Control-flow Enforcement Technology Preview

June 2016

Revision 1.0

Document Number: 334525-001

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.

The products and services described may contain defects or errors known as errata which may cause deviations from published specifications. Current characterized errata are available on request.

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration. No computer system can be absolutely secure. Check with your system manufacturer or retailer or learn more at http://intel.com/.

Copies of documents which have an order number and are referenced in this document may be obtained by calling 1-800-548-4725 or by visiting www.intel.com/design/literature.htm.

Contents

Revision History

Document Number	Revision Number	Description	Date
334525-001	1.0	• Initial release of the document.	June 2016

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 13 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 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. See Figure 1 for an illustration of shadow stack operations on near call and ret 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. are allowed to store return addresses to the shadow stack. 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 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 7 for details) is a new instruction that is used to mark valid indirect call/jmp targets 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 in-order part of 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 fault else the state machine moves back to IDLE state.

2 Shadow Stacks

A shadow stack is a second 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. Shadow stack feature can be enabled for 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 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 SSP holds a linear address and is loaded into the register by FAR RET, IRET and RSTORSSP instructions. The SSP must be loaded with a 32-bit aligned linear address.

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, 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 load/store performed by other instructions like MOV, XSAVES, etc.

The pseudo-code for the instruction operations use a 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 MSR used to enable feature for CPL = 3 *)
                SHADOW_STACK_ENