	If a memory operand is locked.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If a memory operand is not an EPC page.

EMODT—Change the Type of an EPC Page

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 0FH ENCLS[EMODT]	IR	V/V	SGX2	This leaf function changes the type of an existing EPC page.

Instruction Operand Encoding

Op/En	EAX		RBX	RCX
IR	EMODT (In)	Return Error Code (Out)	Address of a SECINFO (In)	Address of the destination EPC page (In)

Description

This leaf function modifies the type of an EPC page. The security attributes are configured to prevent access to the EPC page at its new type until a corresponding invocation of the EACCEPT leaf confirms the modification. This instruction can only be executed when current privilege level is 0.

RBX contains the effective address of a SECINFO structure while RCX contains the effective address of an EPC page. The table below provides additional information on the memory parameter of the EMODT leaf function.

EMODT Memory Parameter Semantics

SECINFO	EPCPAGE
Read access permitted by Non Enclave	Read/Write access permitted by Enclave

EMODT Faulting Conditions

The instruction faults if any of the following:

The operands are not properly aligned.	If unsupported security attributes are set.
The Enclave is not initialized.	SECS is locked by another thread.
The EPC page is locked by another thread.	RCX does not contain an effective address of an EPC page in the running enclave.
The EPC page is not valid.	

The error codes are:

EMODT Return Value in RAX

Error Code	Description
No Error	EMODT successful.
SGX_PAGE_NOT_MODIFIABLE	The EPC page cannot be modified because it is in the PENDING or MODIFIED state.
SGX_EPC_PAGE_CONFLICT	Page is being written by EADD, EAUG, ECREATE, ELDU/B, EMODPR, or EWB.

Concurrency Restrictions

Base Concurrency Restrictions of EMODT

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EMODT	Target [DS:RCX]	Exclusive	SGX_EPC_PAGE_CONFLICT	EPC_PAGE_CONFLICT_ERROR

Additional Concurrency Restrictions of EMODT

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EMODT	Target [DS:RCX]	Exclusive	SGX_EPC_PAGE_CONFLICT	Concurrent		Concurrent	

Operation

Temp Variables in EMODT Operational Flow

Name	Type	Size (bits)	Description
TMP_SECS	Effective Address	32/64	Physical address of SECS to which EPC operand belongs.
SCRATCH_SECINFO	SECINFO	512	Scratch storage for holding the contents of DS:RBX.

```
IF (DS:RBX is not 64Byte Aligned)
    THEN #GP(0); FI;

IF (DS:RCX is not 4KByte Aligned)
    THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;

SCRATCH_SECINFO ← DS:RBX;

(* Check for misconfigured SECINFO flags*)
IF ( (SCRATCH_SECINFO reserved fields are not zero ) or
    !(SCRATCH_SECINFO.FLAGS.PT is PT_TCS or SCRATCH_SECINFO.FLAGS.PT is PT_TRIM) )
    THEN #GP(0); FI;

(* Check concurrency with SGX1 instructions on the EPC page *)
IF (other SGX1 instructions accessing EPC page)
    THEN
        RFLAGS.ZF ← 1;
        RAX ← SGX_EPC_PAGE_CONFLICT;
        GOTO DONE;
FI;

IF (EPCM(DS:RCX).VALID is 0)
    THEN #PF(DS:RCX); FI;

(* Check the EPC page for concurrency *)
IF (EPC page in use by another SGX2 instruction)
    THEN
        RFLAGS.ZF ← 1;
        RAX ← SGX_EPC_PAGE_CONFLICT;
        GOTO DONE;
FI;

IF (!(EPCM(DS:RCX).PT is PT_REG or
    ((EPCM(DS:RCX).PT is PT_TCS or PT_SS_FIRST or PT_SS_REST) and SCRATCH_SECINFO.FLAGS.PT is PT_TRIM)))
        THEN #PF(DS:RCX); FI;

IF (EPCM(DS:RCX).PENDING is not 0 or (EPCM(DS:RCX).MODIFIED is not 0) )
```

```
    THEN
        RFLAGS.ZF ← 1;
        RAX ← SGX_PAGE_NOT_MODIFIABLE;
        GOTO DONE;
FI;

TMP_SECS ← GET_SECS_ADDRESS

IF (TMP_SECS.ATTRIBUTES.INIT = 0)
    THEN #GP(0); FI;

(* Update EPCM fields *)
EPCM(DS:RCX).PR ← 0;
EPCM(DS:RCX).MODIFIED ← 1;
EPCM(DS:RCX).R ← 0;
EPCM(DS:RCX).W ← 0;
EPCM(DS:RCX).X ← 0;
EPCM(DS:RCX).PT ← SCRATCH_SECINFO.FLAGS.PT;

RFLAGS.ZF ← 0;
RAX ← 0;

DONE:
RFLAGS.CF,PF,AF,OF,SF ← 0;
```

Flags Affected

Sets ZF if page is not modifiable or if other SGX2 instructions are executing concurrently, otherwise cleared. Clears CF, PF, AF, OF, SF.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the DS segment limit.
	If a memory operand is not properly aligned.
	If a memory operand is locked.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If a memory operand is not an EPC page.

64-Bit Mode Exceptions

#GP(0)	If a memory operand is non-canonical form.
	If a memory operand is not properly aligned.
	If a memory operand is locked.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If a memory operand is not an EPC page.

EPA—Add Version Array

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 0AH ENCLS[EPA]	IR	V/V	SGX1	This leaf function adds a Version Array to the EPC.

Instruction Operand Encoding

Op/En	EAX	RBX	RCX
IR	EPA (In)	PT_VA (In, Constant)	Effective address of the EPC page (In)

Description

This leaf function creates an empty version array in the EPC page whose logical address is given by DS:RCX, and sets up EPCM attributes for that page. At the time of execution of this instruction, the register RBX must be set to PT_VA.

The table below provides additional information on the memory parameter of EPA leaf function.

EPA Memory Parameter Semantics

EPCPAGE
Write access permitted by Enclave

Concurrency Restrictions

Base Concurrency Restrictions of EPA

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EPA	VA [DS:RCX]	Exclusive	#GP	EPC_PAGE_CONFLICT_EXCEPTION

Additional Concurrency Restrictions of EPA

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EPA	VA [DS:RCX]	Concurrent	L	Concurrent		Concurrent	

Operation

```
IF (RBX ≠ PT_VA or DS:RCX is not 4KByte Aligned)
    THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;

(* Check concurrency with other Intel SGX instructions *)
IF (Other Intel SGX instructions accessing the page)
    THEN
        IF (<<VMX non-root operation>> AND <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>>)
            THEN
                VMCS.Exit_reason ← SGX_CONFLICT;
                VMCS.Exit_qualification.code ← EPC_PAGE_CONFLICT_EXCEPTION;
                VMCS.Exit_qualification.error ← 0;
                VMCS.Guest-physical_address ←<< translation of DS:RCX produced by paging >>;
                VMCS.Guest-linear_address ← DS:RCX;
                    Deliver VMEXIT;
            ELSE
                #GP(0);
        FI;
FI;

(* Check EPC page must be empty *)
IF (EPCM(DS:RCX). VALID ≠ 0)
    THEN #PF(DS:RCX); FI;

(* Clears EPC page *)
DS:RCX[32767:0] ← 0;

EPCM(DS:RCX).PT ← PT_VA;
EPCM(DS:RCX).ENCLAVEADDRESS ← 0;
EPCM(DS:RCX).BLOCKED ← 0;
EPCM(DS:RCX).PENDING ← 0;
EPCM(DS:RCX).MODIFIED ← 0;
EPCM(DS:RCX).PR ← 0;
EPCM(DS:RCX).RWX ← 0;
EPCM(DS:RCX).VALID ← 1;
```

Flags Affected

None

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the DS segment limit.
	If a memory operand is not properly aligned.
	If another Intel SGX instruction is accessing the EPC page.
	If RBX is not set to PT_VA.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If a memory operand is not an EPC page.
	If the EPC page is valid.

64-Bit Mode Exceptions

#GP(0)	If a memory operand is non-canonical form.
	If a memory operand is not properly aligned.
	If another Intel SGX instruction is accessing the EPC page.
	If RBX is not set to PT_VA.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If a memory operand is not an EPC page.
	If the EPC page is valid.

ERDINFO—Read Type and Status Information About an EPC Page

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 10H ENCLS[ERDINFO]	IR	V/V	EAX[6]	This leaf function returns type and status information about an EPC page.

Instruction Operand Encoding

Op/En	EAX	RBX	RCX
IR	ERDINFO (In)	Address of a RDINFO structure (In)	Address of the destination EPC page (In)

Description

This instruction reads type and status information about an EPC page and returns it in a RDINFO structure. The STATUS field of the structure describes the status of the page and determines the validity of the remaining fields. The FLAGS field returns the EPCM permissions of the page; the page type; and the BLOCKED, PENDING, MODIFIED, and PR status of the page. For enclave pages, the ENCLAVECONTEXT field of the structure returns the value of SECS.ENCLAVECONTEXT. For non-enclave pages (e.g., VA) ENCLAVECONTEXT returns 0.

For invalid or non-EPC pages, the instruction returns an information code indicating the page's status, in addition to populating the STATUS field.

ERDINFO returns an error code if the destination EPC page is being modified by a concurrent SGX instruction.

RBX contains the effective address of a RDINFO structure while RCX contains the effective address of an EPC page. The table below provides additional information on the memory parameter of ERDINFO leaf function.

ERDINFO Memory Parameter Semantics

RDINFO	EPCPAGE
Read/Write access permitted by Non Enclave	Read access permitted by Enclave

ERDINFO Faulting Conditions

The instruction faults if any of the following:

A memory operand effective address is outside the DS segment limit (32b mode).	A memory operand is not properly aligned.
DS segment is unusable (32b mode).	A page fault occurs in accessing memory operands.
A memory address is in a non-canonical form (64b mode).	

The error codes are:

ERDINFO Return Value in RAX

Error Code	Value	Description
No Error	0	ERDINFO successful.
SGX_EPC_PAGE_CONFLICT		Failure due to concurrent operation of another SGX instruction.
SGX_PG_INVLD		Target page is not a valid EPC page.
SGX_PG_NONEPC		Page is not an EPC page.

Concurrency Restrictions

Base Concurrency Restrictions of ERDINFO

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
ERDINFO	Target [DS:RCX]	Shared	SGX_EPC_PAGE_CONFLICT	

Additional Concurrency Restrictions of ERDINFO

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
ERDINFO	Target [DS:RCX]	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in ERDINFO Operational Flow

Name	Type	Size (Bits)	Description
TMP_SECS	Physical Address	64	Physical address of the SECS of the page being modified.
TMP_RDINFO	Linear Address	64	Address of the RDINFO structure.

```
(* check alignment of RDINFO structure (RBX) *)
IF (DS:RBX is not 32Byte Aligned) THEN
    #GP(0); FI;

(* check alignment of the EPCPAGE (RCX) *)
IF (DS:RCX is not 4KByte Aligned) THEN
    #GP(0); FI;

(* check that EPCPAGE (DS:RCX) is the address of an EPC page *)
IF (DS:RCX does not resolve within EPC) THEN
    RFLAGS.CF ← 1;
    RFLAGS.ZF ← 0;
    RAX ← SGX_PG_NONEPC;
    goto DONE;
FI;

(* Check the EPC page for concurrency *)
IF (EPC page is being modified) THEN
    RFLAGS.ZF = 1;
    RFLAGS.CF = 0;
    RAX = SGX_EPC_PAGE_CONFLICT;
    goto DONE;
FI;

(* check page validity *)
IF (EPCM(DS:RCX).VALID = 0) THEN
    RFLAGS.CF = 1;
    RFLAGS.ZF = 0;
    RAX = SGX_PG_INVLD;
    goto DONE;
FI;

(* clear the fields of the RDINFO structure *)
TMP_RDINFO ← DS:RBX;
TMP_RDINFO.STATUS ← 0;
TMP_RDINFO.FLAGS ← 0;
TMP_RDINFO.ENCLAVECONTEXT ← 0;

(* store page info in RDINFO structure *)
TMP_RDINFO.FLAGS.RWX ← EPCM(DS:RCX).RWX;
TMP_RDINFO.FLAGS.PENDING ← EPCM(DS:RCX).PENDING;
TMP_RDINFO.FLAGS.MODIFIED ← EPCM(DS:RCX).MODIFIED;
TMP_RDINFO.FLAGS.PR ← EPCM(DS:RCX).PR;
TMP_RDINFO.FLAGS.PAGE_TYPE ← EPCM(DS:RCX).PAGE_TYPE;
TMP_RDINFO.FLAGS.BLOCKED ← EPCM(DS:RCX).BLOCKED;

(* read SECS.ENCLAVECONTEXT for enclave child pages *)
IF ((EPCM(DS:RCX).PAGE_TYPE = PT_REG) or
    (EPCM(DS:RCX).PAGE_TYPE = PT_TCS) or
    (EPCM(DS:RCX).PAGE_TYPE = PT_TRIM) or
    (EPCM(DS:RCX).PAGE_TYPE = PT_SS_FIRST) or
    (EPCM(DS:RCX).PAGE_TYPE = PT_SS_REST)
   ) THEN
    TMP_SECS ← Address of SECS for (DS:RCX);
    TMP_RDINFO.ENCLAVECONTEXT ← SECS(TMP_SECS).ENCLAVECONTEXT;
FI;

(* populate enclave information for SECS pages *)
IF (EPCM(DS:RCX).PAGE_TYPE = PT_SECS) THEN
    IF ((VMX non-root mode) and
        (ENABLE_EPC_VIRTUALIZATION_EXTENSIONS Execution Control = 1)
       ) THEN
        TMP_RDINFO.STATUS.CHILDPRESENT ←
                                ((SECS(DS:RCX).CHLDCNT ≠ 0) or
                                  SECS(DS:RCX).VIRTCHILDCNT ≠ 0);
    ELSE
        TMP_RDINFO.STATUS.CHILDPRESENT ← (SECS(DS:RCX).CHLDCNT ≠ 0);
        TMP_RDINFO.STATUS.VIRTCHILDPRESENT ←
                                (SECS(DS:RCX).VIRTCHILDCNT ≠ 0);
        TMP_RDINFO.ENCLAVECONTEXT ← SECS(DS_RCX).ENCLAVECONTEXT;
    FI;
FI;

RAX ← 0;
RFLAGS.ZF ← 0;
RFLAGS.CF ← 0;

DONE:
(* clear flags *)
RFLAGS.PF ← 0;
RFLAGS.AF ← 0;
RFLAGS.OF ← 0;
RFLAGS.SF ←? 0;
```

Flags Affected

ZF is set if ERDINFO fails due to concurrent operation with another SGX instruction; otherwise cleared.

CF is set if page is not a valid EPC page or not an EPC page; otherwise cleared.

PF, AF, OF and SF are cleared.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the DS segment limit.
	If DS segment is unusable.
	If a memory operand is not properly aligned.
#PF(error code)	If a page fault occurs in accessing memory operands.

64-Bit Mode Exceptions

#GP(0)	If the memory address is in a non-canonical form.
	If a memory operand is not properly aligned.
#PF(error code)	If a page fault occurs in accessing memory operands.

EREMOVE—Remove a page from the EPC

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 03H ENCLS[EREMOVE]	IR	V/V	SGX1	This leaf function removes a page from the EPC.

Instruction Operand Encoding

Op/En	EAX	RCX
IR	EREMOVE (In)	Effective address of the EPC page (In)

Description

This leaf function causes an EPC page to be un-associated with its SECS and be marked as unused. This instruction leaf can only be executed when the current privilege level is 0.

The content of RCX is an effective address of an EPC page. The DS segment is used to create linear address. Segment override is not supported.

The instruction fails if the operand is not properly aligned or does not refer to an EPC page or the page is in use by another thread, or other threads are running in the enclave to which the page belongs. In addition the instruction fails if the operand refers to an SECS with associations.

EREMOVE Memory Parameter Semantics

EPCPAGE
Write access permitted by Enclave

EREMOVE Faulting Conditions

The instruction faults if any of the following:

The memory operand is not properly aligned.	The memory operand does not resolve in an EPC page.
Refers to an invalid SECS.	Refers to an EPC page that is locked by another thread.
Another Intel SGX instruction is accessing the EPC page.	RCX does not contain an effective address of an EPC page.
the EPC page refers to an SECS with associations.	

The error codes are:

EREMOVE Return Value in RAX

Error Code	Description
No Error	EREMOVE successful.
SGX_CHILD_PRESENT	If the SECS still have enclave pages loaded into EPC.
SGX_ENCLAVE_ACT	If there are still logical processors executing inside the enclave.

Concurrency Restrictions

Base Concurrency Restrictions of EREMOVE

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EREMOVE	Target [DS:RCX]	Exclusive	#GP	EPC_PAGE_CONFLICT_EXCEPTION

Additional Concurrency Restrictions of EREMOVE

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EREMOVE	Target [DS:RCX]	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in EREMOVE Operational Flow

Name	Type	Size (Bits)	Description
TMP_SECS	Effective Address	32/64	Effective address of the SECS destination page.

```
IF (DS:RCX is not 4KByte Aligned)
    THEN #GP(0); FI;

IF (DS:RCX does not resolve to an EPC page)
    THEN #PF(DS:RCX); FI;

TMP_SECS ← Get_SECS_ADDRESS();

(* Check the EPC page for concurrency *)
IF (EPC page being referenced by another Intel SGX instruction)
    THEN
        IF (<<VMX non-root operation>> AND <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>>)
            THEN
                VMCS.Exit_reason ← SGX_CONFLICT;
                VMCS.Exit_qualification.code ← EPC_PAGE_CONFLICT_EXCEPTION;
                VMCS.Exit_qualification.error ← 0;
                VMCS.Guest-physical_address ← << translation of DS:RCX produced by paging >>;
                VMCS.Guest-linear_address ← DS:RCX;
                    Deliver VMEXIT;
            ELSE
                #GP(0);
        FI;
FI;

(* if DS:RCX is already unused, nothing to do*)
IF ( (EPCM(DS:RCX).VALID = 0) or (EPCM(DS:RCX).PT = PT_TRIM AND EPCM(DS:RCX).MODIFIED = 0))
    THEN GOTO DONE;
FI;

IF ( (EPCM(DS:RCX).PT = PT_VA) OR
    ((EPCM(DS:RCX).PT = PT_TRIM) AND (EPCM(DS:RCX).MODIFIED = 0)) )
    THEN
        EPCM(DS:RCX).VALID ← 0;
        GOTO DONE;
FI;

IF (EPCM(DS:RCX).PT = PT_SECS)
    THEN
        IF (DS:RCX has an EPC page associated with it)
            THEN
                RFLAGS.ZF ← 1;
                RAX← SGX_CHILD_PRESENT;
                GOTO ERROR_EXIT;
        FI;
        (* treat SECS as having a child page when VIRTCHILDCNT is non-zero *)
        IF (<<in VMX non-root operation>> AND
              <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>> AND
              (SECS(DS:RCX).VIRTCHILDCNT ≠ 0))
            THEN
                RFLAGS.ZF ← 1;
                    RAX ← SGX_CHILD_PRESENT
                GOTO ERROR_EXIT
        FI;
        EPCM(DS:RCX).VALID ← 0;
        GOTO DONE;
FI;

IF (Other threads active using SECS)
    THEN
        RFLAGS.ZF ← 1;
        RAX← SGX_ENCLAVE_ACT;
        GOTO ERROR_EXIT;
FI;

IF ( (EPCM(DS:RCX).PT is PT_REG) or (EPCM(DS:RCX).PT is PT_TCS) or (EPCM(DS:RCX).PT is PT_TRIM) or
     (EPCM(DS:RCX).PT is PT_SS_FIRST) or (EPCM(DS:RCX).PT is PT_SS_REST))
    THEN
        EPCM(DS:RCX).VALID ← 0;
        GOTO DONE;
FI;

DONE:
RAX← 0;
RFLAGS.ZF ← 0;

ERROR_EXIT:
RFLAGS.CF,PF,AF,OF,SF ← 0;
```

Flags Affected

Sets ZF if unsuccessful, otherwise cleared and RAX returns error code. Clears CF, PF, AF, OF, SF.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the DS segment limit.
	If a memory operand is not properly aligned.
	If another Intel SGX instruction is accessing the page.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If the memory operand is not an EPC page.

64-Bit Mode Exceptions

#GP(0)	If the memory operand is non-canonical form.
	If a memory operand is not properly aligned.
	If another Intel SGX instruction is accessing the page.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If the memory operand is not an EPC page.

ETRACK—Activates EBLOCK Checks

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 0CH ENCLS[ETRACK]	IR	V/V	SGX1	This leaf function activates EBLOCK checks.

Instruction Operand Encoding

Op/En	EAX		RCX
IR	ETRACK (In)	Return error code (Out)	Pointer to the SECS of the EPC page (In)

Description

This leaf function provides the mechanism for hardware to track that software has completed the required TLB address clears successfully. The instruction can only be executed when the current privilege level is 0.

The content of RCX is an effective address of an EPC page.

The table below provides additional information on the memory parameter of ETRACK leaf function.

ETRACK Memory Parameter Semantics

EPCPAGE
Read/Write access permitted by Enclave

The error codes are:

ETRACK Return Value in RAX

Error Code	Description
No Error	ETRACK successful.
SGX_PREV_TRK_INCMPL	All processors did not complete the previous shoot-down sequence.

Concurrency Restrictions

Base Concurrency Restrictions of ETRACK

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
ETRACK	SECS [DS:RCX]	Shared	#GP	

Additional Concurrency Restrictions of ETRACK

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
ETRACK	SECS [DS:RCX]	Concurrent		Concurrent		Exclusive	SGX_EPC_PAGE_CONFLICT

Operation

```
IF (DS:RCX is not 4KByte Aligned)
   THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)
   THEN #PF(DS:RCX); FI;

(* Check concurrency with other Intel SGX instructions *)
IF (Other Intel SGX instructions using tracking facility on this SECS)
   THEN
       IF (<<VMX non-root operation>> AND <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>>)
           THEN
               VMCS.Exit_reason ← SGX_CONFLICT;
               VMCS.Exit_qualification.code ← TRACKING_RESOURCE_CONFLICT;
               VMCS.Exit_qualification.error ← 0;
               VMCS.Guest-physical_address ← SECS(TMP_SECS).ENCLAVECONTEXT;
               VMCS.Guest-linear_address ← 0;
                   Deliver VMEXIT;
           ELSE
               #GP(0);
       FI;
FI;

IF (EPCM(DS:RCX). VALID = 0)
   THEN #PF(DS:RCX); FI;

IF (EPCM(DS:RCX).PT ≠ PT_SECS)
   THEN #PF(DS:RCX); FI;

(* All processors must have completed the previous tracking cycle*)
IF ( (DS:RCX).TRACKING ≠ 0) )
   THEN
       IF (<<VMX non-root operation>> AND <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>>)
           THEN
               VMCS.Exit_reason ← SGX_CONFLICT;
               VMCS.Exit_qualification.code ← TRACKING_REFERENCE_CONFLICT;
               VMCS.Exit_qualification.error ← 0;
               VMCS.Guest-physical_address ← SECS(TMP_SECS).ENCLAVECONTEXT;
               VMCS.Guest-linear_address ← 0;
                   Deliver VMEXIT;
       FI;
   RFLAGS.ZF ← 1;
       RAX← SGX_PREV_TRK_INCMPL;
       GOTO DONE;
   ELSE
       RAX← 0;
       RFLAGS.ZF ← 0;
FI;

DONE:
RFLAGS.CF,PF,AF,OF,SF ← 0;
```

Flags Affected

Sets ZF if SECS is in use or invalid, otherwise cleared. Clears CF, PF, AF, OF, SF.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the DS segment limit.
	If a memory operand is not properly aligned.
	If another thread is concurrently using the tracking facility on this SECS.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If a memory operand is not an EPC page.

64-Bit Mode Exceptions

#GP(0)	If a memory operand is non-canonical form.
	If a memory operand is not properly aligned.
	If the specified EPC resource is in use.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If a memory operand is not an EPC page.

ETRACKC—Activates EBLOCK Checks

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 11H ENCLS[ETRACKC]	IR	V/V	EAX[6]	This leaf function activates EBLOCK checks.

Instruction Operand Encoding

Op/En	EAX		RCX	
IR	ETRACK (In)	Return error code (Out)	Address of the destination EPC page (In, EA)	Address of the SECS page (In, EA)

Description

The ETRACKC instruction is thread safe variant of ETRACK leaf and can be executed concurrently with other CPU threads operating on the same SECS.

This leaf function provides the mechanism for hardware to track that software has completed the required TLB address clears successfully. The instruction can only be executed when the current privilege level is 0.

The content of RCX is an effective address of an EPC page.

The table below provides additional information on the memory parameter of ETRACK leaf function.

ETRACKC Memory Parameter Semantics

EPCPAGE
Read/Write access permitted by Enclave

The error codes are:

ETRACKC Return Value in RAX

Error Code	Value	Description
No Error	0	ETRACKC successful.
SGX_EPC_PAGE_CONFLICT	7	Failure due to concurrent operation of another SGX instruction.
SGX_PG_INVLD	6	Target page is not a VALID EPC page.
SGX_PREV_TRK_INCMPL	17	All processors did not complete the previous tracking sequence.
SGX_TRACK_NOT_REQUIRED	27	Target page type does not require tracking.

Concurrency Restrictions

Base Concurrency Restrictions of ETRACKC

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
ETRACKC	Target [DS:RCX]	Shared	SGX_EPC_PAGE_ CONFLICT	
	SECS implicit	Concurrent		

Additional Concurrency Restrictions of ETRACKC

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
ETRACKC	Target [DS:RCX]	Concurrent		Concurrent		Concurrent	
	SECS implicit	Concurrent		Concurrent		Exclusive	SGX_EPC_PAG E_CONFLICT

Operation

Temp Variables in ETRACKC Operational Flow

Name	Type	Size (Bits)	Description
TMP_SECS	Physical Address	64	Physical address of the SECS of the page being modified.

```
(* check alignment of EPCPAGE (RCX) *)
IF (DS:RCX is not 4KByte Aligned) THEN
#GP(0); FI;

(* check that EPCPAGE (DS:RCX) is the address of an EPC page *)
IF (DS:RCX does not resolve within an EPC) THEN
#PF(DS:RCX, PFEC.SGX); FI;

(* Check the EPC page for concurrency *)
```

```
IF (EPC page is being modified) THEN
    RFLAGS.ZF ← 1;
    RFLAGS.CF ← 0;
    RAX ← SGX_EPC_PAGE_CONFLICT;
    goto DONE_POST_LOCK_RELEASE;
FI;

(* check to make sure the page is valid *)
IF (EPCM(DS:RCX).VALID = 0) THEN
    RFLAGS.ZF ← 1;
    RFLAGS.CF ← 0;
    RAX ← SGX_PG_INVLD;
    GOTO DONE;
FI;

(* find out the target SECS page *)
IF (EPCM(DS:RCX).PT is PT_REG or PT_TCS or PT_TRIM or PT_SS_FIRST or PT_SS_REST) THEN
    TMP_SECS ← Obtain SECS through EPCM(DS:RCX).ENCLAVESECS;
ELSE IF (EPCM(DS:RCX).PT is PT_SECS) THEN
    TMP_SECS ← Obtain SECS through (DS:RCX);
ELSE
    RFLAGS.ZF ← 0;
    RFLAGS.CF ← 1;
    RAX ← SGX_TRACK_NOT_REQUIRED;
    GOTO DONE;
FI;

(* Check concurrency with other Intel SGX instructions *)
IF (Other Intel SGX instructions using tracking facility on this SECS) THEN
   IF ((VMX non-root mode) and
   (ENABLE_EPC_VIRTUALIZATION_EXTENSIONS Execution Control = 1)) THEN
            VMCS.Exit_reason ← SGX_CONFLICT;
            VMCS.Exit_qualification.code ← TRACKING_RESOURCE_CONFLICT;
            VMCS.Exit_qualification.error ← 0;
            VMCS.Guest-physical_address ←
        SECS(TMP_SECS).ENCLAVECONTEXT;
            VMCS.Guest-linear_address ← 0;
            Deliver VMEXIT;
      FI;

    RFLAGS.ZF ← 1;
    RFLAGS.CF ← 0;
    RAX ← SGX_EPC_PAGE_CONFLICT;
    GOTO DONE;
FI;
(* All processors must have completed the previous tracking cycle*)
IF ( (TMP_SECS).TRACKING ≠ 0) )
THEN
   IF ((VMX non-root mode) and
   (ENABLE_EPC_VIRTUALIZATION_EXTENSIONS Execution Control = 1)) THEN
            VMCS.Exit_reason ← SGX_CONFLICT;
            VMCS.Exit_qualification.code ← TRACKING_REFERENCE_CONFLICT;
            VMCS.Exit_qualification.error ← 0;
```

```
                    VMCS.Guest-physical_address ←
                SECS(TMP_SECS).ENCLAVECONTEXT;
                    VMCS.Guest-linear_address ← 0;
                    Deliver VMEXIT;
            FI;

        RFLAGS.ZF ← 1;
        RFLAGS.CF ← 0;
        RAX ← SGX_PREV_TRK_INCMPL;
        GOTO DONE;
FI;

RFLAGS.ZF ← 0;
RFLAGS.CF ← 0;
RAX ← 0;

DONE:
(* clear flags *)
RFLAGS.PF,AF,OF,SF ← 0;
```

Flags Affected

ZF is set if ETRACKC fails due to concurrent operations with another SGX instructions or target page is an invalid EPC page or tracking is not completed on SECS page; otherwise cleared.

CF is set if target page is not of a type that requires tracking; otherwise cleared.

PF, AF, OF and SF are cleared.

Protected Mode Exceptions

#GP(0)	If the memory operand violates access-control policies of DS segment.
	If DS segment is unusable.
	If the memory operand is not properly aligned.
#PF(error code)	If the memory operand expected to be in EPC does not resolve to an EPC page.
	If a page fault occurs in access memory operand.

64-Bit Mode Exceptions

#GP(0)	If a memory address is in a non-canonical form.
	If a memory operand is not properly aligned.
#PF(error code)	If the memory operand expected to be in EPC does not resolve to an EPC page.
	If a page fault occurs in access memory operand.

EWB—Invalidate an EPC Page and Write out to Main Memory

Opcode/ Instruction	**Op/En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
EAX = 0BH ENCLS[EWB]	IR	V/V	SGX1	This leaf function invalidates an EPC page and writes it out to main memory.

Instruction Operand Encoding

Op/En	EAX		RBX	RCX	RDX
IR	EWB (In)	Error code (Out)	Address of an PAGEINFO (In)	Address of the EPC page (In)	Address of a VA slot (In)

Description

This leaf function copies a page from the EPC to regular main memory. As part of the copying process, the page is cryptographically protected. This instruction can only be executed when current privilege level is 0.

The table below provides additional information on the memory parameter of EPA leaf function.

EWB Memory Parameter Semantics

PAGEINFO	PAGEINFO.SRCPGE	PAGEINFO.PCMD	EPCPAGE	VASLOT
Non-EPC R/W access	Non-EPC R/W access	Non-EPC R/W access	EPC R/W access	EPC R/W access

The error codes are:

EWB Return Value in RAX

Error Code	**Description**
No Error	EWB successful.
SGX_PAGE_NOT_BLOCKED	If page is not marked as blocked.
SGX_NOT_TRACKED	If EWB is racing with ETRACK instruction.
SGX_VA_SLOT_OCCUPIED	Version array slot contained valid entry.
SGX_CHILD_PRESENT	Child page present while attempting to page out enclave.

Concurrency Restrictions

Base Concurrency Restrictions of EWB

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EWB	Source [DS:RCX]	Exclusive	#GP	EPC_PAGE_CONFLICT_EXCEPTION
	VA [DS:RDX]	Shared	#GP	

Additional Concurrency Restrictions of EWB

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EWB	Source [DS:RCX]	Concurrent		Concurrent		Concurrent	
	VA [DS:RDX]	Concurrent		Concurrent		Exclusive	

Operation

Temp Variables in EWB Operational Flow

Name	Type	Size (Bytes)	Description
TMP_SRCPGE	Memory page	4096	
TMP_PCMD	PCMD	128	
TMP_SECS	SECS	4096	
TMP_BPEPOCH	UINT64	8	
TMP_BPREFCOUNT	UINT64	8	
TMP_HEADER	MAC Header	128	
TMP_PCMD_ENCLAVEID	UINT64	8	

TMP_VER	UINT64	8	
TMP_PK	UINT128	16	

```
IF ( (DS:RBX is not 32Byte Aligned) or (DS:RCX is not 4KByte Aligned) )
    THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;

IF (DS:RDX is not 8Byte Aligned)
    THEN #GP(0); FI;

IF (DS:RDX does not resolve within an EPC)
    THEN #PF(DS:RDX); FI;

(* EPCPAGE and VASLOT should not resolve to the same EPC page*)
IF (DS:RCX and DS:RDX resolve to the same EPC page)
    THEN #GP(0); FI;

TMP_SRCPGE ← DS:RBX.SRCPGE;
(* Note PAGEINFO.PCMD is overlaid on top of PAGEINFO.SECINFO *)
TMP_PCMD ← DS:RBX.PCMD;

If (DS:RBX.LINADDR ≠ 0) OR (DS:RBX.SECS ≠ 0)
    THEN #GP(0); FI;

IF ( (DS:TMP_PCMD is not 128Byte Aligned) or (DS:TMP_SRCPGE is not 4KByte Aligned) )
    THEN #GP(0); FI;

(* Check for concurrent Intel SGX instruction access to the page *)
IF (Other Intel SGX instruction is accessing page)
    THEN
        IF (<<VMX non-root operation>> AND <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>>)
            THEN
                VMCS.Exit_reason ← SGX_CONFLICT;
                VMCS.Exit_qualification.code ← EPC_PAGE_CONFLICT_EXCEPTION;
                VMCS.Exit_qualification.error ← 0;
                VMCS.Guest-physical_address ← << translation of DS:RCX produced by paging >>;
                VMCS.Guest-linear_address ← DS:RCX;
                Deliver VMEXIT;
            ELSE
                #GP(0);
        FI;
FI;

(*Check if the VA Page is being removed or changed*)
IF (VA Page is being modified)
    THEN #GP(0); FI;

(* Verify that EPCPAGE and VASLOT page are valid EPC pages and DS:RDX is VA *)
IF (EPCM(DS:RCX).VALID = 0)
```

```
   THEN #PF(DS:RCX); FI;

IF ( (EPCM(DS:RDX & ~0FFFH).VALID = 0) or (EPCM(DS:RDX & ~FFFH).PT is not PT_VA) )
   THEN #PF(DS:RDX); FI;

(* Perform page-type-specific exception checks *)
IF ( (EPCM(DS:RCX).PT is PT_REG) or (EPCM(DS:RCX).PT is PT_TCS) or (EPCM(DS:RCX).PT is PT_TRIM )
     or (EPCM(DS:RCX).PT is PT_SS_FIRST ) or (EPCM(DS:RCX).PT is PT_SS_REST))
   THEN
       TMP_SECS = Obtain SECS through EPCM(DS:RCX)
   (* Check that EBLOCK has occurred correctly *)
   IF (EBLOCK is not correct)
       THEN #GP(0); FI;
FI;

RFLAGS.ZF,CF,PF,AF,OF,SF ← 0;
RAX ← 0;

(* Perform page-type-specific checks *)
IF ( (EPCM(DS:RCX).PT is PT_REG) or (EPCM(DS:RCX).PT is PT_TCS) or (EPCM(DS:RCX).PT is PT_TRIM )
     or (EPCM(DS:RCX).PT is PT_SS_FIRST ) or (EPCM(DS:RCX).PT is PT_SS_REST))
   THEN
       (* check to see if the page is evictable *)
       IF (EPCM(DS:RCX).BLOCKED = 0)
           THEN
               RAX ← SGX_PAGE NOT_BLOCKED;
               RFLAGS.ZF ← 1;
               GOTO ERROR_EXIT;
       FI;
       (* Check if tracking done correctly *)
       IF (Tracking not correct)
           THEN
               RAX ← SGX_NOT_TRACKED;
               RFLAGS.ZF ← 1;
               GOTO ERROR_EXIT;
       FI;

       (* Obtain EID to establish cryptographic binding between the paged-out page and the enclave *)
       TMP_HEADER.EID ← TMP_SECS.EID;

       (* Obtain EID as an enclave handle for software *)
       TMP_PCMD_ENCLAVEID ← TMP_SECS.EID;
   ELSE IF (EPCM(DS:RCX).PT is PT_SECS)
       (*check that there are no child pages inside the enclave *)
       IF (DS:RCX has an EPC page associated with it)
           THEN
               RAX ← SGX_CHILD_PRESENT;
               RFLAGS.ZF ← 1;
               GOTO ERROR_EXIT;
```

```
            FI:
            (* treat SECS as having a child page when VIRTCHILDCNT is non-zero *)
            IF (<<in VMX non-root operation>> AND
              <<ENABLE_EPC_VIRTUALIZATION_EXTENSIONS>> AND
              (SECS(DS:RCX).VIRTCHILDCNT ≠ 0))
                THEN
                    RFLAGS.ZF ← 1;
                        RAX ← SGX_CHILD_PRESENT;
                    GOTO ERROR_EXIT;
            FI;
            TMP_HEADER.EID ← 0;
            (* Obtain EID as an enclave handle for software *)
            TMP_PCMD_ENCLAVEID ← (DS:RCX).EID;
      ELSE IF (EPCM(DS:RCX).PT is PT_VA)
            TMP_HEADER.EID ← 0; // Zero is not a special value
            (* No enclave handle for VA pages*)
            TMP_PCMD_ENCLAVEID ← 0;
FI;

(* Zero out TMP_HEADER*)
TMP_HEADER[ sizeof(TMP_HEADER)-1 : 0] ← 0;

TMP_HEADER.LINADDR ← EPCM(DS:RCX).ENCLAVEADDRESS;
TMP_HEADER.SECINFO.FLAGS.PT ← EPCM(DS:RCX).PT;
TMP_HEADER.SECINFO.FLAGS.RWX ← EPCM(DS:RCX).RWX;
TMP_HEADER.SECINFO.FLAGS.PENDING ← EPCM(DS:RCX).PENDING;
TMP_HEADER.SECINFO.FLAGS.MODIFIED ← EPCM(DS:RCX).MODIFIED;
TMP_HEADER.SECINFO.FLAGS.PR ← EPCM(DS:RCX).PR;

(* Encrypt the page, DS:RCX could be encrypted in place. AES-GCM produces 2 values, {ciphertext, MAC}. *)
(* AES-GCM input parameters: key, GCM Counter, MAC_HDR, MAC_HDR_SIZE, SRC, SRC_SIZE)*)
{DS:TMP_SRCPGE, DS:TMP_PCMD.MAC} ← AES_GCM_ENC(CR_BASE_PK                    ), (TMP_VER << 32),
      TMP_HEADER, 128, DS:RCX, 4096);

(* Write the output *)
Zero out DS:TMP_PCMD.SECINFO
DS:TMP_PCMD.SECINFO.FLAGS.PT ← EPCM(DS:RCX).PT;
DS:TMP_PCMD.SECINFO.FLAGS.RWX ← EPCM(DS:RCX).RWX;
DS:TMP_PCMD.SECINFO.FLAGS.PENDING ← EPCM(DS:RCX).PENDING;
DS:TMP_PCMD.SECINFO.FLAGS.MODIFIED ← EPCM(DS:RCX).MODIFIED;
DS:TMP_PCMD.SECINFO.FLAGS.PR ← EPCM(DS:RCX).PR;
DS:TMP_PCMD.RESERVED ← 0;
DS:TMP_PCMD.ENCLAVEID ← TMP_PCMD_ENCLAVEID;
DS:RBX.LINADDR ← EPCM(DS:RCX).ENCLAVEADDRESS;

(*Check if version array slot was empty *)
IF ([DS.RDX])
      THEN
            RAX ← SGX_VA_SLOT_OCCUPIED
            RFLAGS.CF ← 1;
FI;

(* Write version to Version Array slot *)
```

```
[DS.RDX] ← TMP_VER;

(* Free up EPCM Entry *)
EPCM.(DS:RCX).VALID ← 0;
ERROR_EXIT:
```

Flags Affected

ZF is set if page is not blocked, not tracked, or a child is present. Otherwise cleared.

CF is set if VA slot is previously occupied, Otherwise cleared.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the DS segment limit.
	If a memory operand is not properly aligned.
	If the EPC page and VASLOT resolve to the same EPC page.
	If another Intel SGX instruction is concurrently accessing either the target EPC, VA, or SECS pages.
	If the tracking resource is in use.
	If the EPC page or the version array page is invalid.
	If the parameters fail consistency checks.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If a memory operand is not an EPC page.
	If one of the EPC memory operands has incorrect page type.

64-Bit Mode Exceptions

#GP(0)	If a memory operand is non-canonical form.
	If a memory operand is not properly aligned.
	If the EPC page and VASLOT resolve to the same EPC page.
	If another Intel SGX instruction is concurrently accessing either the target EPC, VA, or SECS pages.
	If the tracking resource is in use.
	If the EPC page or the version array page in invalid.
	If the parameters fail consistency checks.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If a memory operand is not an EPC page.
	If one of the EPC memory operands has incorrect page type.

16.4 Intel® SGX User Leaf Function Reference

Leaf functions available with the ENCLU instruction mnemonic are covered in this section. In general, each instruction leaf requires EAX to specify the leaf function index and/or additional registers specifying leaf-specific input parameters. An instruction operand encoding table provides details of the implicitly-encoded register usage and associated input/output semantics.

In many cases, an input parameter specifies an effective address associated with a memory object inside or outside the EPC, the memory addressing semantics of these memory objects are also summarized in a separate table.

EACCEPT—Accept Changes to an EPC Page

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 05H ENCLU[EACCEPT]	IR	V/V	SGX2	This leaf function accepts changes made by system software to an EPC page in the running enclave.

Instruction Operand Encoding

Op/En	EAX		RBX	RCX
IR	EACCEPT (In)	Return Error Code (Out)	Address of a SECINFO (In)	Address of the destination EPC page (In)

Description

This leaf function accepts changes to a page in the running enclave by verifying that the security attributes specified in the SECINFO match the security attributes of the page in the EPCM. This instruction leaf can only be executed when inside the enclave.

RBX contains the effective address of a SECINFO structure while RCX contains the effective address of an EPC page. The table below provides additional information on the memory parameter of the EACCEPT leaf function.

EACCEPT Memory Parameter Semantics

SECINFO	EPCPAGE (Destination)
Read access permitted by Non Enclave	Read access permitted by Enclave

EACCEPT Faulting Conditions

The instruction faults if any of the following:

The operands are not properly aligned.	RBX does not contain an effective address in an EPC page in the running enclave.
The EPC page is locked by another thread.	RCX does not contain an effective address of an EPC page in the running enclave.
The EPC page is not valid.	Page type is PT_REG and MODIFIED bit is 0.
SECINFO contains an invalid request.	Page type is PT_TCS or PT_TRIM and PENDING bit is 0 and MODIFIED bit is 1.
If security attributes of the SECINFO page make the page inaccessible.	

The error codes are:

EACCEPT Return Value in RAX

Error Code	Description
No Error	EACCEPT successful.
SGX_PAGE_ATTRIBUTES_MISMATCH	The attributes of the target EPC page do not match the expected values.
SGX_NOT_TRACKED	The OS did not complete an ETRACK on the target page.

Concurrency Restrictions

Base Concurrency Restrictions of EACCEPT

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EACCEPT	Target [DS:RCX]	Shared	#GP	
	SECINFO [DS:RBX]	Concurrent		

Additional Concurrency Restrictions of EACCEPT

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EACCEPT	Target [DS:RCX]	Exclusive	#GP	Concurrent		Concurrent	
	SECINFO [DS:RBX]	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in EACCEPT Operational Flow

Name	Type	Size (bits)	Description
TMP_SECS	Effective Address	32/64	Physical address of SECS to which EPC operands belongs.
SCRATCH_SECINFO	SECINFO	512	Scratch storage for holding the contents of DS:RBX.

```
IF (DS:RBX is not 64Byte Aligned)
    THEN #GP(0); FI;

IF (DS:RBX is not within CR_ELRANGE)
    THEN #GP(0); FI;

IF (DS:RBX does not resolve within an EPC)
    THEN #PF(DS:RBX); FI;

IF ( (EPCM(DS:RBX &~FFFH).VALID = 0) or (EPCM(DS:RBX &~FFFH).R = 0) or (EPCM(DS:RBX &~FFFH).PENDING ≠ 0) or
    (EPCM(DS:RBX &~FFFH).MODIFIED ≠ 0) or (EPCM(DS:RBX &~FFFH).BLOCKED ≠ 0) or
    (EPCM(DS:RBX &~FFFH).PT ≠ PT_REG) or (EPCM(DS:RBX &~FFFH).ENCLAVESECS ≠ CR_ACTIVE_SECS) or
    (EPCM(DS:RBX &~FFFH).ENCLAVEADDRESS ≠ (DS:RBX & FFFH)) )
    THEN #PF(DS:RBX); FI;

(* Copy 64 bytes of contents *)
SCRATCH_SECINFO ← DS:RBX;

(* Check for misconfigured SECINFO flags*)
IF (SCRATCH_SECINFO reserved fields are not zero )
    THEN #GP(0); FI;

IF (DS:RCX is not 4KByte Aligned)
    THEN #GP(0); FI;

IF (DS:RCX is not within CR_ELRANGE)
    THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;

(* Check that the combination of requested PT, PENDING and MODIFIED is legal *)
IF (CPUID.(EAX=12H, ECX=1):EAX[6] = 0 )
    THEN
        IF (NOT (((SCRATCH_SECINFO.FLAGS.PT is PT_REG) and
            ((SCRATCH_SECINFO.FLAGS.PR is 1) or
            (SCRATCH_SECINFO.FLAGS.PENDING is 1)) and
            (SCRATCH_SECINFO.FLAGS.MODIFIED is 0)) or
            ((SCRATCH_SECINFO.FLAGS.PT is PT_TCS or PT_TRIM) and
            (SCRATCH_SECINFO.FLAGS.PR is 0) and
            (SCRATCH_SECINFO.FLAGS.PENDING is 0) and
            (SCRATCH_SECINFO.FLAGS.MODIFIED is 1) )))
            THEN #GP(0); FI
    ELSE
        IF (NOT (((SCRATCH_SECINFO.FLAGS.PT is PT_REG) AND
                    ((SCRATCH_SECINFO.FLAGS.PR is 1) OR
                    (SCRATCH_SECINFO.FLAGS.PENDING is 1)) AND
                    (SCRATCH_SECINFO.FLAGS.MODIFIED is 0)) OR
            ((SCRATCH_SECINFO.FLAGS.PT is PT_TCS OR PT_TRIM) AND
                    (SCRATCH_SECINFO.FLAGS.PENDING is 0) AND
                    (SCRATCH_SECINFO.FLAGS.MODIFIED is 1) AND
                    (SCRATCH_SECINFO.FLAGS.PR is 0)) OR
            ((SCRATCH_SECINFO.FLAGS.PT is PT_SS_FIRST or PT_SS_REST) AND
                    (SCRATCH_SECINFO.FLAGS.PENDING is 1) AND
                    (SCRATCH_SECINFO.FLAGS.MODIFIED is 0) AND
                    (SCRATCH_SECINFO.FLAGS.PR is 0))))
            THEN #GP(0); FI;
    FI;


(* Check security attributes of the destination EPC page *)
If ( (EPCM(DS:RCX).VALID is 0) or       (EPCM(DS:RCX).BLOCKED is not 0) or
    ((EPCM(DS:RCX).PT is not PT_REG) and (EPCM(DS:RCX).PT is not PT_TCS) and (EPCM(DS:RCX).PT is not PT_TRIM)
     and (EPCM(DS:RCX).PT is not PT_SS_FIRST) and (EPCM(DS:RCX).PT is not PT_SS_REST)) or
    (EPCM(DS:RCX).ENCLAVESECS ≠ CR_ACTIVE_SECS))
    THEN #PF((DS:RCX); FI;

(* Check the destination EPC page for concurrency *)
IF ( EPC page in use )
    THEN #GP(0); FI;

(* Re-Check security attributes of the destination EPC page *)
IF ( (EPCM(DS:RCX).VALID is 0) or (EPCM(DS:RCX).ENCLAVESECS ≠ CR_ACTIVE_SECS) )
    THEN #PF(DS:RCX); FI;

(* Verify that accept request matches current EPC page settings *)
IF ( (EPCM(DS:RCX).ENCLAVEADDRESS ≠ DS:RCX) or (EPCM(DS:RCX).PENDING ≠ SCRATCH_SECINFO.FLAGS.PENDING) or
    (EPCM(DS:RCX).MODIFIED ≠ SCRATCH_SECINFO.FLAGS.MODIFIED) or (EPCM(DS:RCX).R ≠ SCRATCH_SECINFO.FLAGS.R) or
    (EPCM(DS:RCX).W ≠ SCRATCH_SECINFO.FLAGS.W) or (EPCM(DS:RCX).X ≠ SCRATCH_SECINFO.FLAGS.X) or
    (EPCM(DS:RCX).PT ≠ SCRATCH_SECINFO.FLAGS.PT) )
    THEN
        RFLAGS.ZF ← 1;
        RAX ← SGX_PAGE_ATTRIBUTES_MISMATCH;
        GOTO DONE;
FI;
(* Check that all required threads have left enclave *)
IF (Tracking not correct)
    THEN
        RFLAGS.ZF ← 1;
        RAX ← SGX_NOT_TRACKED;
        GOTO DONE;
FI;

(* Get pointer to the SECS to which the EPC page belongs *)
TMP_SECS = << Obtain physical address of SECS through EPCM(DS:RCX)>>
(* For TCS pages, perform additional checks *)
IF (SCRATCH_SECINFO.FLAGS.PT = PT_TCS)
   THEN
        IF (DS:RCX.RESERVED ≠ 0) #GP(0); FI;
FI;

(* Check that TCS.FLAGS.DBGOPTIN, TCS stack, and TCS status are correctly initialized *)
(* check that TCS.PREVSSP is 0 *)
IF ( ((DS:RCX).FLAGS.DBGOPTIN is not 0) or ((DS:RCX).CSSA ≥ (DS:RCX).NSSA) or ((DS:RCX).AEP is not 0) or ((DS:RCX).STATE is not 0) or
((CPUID.(EAX=12H, ECX=1):EAX[6] = 1) AND ((DS:RCX).PREVSSP != 0)))
   THEN #GP(0); FI;

(* Check consistency of FS & GS Limit *)
IF ( (TMP_SECS.ATTRIBUTES.MODE64BIT is 0) and ((DS:RCX.FSLIMIT & FFFH ≠ FFFH) or (DS:RCX.GSLIMIT & FFFH ≠ FFFH)) )
   THEN #GP(0); FI;

(* Clear PENDING/MODIFIED flags to mark accept operation complete *)
EPCM(DS:RCX).PENDING ← 0;
EPCM(DS:RCX).MODIFIED ← 0;
EPCM(DS:RCX).PR ← 0;

(* Clear EAX and ZF to indicate successful completion *)
RFLAGS.ZF ← 0;
RAX ← 0;

DONE:
RFLAGS.CF,PF,AF,OF,SF ← 0;
```

Flags Affected

Sets ZF if page cannot be accepted, otherwise cleared. Clears CF, PF, AF, OF, SF

Protected Mode Exceptions

#GP(0)	If executed outside an enclave.
	If a memory operand effective address is outside the DS segment limit.
	If a memory operand is not properly aligned.
	If a memory operand is locked.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If a memory operand is not an EPC page.
	If EPC page has incorrect page type or security attributes.

64-Bit Mode Exceptions

#GP(0)	If executed outside an enclave.
	If a memory operand is non-canonical form.
	If a memory operand is not properly aligned.
	If a memory operand is locked.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If a memory operand is not an EPC page.
	If EPC page has incorrect page type or security attributes.

EACCEPTCOPY—Initialize a Pending Page

Opcode/ Instruction	**Op/En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
EAX = 07H ENCLU[EACCEPTCOPY]	IR	V/V	SGX2	This leaf function initializes a dynamically allocated EPC page from another page in the EPC.

Instruction Operand Encoding

Op/En	EAX		RBX	RCX	RDX
IR	EACCEPTCOPY (In)	Return Error Code (Out)	Address of a SECINFO (In)	Address of the destination EPC page (In)	Address of the source EPC page (In)

Description

This leaf function copies the contents of an existing EPC page into an uninitialized EPC page (created by EAUG). After initialization, the instruction may also modify the access rights associated with the destination EPC page. This instruction leaf can only be executed when inside the enclave.

RBX contains the effective address of a SECINFO structure while RCX and RDX each contain the effective address of an EPC page. The table below provides additional information on the memory parameter of the EACCEPTCOPY leaf function.

EACCEPTCOPY Memory Parameter Semantics

SECINFO	EPCPAGE (Destination)	EPCPAGE (Source)
Read access permitted by Non Enclave	Read/Write access permitted by Enclave	Read access permitted by Enclave

EACCEPTCOPY Faulting Conditions

The instruction faults if any of the following:

The operands are not properly aligned.	If security attributes of the SECINFO page make the page inaccessible.
The EPC page is locked by another thread.	If security attributes of the source EPC page make the page inaccessible.
The EPC page is not valid.	RBX does not contain an effective address in an EPC page in the running enclave.
SECINFO contains an invalid request.	RCX/RDX does not contain an effective address of an EPC page in the running enclave.

The error codes are:

EACCEPTCOPY Return Value in RAX

Error Code	Description
No Error	EACCEPTCOPY successful.
SGX_PAGE_ATTRIBUTES_MISMATCH	The attributes of the target EPC page do not match the expected values.

Concurrency Restrictions

Base Concurrency Restrictions of EACCEPTCOPY

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EACCEPTCOPY	Target [DS:RCX]	Concurrent		
	Source [DS:RDX]	Concurrent		
	SECINFO [DS:RBX]	Concurrent		

Additional Concurrency Restrictions of EACCEPTCOPY

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EACCEPTCOPY	Target [DS:RCX]	Exclusive	#GP	Concurrent		Concurrent	
	Source [DS:RDX]	Concurrent		Concurrent		Concurrent	
	SECINFO [DS:RBX]	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in EACCEPTCOPY Operational Flow

Name	Type	Size (bits)	Description
SCRATCH_SECINFO	SECINFO	512	Scratch storage for holding the contents of DS:RBX.

```
IF (DS:RBX is not 64Byte Aligned)
    THEN #GP(0); FI;

IF ( (DS:RCX is not 4KByte Aligned) or (DS:RDX is not 4KByte Aligned) )
    THEN #GP(0); FI;

IF ((DS:RBX is not within CR_ELRANGE) or (DS:RCX is not within CR_ELRANGE) or (DS:RDX is not within CR_ELRANGE))
    THEN #GP(0); FI;

IF (DS:RBX does not resolve within an EPC)
    THEN #PF(DS:RBX); FI;

IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;

IF (DS:RDX does not resolve within an EPC)
    THEN #PF(DS:RDX); FI;

IF ( (EPCM(DS:RBX &~FFFH).VALID = 0) or (EPCM(DS:RBX &~FFFH).R = 0) or (EPCM(DS:RBX &~FFFH).PENDING ≠ 0) or
    (EPCM(DS:RBX &~FFFH).MODIFIED ≠ 0) or (EPCM(DS:RBX &~FFFH).BLOCKED ≠ 0) or (EPCM(DS:RBX &~FFFH).PT ≠ PT_REG) or
    (EPCM(DS:RBX &~FFFH).ENCLAVESECS ≠ CR_ACTIVE_SECS) or
    (EPCM(DS:RBX &~FFFH).ENCLAVEADDRESS ≠ DS:RBX) )
    THEN #PF(DS:RBX); FI;

(* Copy 64 bytes of contents *)
SCRATCH_SECINFO ← DS:RBX;

(* Check for misconfigured SECINFO flags*)
IF ( (SCRATCH_SECINFO reserved fields are not zero ) or (SCRATCH_SECINFO.FLAGS.R=0) AND(SCRATCH_SECINFO.FLAGS.W≠0 ) or
    (SCRATCH_SECINFO.FLAGS.PT is not PT_REG) )
    THEN #GP(0); FI;

(* Check security attributes of the source EPC page *)
IF ( (EPCM(DS:RDX).VALID = 0) or (EPCM(DS:RCX).R = 0) or (EPCM(DS:RDX).PENDING ≠ 0) or (EPCM(DS:RDX).MODIFIED ≠ 0) or
    (EPCM(DS:RDX).BLOCKED ≠ 0) or (EPCM(DS:RDX).PT ≠ PT_REG) or (EPCM(DS:RDX).ENCLAVESECS ≠ CR_ACTIVE_SECS) or
    (EPCM(DS:RDX).ENCLAVEADDRESS ≠ DS:RDX))
    THEN #PF(DS:RDX); FI;

(* Check security attributes of the destination EPC page *)
IF ( (EPCM(DS:RCX).VALID = 0) or (EPCM(DS:RCX).PENDING ≠ 1) or (EPCM(DS:RCX).MODIFIED ≠ 0) or
    (EPCM(DS:RDX).BLOCKED ≠ 0) or (EPCM(DS:RCX).PT ≠ PT_REG) or (EPCM(DS:RCX).ENCLAVESECS ≠ CR_ACTIVE_SECS) )
    THEN
        RFLAGS.ZF ← 1;
        RAX ← SGX_PAGE_ATTRIBUTES_MISMATCH;
        GOTO DONE;
FI;

(* Check the destination EPC page for concurrency *)
```

```
IF (destination EPC page in use )
    THEN #GP(0); FI;

(* Re-Check security attributes of the destination EPC page *)
IF ( (EPCM(DS:RCX).VALID = 0) or (EPCM(DS:RCX).PENDING ≠ 1) or (EPCM(DS:RCX).MODIFIED ≠ 0) or
    (EPCM(DS:RCX).R ≠ 1) or (EPCM(DS:RCX).W ≠ 1) or (EPCM(DS:RCX).X ≠ 0) or
    (EPCM(DS:RCX).PT ≠ SCRATCH_SECINFO.FLAGS.PT) or (EPCM(DS:RCX).ENCLAVESECS ≠ CR_ACTIVE_SECS) or
    (EPCM(DS:RCX).ENCLAVEADDRESS ≠ DS:RCX))
    THEN
        RFLAGS.ZF ← 1;
        RAX ← SGX_PAGE_ATTRIBUTES_MISMATCH;
        GOTO DONE;
FI;

(* Copy 4KBbytes form the source to destination EPC page*)
DS:RCX[32767:0] ← DS:RDX[32767:0];

(* Update EPCM permissions *)
EPCM(DS:RCX).R ← SCRATCH_SECINFO.FLAGS.R;
EPCM(DS:RCX).W ← SCRATCH_SECINFO.FLAGS.W;
EPCM(DS:RCX).X ← SCRATCH_SECINFO.FLAGS.X;
EPCM(DS:RCX).PENDING ← 0;

RFLAGS.ZF ← 0;
RAX ← 0;

DONE:
RFLAGS.CF,PF,AF,OF,SF ← 0;
```

Flags Affected

Sets ZF if page is not modifiable, otherwise cleared. Clears CF, PF, AF, OF, SF

Protected Mode Exceptions

#GP(0)	If executed outside an enclave.
	If a memory operand effective address is outside the DS segment limit.
	If a memory operand is not properly aligned.
	If a memory operand is locked.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If a memory operand is not an EPC page.
	If EPC page has incorrect page type or security attributes.

64-Bit Mode Exceptions

#GP(0)	If executed outside an enclave.
	If a memory operand is non-canonical form.
	If a memory operand is not properly aligned.
	If a memory operand is locked.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If a memory operand is not an EPC page.
	If EPC page has incorrect page type or security attributes.

EENTER—Enters an Enclave

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 02H ENCLU[EENTER]	IR	V/V	SGX1	This leaf function is used to enter an enclave.

Instruction Operand Encoding

Op/En	EAX		RBX	RCX	
IR	EENTER (In)	Content of RBX.CSSA (Out)	Address of a TCS (In)	Address of AEP (In)	Address of IP following EENTER (Out)

Description

The ENCLU[EENTER] instruction transfers execution to an enclave. At the end of the instruction, the logical processor is executing in enclave mode at the RIP computed as EnclaveBase + TCS.OENTRY. If the target address is not within the CS segment (32-bit) or is not canonical (64-bit), a #GP(0) results.

EENTER Memory Parameter Semantics

TCS
Enclave access

EENTER is a serializing instruction. The instruction faults if any of the following occurs:

Address in RBX is not properly aligned.	Any TCS.FLAGS's must-be-zero bit is not zero.
TCS pointed to by RBX is not valid or available or locked.	Current 32/64 mode does not match the enclave mode in SECS.ATTRIBUTES.MODE64.
The SECS is in use.	Either of TCS-specified FS and GS segment is not a subsets of the current DS segment.
Any one of DS, ES, CS, SS is not zero.	If XSAVE available, CR4.OSXSAVE = 0, but SECS.ATTRIBUTES.XFRM ≠ 3.
CR4.OSFXSR ≠ 1.	If CR4.OSXSAVE = 1, SECS.ATTRIBUTES.XFRM is not a subset of XCR0.

The following operations are performed by EENTER:

- RSP and RBP are saved in the current SSA frame on EENTER and are automatically restored on EEXIT or interrupt.
- The AEP contained in RCX is stored into the TCS for use by AEXs.FS and GS (including hidden portions) are saved and new values are constructed using TCS.OFSBASE/GSBASE (32 and 64-bit mode) and TCS.OFSLIMIT/GSLIMIT (32-bit mode only). The resulting segments must be a subset of the DS segment.
- If CR4.OSXSAVE == 1, XCR0 is saved and replaced by SECS.ATTRIBUTES.XFRM. The effect of RFLAGS.TF depends on whether the enclave entry is opt-in or opt-out:
  - On opt-out entry, TF is saved and cleared (it is restored on EEXIT or AEX). Any attempt to set TF via a POPF instruction while inside the enclave clears TF.
  - On opt-in entry, a single-step debug exception is pended on the instruction boundary immediately after EENTER.
- All code breakpoints that do not overlap with ELRANGE are also suppressed. If the entry is an opt-out entry, all code and data breakpoints that overlap with the ELRANGE are suppressed.
- On opt-out entry, a number of performance monitoring counters and behaviors are modified or suppressed:
  - All performance monitoring activity on the current thread is suppressed except for incrementing and firing of FIXED_CTR1 and FIXED_CTR2.
  - PEBS is suppressed.
  - AnyThread counting on other threads is demoted to MyThread mode and IA32_PERF_GLOBAL_STATUS[60] on that thread is set
  - If the opt-out entry on a hardware thread results in suppression of any performance monitoring, then the processor sets IA32_PERF_GLOBAL_STATUS[60] and IA32_PERF_GLOBAL_STATUS[63].

Concurrency Restrictions

Base Concurrency Restrictions of EENTER

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EENTER	TCS [DS:RBX]	Shared	#GP	

Additional Concurrency Restrictions of EENTER

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EENTER	TCS [DS:RBX]	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in EENTER Operational Flow

Name	Type	Size (Bits)	Description
TMP_FSBASE	Effective Address	32/64	Proposed base address for FS segment.
TMP_GSBASE	Effective Address	32/64	Proposed base address for FS segment.
TMP_FSLIMIT	Effective Address	32/64	Highest legal address in proposed FS segment.
TMP_GSLIMIT	Effective Address	32/64	Highest legal address in proposed GS segment.
TMP_XSIZE	integer	64	Size of XSAVE area based on SECS.ATTRIBUTES.XFRM.
TMP_SSA_PAGE	Effective Address	32/64	Pointer used to iterate over the SSA pages in the current frame.
TMP_GPR	Effective Address	32/64	Address of the GPR area within the current SSA frame.

```
TMP_MODE64 ← ((IA32_EFER.LMA = 1) && (CS.L = 1));

(* Make sure DS is usable, expand up *)
IF (TMP_MODE64 = 0 and (DS not usable or ( ( DS[S] = 1) and (DS[bit 11] = 0) and DS[bit 10] = 1) ) )
    THEN #GP(0); FI;

(* Check that CS, SS, DS, ES.base is 0 *)
IF (TMP_MODE64 = 0)
    THEN
        IF(CS.base ≠ 0 or DS.base ≠ 0) #GP(0); FI;
        IF(ES usable and ES.base ≠ 0) #GP(0); FI;
        IF(SS usable and SS.base ≠ 0) #GP(0); FI;
        IF(SS usable and SS.B = 0) #GP(0); FI;
FI;

IF (DS:RBX is not 4KByte Aligned)
    THEN #GP(0); FI;

IF (DS:RBX does not resolve within an EPC)
    THEN #PF(DS:RBX); FI;

(* Check AEP is canonical*)
IF (TMP_MODE64 = 1 and (CS:RCX is not canonical) )
    THEN #GP(0); FI;

(* Check concurrency of TCS operation*)
IF (Other Intel SGX instructions is operating on TCS)
```

```
    THEN #GP(0); FI;

(* TCS verification *)
IF (EPCM(DS:RBX).VALID = 0)
    THEN #PF(DS:RBX); FI;

IF (EPCM(DS:RBX).BLOCKED = 1)
    THEN #PF(DS:RBX); FI;

IF ( (EPCM(DS:RBX).ENCLAVEADDRESS ≠ DS:RBX) or (EPCM(DS:RBX).PT ≠ PT_TCS) )
    THEN #PF(DS:RBX); FI;

IF ((EPCM(DS:RBX).PENDING = 1) or (EPCM(DS:RBX).MODIFIED = 1))
    THEN #PF(DS:RBX); FI;

IF ( (DS:RBX).OSSA is not 4KByte Aligned)
    THEN #GP(0); FI;

(* Check proposed FS and GS *)
IF ( ( (DS:RBX).OFSBASE is not 4KByte Aligned) or ( (DS:RBX).OGSBASE is not 4KByte Aligned) )
    THEN #GP(0); FI;

(* Get the SECS for the enclave in which the TCS resides *)
TMP_SECS ← Address of SECS for TCS;

(* Check proposed FS/GS segments fall within DS *)
IF (TMP_MODE64 = 0)
    THEN
        TMP_FSBASE ← (DS:RBX).OFSBASE + TMP_SECS.BASEADDR;
        TMP_FSLIMIT ← (DS:RBX).OFSBASE + TMP_SECS.BASEADDR + (DS:RBX).FSLIMIT;
        TMP_GSBASE ← (DS:RBX).OGSBASE + TMP_SECS.BASEADDR;
        TMP_GSLIMIT ← (DS:RBX).OGSBASE + TMP_SECS.BASEADDR + (DS:RBX).GSLIMIT;
        (* if FS wrap-around, make sure DS has no holes*)
        IF (TMP_FSLIMIT < TMP_FSBASE)
            THEN
                IF (DS.limit < 4GB) THEN #GP(0); FI;
            ELSE
                IF (TMP_FSLIMIT > DS.limit) THEN #GP(0); FI;
        FI;
        (* if GS wrap-around, make sure DS has no holes*)
        IF (TMP_GSLIMIT < TMP_GSBASE)
            THEN
                IF (DS.limit < 4GB) THEN #GP(0); FI;
            ELSE
                IF (TMP_GSLIMIT > DS.limit) THEN #GP(0); FI;
        FI;
    ELSE
        TMP_FSBASE ← (DS:RBX).OFSBASE + TMP_SECS.BASEADDR;
        TMP_GSBASE ← (DS:RBX).OGSBASE + TMP_SECS.BASEADDR;
        IF ( (TMP_FSBASE is not canonical) or (TMP_GSBASE is not canonical))
            THEN #GP(0); FI;
FI;
```

```
(* Ensure that the FLAGS field in the TCS does not have any reserved bits set *)
IF ( ( (DS:RBX).FLAGS & FFFFFFFFFFFFFFFEH) ≠ 0)
    THEN #GP(0); FI;

(* SECS must exist and enclave must have previously been EINITted *)
IF (the enclave is not already initialized)
    THEN #GP(0); FI;

(* make sure the logical processor's operating mode matches the enclave *)
IF ( (TMP_MODE64 ≠ TMP_SECS.ATTRIBUTES.MODE64BIT) )
    THEN #GP(0); FI;

IF (CR4.OSFXSR = 0)
    THEN #GP(0); FI;

(* Check for legal values of SECS.ATTRIBUTES.XFRM *)
IF (CR4.OSXSAVE = 0)
    THEN
        IF (TMP_SECS.ATTRIBUTES.XFRM ≠ 03H) THEN #GP(0); FI;
    ELSE
        IF ( (TMP_SECS.ATTRIBUTES.XFRM & XCR0) ≠ TMP_SECS.ATTRIBUES.XFRM) THEN #GP(0); FI;
FI;

(* Make sure the SSA contains at least one more frame *)
IF ( (DS:RBX).CSSA ≥ (DS:RBX).NSSA)
    THEN #GP(0); FI;

(* Compute linear address of SSA frame *)
TMP_SSA ← (DS:RBX).OSSA + TMP_SECS.BASEADDR + 4096 * TMP_SECS.SSAFRAMESIZE * (DS:RBX).CSSA;
TMP_XSIZE ← compute_XSAVE_frame_size(TMP_SECS.ATTRIBUTES.XFRM);

FOR EACH TMP_SSA_PAGE = TMP_SSA to TMP_SSA + TMP_XSIZE
    (* Check page is read/write accessible *)
    Check that DS:TMP_SSA_PAGE is read/write accessible;
    If a fault occurs, release locks, abort and deliver that fault;

    IF (DS:TMP_SSA_PAGE does not resolve to EPC page)
        THEN #PF(DS:TMP_SSA_PAGE); FI;
    IF (EPCM(DS:TMP_SSA_PAGE).VALID = 0)
        THEN #PF(DS:TMP_SSA_PAGE); FI;
    IF (EPCM(DS:TMP_SSA_PAGE).BLOCKED = 1)
        THEN #PF(DS:TMP_SSA_PAGE); FI;
    IF ((EPCM(DS:TMP_SSA_PAGE).PENDING = 1) or (EPCM(DS:TMP_SSA_PAGE).MODIFIED = 1))
        THEN #PF(DS:TMP_SSA_PAGE); FI;
    IF ( ( EPCM(DS:TMP_SSA_PAGE).ENCLAVEADDRESS ≠ DS:TMP_SSA_PAGE) or (EPCM(DS:TMP_SSA_PAGE).PT ≠ PT_REG) or
        (EPCM(DS:TMP_SSA_PAGE).ENCLAVESECS ≠ EPCM(DS:RBX).ENCLAVESECS) or
        (EPCM(DS:TMP_SSA_PAGE).R = 0) or (EPCM(DS:TMP_SSA_PAGE).W = 0) )
        THEN #PF(DS:TMP_SSA_PAGE); FI;
    CR_XSAVE_PAGE_n ← Physical_Address(DS:TMP_SSA_PAGE);
```

```
ENDFOR

(* Compute address of GPR area*)
TMP_GPR ← TMP_SSA + 4096 * DS:TMP_SECS.SSAFRAMESIZE - sizeof(GPRSGX_AREA);
If a fault occurs; release locks, abort and deliver that fault;

IF (DS:TMP_GPR does not resolve to EPC page)
    THEN #PF(DS:TMP_GPR); FI;
IF (EPCM(DS:TMP_GPR).VALID = 0)
    THEN #PF(DS:TMP_GPR); FI;
IF (EPCM(DS:TMP_GPR).BLOCKED = 1)
    THEN #PF(DS:TMP_GPR); FI;
IF ((EPCM(DS:TMP_GPR).PENDING = 1) or (EPCM(DS:TMP_GPR).MODIFIED = 1))
    THEN #PF(DS:TMP_GPR); FI;
IF ( ( EPCM(DS:TMP_GPR).ENCLAVEADDRESS ≠ DS:TMP_GPR) or (EPCM(DS:TMP_GPR).PT ≠ PT_REG) or
    (EPCM(DS:TMP_GPR).ENCLAVESECS EPCM(DS:RBX).ENCLAVESECS) or
    (EPCM(DS:TMP_GPR).R = 0) or (EPCM(DS:TMP_GPR).W = 0) )
    THEN #PF(DS:TMP_GPR); FI;

IF (TMP_MODE64 = 0)
    THEN
        IF (TMP_GPR + (GPR_SIZE -1) is not in DS segment) THEN #GP(0); FI;
FI;

CR_GPR_PA ← Physical_Address (DS: TMP_GPR);

(* Validate TCS.OENTRY *)
TMP_TARGET ← (DS:RBX).OENTRY + TMP_SECS.BASEADDR;
IF (TMP_MODE64 = 1)
    THEN
        IF (TMP_TARGET is not canonical) THEN #GP(0); FI;
    ELSE
        IF (TMP_TARGET > CS limit) THEN #GP(0); FI;
FI;

(* Ensure the enclave is not already active and this thread is the only one using the TCS*)
IF (DS:RBX.STATE = ACTIVE)
    THEN #GP(0); FI;

IF CPUID.(EAX=12H, ECX=1):EAX[6] = 1
     THEN
        IF ( CR4.CET = 0 )
            THEN
                (* If part does not support CET or CET has not been enabled  and enclave requires CET then fail  *)
                IF ( TMP_SECS.CET_ATTRIBUTES ≠ 0 OR TMP_SECS.CET_LEG_BITMAP_OFFSET ≠ 0 )  #GP(0);  FI;
            FI;
        (* If indirect branch tracking or shadow stacks enabled but CET  state save area is not 16B aligned then fail EENTER  *)
        IF ( TMP_SECS.CET_ATTRIBUTES.SH_STK_EN = 1 OR  TMP_SECS.CET_ATTRIBUTES.ENDBR_EN = 1 )
            THEN
                 IF (DS:RBX.OCETSSA is not 16B aligned)    #GP(0);  FI;
            FI;
        TMP_IA32_U_CET ← 0;
        TMP_SSP ← 0;
    IF (TMP_SECS.CET_ATTRIBUTES.SH_STK_EN OR TMP_SECS.CET_ATTRIBUTES.ENDBR_EN)
        THEN
            (* Setup CET state from SECS, note tracker goes to IDLE  *)
             TMP_IA32_U_CET = TMP_SECS.CET_ATTRIBUTES;
             IF (TMP_IA32_U_CET.LEG_IW_EN = 1 AND TMP_IA32_U_CET.ENDBR_EN = 1 )
                 THEN
                     TMP_IA32_U_CET ← TMP_IA32_U_CET + TMP_SECS.BASEADDR;
                       TMP_IA32_U_CET ← TMP_IA32_U_CET + TMP_SECS.CET_LEG_BITMAP_BASE;
                  FI;

             (* Compute linear address of what will become  new CET state save area and cache its PA  *)
             TMP_CET_SAVE_AREA = DS:RBX.OCETSSA + TMP_SECS.BASEADDR + (DS:RBX.CSSA) * 16
             TMP_CET_SAVE_PAGE = TMP_CET_SAVE_AREA & ~0xFFF;

              Check the TMP_CET_SAVE_PAGE page is read/write accessible
              If fault occurs release locks, abort and deliver fault

              (* read the EPCM VALID, PENDING, MODIFIED, BLOCKED and PT fields atomically  *)
            IF ((DS:TMP_CET_SAVE_PAGE Does NOT RESOLVE TO EPC PAGE) OR
                 (EPCM(DS:TMP_CET_SAVE_PAGE).VALID = 0) OR
                 (EPCM(DS:TMP_CET_SAVE_PAGE).PENDING = 1) OR
                 (EPCM(DS:TMP_CET_SAVE_PAGE).MODIFIED = 1) OR
                 (EPCM(DS:TMP_CET_SAVE_PAGE).BLOCKED = 1) OR
                 (EPCM(DS:TMP_CET_SAVE_PAGE).R = 0) OR
                 (EPCM(DS:TMP_CET_SAVE_PAGE).W = 0) OR
                 (EPCM(DS:TMP_CET_SAVE_PAGE).ENCLAVEADDRESS ≠ DS:TMP_CET_SAVE_PAGE) OR
                 (EPCM(DS:TMP_CET_SAVE_PAGE).PT ≠ PT_SS_REST) OR
                 (EPCM(DS:TMP_CET_SAVE_PAGE).ENCLAVESECS ≠ EPCM(DS:RBX).ENCLAVESECS))
                 THEN
                       #PF(DS:TMP_CET_SAVE_PAGE);
                 FI;

              CR_CET_SAVE_AREA_PA ← Physical address(DS:TMP_CET_SAVE_AREA)

             IF TMP_IA32_U_CET.SH_STK_EN = 1
                 THEN
                     TMP_SSP = TCS.PREVSSP;
                 FI;
        FI;
  FI;

CR_ENCLAVE_MODE ← 1;
CR_ACTIVE_SECS ← TMP_SECS;
CR_ELRANGE ← (TMPSECS.BASEADDR, TMP_SECS.SIZE);

(* Save state for possible AEXs *)
CR_TCS_PA ← Physical_Address (DS:RBX);
```

```
CR_TCS_LA ← RBX;
CR_TCS_LA.AEP ← RCX;

(* Save the hidden portions of FS and GS *)
CR_SAVE_FS_selector ← FS.selector;
CR_SAVE_FS_base ← FS.base;
CR_SAVE_FS_limit ← FS.limit;
CR_SAVE_FS_access_rights ← FS.access_rights;
CR_SAVE_GS_selector ← GS.selector;
CR_SAVE_GS_base ← GS.base;
CR_SAVE_GS_limit ← GS.limit;
CR_SAVE_GS_access_rights ← GS.access_rights;

(* If XSAVE is enabled, save XCR0 and replace it with SECS.ATTRIBUTES.XFRM*)
IF (CR4.OSXSAVE = 1)
    CR_SAVE_XCR0 ← XCR0;
    XCR0 ← TMP_SECS.ATTRIBUTES.XFRM;
FI;

RCX ← RIP;
RIP ← TMP_TARGET;
RAX ← (DS:RBX).CSSA;
(* Save the outside RSP and RBP so they can be restored on interrupt or EEXIT *)
DS:TMP_SSA.U_RSP ← RSP;
DS:TMP_SSA.U_RBP ← RBP;

(* Do the FS/GS swap *)
FS.base ← TMP_FSBASE;
FS.limit ← DS:RBX.FSLIMIT;
FS.type ← 0001b;
FS.W ← DS.W;
FS.S ← 1;
FS.DPL ← DS.DPL;
FS.G ← 1;
FS.B ← 1;
FS.P ← 1;
FS.AVL ← DS.AVL;
FS.L ← DS.L;
FS.unusable ← 0;
FS.selector ← 0BH;

GS.base ← TMP_GSBASE;
GS.limit ← DS:RBX.GSLIMIT;
GS.type ← 0001b;
GS.W ← DS.W;
GS.S ← 1;
GS.DPL ← DS.DPL;
GS.G ← 1;
GS.B ← 1;
GS.P ← 1;
GS.AVL ← DS.AVL;
GS.L ← DS.L;
GS.unusable ← 0;
GS.selector ← 0BH;

CR_DBGOPTIN ← TCS.FLAGS.DBGOPTIN;
Suppress_all_code_breakpoints_that_are_outside_ELRANGE;

IF (CR_DBGOPTIN = 0)
    THEN
        Suppress_all_code_breakpoints_that_overlap_with_ELRANGE;
        CR_SAVE_TF ← RFLAGS.TF;
        RFLAGS.TF ← 0;
        Suppress_monitor_trap_flag for the source of the execution of the enclave;
        Suppress any pending debug exceptions;
        Suppress any pending MTF VM exit;
    ELSE
        IF RFLAGS.TF = 1
            THEN pend a single-step #DB at the end of EENTER; FI;
        IF the "monitor trap flag" VM-execution control is set
            THEN pend an MTF VM  exit at the end of EENTER; FI;
FI;
IF (CPUID.(EAX=12H, ECX=1):EAX[6] = 1)
    THEN
        (* Save enclosing application CET state into save registers  *)
        CR_SAVE_IA32_U_CET ← IA32_U_CET
        (* Setup enclave CET state  *)
        IF CPUID.(EAX=07H, ECX=00h):ECX[CET_SS] = 1
            THEN
                CR_SAVE_SSP ← SSP
                SSP ← TMP_SSP;
            FI;
        IA32_U_CET ← TMP_IA32_U_CET;
     FI;
Flush_linear_context;
Allow_front_end_to_begin_fetch_at_new_RIP;
```

Flags Affected

RFLAGS.TF is cleared on opt-out entry

Protected Mode Exceptions

#GP(0)	If DS:RBX is not page aligned.
	If the enclave is not initialized.
	If part or all of the FS or GS segment specified by TCS is outside the DS segment or not properly aligned.
	If the thread is not in the INACTIVE state.
	If CS, DS, ES or SS bases are not all zero.
	If executed in enclave mode.
	If any reserved field in the TCS FLAG is set.
	If the target address is not within the CS segment.
	If CR4.OSFXSR = 0.

	If CR4.OSXSAVE = 0 and SECS.ATTRIBUTES.XFRM ≠ 3.
	If CR4.OSXSAVE = 1and SECS.ATTRIBUTES.XFRM is not a subset of XCR0.
#PF(error code)	If a page fault occurs in accessing memory.
	If DS:RBX does not point to a valid TCS.
	If one or more pages of the current SSA frame are not readable/writable, or do not resolve to a valid PT_REG EPC page.

64-Bit Mode Exceptions

#GP(0)	If DS:RBX is not page aligned.
	If the enclave is not initialized.
	If the thread is not in the INACTIVE state.
	If CS, DS, ES or SS bases are not all zero.
	If executed in enclave mode.
	If part or all of the FS or GS segment specified by TCS is outside the DS segment or not properly aligned.
	If the target address is not canonical.
	If CR4.OSFXSR = 0.
	If CR4.OSXSAVE = 0 and SECS.ATTRIBUTES.XFRM ≠ 3.
	If CR4.OSXSAVE = 1and SECS.ATTRIBUTES.XFRM is not a subset of XCR0.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If DS:RBX does not point to a valid TCS.
	If one or more pages of the current SSA frame are not readable/writable, or do not resolve to a valid PT_REG EPC page.

EEXIT—Exits an Enclave

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 04H ENCLU[EEXIT]	IR	V/V	SGX1	This leaf function is used to exit an enclave.

Instruction Operand Encoding

Op/En	EAX	RBX	RCX
IR	EEXIT (In)	Target address outside the enclave (In)	Address of the current AEP (In)

Description

The ENCLU[EEXIT] instruction exits the currently executing enclave and branches to the location specified in RBX. RCX receives the current AEP. If RBX is not within the CS (32-bit mode) or is not canonical (64-bit mode) a #GP(0) results.

EEXIT Memory Parameter Semantics

Target Address
Non-Enclave read and execute access

If RBX specifies an address that is inside the enclave, the instruction will complete normally. The fetch of the next instruction will occur in non-enclave mode, but will attempt to fetch from inside the enclave. This fetch returns a fixed data pattern.

If secrets are contained in any registers, it is responsibility of enclave software to clear those registers.

If XCR0 was modified on enclave entry, it is restored to the value it had at the time of the most recent EENTER or ERESUME.

If the enclave is opt-out, RFLAGS.TF is loaded from the value previously saved on EENTER.

Code and data breakpoints are unsuppressed.

Performance monitoring counters are unsuppressed.

Concurrency Restrictions

Base Concurrency Restrictions of EEXIT

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EEXIT		Concurrent		

Additional Concurrency Restrictions of EEXIT

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EEXIT		Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in EEXIT Operational Flow

Name	Type	Size (Bits)	Description
TMP_RIP	Effective Address	32/64	Saved copy of CRIP for use when creating LBR.

```
TMP_MODE64 ← ((IA32_EFER.LMA = 1) && (CS.L = 1));

IF (TMP_MODE64 = 1)
    THEN
        IF (RBX is not canonical) THEN #GP(0); FI;
    ELSE
        IF (RBX > CS limit) THEN #GP(0); FI;
FI;

TMP_RIP ← CRIP;
RIP ← RBX;

(* Return current AEP in RCX *)
RCX ← CR_TCS_PA.AEP;

(* Do the FS/GS swap *)
FS.selector ← CR_SAVE_FS.selector;
FS.base ← CR_SAVE_FS.base;
FS.limit ← CR_SAVE_FS.limit;
FS.access_rights ← CR_SAVE_FS.access_rights;
GS.selector ← CR_SAVE_GS.selector;
GS.base ← CR_SAVE_GS.base;
GS.limit ← CR_SAVE_GS.limit;
GS.access_rights ← CR_SAVE_GS.access_rights;

(* Restore XCR0 if needed *)
IF (CR4.OSXSAVE = 1)
    XCR0 ← CR_SAVE__XCR0;
FI;

Unsuppress_all_code_breakpoints_that_are_outside_ELRANGE;

IF (CR_DBGOPTIN = 0)
    THEN
        UnSuppress_all_code_breakpoints_that_overlap_with_ELRANGE;
        Restore suppressed breakpoint matches;
        RFLAGS.TF ← CR_SAVE_TF;
        UnSuppress_montior_trap_flag;
        UnSuppress_LBR_Generation;
        UnSuppress_performance monitoring_activity;
        Restore performance monitoring counter AnyThread demotion to MyThread in enclave back to AnyThread
FI;


IF RFLAGS.TF = 1
    THEN Pend Single-Step #DB at the end of EEXIT;
FI;


IF the “monitor trap flag” VM-execution control is set
    THEN pend a MTF VM  exit at the end of EEXIT;
FI;
IF (CPUID.(EAX=12H, ECX=1):EAX[6] = 1)
     THEN
         (* Record PREVSSP *)
        IF (IA32_U_CET.SH_STK_EN == 1)
            THEN  CR_TCS_PA.PREVSSP = SSP;  FI;

        (* Restore enclosing apps CET state from the save registers *)
        IA32_U_CET ← CR_SAVE_IA32_U_CET;
        IF CPUID.(EAX=07H, ECX=00h):ECX[CET_SS] = 1
             THEN SSP ← CR_SAVE_SSP;  FI;

        (* Update enclosing apps TRACKER if enclosing app has indirect branch tracking enabled  *)
        IF (CR4.CET = 1 AND IA32_U_CET.ENDBR_EN = 1)
            THEN
                IA32_U_CET.TRACKER ← WAIT_FOR_ENDBRANCH;
                IA32_U_CET.SUPPRESS ← 0
            FI;
    FI;
CR_ENCLAVE_MODE ← 0;
CR_TCS_PA.STATE ← INACTIVE;

(* Assure consistent translations *)
Flush_linear_context;
```

Flags Affected

RFLAGS.TF is restored from the value previously saved in EENTER or ERESUME.

Protected Mode Exceptions

#GP(0)	If executed outside an enclave.
	If RBX is outside the CS segment.
#PF(error code)	If a page fault occurs in accessing memory.

64-Bit Mode Exceptions

#GP(0)	If executed outside an enclave.
	If RBX is not canonical.
#PF(error code)	If a page fault occurs in accessing memory operands.

EGETKEY—Retrieves a Cryptographic Key

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 01H ENCLU[EGETKEY]	IR	V/V	SGX1	This leaf function retrieves a cryptographic key.

Instruction Operand Encoding

Op/En	EAX	RBX	RCX
IR	EGETKEY (In)	Address to a KEYREQUEST (In)	Address of the OUTPUTDATA (In)

Description

The ENCLU[EGETKEY] instruction returns a 128-bit secret key from the processor specific key hierarchy. The register RBX contains the effective address of a KEYREQUEST structure, which the instruction interprets to determine the key being requested. The Requesting Keys section below provides a description of the keys that can be requested. The RCX register contains the effective address where the key will be returned. Both the addresses in RBX & RCX should be locations inside the enclave.

EGETKEY derives keys using a processor unique value to create a specific key based on a number of possible inputs. This instruction leaf can only be executed inside an enclave.

EEGETKEY Memory Parameter Semantics

KEYREQUEST	OUTPUTDATA
Enclave read access	Enclave write access

After validating the operands, the instruction determines which key is to be produced and performs the following actions:

- The instruction assembles the derivation data for the key based on Table 41.
- Computes derived key using the derivation data and package specific value.
- Outputs the calculated key to the address in RCX.

The instruction fails with #GP(0) if the operands are not properly aligned. Successful completion of the instruction will clear RFLAGS.{ZF, CF, AF, OF, SF, PF}. The instruction returns an error code if the user tries to request a key based on an invalid CPUSVN or ISVSVN (when the user request is accepted, see the table below), requests a key for which it has not been granted the attribute to request, or requests a key that is not supported by the hardware. These checks may be performed in any order. Thus, an indication by error number of one cause (for example, invalid attribute) does not imply that there are not also other errors. Different processors may thus give different error numbers for the same Enclave. The correctness of software should not rely on the order resulting from the checks documented in this section. In such cases the ZF flag is set and the corresponding error bit (SGX_INVALID_SVN, SGX_INVALID_ATTRIBUTE, SGX_INVALID_KEYNAME) is set in RAX and the data at the address specified by RCX is unmodified.

Requesting Keys

The KEYREQUEST structure identifies the key to be provided. The Keyrequest.KeyName field identifies which type of key is requested.

Deriving Keys

Key derivation is based on a combination of the enclave specific values (see Table 41) and a processor key. Depending on the key being requested a field may either be included by definition or the value may be included from the KeyRequest. A "yes" in Table 41 indicates the value for the field is included from its default location, identified in the source row, and a "request" indicates the values for the field is included from its corresponding KeyRequest field.

Table 41 Key Derivation

	Key Name	Attributes	Owner Epoch	CPU SVN	ISV SVN	ISV PRODID	ISVEXT PRODID	ISVFAMILYID	MRENCLAVE	MRSIGNER	CONFIGID	CONFIG SVN	RAND
Source	Key Dependent Constant	Y← SECS.ATTRIBUTES and SECS.MISCSELECT and SECS.CET_ATTRIBUTES;	CR_SGXOWNEREPOCH	Y← CPUSVN Register;	R← Req.ISVSVN;	SECS. ISVID	SECS.ISVEXTPRODID	SECS.ISVFAMILYID	SECS. MRENCLAVE	SECS. MRSIGNER	SECS.CONFIGID	SECS.CONFIGSVN	Req. KEYID
		R←AttribMask & SECS.ATTRIBUTES and SECS.MISCSELECT and SECS.CET_ATTRIBUTES;		R← Req.CPUSVN;									
EINITTOKEN	Yes	Request	Yes	Request	Request	Yes	No	No	No	Yes	No	No	Request
Report	Yes	Yes	Yes	Yes	No	No	No	No	Yes	No	Yes	Yes	Request
Seal	Yes	Request	Yes	Request	Request	Request	Request	Request	Request	Request	Request	Request	Request
Provisioning	Yes	Request	No	Request	Request	Yes	No	No	No	Yes	No	No	Yes
Provisioning Seal	Yes	Request	No	Request	Request	Request	Request	Request	No	Yes	Request	Request	Yes

Keys that permit the specification of a CPU or ISV's code's, or enclave configuration's SVNs have additional requirements. The caller may not request a key for an SVN beyond the current CPU, ISV or enclave configuration's SVN, respectively.

Several keys are access controlled. Access to the Provisioning Key and Provisioning Seal key requires the enclave's ATTRIBUTES.PROVISIONKEY be set. The EINITTOKEN Key requires ATTRIBUTES.EINITTOKEN_KEY be set and SECS.MRSIGNER equal IA32_SGXLEPUBKEYHASH.

Some keys are derived based on a hardcode PKCS padding constant (352 byte string):

HARDCODED_PKCS1_5_PADDING[15:0] ← 0100H;

HARDCODED_PKCS1_5_PADDING[2655:16] ← SignExtend330Byte(-1); // 330 bytes of 0FFH

HARDCODED_PKCS1_5_PADDING[2815:2656] ← 2004000501020403650148866009060D30313000H;

The error codes are:

EGETKEY Return Value in RAX

Error Code	Value	Description
No Error	0	EGETKEY successful.
SGX_INVALID_ATTRIBUTE		The KEYREQUEST contains a KEYNAME for which the enclave is not authorized.
SGX_INVALID_CPUSVN		If KEYREQUEST.CPUSVN is an unsupported platforms CPUSVN value.
SGX_INVALID_ISVSVN		If KEYREQUEST software SVN (ISVSVN or CONFIGSVN) is greater than the enclave's corresponding SVN.
SGX_INVALID_KEYNAME		If KEYREQUEST.KEYNAME is an unsupported value.

Concurrency Restrictions

Base Concurrency Restrictions of EGETKEY

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EGETKEY	KEYREQUEST [DS:RBX]	Concurrent		
	OUTPUTDATA [DS:RCX]	Concurrent		

Additional Concurrency Restrictions of EGETKEY

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EGETKEY	KEYREQUEST [DS:RBX]	Concurrent		Concurrent		Concurrent	
	OUTPUTDATA [DS:RCX]	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in EGETKEY Operational Flow

Name	Type	Size (Bits)	Description
TMP_CURRENTSECS			Address of the SECS for the currently executing enclave.
TMP_KEYDEPENDENCIES			Temp space for key derivation.
TMP_ATTRIBUTES		128	Temp Space for the calculation of the sealable Attributes.
TMP_ISVEXTPRODID		16 bytes	Temp Space for ISVEXTPRODID.
TMP_ISVPRODID		2 bytes	Temp Space for ISVPRODID.
TMP_ISVFAMILYID		16 bytes	Temp Space for ISVFAMILYID.
TMP_CONFIGID		64 bytes	Temp Space for CONFIGID.
TMP_CONFIGSVN		2 bytes	Temp Space for CONFIGSVN.
TMP_OUTPUTKEY		128	Temp Space for the calculation of the key.

```
(* Make sure KEYREQUEST is properly aligned and inside the current enclave *)
IF ( (DS:RBX is not 512Byte aligned) or (DS:RBX is within CR_ELRANGE) )
    THEN #GP(0); FI;

(* Make sure DS:RBX is an EPC address and the EPC page is valid *)
IF ( (DS:RBX does not resolve to an EPC address) or (EPCM(DS:RBX).VALID = 0) )
    THEN #PF(DS:RBX); FI;

IF (EPCM(DS:RBX).BLOCKED = 1)
    THEN #PF(DS:RBX); FI;

(* Check page parameters for correctness *)
IF ( (EPCM(DS:RBX).PT ≠ PT_REG) or (EPCM(DS:RBX).ENCLAVESECS ≠ CR_ACTIVE_SECS) or (EPCM(DS:RBX).PENDING = 1) or
    (EPCM(DS:RBX).MODIFIED = 1) or (EPCM(DS:RBX).ENCLAVEADDRESS ≠ (DS:RBX & ~0FFFH) ) or (EPCM(DS:RBX).R = 0) )
    THEN #PF(DS:RBX);
FI;

(* Make sure OUTPUTDATA is properly aligned and inside the current enclave *)
IF ( (DS:RCX is not 16Byte aligned) or (DS:RCX is not within CR_ELRANGE) )
    THEN #GP(0); FI;

(* Make sure DS:RCX is an EPC address and the EPC page is valid *)
IF ( (DS:RCX does not resolve to an EPC address) or (EPCM(DS:RCX).VALID = 0) )
    THEN #PF(DS:RCX); FI;

IF (EPCM(DS:RCX).BLOCKED = 1)
    THEN #PF(DS:RCX); FI;
```

```
(* Check page parameters for correctness *)
IF ( (EPCM(DS:RCX).PT ≠ PT_REG) or (EPCM(DS:RCX).ENCLAVESECS ≠ CR_ACTIVE_SECS) or (EPCM(DS:RCX).PENDING = 1) or
    (EPCM(DS:RCX).MODIFIED = 1) or (EPCM(DS:RCX).ENCLAVEADDRESS ≠ (DS:RCX & ~0FFFH) ) or (EPCM(DS:RCX).W = 0) )
    THEN #PF(DS:RCX);
FI;

(* Verify RESERVED spaces in KEYREQUEST are valid *)
IF ( (DS:RBX).RESERVED ≠ 0) or (DS:RBX.KEYPOLICY.RESERVED ≠ 0) )
    THEN #GP(0); FI;

TMP_CURRENTSECS ← CR_ACTIVE_SECS;

(* Verify that CONFIGSVN & New Policy bits are not used if KSS is not enabled *)
IF ((TMP_CURRENTSECS.ATTRIBUTES.KSS == 0) AND ((DS:RBX.KEYPOLICY & 0x003C ≠ 0) OR (DS:RBX.CONFIGSVN > 0)))
    THEN #GP(0); FI;
(* Determine which enclave attributes that must be included in the key. Attributes that must always be include INIT & DEBUG *)
REQUIRED_SEALING_MASK[127:0] ← 00000000 00000000 00000000 00000003H;
TMP_ATTRIBUTES ← (DS:RBX.ATTRIBUTEMASK | REQUIRED_SEALING_MASK) & TMP_CURRENTSECS.ATTRIBUTES;

(* Compute MISCSELECT fields to be included *)
TMP_MISCSELECT ← DS:RBX.MISCMASK & TMP_CURRENTSECS.MISCSELECT

(* Compute CET_ATTRIBUTES fields to be included *)
IF (CPUID.(EAX=12H, ECX=1):EAX[6] = 1)
    THEN TMP_CET_ATTRIBUTES ← DS:RBX.CET_ATTRIBUTES_ MASK & TMP_CURRENTSECS.CET_ATTRIBUTES; FI;
TMP_KEYDEPENDENCIES ← 0;

CASE (DS:RBX.KEYNAME)
    SEAL_KEY:
        IF (DS:RBX.CPUSVN is beyond current CPU configuration)
            THEN
                RFLAGS.ZF ← 1;
                RAX ← SGX_INVALID_CPUSVN;
                GOTO EXIT;
        FI;
        IF (DS:RBX.ISVSVN > TMP_CURRENTSECS.ISVSVN)
            THEN
                RFLAGS.ZF ← 1;
                RAX ← SGX_INVALID_ISVSVN;
                GOTO EXIT;
        FI;
        IF (DS:RBX.CONFIGSVN > TMP_CURRENTSECS.CONFIGSVN)
            THEN
                RFLAGS.ZF ← 1;
                RAX ← SGX_INVALID_ISVSVN;
                GOTO EXIT;
        FI;
        (*Include enclave identity?*)
        TMP_MRENCLAVE ← 0;
        IF (DS:RBX.KEYPOLICY.MRENCLAVE = 1)
            THEN TMP_MRENCLAVE ← TMP_CURRENTSECS.MRENCLAVE;
        FI;
        (*Include enclave author?*)
        TMP_MRSIGNER ← 0;
        IF (DS:RBX.KEYPOLICY.MRSIGNER = 1)
            THEN TMP_MRSIGNER ← TMP_CURRENTSECS.MRSIGNER;
        FI;
(* Include enclave product family ID? *)
    TMP_ISVFAMILYID ← 0;
    IF (DS:RBX.KEYPOLICY.ISVFAMILYID = 1)
        THEN TMP_ISVFAMILYID ← TMP_CURRENTSECS.ISVFAMILYID;
        FI;

    (* Include enclave product ID? *)
    TMP_ISVPRODID ← 0;
    IF (DS:RBX.KEYPOLICY.NOISVPRODID = 0)
        TMP_ISVPRODID ←TMP_CURRENTSECS.ISVPRODID;
        FI;

    (* Include enclave Config ID? *)
    TMP_CONFIGID ← 0;
    TMP_CONFIGSVN ← 0;
    IF (DS:RBX.KEYPOLICY.CONFIGID = 1)
        TMP_CONFIGID ← TMP_CURRENTSECS.CONFIGID;
        TMP_CONFIGSVN ← DS:RBX.CONFIGSVN;
        FI;

    (* Include enclave extended product ID? *)
    TMP_ISVEXTPRODID ← 0;
    IF (DS:RBX.KEYPOLICY.ISVEXTPRODID = 1 )
        TMP_ISVEXTPRODID ← TMP_CURRENTSECS.ISVEXTPRODID;
    FI;

        //Determine values key is based on
        TMP_KEYDEPENDENCIES.KEYNAME ← SEAL_KEY;
        TMP_KEYDEPENDENCIES.ISVFAMILYID ← TMP_ISVFAMILYID;
        TMP_KEYDEPENDENCIES.ISVEXTPRODID ← TMP_ISVEXTPRODID;
        TMP_KEYDEPENDENCIES.ISVPRODID ← TMP_ISVPRODID;
        TMP_KEYDEPENDENCIES.ISVSVN ← DS:RBX.ISVSVN;
        TMP_KEYDEPENDENCIES.SGXOWNEREPOCH ← CR_SGXOWNEREPOCH;
        TMP_KEYDEPENDENCIES.ATTRIBUTES ← TMP_ATTRIBUTES;
        TMP_KEYDEPENDENCIES.ATTRIBUTESMASK ← DS:RBX.ATTRIBUTEMASK;
        TMP_KEYDEPENDENCIES.MRENCLAVE ← TMP_MRENCLAVE;
        TMP_KEYDEPENDENCIES.MRSIGNER ← TMP_MRSIGNER;
        TMP_KEYDEPENDENCIES.KEYID ← DS:RBX.KEYID;
        TMP_KEYDEPENDENCIES.SEAL_KEY_FUSES ← CR_SEAL_FUSES;
        TMP_KEYDEPENDENCIES.CPUSVN ← DS:RBX.CPUSVN;
        TMP_KEYDEPENDENCIES.PADDING ← TMP_CURRENTSECS.PADDING;
        TMP_KEYDEPENDENCIES.MISCSELECT ← TMP_MISCSELECT;
        TMP_KEYDEPENDENCIES.MISCMASK ← ~DS:RBX.MISCMASK;
        TMP_KEYDEPENDENCIES.KEYPOLICY ← DS:RBX.KEYPOLICY;
        TMP_KEYDEPENDENCIES.CONFIGID ← TMP_CONFIGID;
        TMP_KEYDEPENDENCIES.CONFIGSVN ← TMP_CONFIGSVN;
          IF CPUID.(EAX=12H, ECX=1):EAX[6] = 1
            THEN
                TMP_KEYDEPENDENCIES.CET_ATTRIBUTES ← TMP_CET_ATTRIBUTES;
               TMP_KEYDEPENDENCIES.CET_ATTRIBUTES _MASK ← DS:RBX.CET_ATTRIBUTES _MASK;
            FI;
        BREAK;
    REPORT_KEY:
        //Determine values key is based on
        TMP_KEYDEPENDENCIES.KEYNAME ← REPORT_KEY;
        TMP_KEYDEPENDENCIES.ISVFAMILYID ← 0;
        TMP_KEYDEPENDENCIES.ISVEXTPRODID ← 0;
        TMP_KEYDEPENDENCIES.ISVPRODID ← 0;
        TMP_KEYDEPENDENCIES.ISVSVN ← 0;
        TMP_KEYDEPENDENCIES.SGXOWNEREPOCH ← CR_SGXOWNEREPOCH;
        TMP_KEYDEPENDENCIES.ATTRIBUTES ← TMP_CURRENTSECS.ATTRIBUTES;
        TMP_KEYDEPENDENCIES.ATTRIBUTESMASK ← 0;
        TMP_KEYDEPENDENCIES.MRENCLAVE ← TMP_CURRENTSECS.MRENCLAVE;
        TMP_KEYDEPENDENCIES.MRSIGNER ← 0;
        TMP_KEYDEPENDENCIES.KEYID ← DS:RBX.KEYID;
        TMP_KEYDEPENDENCIES.SEAL_KEY_FUSES ← CR_SEAL_FUSES;
        TMP_KEYDEPENDENCIES.CPUSVN ← CR_CPUSVN;
        TMP_KEYDEPENDENCIES.PADDING ← HARDCODED_PKCS1_5_PADDING;
        TMP_KEYDEPENDENCIES.MISCSELECT ← TMP_CURRENTSECS.MISCSELECT;
        TMP_KEYDEPENDENCIES.MISCMASK ← 0;
        TMP_KEYDEPENDENCIES.KEYPOLICY ← 0;
        TMP_KEYDEPENDENCIES.CONFIGID ← TMP_CURRENTSECS.CONFIGID;
        TMP_KEYDEPENDENCIES.CONFIGSVN ← TMP_CURRENTSECS.CONFIGSVN;
         IF (CPUID.(EAX=12H, ECX=1):EAX[6] = 1)
            THEN
                 TMP_KEYDEPENDENCIES.CET_ATTRIBUTES ← TMP_CURRENTSECS.CET_ATTRIBUTES;
                TMP_KEYDEPENDENCIES.CET_ATTRIBUTES_MASK ← 0;
            FI;
        BREAK;
    EINITTOKEN_KEY:
        (* Check ENCLAVE has LAUNCH capability *)
        IF (TMP_CURRENTSECS.ATTRIBUTES.LAUNCHKEY = 0)
            THEN
                RFLAGS.ZF ← 1;
                RAX ← SGX_INVALID_ATTRIBUTE;
                GOTO EXIT;
        FI;
        IF (DS:RBX.CPUSVN is beyond current CPU configuration)
            THEN
                RFLAGS.ZF ← 1;
                RAX ← SGX_INVALID_CPUSVN;
                GOTO EXIT;
        FI;
        IF (DS:RBX.ISVSVN > TMP_CURRENTSECS.ISVSVN)
            THEN
                RFLAGS.ZF ← 1;
                RAX ← SGX_INVALID_ISVSVN;
                GOTO EXIT;
        FI;
        (* Determine values key is based on *)
        TMP_KEYDEPENDENCIES.KEYNAME ← EINITTOKEN_KEY;
        TMP_KEYDEPENDENCIES.ISVFAMILYID ← 0;
        TMP_KEYDEPENDENCIES.ISVEXTPRODID ← 0;
        TMP_KEYDEPENDENCIES.ISVPRODID ← TMP_CURRENTSECS.ISVPRODID
        TMP_KEYDEPENDENCIES.ISVSVN ← DS:RBX.ISVSVN;
        TMP_KEYDEPENDENCIES.SGXOWNEREPOCH ← CR_SGXOWNEREPOCH;
        TMP_KEYDEPENDENCIES.ATTRIBUTES ← TMP_ATTRIBUTES;
        TMP_KEYDEPENDENCIES.ATTRIBUTESMASK ← 0;
        TMP_KEYDEPENDENCIES.MRENCLAVE ← 0;
        TMP_KEYDEPENDENCIES.MRSIGNER ← TMP_CURRENTSECS.MRSIGNER;
        TMP_KEYDEPENDENCIES.KEYID ← DS:RBX.KEYID;
        TMP_KEYDEPENDENCIES.SEAL_KEY_FUSES ← CR_SEAL_FUSES;
        TMP_KEYDEPENDENCIES.CPUSVN ← DS:RBX.CPUSVN;
        TMP_KEYDEPENDENCIES.PADDING ← TMP_CURRENTSECS.PADDING;
        TMP_KEYDEPENDENCIES.MISCSELECT ← TMP_MISCSELECT;
        TMP_KEYDEPENDENCIES.MISCMASK ← 0;
        TMP_KEYDEPENDENCIES.KEYPOLICY ← 0;
        TMP_KEYDEPENDENCIES.CONFIGID ← 0;
        TMP_KEYDEPENDENCIES.CONFIGSVN ← 0;
         IF (CPUID.(EAX=12H, ECX=1):EAX[6] = 1)
            THEN
                TMP_KEYDEPENDENCIES.CET_ATTRIBUTES ← TMP_CET_ATTRIBUTES;
                TMP_KEYDEPENDENCIES.CET_ATTRIBUTES _MASK ← 0;
            FI;
        BREAK;
    PROVISION_KEY:
    (* Check ENCLAVE has PROVISIONING capability *)
        IF (TMP_CURRENTSECS.ATTRIBUTES.PROVISIONKEY = 0)
            THEN
                RFLAGS.ZF ← 1;
                RAX ← SGX_INVALID_ATTRIBUTE;
                GOTO EXIT;
        FI;
        IF (DS:RBX.CPUSVN is beyond current CPU configuration)
            THEN
                RFLAGS.ZF ← 1;
                RAX ← SGX_INVALID_CPUSVN;
                GOTO EXIT;
        FI;
        IF (DS:RBX.ISVSVN > TMP_CURRENTSECS.ISVSVN)
            THEN
                RFLAGS.ZF ← 1;
                RAX ← SGX_INVALID_ISVSVN;
                GOTO EXIT;
        FI;
```

```
            (* Determine values key is based on *)
            TMP_KEYDEPENDENCIES.KEYNAME ← PROVISION_KEY;
            TMP_KEYDEPENDENCIES.ISVFAMILYID ← 0;
            TMP_KEYDEPENDENCIES.ISVEXTPRODID ← 0;
            TMP_KEYDEPENDENCIES.ISVPRODID ← TMP_CURRENTSECS.ISVPRODID;
            TMP_KEYDEPENDENCIES.ISVSVN ← DS:RBX.ISVSVN;
            TMP_KEYDEPENDENCIES.SGXOWNEREPOCH ← 0;
            TMP_KEYDEPENDENCIES.ATTRIBUTES ← TMP_ATTRIBUTES;
            TMP_KEYDEPENDENCIES.ATTRIBUTESMASK ← DS:RBX.ATTRIBUTEMASK;
            TMP_KEYDEPENDENCIES.MRENCLAVE ← 0;
            TMP_KEYDEPENDENCIES.MRSIGNER ← TMP_CURRENTSECS.MRSIGNER;
            TMP_KEYDEPENDENCIES.KEYID ← 0;
            TMP_KEYDEPENDENCIES.SEAL_KEY_FUSES ← 0;
            TMP_KEYDEPENDENCIES.CPUSVN ← DS:RBX.CPUSVN;
            TMP_KEYDEPENDENCIES.PADDING ← TMP_CURRENTSECS.PADDING;
            TMP_KEYDEPENDENCIES.MISCSELECT ← TMP_MISCSELECT;
            TMP_KEYDEPENDENCIES.MISCMASK ← ~DS:RBX.MISCMASK;
            TMP_KEYDEPENDENCIES.KEYPOLICY ← 0;
            TMP_KEYDEPENDENCIES.CONFIGID ← 0;
             IF (CPUID.(EAX=12H, ECX=1):EAX[6] = 1)
                THEN
                    TMP_KEYDEPENDENCIES.CET_ATTRIBUTES ← TMP_CET_ATTRIBUTES;
                    TMP_KEYDEPENDENCIES.CET_ATTRIBUTES _MASK ← 0;
                FI;
            BREAK;
        PROVISION_SEAL_KEY:
            (* Check ENCLAVE has PROVISIONING capability *)
            IF (TMP_CURRENTSECS.ATTRIBUTES.PROVISIONKEY = 0)
                THEN
                    RFLAGS.ZF ← 1;
                    RAX ← SGX_INVALID_ATTRIBUTE;
                    GOTO EXIT;
            FI;
            IF (DS:RBX.CPUSVN is beyond current CPU configuration)
                THEN
                    RFLAGS.ZF ← 1;
                    RAX ← SGX_INVALID_CPUSVN;
                    GOTO EXIT;
            FI;
            IF (DS:RBX.ISVSVN > TMP_CURRENTSECS.ISVSVN)
                THEN
                    RFLAGS.ZF ← 1;
                    RAX ← SGX_INVALID_ISVSVN;
                    GOTO EXIT;
            FI;
(* Include enclave product family ID? *)
        TMP_ISVFAMILYID ← 0;
        IF (DS:RBX.KEYPOLICY.ISVFAMILYID = 1)
            THEN TMP_ISVFAMILYID ← TMP_CURRENTSECS.ISVFAMILYID;
            FI;

        (* Include enclave product ID? *)
        TMP_ISVPRODID ← 0;
        IF (DS:RBX.KEYPOLICY.NOISVPRODID = 0)
                TMP_ISVPRODID ←TMP_CURRENTSECS.ISVPRODID;
            FI;

        (* Include enclave Config ID? *)
        TMP_CONFIGID ← 0;
        TMP_CONFIGSVN ← 0;
        IF (DS:RBX.KEYPOLICY.CONFIGID = 1)
                TMP_CONFIGID ← TMP_CURRENTSECS.CONFIGID;
                TMP_CONFIGSVN ← DS:RBX.CONFIGSVN;
            FI;

        (* Include enclave extended product ID? *)
        TMP_ISVEXTPRODID ← 0;
        IF (DS:RBX.KEYPOLICY.ISVEXTPRODID = 1)
                TMP_ISVEXTPRODID ← TMP_CURRENTSECS.ISVEXTPRODID;
        FI;

            (* Determine values key is based on *)
            TMP_KEYDEPENDENCIES.KEYNAME ← PROVISION_SEAL_KEY;
            TMP_KEYDEPENDENCIES.ISVFAMILYID ← TMP_ISVFAMILYID;
            TMP_KEYDEPENDENCIES.ISVEXTPRODID ← TMP_ISVEXTPRODID;
            TMP_KEYDEPENDENCIES.ISVPRODID ← TMP_ISVPRODID;
            TMP_KEYDEPENDENCIES.ISVSVN ← DS:RBX.ISVSVN;
            TMP_KEYDEPENDENCIES.SGXOWNEREPOCH ← 0;
            TMP_KEYDEPENDENCIES.ATTRIBUTES ← TMP_ATTRIBUTES;
            TMP_KEYDEPENDENCIES.ATTRIBUTESMASK ← DS:RBX.ATTRIBUTEMASK;
            TMP_KEYDEPENDENCIES.MRENCLAVE ← 0;
            TMP_KEYDEPENDENCIES.MRSIGNER ← TMP_CURRENTSECS.MRSIGNER;
            TMP_KEYDEPENDENCIES.KEYID ← 0;
            TMP_KEYDEPENDENCIES.SEAL_KEY_FUSES ← CR_SEAL_FUSES;
            TMP_KEYDEPENDENCIES.CPUSVN ← DS:RBX.CPUSVN;
            TMP_KEYDEPENDENCIES.PADDING ← TMP_CURRENTSECS.PADDING;
            TMP_KEYDEPENDENCIES.MISCSELECT ← TMP_MISCSELECT;
            TMP_KEYDEPENDENCIES.MISCMASK ← ~DS:RBX.MISCMASK;
            TMP_KEYDEPENDENCIES.KEYPOLICY ← DS:RBX.KEYPOLICY;
            TMP_KEYDEPENDENCIES.CONFIGID ← TMP_CONFIGID;
            TMP_KEYDEPENDENCIES.CONFIGSVN ← TMP_CONFIGSVN;
             IF (CPUID.(EAX=12H, ECX=1):EAX[6] = 1)
                THEN
                    TMP_KEYDEPENDENCIES.CET_ATTRIBUTES ← TMP_CET_ATTRIBUTES;
                    TMP_KEYDEPENDENCIES.CET_ATTRIBUTES _MASK ← 0;
                FI;
            BREAK;
        DEFAULT:
            (* The value of KEYNAME is invalid *)
            RFLAGS.ZF ← 1;
            RAX ← SGX_INVALID_KEYNAME;
            GOTO EXIT:
```

```
ESAC;

(* Calculate the final derived key and output to the address in RCX *)
TMP_OUTPUTKEY ← derivekey(TMP_KEYDEPENDENCIES);
DS:RCX[15:0] ← TMP_OUTPUTKEY;
RAX ← 0;
RFLAGS.ZF ← 0;

EXIT:
RFLAGS.CF ← 0;
RFLAGS.PF ← 0;
RFLAGS.AF ← 0;
RFLAGS.OF ← 0;
RFLAGS.SF ← 0;
```

Flags Affected

ZF is cleared if successful, otherwise ZF is set. CF, PF, AF, OF, SF are cleared.

Protected Mode Exceptions

#GP(0)	If executed outside an enclave.
	If a memory operand effective address is outside the current enclave.
	If an effective address is not properly aligned.
	If an effective address is outside the DS segment limit.
	If KEYREQUEST format is invalid.
#PF(error code)	If a page fault occurs in accessing memory.

64-Bit Mode Exceptions

#GP(0)	If executed outside an enclave.
	If a memory operand effective address is outside the current enclave.
	If an effective address is not properly aligned.
	If an effective address is not canonical.
	If KEYREQUEST format is invalid.
#PF(error code)	If a page fault occurs in accessing memory operands.

EMODPE—Extend an EPC Page Permissions

Opcode/ Instruction	**Op/En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
EAX = 06H ENCLU[EMODPE]	IR	V/V	SGX2	This leaf function extends the access rights of an existing EPC page.

Instruction Operand Encoding

Op/En	EAX	RBX	RCX
IR	EMODPE (In)	Address of a SECINFO (In)	Address of the destination EPC page (In)

Description

This leaf function extends the access rights associated with an existing EPC page in the running enclave. THE RWX bits of the SECINFO parameter are treated as a permissions mask; supplying a value that does not extend the page permissions will have no effect. This instruction leaf can only be executed when inside the enclave.

RBX contains the effective address of a SECINFO structure while RCX contains the effective address of an EPC page. The table below provides additional information on the memory parameter of the EMODPE leaf function.

EMODPE Memory Parameter Semantics

SECINFO	EPCPAGE
Read access permitted by Non Enclave	Read access permitted by Enclave

EMODPE Faulting Conditions

The instruction faults if any of the following:

The operands are not properly aligned.	If security attributes of the SECINFO page make the page inaccessible.
The EPC page is locked by another thread.	RBX does not contain an effective address in an EPC page in the running enclave.
The EPC page is not valid.	RCX does not contain an effective address of an EPC page in the running enclave.
SECINFO contains an invalid request.	

Concurrency Restrictions

Base Concurrency Restrictions of EMODPE

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EMODPE	Target [DS:RCX]	Concurrent		
	SECINFO [DS:RBX]	Concurrent		

Additional Concurrency Restrictions of EMODPE

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EMODPE	Target [DS:RCX]	Exclusive	#GP	Concurrent		Concurrent	
	SECINFO [DS:RBX]	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in EMODPE Operational Flow

Name	Type	Size (bits)	Description
SCRATCH_SECINFO	SECINFO	512	Scratch storage for holding the contents of DS:RBX.

```
IF (DS:RBX is not 64Byte Aligned)
    THEN #GP(0); FI;

IF (DS:RCX is not 4KByte Aligned)
    THEN #GP(0); FI;

IF ((DS:RBX is not within CR_ELRANGE) or (DS:RCX is not within CR_ELRANGE) )
    THEN #GP(0); FI;

IF (DS:RBX does not resolve within an EPC)
    THEN #PF(DS:RBX); FI;
```

```
IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;

IF ( (EPCM(DS:RBX).VALID = 0) or (EPCM(DS:RBX).R = 0) or (EPCM(DS:RBX).PENDING ≠ 0) or (EPCM(DS:RBX).MODIFIED ≠ 0) or
    (EPCM(DS:RBX).BLOCKED ≠ 0) or (EPCM(DS:RBX).PT ≠ PT_REG) or (EPCM(DS:RBX).ENCLAVESECS ≠ CR_ACTIVE_SECS) or
    (EPCM(DS:RBX).ENCLAVEADDRESS ≠ (DS:RBX & ~0xFFF)) )
    THEN #PF(DS:RBX); FI;

SCRATCH_SECINFO ← DS:RBX;

(* Check for misconfigured SECINFO flags*)
IF (SCRATCH_SECINFO reserved fields are not zero )
    THEN #GP(0); FI;

(* Check security attributes of the EPC page *)
IF ( (EPCM(DS:RCX).VALID = 0) or (EPCM(DS:RCX).PENDING ≠ 0) or (EPCM(DS:RCX).MODIFIED ≠ 0) or
    (EPCM(DS:RCX).BLOCKED ≠ 0) or (EPCM(DS:RCX).PT ≠ PT_REG) or (EPCM(DS:RCX).ENCLAVESECS ≠ CR_ACTIVE_SECS) )
    THEN #PF(DS:RCX); FI;

(* Check the EPC page for concurrency *)
IF (EPC page in use by another SGX2 instruction)
    THEN #GP(0); FI;

(* Re-Check security attributes of the EPC page *)
IF ( (EPCM(DS:RCX).VALID = 0) or (EPCM(DS:RCX).PENDING ≠ 0) or (EPCM(DS:RCX).MODIFIED ≠ 0) or
    (EPCM(DS:RCX).PT ≠ PT_REG) or (EPCM(DS:RCX).ENCLAVESECS ≠ CR_ACTIVE_SECS) or
    (EPCM(DS:RCX).ENCLAVEADDRESS ≠ DS:RCX))
    THEN #PF(DS:RCX); FI;

(* Check for misconfigured SECINFO flags*)
IF ( (EPCM(DS:RCX).R = 0) and (SCRATCH_SECINFO.FLAGS.R = 0) and (SCRATCH_SECINFO.FLAGS.W ≠ 0) )
    THEN #GP(0); FI;

(* Update EPCM permissions *)
EPCM(DS:RCX).R ← EPCM(DS:RCX).R | SCRATCH_SECINFO.FLAGS.R;
EPCM(DS:RCX).W ← EPCM(DS:RCX).W | SCRATCH_SECINFO.FLAGS.W;
EPCM(DS:RCX).X ← EPCM(DS:RCX).X | SCRATCH_SECINFO.FLAGS.X;
```

Flags Affected

None

Protected Mode Exceptions

#GP(0)	If executed outside an enclave.
	If a memory operand effective address is outside the DS segment limit.
	If a memory operand is not properly aligned.
	If a memory operand is locked.
#PF(error code)	If a page fault occurs in accessing memory operands.

64-Bit Mode Exceptions

#GP(0)	If executed outside an enclave.
	If a memory operand is non-canonical form.
	If a memory operand is not properly aligned.
	If a memory operand is locked.
#PF(error code)	If a page fault occurs in accessing memory operands.

EREPORT—Create a Cryptographic Report of the Enclave

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 00H ENCLU[EREPORT]	IR	V/V	SGX1	This leaf function creates a cryptographic report of the enclave.

Instruction Operand Encoding

Op/En	EAX	RBX	RCX	RDX
IR	EREPORT (In)	Address of TARGETINFO (In)	Address of RE-PORTDATA (In)	Address where the REPORT is written to in an OUTPUTDATA (In)

Description

This leaf function creates a cryptographic REPORT that describes the contents of the enclave. This instruction leaf can only be executed when inside the enclave. The cryptographic report can be used by other enclaves to determine that the enclave is running on the same platform.

RBX contains the effective address of the MRENCLAVE value of the enclave that will authenticate the REPORT output, using the REPORT key delivered by EGETKEY command for that enclave. RCX contains the effective address of a 64-byte REPORTDATA structure, which allows the caller of the instruction to associate data with the enclave from which the instruction is called. RDX contains the address where the REPORT will be output by the instruction.

EREPORT Memory Parameter Semantics

TARGETINFO	REPORTDATA	OUTPUTDATA
Read access by Enclave	Read access by Enclave	Read/Write access by Enclave

This instruction leaf perform the following:

1. Validate the 3 operands (RBX, RCX, RDX) are inside the enclave.
2. Compute a report key for the target enclave, as indicated by the value located in RBX(TARGETINFO).
3. Assemble the enclave SECS data to complete the REPORT structure (including the data provided using the RCX (REPORTDATA) operand).
4. Computes a cryptographic hash over REPORT structure.
5. Add the computed hash to the REPORT structure.
6. Output the completed REPORT structure to the address in RDX (OUTPUTDATA).

The instruction fails if the operands are not properly aligned.

CR_REPORT_KEYID, used to provide key wearout protection, is populated with a statistically unique value on boot of the platform by a trusted entity within the SGX TCB.

EREPORT Faulting Conditions

The instruction faults if any of the following:

An effective address not properly aligned.	An memory address does not resolve in an EPC page.
If accessing an invalid EPC page.	If the EPC page is blocked.
May page fault.	

Concurrency Restrictions

Base Concurrency Restrictions of EREPORT

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EREPORT	TARGETINFO [DS:RBX]	Concurrent		
	REPORTDATA [DS:RCX]	Concurrent		
	OUTPUTDATA [DS:RDX]	Concurrent		

Additional Concurrency Restrictions of EREPORT

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EREPORT	TARGETINFO [DS:RBX]	Concurrent		Concurrent		Concurrent	
	REPORTDATA [DS:RCX]	Concurrent		Concurrent		Concurrent	
	OUTPUTDATA [DS:RDX]	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in EREPORT Operational Flow

Name	Type	Size (bits)	Description
TMP_ATTRIBUTES		32	Physical address of SECS of the enclave to which source operand belongs.
TMP_CURRENTSECS			Address of the SECS for the currently executing enclave.
TMP_KEYDEPENDENCIES			Temp space for key derivation.
TMP_REPORTKEY		128	REPORTKEY generated by the instruction.
TMP_REPORT		3712	

```
TMP_MODE64 ← ((IA32_EFER.LMA = 1) && (CS.L = 1));

(* Address verification for TARGETINFO (RBX) *)
IF ( (DS:RBX is not 512Byte Aligned) or (DS:RBX is not within CR_ELRANGE) )
    THEN #GP(0); FI;

IF (DS:RBX does not resolve within an EPC)
    THEN #PF(DS:RBX); FI;

IF (EPCM(DS:RBX).VALID = 0)
    THEN #PF(DS:RBX); FI;

IF (EPCM(DS:RBX).BLOCKED = 1)
    THEN #PF(DS:RBX); FI;

(* Check page parameters for correctness *)
IF ( (EPCM(DS:RBX).PT ≠ PT_REG) or (EPCM(DS:RBX).ENCLAVESECS ≠ CR_ACTIVE_SECS) or (EPCM(DS:RBX).PENDING = 1) or
    (EPCM(DS:RBX).MODIFIED = 1) or (EPCM(DS:RBX).ENCLAVEADDRESS ≠ (DS:RBX & ~0FFFH) ) or (EPCM(DS:RBX).R = 0) )
    THEN #PF(DS:RBX);
FI;

(* Address verification for REPORTDATA (RCX) *)
IF ( (DS:RCX is not 128Byte Aligned) or (DS:RCX is not within CR_ELRANGE) )
    THEN #GP(0); FI;

IF (DS:RCX does not resolve within an EPC)
    THEN #PF(DS:RCX); FI;

IF (EPCM(DS:RCX).VALID = 0)
    THEN #PF(DS:RCX); FI;

IF (EPCM(DS:RCX).BLOCKED = 1)
    THEN #PF(DS:RCX); FI;
```

```
(* Check page parameters for correctness *)
IF ( (EPCM(DS:RCX).PT ≠ PT_REG) or (EPCM(DS:RCX).ENCLAVESECS ≠ CR_ACTIVE_SECS) or (EPCM(DS:RCX).PENDING = 1) or
    (EPCM(DS:RCX).MODIFIED = 1) or (EPCM(DS:RCX).ENCLAVEADDRESS ≠ (DS:RCX & ~0FFFH) ) or (EPCM(DS:RCX).R = 0) )
    THEN #PF(DS:RCX);
FI;

(* Address verification for OUTPUTDATA (RDX) *)
IF ( (DS:RDX is not 512Byte Aligned) or (DS:RDX is not within CR_ELRANGE) )
    THEN #GP(0); FI;

IF (DS:RDX does not resolve within an EPC)
    THEN #PF(DS:RDX); FI;

IF (EPCM(DS:RDX).VALID = 0)
    THEN #PF(DS:RDX); FI;

IF (EPCM(DS:RDX).BLOCKED = 1)
    THEN #PF(DS:RDX); FI;

(* Check page parameters for correctness *)
IF ( (EPCM(DS:RDX).PT ≠ PT_REG) or (EPCM(DS:RDX).ENCLAVESECS ≠ CR_ACTIVE_SECS) or (EPCM(DS:RCX).PENDING = 1) or
    (EPCM(DS:RCX).MODIFIED = 1) or (EPCM(DS:RDX).ENCLAVEADDRESS ≠ (DS:RDX & ~0FFFH) ) or (EPCM(DS:RDX).W = 0) )
    THEN #PF(DS:RDX);
FI;

(* REPORT MAC needs to be computed over data which cannot be modified *)
TMP_REPORT.CPUSVN ← CR_CPUSVN;
TMP_REPORT.ISVFAMILYID ← TMP_CURRENTSECS.ISVFAMILYID;
TMP_REPORT.ISVEXTPRODID ← TMP_CURRENTSECS.ISVEXTPRODID;
TMP_REPORT.ISVPRODID ← TMP_CURRENTSECS.ISVPRODID;
TMP_REPORT.ISVSVN ← TMP_CURRENTSECS.ISVSVN;
TMP_REPORT.ATTRIBUTES ← TMP_CURRENTSECS.ATTRIBUTES;
TMP_REPORT.REPORTDATA ← DS:RCX[511:0];
TMP_REPORT.MRENCLAVE ← TMP_CURRENTSECS.MRENCLAVE;
TMP_REPORT.MRSIGNER ← TMP_CURRENTSECS.MRSIGNER;
TMP_REPORT.MRRESERVED ← 0;
TMP_REPORT.KEYID[255:0] ← CR_REPORT_KEYID;
TMP_REPORT.MISCSELECT ← TMP_CURRENTSECS.MISCSELECT;
TMP_REPORT.CONFIGID ← TMP_CURRENTSECS.CONFIGID;
TMP_REPORT.CONFIGSVN ← TMP_CURRENTSECS.CONFIGSVN;
IF (CPUID.(EAX=12H, ECX=1):EAX[6] = 1)
    THEN TMP_REPORT.CET_ATTRIBUTES ← TMP_CURRENTSECS.CET_ATTRIBUTES; FI;

(* Derive the report key *)
TMP_KEYDEPENDENCIES.KEYNAME ← REPORT_KEY;
TMP_KEYDEPENDENCIES.ISVFAMILYID ← 0;
TMP_KEYDEPENDENCIES.ISVEXTPRODID ← 0;
TMP_KEYDEPENDENCIES.ISVPRODID ← 0;
TMP_KEYDEPENDENCIES.ISVSVN ← 0;
TMP_KEYDEPENDENCIES.SGXOWNEREPOCH ← CR_SGXOWNEREPOCH;
TMP_KEYDEPENDENCIES.ATTRIBUTES ← DS:RBX.ATTRIBUTES;
TMP_KEYDEPENDENCIES.ATTRIBUTESMASK ← 0;
TMP_KEYDEPENDENCIES.MRENCLAVE ← DS:RBX.MEASUREMENT;
TMP_KEYDEPENDENCIES.MRSIGNER ← 0;
TMP_KEYDEPENDENCIES.KEYID ← TMP_REPORT.KEYID;
TMP_KEYDEPENDENCIES.SEAL_KEY_FUSES ← CR_SEAL_FUSES;
TMP_KEYDEPENDENCIES.CPUSVN ← CR_CPUSVN;
TMP_KEYDEPENDENCIES.PADDING ← TMP_CURRENTSECS.PADDING;
TMP_KEYDEPENDENCIES.MISCSELECT ← DS:RBX.MISCSELECT;
TMP_KEYDEPENDENCIES.MISCMASK ← 0;
TMP_KEYDEPENDENCIES.KEYPOLICY ← 0;
TMP_KEYDEPENDENCIES.CONFIGID ← DS:RBX.CONFIGID;
TMP_KEYDEPENDENCIES.CONFIGSVN ← DS:RBX.CONFIGSVN;
IF (CPUID.(EAX=12H, ECX=1):EAX[6] = 1)
    THEN
        TMP_KEYDEPENDENCIES.CET_ATTRIBUTES ← DS:RBX.CET_ATTRIBUTES;
        TMP_KEYDEPENDENCIES.CET_ATTRIBUTES _MASK ← 0;
    FI;

(* Calculate the derived key*)
TMP_REPORTKEY ← derive_key(TMP_KEYDEPENDENCIES);

(* call cryptographic CMAC function, CMAC data are not including MAC&KEYID *)
TMP_REPORT.MAC ← cmac(TMP_REPORTKEY, TMP_REPORT[3071:0] );
DS:RDX[3455: 0] ← TMP_REPORT;
```

Flags Affected

None

Protected Mode Exceptions

#GP(0)	If executed outside an enclave.
	If the address in RCS is outside the DS segment limit.
	If a memory operand is not properly aligned.
	If a memory operand is not in the current enclave.
#PF(error code)	If a page fault occurs in accessing memory operands.

64-Bit Mode Exceptions

#GP(0)	If executed outside an enclave.
	If RCX is non-canonical form.
	If a memory operand is not properly aligned.
	If a memory operand is not in the current enclave.
#PF(error code)	If a page fault occurs in accessing memory operands.

ERESUME—Re-Enters an Enclave

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 03H ENCLU[ERESUME]	IR	V/V	SGX1	This leaf function is used to re-enter an enclave after an interrupt.

Instruction Operand Encoding

Op/En	RAX	RBX	RCX
IR	ERESUME (In)	Address of a TCS (In)	Address of AEP (In)

Description

ERESUME Memory Parameter Semantics

The ENCLU[ERESUME] instruction resumes execution of an enclave that was interrupted due to an exception or interrupt, using the machine state previously stored in the SSA.

TCS
Enclave read/write access

The instruction faults if any of the following:

Address in RBX is not properly aligned.	Any TCS.FLAGS's must-be-zero bit is not zero.
TCS pointed to by RBX is not valid or available or locked.	Current 32/64 mode does not match the enclave mode in SECS.ATTRIBUTES.MODE64.
The SECS is in use by another enclave.	Either of TCS-specified FS and GS segment is not a subset of the current DS segment.
Any one of DS, ES, CS, SS is not zero.	If XSAVE available, CR4.OSXSAVE = 0, but SECS.ATTRIBUTES.XFRM ≠ 3.
CR4.OSFXSR ≠ 1.	If CR4.OSXSAVE = 1, SECS.ATTRIBUTES.XFRM is not a subset of XCR0.
Offsets 520-535 of the XSAVE area not 0.	The bit vector stored at offset 512 of the XSAVE area must be a subset of SECS.ATTRIBUTES.XFRM.
The SSA frame is not valid or in use.	

The following operations are performed by ERESUME:

- RSP and RBP are saved in the current SSA frame on EENTER and are automatically restored on EEXIT or an asynchronous exit due to any Interrupt event.
- The AEP contained in RCX is stored into the TCS for use by AEXs.FS and GS (including hidden portions) are saved and new values are constructed using TCS.OFSBASE/GSBASE (32 and 64-bit mode) and TCS.OFSLIMIT/GSLIMIT (32-bit mode only). The resulting segments must be a subset of the DS segment.
- If CR4.OSXSAVE == 1, XCR0 is saved and replaced by SECS.ATTRIBUTES.XFRM. The effect of RFLAGS.TF depends on whether the enclave entry is opt-in or opt-out:
  - On opt-out entry, TF is saved and cleared (it is restored on EEXIT or AEX). Any attempt to set TF via a POPF instruction while inside the enclave clears TF.
  - On opt-in entry, a single-step debug exception is pended on the instruction boundary immediately after EENTER.
- All code breakpoints that do not overlap with ELRANGE are also suppressed. If the entry is an opt-out entry, all code and data breakpoints that overlap with the ELRANGE are suppressed.
- On opt-out entry, a number of performance monitoring counters and behaviors are modified or suppressed:
  - All performance monitoring activity on the current thread is suppressed except for incrementing and firing of FIXED_CTR1 and FIXED_CTR2.
  - PEBS is suppressed.
  - AnyThread counting on other threads is demoted to MyThread mode and IA32_PERF_GLOBAL_STATUS[60] on that thread is set.
  - If the opt-out entry on a hardware thread results in suppression of any performance monitoring, then the processor sets IA32_PERF_GLOBAL_STATUS[60] and IA32_PERF_GLOBAL_STATUS[63].

Concurrency Restrictions

Base Concurrency Restrictions of ERESUME

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
ERESUME	TCS [DS:RBX]	Shared	#GP	

Additional Concurrency Restrictions of ERESUME

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
ERESUME	TCS [DS:RBX]	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in ERESUME Operational Flow

Name	Type	Size	Description
TMP_FSBASE	Effective Address	32/64	Proposed base address for FS segment.
TMP_GSBASE	Effective Address	32/64	Proposed base address for FS segment.
TMP_FSLIMIT	Effective Address	32/64	Highest legal address in proposed FS segment.
TMP_GSLIMIT	Effective Address	32/64	Highest legal address in proposed GS segment.
TMP_TARGET	Effective Address	32/64	Address of first instruction inside enclave at which execution is to resume.
TMP_SECS	Effective Address	32/64	Physical address of SECS for this enclave.
TMP_SSA	Effective Address	32/64	Address of current SSA frame.
TMP_XSIZE	integer	64	Size of XSAVE area based on SECS.ATTRIBUTES.XFRM.
TMP_SSA_PAGE	Effective Address	32/64	Pointer used to iterate over the SSA pages in the current frame.
TMP_GPR	Effective Address	32/64	Address of the GPR area within the current SSA frame.
TMP_BRANCH_RECORD	LBR Record		From/to addresses to be pushed onto the LBR stack.

```
TMP_MODE64 ← ((IA32_EFER.LMA = 1) && (CS.L = 1));

(* Make sure DS is usable, expand up *)
IF (TMP_MODE64 = 0 and (DS not usable or ( ( DS[S] = 1) and (DS[bit 11] = 0) and DS[bit 10] = 1) ) ) )
    THEN #GP(0); FI;

(* Check that CS, SS, DS, ES.base is 0 *)
IF (TMP_MODE64 = 0)
    THEN
        IF(CS.base ≠ 0 or DS.base ≠ 0) #GP(0); FI;
        IF(ES usable and ES.base ≠ 0) #GP(0); FI;
        IF(SS usable and SS.base ≠ 0) #GP(0); FI;
        IF(SS usable and SS.B = 0) #GP(0); FI;
FI;

IF (DS:RBX is not 4KByte Aligned)
    THEN #GP(0); FI;
```

```
IF (DS:RBX does not resolve within an EPC)
    THEN #PF(DS:RBX); FI;

(* Check AEP is canonical*)
IF (TMP_MODE64 = 1 and (CS:RCX is not canonical) )
    THEN #GP(0); FI;

(* Check concurrency of TCS operation*)
IF (Other Intel SGX instructions is operating on TCS)
    THEN #GP(0); FI;

(* TCS verification *)
IF (EPCM(DS:RBX).VALID = 0)
    THEN #PF(DS:RBX); FI;

IF (EPCM(DS:RBX).BLOCKED = 1)
    THEN #PF(DS:RBX); FI;

IF ((EPCM(DS:RBX).PENDING = 1) or (EPCM(DS:RBX).MODIFIED = 1))
    THEN #PF(DS:RBX); FI;

IF ( (EPCM(DS:RBX).ENCLAVEADDRESS ≠ DS:RBX) or (EPCM(DS:RBX).PT ≠ PT_TCS) )
    THEN #PF(DS:RBX); FI;

IF ( (DS:RBX).OSSA is not 4KByte Aligned)
    THEN #GP(0); FI;

(* Check proposed FS and GS *)
IF ( ( (DS:RBX).OFSBASE is not 4KByte Aligned) or ( (DS:RBX).OGSBASE is not 4KByte Aligned) )
    THEN #GP(0); FI;

(* Get the SECS for the enclave in which the TCS resides *)
TMP_SECS ← Address of SECS for TCS;

(* Make sure that the FLAGS field in the TCS does not have any reserved bits set *)
IF ( ( (DS:RBX).FLAGS & FFFFFFFFFFFFFFFEH) ≠ 0)
    THEN #GP(0); FI;

(* SECS must exist and enclave must have previously been EINITted *)
IF (the enclave is not already initialized)
    THEN #GP(0); FI;

(* make sure the logical processor's operating mode matches the enclave *)
IF ( (TMP_MODE64 ≠ TMP_SECS.ATTRIBUTES.MODE64BIT) )
    THEN #GP(0); FI;

IF (CR4.OSFXSR = 0)
    THEN #GP(0); FI;

(* Check for legal values of SECS.ATTRIBUTES.XFRM *)
IF (CR4.OSXSAVE = 0)
    THEN
        IF (TMP_SECS.ATTRIBUTES.XFRM ≠ 03H) THEN #GP(0); FI;
    ELSE
        IF ( (TMP_SECS.ATTRIBUTES.XFRM & XCR0) ≠ TMP_SECS.ATTRIBUTES.XFRM) THEN #GP(0); FI;
FI;

(* Make sure the SSA contains at least one active frame *)
IF ( (DS:RBX).CSSA = 0)
    THEN #GP(0); FI;

(* Compute linear address of SSA frame *)
TMP_SSA ← (DS:RBX).OSSA + TMP_SECS.BASEADDR + 4096 * TMP_SECS.SSAFRAMESIZE * ( (DS:RBX).CSSA - 1);
TMP_XSIZE ← compute_XSAVE_frame_size(TMP_SECS.ATTRIBUTES.XFRM);

FOR EACH TMP_SSA_PAGE = TMP_SSA to TMP_SSA + TMP_XSIZE
    (* Check page is read/write accessible *)
    Check that DS:TMP_SSA_PAGE is read/write accessible;
    If a fault occurs, release locks, abort and deliver that fault;
    IF (DS:TMP_SSA_PAGE does not resolve to EPC page)
        THEN #PF(DS:TMP_SSA_PAGE); FI;
    IF (EPCM(DS:TMP_SSA_PAGE).VALID = 0)
        THEN #PF(DS:TMP_SSA_PAGE); FI;
    IF (EPCM(DS:TMP_SSA_PAGE).BLOCKED = 1)
        THEN #PF(DS:TMP_SSA_PAGE); FI;
    IF ((EPCM(DS:TMP_SSA_PAGE).PENDING = 1) or (EPCM(DS:TMP_SSA_PAGE_.MODIFIED = 1))
        THEN #PF(DS:TMP_SSA_PAGE); FI;
    IF ( ( EPCM(DS:TMP_SSA_PAGE).ENCLAVEADDRESS ≠ DS:TMPSSA_PAGE) or (EPCM(DS:TMP_SSA_PAGE).PT ≠ PT_REG) or
        (EPCM(DS:TMP_SSA_PAGE).ENCLAVESECS ≠ EPCM(DS:RBX).ENCLAVESECS) or
        (EPCM(DS:TMP_SSA_PAGE).R = 0) or (EPCM(DS:TMP_SSA_PAGE).W = 0) )
        THEN #PF(DS:TMP_SSA_PAGE); FI;
    CR_XSAVE_PAGE_n ← Physical_Address(DS:TMP_SSA_PAGE);
ENDFOR

(* Compute address of GPR area*)
TMP_GPR ← TMP_SSA + 4096 * DS:TMP_SECS.SSAFRAMESIZE - sizeof(GPRSGX_AREA);
Check that DS:TMP_SSA_PAGE is read/write accessible;
If a fault occurs, release locks, abort and deliver that fault;
IF (DS:TMP_GPR does not resolve to EPC page)
    THEN #PF(DS:TMP_GPR); FI;
IF (EPCM(DS:TMP_GPR).VALID = 0)
    THEN #PF(DS:TMP_GPR); FI;
IF (EPCM(DS:TMP_GPR).BLOCKED = 1)
    THEN #PF(DS:TMP_GPR); FI;
IF ((EPCM(DS:TMP_GPR).PENDING = 1) or (EPCM(DS:TMP_GPR).MODIFIED = 1))
    THEN #PF(DS:TMP_GPR); FI;
IF ( ( EPCM(DS:TMP_GPR).ENCLAVEADDRESS ≠ DS:TMP_GPR) or (EPCM(DS:TMP_GPR).PT ≠ PT_REG) or
    (EPCM(DS:TMP_GPR).ENCLAVESECS ≠ EPCM(DS:RBX).ENCLAVESECS) or
    (EPCM(DS:TMP_GPR).R = 0) or (EPCM(DS:TMP_GPR).W = 0) )
    THEN #PF(DS:TMP_GPR); FI;

IF (TMP_MODE64 = 0)
```

```
    THEN
        IF (TMP_GPR + (GPR_SIZE -1) is not in DS segment) THEN #GP(0); FI;
FI;

CR_GPR_PA ← Physical_Address (DS: TMP_GPR);

TMP_TARGET ← (DS:TMP_GPR).RIP;
IF (TMP_MODE64 = 1)
    THEN
        IF (TMP_TARGET is not canonical) THEN #GP(0); FI;
    ELSE
        IF (TMP_TARGET > CS limit) THEN #GP(0); FI;
FI;

(* Check proposed FS/GS segments fall within DS *)
IF (TMP_MODE64 = 0)
    THEN
        TMP_FSBASE ← (DS:RBX).OFSBASE + TMP_SECS.BASEADDR;
        TMP_FSLIMIT ← (DS:RBX).OFSBASE + TMP_SECS.BASEADDR + (DS:RBX).FSLIMIT;
        TMP_GSBASE ← (DS:RBX).OGSBASE + TMP_SECS.BASEADDR;
        TMP_GSLIMIT ← (DS:RBX).OGSBASE + TMP_SECS.BASEADDR + (DS:RBX).GSLIMIT;
        (* if FS wrap-around, make sure DS has no holes*)
        IF (TMP_FSLIMIT < TMP_FSBASE)
            THEN
                IF (DS.limit < 4GB) THEN #GP(0); FI;
            ELSE
                IF (TMP_FSLIMIT > DS.limit) THEN #GP(0); FI;
        FI;
        (* if GS wrap-around, make sure DS has no holes*)
        IF (TMP_GSLIMIT < TMP_GSBASE)
            THEN
                IF (DS.limit < 4GB) THEN #GP(0); FI;
            ELSE
                IF (TMP_GSLIMIT > DS.limit) THEN #GP(0); FI;
        FI;
    ELSE
        TMP_FSBASE ← DS:TMP_GPR.FSBASE;
        TMP_GSBASE ← DS:TMP_GPR.GSBASE;
        IF ( (TMP_FSBASE is not canonical) or (TMP_GSBASE is not canonical))
            THEN #GP(0); FI;
FI;

(* Ensure the enclave is not already active and this thread is the only one using the TCS*)
IF (DS:RBX.STATE = ACTIVE))
    THEN #GP(0); FI;

IF (CPUID.(EAX=12H, ECX=1):EAX[6] = 1)
    THEN
        IF ( CR4.CET = 0 )
            THEN
                (* If part does not support CET or CET has not been enabled and enclave requires CET then fail *)
                IF ( TMP_SECS.CET_ATTRIBUTES ≠ 0 OR TMP_SECS.CET_LEG_BITMAP_OFFSET ≠ 0 ) #GP(0); FI;
            FI;
```

```
(* If indirect branch tracking or shadow stacks enabled but CET state save area is not 16B aligned then fail ERESUME *)
IF ( TMP_SECS.CET_ATTRIBUTES.SH_STK_EN = 1 OR TMP_SECS.CET_ATTRIBUTES.ENDBR_EN = 1 )
    THEN
        IF (DS:RBX.OCETSSA is not 16B aligned) #GP(0); FI;
    FI;

TMP_IA32_U_CET ← 0;
TMP_SSP ← 0;

IF (TMP_SECS.CET_ATTRIBUTES.SH_STK_EN OR TMP_SECS.CET_ATTRIBUTES.ENDBR_EN)
    THEN
        (* Setup CET state from SECS, note tracker goes to IDLE *)
        TMP_IA32_U_CET = TMP_SECS.CET_ATTRIBUTES;
        IF (TMP_IA32_U_CET.LEG_IW_EN = 1 AND TMP_IA32_U_CET.ENDBR_EN = 1 )
            THEN
                TMP_IA32_U_CET ← TMP_IA32_U_CET + TMP_SECS.BASEADDR;
                TMP_IA32_U_CET ← TMP_IA32_U_CET + TMP_SECS.CET_LEG_BITMAP_BASE;
            FI;

        (* Compute linear address of what will become new CET state save area and cache its PA *)
        TMP_CET_SAVE_AREA = DS:RBX.OCETSSA + TMP_SECS.BASEADDR + (DS:RBX.CSSA - 1) * 16
        TMP_CET_SAVE_PAGE = TMP_CET_SAVE_AREA & ~0xFFF;

         Check the TMP_CET_SAVE_PAGE page is read/write accessible
         If fault occurs release locks, abort and deliver fault

         (* read the EPCM VALID, PENDING, MODIFIED, BLOCKED and PT fields atomically *)
        IF ((DS:TMP_CET_SAVE_PAGE Does NOT RESOLVE TO EPC PAGE) OR
            (EPCM(DS:TMP_CET_SAVE_PAGE).VALID = 0) OR
            (EPCM(DS:TMP_CET_SAVE_PAGE).PENDING = 1) OR
            (EPCM(DS:TMP_CET_SAVE_PAGE).MODIFIED = 1) OR
            (EPCM(DS:TMP_CET_SAVE_PAGE).BLOCKED = 1) OR
            (EPCM(DS:TMP_CET_SAVE_PAGE).R = 0) OR
            (EPCM(DS:TMP_CET_SAVE_PAGE).W = 0) OR
            (EPCM(DS:TMP_CET_SAVE_PAGE).ENCLAVEADDRESS ≠ DS:TMP_CET_SAVE_PAGE) OR
            (EPCM(DS:TMP_CET_SAVE_PAGE).PT ≠ PT_SS_REST) OR
            (EPCM(DS:TMP_CET_SAVE_PAGE).ENCLAVESECS ≠ EPCM(DS:RBX).ENCLAVESECS))
            THEN
                #PF(DS:TMP_CET_SAVE_PAGE);
            FI;

         CR_CET_SAVE_AREA_PA ← Physical address(DS:TMP_CET_SAVE_AREA)

         TMP_SSP = CR_CET_SAVE_AREA_PA.SSP
         TMP_IA32_U_CET.TRACKER = CR_CET_SAVE_AREA_PA.TRACKER;
         TMP_IA32_U_CET.SUPPRESS = CR_CET_SAVE_AREA_PA.SUPPRESS;

         If ( (TMP_MODE64 = 1 AND TMP_SSP is not machine canonical) OR
              (TMP_MODE64 = 0 AND (TMP_SSP & 0xFFFFFFFF00000000) ≠ 0) OR
              (TMP_SSP is not 4 byte aligned) OR
              (TMP_IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH AND TMP_IA32_U_CET.SUPPRESS = 1) OR
              (CR_CET_SAVE_AREA_PA.Reserved ≠ 0) ) #GP(0); FI;
            FI;
    FI;

(* SECS.ATTRIBUTES.XFRM selects the features to be saved. *)
(* CR_XSAVE_PAGE_n: A list of 1 or more physical address of pages that contain the XSAVE area. *)
XRSTOR(TMP_MODE64, SECS.ATTRIBUTES.XFRM, CR_XSAVE_PAGE_n);

IF (XRSTOR failed with #GP)
    THEN
        DS:RBX.STATE ← INACTIVE;
        #GP(0);
FI;

CR_ENCLAVE_MODE ← 1;
CR_ACTIVE_SECS ← TMP_SECS;
CR_ELRANGE ← (TMP_SECS.BASEADDR, TMP_SECS.SIZE);

(* Save sate for possible AEXs *)
CR_TCS_PA ← Physical_Address (DS:RBX);
CR_TCS_LA ← RBX;
CR_TCS_LA.AEP ← RCX;

(* Save the hidden portions of FS and GS *)
CR_SAVE_FS_selector ← FS.selector;
CR_SAVE_FS_base ← FS.base;
CR_SAVE_FS_limit ← FS.limit;
CR_SAVE_FS_access_rights ← FS.access_rights;
CR_SAVE_GS_selector ← GS.selector;
CR_SAVE_GS_base ← GS.base;
CR_SAVE_GS_limit ← GS.limit;
CR_SAVE_GS_access_rights ← GS.access_rights;

RIP ← TMP_TARGET;

Restore_GPRs from DS:TMP_GPR;

(*Restore the RFLAGS values from SSA*)
RFLAGS.CF ← DS:TMP_GPR.RFLAGS.CF;
RFLAGS.PF ← DS:TMP_GPR.RFLAGS.PF;
RFLAGS.AF ← DS:TMP_GPR.RFLAGS.AF;
RFLAGS.ZF ← DS:TMP_GPR.RFLAGS.ZF;
RFLAGS.SF ← DS:TMP_GPR.RFLAGS.SF;
RFLAGS.DF ← DS:TMP_GPR.RFLAGS.DF;
RFLAGS.OF ← DS:TMP_GPR.RFLAGS.OF;
RFLAGS.NT ← DS:TMP_GPR.RFLAGS.NT;
RFLAGS.AC ← DS:TMP_GPR.RFLAGS.AC;
RFLAGS.ID ← DS:TMP_GPR.RFLAGS.ID;
RFLAGS.RF ← DS:TMP_GPR.RFLAGS.RF;
RFLAGS.VM ← 0;
IF (RFLAGS.IOPL = 3)
    THEN RFLAGS.IF ← DS:TMP_GPR.RFLAGS.IF; FI;

IF (TCS.FLAGS.OPTIN = 0)
```

```
    THEN RFLAGS.TF ← 0; FI;

(* If XSAVE is enabled, save XCR0 and replace it with SECS.ATTRIBUTES.XFRM*)
IF (CR4.OSXSAVE = 1)
    CR_SAVE_XCR0 ← XCR0;
    XCR0 ← TMP_SECS.ATTRIBUTES.XFRM;
FI;

(* Pop the SSA stack*)
(DS:RBX).CSSA ← (DS:RBX).CSSA -1;

(* Do the FS/GS swap *)
FS.base ← TMP_FSBASE;
FS.limit ← DS:RBX.FSLIMIT;
FS.type ← 0001b;
FS.W ← DS.W;
FS.S ← 1;
FS.DPL ← DS.DPL;
FS.G ← 1;
FS.B ← 1;
FS.P ← 1;
FS.AVL ← DS.AVL;
FS.L ← DS.L;
FS.unusable ← 0;
FS.selector ← 0BH;

GS.base ← TMP_GSBASE;
GS.limit ← DS:RBX.GSLIMIT;
GS.type ← 0001b;
GS.W ← DS.W;
GS.S ← 1;
GS.DPL ← DS.DPL;
GS.G ← 1;
GS.B ← 1;
GS.P ← 1;
GS.AVL ← DS.AVL;
GS.L ← DS.L;
GS.unusable ← 0;
GS.selector ← 0BH;

CR_DBGOPTIN ← TCS.FLAGS.DBGOPTIN;
Suppress all code breakpoints that are outside ELRANGE;

IF (CR_DBGOPTIN = 0)
    THEN
        Suppress all code breakpoints that overlap with ELRANGE;
        CR_SAVE_TF ← RFLAGS.TF;
        RFLAGS.TF ← 0;
        Suppress any MTF VM exits during execution of the enclave;
        Clear all pending debug exceptions;
        Clear any pending MTF VM exit;
    ELSE
        Clear all pending debug exceptions;
        Clear pending MTF VM exits;
FI;


IF (CPUID.(EAX=12H, ECX=1):EAX[6] = 1)
    THEN
        (* Save enclosing application CET state into save registers *)
        CR_SAVE_IA32_U_CET ← IA32_U_CET
        (* Setup enclave CET state *)
        IF CPUID.(EAX=07H, ECX=00h):ECX[CET_SS] = 1
            THEN
                CR_SAVE_SSP ← SSP
                SSP ← TMP_SSP;
            FI;
        IA32_U_CET ← TMP_IA32_U_CET;
    FI;

(* Assure consistent translations *)
Flush_linear_context;
Clear_Monitor_FSM;
Allow_front_end_to_begin_fetch_at_new_RIP;
```

Flags Affected

RFLAGS.TF is cleared on opt-out entry

Protected Mode Exceptions

#GP(0)	If DS:RBX is not page aligned.
	If the enclave is not initialized.
	If the thread is not in the INACTIVE state.
	If CS, DS, ES or SS bases are not all zero.
	If executed in enclave mode.
	If part or all of the FS or GS segment specified by TCS is outside the DS segment.
	If any reserved field in the TCS FLAG is set.
	If the target address is not within the CS segment.
	If CR4.OSFXSR = 0.
	If CR4.OSXSAVE = 0 and SECS.ATTRIBUTES.XFRM ≠ 3.
	If CR4.OSXSAVE = 1and SECS.ATTRIBUTES.XFRM is not a subset of XCR0.
#PF(error code)	If a page fault occurs in accessing memory.
	If DS:RBX does not point to a valid TCS.
	If one or more pages of the current SSA frame are not readable/writable, or do not resolve to a valid PT_REG EPC page.

64-Bit Mode Exceptions

#GP(0)	If DS:RBX is not page aligned.
	If the enclave is not initialized.
	If the thread is not in the INACTIVE state.
	If CS, DS, ES or SS bases are not all zero.
	If executed in enclave mode.
	If part or all of the FS or GS segment specified by TCS is outside the DS segment.
	If any reserved field in the TCS FLAG is set.
	If the target address is not canonical.
	If CR4.OSFXSR = 0.
	If CR4.OSXSAVE = 0 and SECS.ATTRIBUTES.XFRM ≠ 3.
	If CR4.OSXSAVE = 1and SECS.ATTRIBUTES.XFRM is not a subset of XCR0.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If DS:RBX does not point to a valid TCS.
	If one or more pages of the current SSA frame are not readable/writable, or do not resolve to a valid PT_REG EPC page.

16.5 Intel® SGX VIRTUALIZATION Leaf Function Reference

Leaf functions available with the ENCLV instruction mnemonic are covered in this section. In general, each instruction leaf requires EAX to specify the leaf function index and/or additional implicit registers specifying leaf-specific input parameters. An instruction operand encoding table provides details of each implicit register usage and associated input/output semantics.

In many cases, an input parameter specifies an effective address associated with a memory object inside or outside the EPC, the memory addressing semantics of these memory objects are also summarized in a separate table.

EDECVIRTCHILD—Decrement VIRTCHILDCNT in SECS

Opcode/ Instruction	**Op/En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
EAX = 00H ENCLV[EDECVIRTCHILD]	IR	V/V	EAX[5]	This leaf function decrements the SECS VIRTCHILDCNT field.

Instruction Operand Encoding

Op/En	EAX	RBX	RCX
IR	EDECVIRTCHILD (In)	Address of an enclave page (In)	Address of an SECS page (In)

Description

This instruction decrements the SECS VIRTCHILDCNT field. This instruction can only be executed when current privilege level is 0.

The content of RCX is an effective address of an EPC page. The DS segment is used to create linear address. Segment override is not supported.

EDECVIRTCHILD Memory Parameter Semantics

EPCPAGE	SECS
Read/Write access permitted by Non Enclave	Read access permitted by Enclave

EDECVIRTCHILD Faulting Conditions

The instruction faults if any of the following:

A memory operand effective address is outside the DS segment limit (32b mode).	A page fault occurs in accessing memory operands.
DS segment is unusable (32b mode).	RBX does not refer to an enclave page (REG, TCS, TRIM, SECS).
A memory address is in a non-canonical form (64b mode).	RCX does not refer to an SECS page.
A memory operand is not properly aligned.	RBX does not refer to an enclave page associated with SECS referenced in RCX.

Concurrency Restrictions

Base Concurrency Restrictions of EDECVIRTCHILD

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EDECVIRTCHILD	Target [DS:RBX]	Shared	SGX_EPC_PAGE_ CONFLICT	
	SECS [DS:RCX]	Concurrent		

Additional Concurrency Restrictions of EDECVIRTCHILD

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EDECVIRTCHILD	Target [DS:RBX]	Concurrent		Concurrent		Concurrent	
	SECS [DS:RCX]	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in EDECVIRTCHILD Operational Flow

Name	Type	Size (bits)	Description
TMP_SECS	Physical Address	64	Physical address of the SECS of the page being modified.
TMP_VIRTCHILDCNT	Integer	64	Number of virtual child pages.

EDECVIRTCHILD Return Value in RAX

Error	Value	Description
No Error	0	EDECVIRTCHILD Successful.
SGX_EPC_PAGE_CONFLICT		Failure due to concurrent operation of another SGX instruction.
SGX_INVALID_COUNTER		Attempt to decrement counter that is already zero.

```
(* check alignment of DS:RBX *)
IF (DS:RBX is not 4K aligned) THEN
    #GP(0); FI;

(* check DS:RBX is an linear address of an EPC page *)
IF (DS:RBX does not resolve within an EPC) THEN
    #PF(DS:RBX, PFEC.SGX); FI;

(* check DS:RCX is an linear address of an EPC page *)
IF (DS:RCX does not resolve within an EPC) THEN
    #PF(DS:RCX, PFEC.SGX); FI;

(* Check the EPCPAGE for concurrency *)
IF (EPCPAGE is being modified) THEN
    RFLAGS.ZF = 1;
    RAX = SGX_EPC_PAGE_CONFLICT;
    goto DONE;
FI;

(* check that the EPC page is valid *)
IF (EPCM(DS:RBX).VALID = 0) THEN
    #PF(DS:RBX, PFEC.SGX); FI;

(* check that the EPC page has the correct type and that the back pointer matches the pointer passed as the pointer to parent *)
```

```
IF ((EPCM(DS:RBX).PAGE_TYPE = PT_REG) or
    (EPCM(DS:RBX).PAGE_TYPE = PT_TCS) or
    (EPCM(DS:RBX).PAGE_TYPE = PT_TRIM) or
    (EPCM(DS:RBX).PAGE_TYPE = PT_SS_FIRST) or
    (EPCM(DS:RBX).PAGE_TYPE = PT_SS_REST) )
   THEN
    (* get the SECS of DS:RBX *)
    TMP_SECS ← Address of SECS for (DS:RBX);
ELSE IF (EPCM(DS:RBX).PAGE_TYPE = PT_SECS) THEN
    (* get the physical address of DS:RBX *)
    TMP_SECS ← Physical_Address(DS:RBX);
ELSE
    (* EDECVIRTCHILD called on page of incorrect type *)
    #PF(DS:RBX, PFEC.SGX); FI;

IF (TMP_SECS ≠ Physical_Address(DS:RCX)) THEN
    #GP(0); FI;

(* Atomically decrement virtchild counter and check for underflow *)
Locked_Decrement(SECS(TMP_SECS).VIRTCHILDCNT);
IF (There was an underflow) THEN
    Locked_Increment(SECS(TMP_SECS).VIRTCHILDCNT);
    RFLAGS.ZF ← 1;
    RAX ← SGX_INVALID_COUNTER;
    goto DONE;
FI;

RFLAGS.ZF ← 0;
RAX ← 0;

DONE:
(* clear flags *)
RFLAGS.CF ← 0;
RFLAGS.PF ← 0;
RFLAGS.AF ← 0;
RFLAGS.OF ← 0;
RFLAGS.SF ← 0;
```

Flags Affected

ZF is set if EDECVIRTCHILD fails due to concurrent operation with another SGX instruction, or if there is a VIRTCHILDCNT underflow. Otherwise cleared.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the DS segment limit.
	If DS segment is unusable.
	If a memory operand is not properly aligned.
	RBX does not refer to an enclave page associated with SECS referenced in RCX.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If RBX does not refer to an enclave page (REG, TCS, TRIM, SECS).
	If RCX does not refer to an SECS page.

64-Bit Mode Exceptions

#GP(0)	If a memory address is in a non-canonical form.
	If a memory operand is not properly aligned.
	RBX does not refer to an enclave page associated with SECS referenced in RCX.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If RBX does not refer to an enclave page (REG, TCS, TRIM, SECS).
	If RCX does not refer to an SECS page.

EINCVIRTCHILD—Increment VIRTCHILDCNT in SECS

Opcode/ Instruction	**Op/En**	**64/32 bit Mode Support**	**CPUID Feature Flag**	**Description**
EAX = 01H ENCLV[EINCVIRTCHILD]	IR	V/V	EAX[5]	This leaf function increments the SECS VIRTCHILDCNT field.

Instruction Operand Encoding

Op/En	EAX	RBX	RCX
IR	EINCVIRTCHILD (In)	Address of an enclave page (In)	Address of an SECS page (In)

Description

This instruction increments the SECS VIRTCHILDCNT field. This instruction can only be executed when the current privilege level is 0.

The content of RCX is an effective address of an EPC page. The DS segment is used to create a linear address. Segment override is not supported.

EINCVIRTCHILD Memory Parameter Semantics

EPCPAGE	SECS
Read/Write access permitted by Non Enclave	Read access permitted by Enclave

EINCVIRTCHILD Faulting Conditions

The instruction faults if any of the following:

A memory operand effective address is outside the DS segment limit (32b mode).	A page fault occurs in accessing memory operands.
DS segment is unusable (32b mode).	RBX does not refer to an enclave page (REG, TCS, TRIM, SECS).
A memory address is in a non-canonical form (64b mode).	RCX does not refer to an SECS page.
A memory operand is not properly aligned.	RBX does not refer to an enclave page associated with SECS referenced in RCX.

Concurrency Restrictions

Base Concurrency Restrictions of EINCVIRTCHILD

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
EINCVIRTCHILD	Target [DS:RBX]	Shared	SGX_EPC_PAGE_ CONFLICT	
	SECS [DS:RCX]	Concurrent		

Additional Concurrency Restrictions of EINCVIRTCHILD

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
EINCVIRTCHILD	Target [DS:RBX]	Concurrent		Concurrent		Concurrent	
	SECS [DS:RCX]	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in EINCVIRTCHILD Operational Flow

Name	Type	Size (bits)	Description
TMP_SECS	Physical Address	64	Physical address of the SECS of the page being modified.

EINCVIRTCHILD Return Value in RAX

Error	Value	Description
No Error	0	EINCVIRTCHILD Successful.
SGX_EPC_PAGE_CONFLICT		Failure due to concurrent operation of another SGX instruction.

```
(* check alignment of DS:RBX *)
IF (DS:RBX is not 4K aligned) THEN
    #GP(0); FI;

(* check DS:RBX is an linear address of an EPC page *)
IF (DS:RBX does not resolve within an EPC) THEN
    #PF(DS:RBX, PFEC.SGX); FI;

(* check DS:RCX is an linear address of an EPC page *)
IF (DS:RCX does not resolve within an EPC) THEN
    #PF(DS:RCX, PFEC.SGX); FI;

(* Check the EPCPAGE for concurrency *)
IF (EPCPAGE is being modified) THEN
    RFLAGS.ZF = 1;
    RAX = SGX_EPC_PAGE_CONFLICT;
    goto DONE;
FI;

(* check that the EPC page is valid *)
IF (EPCM(DS:RBX).VALID = 0) THEN
    #PF(DS:RBX, PFEC.SGX); FI;

(* check that the EPC page has the correct type and that the back pointer matches the pointer passed as the pointer to parent *)
IF ((EPCM(DS:RBX).PAGE_TYPE = PT_REG) or
    (EPCM(DS:RBX).PAGE_TYPE = PT_TCS) or
    (EPCM(DS:RBX).PAGE_TYPE = PT_TRIM) or
    (EPCM(DS:RBX).PAGE_TYPE = PT_SS_FIRST) or
    (EPCM(DS:RBX).PAGE_TYPE = PT_SS_REST) )
   THEN
    (* get the SECS of DS:RBX *)
    TMP_SECS ← Address of SECS for (DS:RBX);
ELSE IF (EPCM(DS:RBX).PAGE_TYPE = PT_SECS) THEN
```

```
    (* get the physical address of DS:RBX *)
    TMP_SECS ← Physical_Address(DS:RBX);
ELSE
    (* EINCVIRTCHILD called on page of incorrect type *)
    #PF(DS:RBX, PFEC.SGX); FI;

IF (TMP_SECS ≠ Physical_Address(DS:RCX)) THEN
    #GP(0); FI;

(* Atomically increment virtchild counter *)
Locked_Increment(SECS(TMP_SECS).VIRTCHILDCNT);

RFLAGS.ZF ← 0;
RAX ← 0;

DONE:
(* clear flags *)
RFLAGS.CF ← 0;
RFLAGS.PF ← 0;
RFLAGS.AF ← 0;
RFLAGS.OF ← 0;
RFLAGS.SF ← 0;
```

Flags Affected

ZF is set if EINCVIRTCHILD fails due to concurrent operation with another SGX instruction; otherwise cleared.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the DS segment limit.
	If DS segment is unusable.
	If a memory operand is not properly aligned.
	RBX does not refer to an enclave page associated with SECS referenced in RCX.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If RBX does not refer to an enclave page (REG, TCS, TRIM, SECS).
	If RCX does not refer to an SECS page.

64-Bit Mode Exceptions

#GP(0)	If a memory address is in a non-canonical form.
	If a memory operand is not properly aligned.
	RBX does not refer to an enclave page associated with SECS referenced in RCX.
#PF(error code)	If a page fault occurs in accessing memory operands.
	If RBX does not refer to an enclave page (REG, TCS, TRIM, SECS).
	If RCX does not refer to an SECS page.

ESETCONTEXT— Set the ENCLAVECONTEXT Field in SECS

Opcode/ Instruction	Op/En	64/32 bit Mode Support	CPUID Feature Flag	Description
EAX = 02H ENCLV[ESETCONTEXT]	IR	V/V	EAX[5]	This leaf function sets the ENCLAVECONTEXT field in SECS.

Instruction Operand Encoding

Op/En	EAX	RCX	RDX
IR	ESETCONTEXT (In)	Address of the destination EPC page (In, EA)	Context Value (In, EA)

Description

The ESETCONTEXT leaf overwrites the ENCLAVECONTEXT field in the SECS. ECREATE and ELD of an SECS set the ENCLAVECONTEXT field in the SECS to the address of the SECS (for access later in ERDINFO). The ESETCONTEXT instruction allows a VMM to overwrite the default context value if necessary, for example, if the VMM is emulating ECREATE or ELD on behalf of the guest.

The content of RCX is an effective address of the SECS page to be updated, RDX contains the address pointing to the value to be stored in the SECS. The DS segment is used to create linear address. Segment override is not supported.

The instruction fails if:

- The operand is not properly aligned.
- RCX does not refer to an SECS page.

ESETCONTEXT Memory Parameter Semantics

EPCPAGE	CONTEXT
Read access permitted by Enclave	Read/Write access permitted by Non Enclave

ESETCONTEXT Faulting Conditions

The instruction faults if any of the following:

A memory operand effective address is outside the DS segment limit (32b mode).	A memory operand is not properly aligned.
DS segment is unusable (32b mode).	A page fault occurs in accessing memory operands.
A memory address is in a non-canonical form (64b mode).	

Concurrency Restrictions

Base Concurrency Restrictions of ESETCONTEXT

Leaf	Parameter	Base Concurrency Restrictions		
		Access	On Conflict	SGX_CONFLICT VM Exit Qualification
ESETCONTEXT	SECS [DS:RCX]	Shared	SGX_EPC_PAGE_ CONFLICT	

Additional Concurrency Restrictions of ESETCONTEXT

Leaf	Parameter	Additional Concurrency Restrictions					
		vs. EACCEPT, EACCEPTCOPY, EMODPE, EMODPR, EMODT		vs. EADD, EEXTEND, EINIT		vs. ETRACK, ETRACKC	
		Access	On Conflict	Access	On Conflict	Access	On Conflict
ESETCONTEXT	SECS [DS:RCX]	Concurrent		Concurrent		Concurrent	

Operation

Temp Variables in ESETCONTEXT Operational Flow

Name	Type	Size (bits)	Description
TMP_SECS	Physical Address	64	Physical address of the SECS of the page being modified.
TMP_CONTEXT	CONTEXT	64	Data Value of CONTEXT.

ESETCONTEXT Return Value in RAX

Error	Value	Description
No Error	0	ESETCONTEXT Successful.
SGX_EPC_PAGE_CONFLICT		Failure due to concurrent operation of another SGX instruction.

```
(* check alignment of the EPCPAGE (RCX) *)
IF (DS:RCX is not 4KByte Aligned) THEN
    #GP(0); FI;

 (* check that EPCPAGE (DS:RCX) is the address of an EPC page *)
IF (DS:RCX does not resolve within an EPC)THEN
    #PF(DS:RCX, PFEC.SGX); FI;

(* check alignment of the CONTEXT field (RDX) *)
IF (DS:RDX is not 8Byte Aligned) THEN
    #GP(0); FI;

 (* Load CONTEXT into local variable *)
TMP_CONTEXT ← DS:RDX

(* Check the EPC page for concurrency *)
IF (EPC page is being modified) THEN
    RFLAGS.ZF ← 1;
    RFLAGS.CF ← 0;
    RAX ← SGX_EPC_PAGE_CONFLICT;
    goto DONE;
FI;

(* check page validity *)
IF (EPCM(DS:RCX).VALID = 0) THEN
    #PF(DS:RCX, PFEC.SGX);
FI;

(* check EPC page is an SECS page *)
IF (EPCM(DS:RCX).PT is not PT_SECS) THEN
    #PF(DS:RCX, PFEC.SGX);
FI;
```

```
(* load the context value into SECS(DS:RCX).ENCLAVECONTEXT *)
SECS(DS:RCX).ENCLAVECONTEXT ← TMP_CONTEXT;

RAX ← 0;
RFLAGS.ZF ← 0;

DONE:
(* clear flags *)
RFLAGS.CF,PF,AF,OF,SF ← 0;
```

Flags Affected

ZF is set if ESETCONTEXT fails due to concurrent operation with another SGX instruction; otherwise cleared.

CF, PF, AF, OF and SF are cleared.

Protected Mode Exceptions

#GP(0)	If a memory operand effective address is outside the DS segment limit.
	If DS segment is unusable.
	If a memory operand is not properly aligned.
#PF(error code)	If a page fault occurs in accessing memory operands.

64-Bit Mode Exceptions

#GP(0)	If a memory address is in a non-canonical form.
	If a memory operand is not properly aligned.
#PF(error code)	If a page fault occurs in accessing memory operands.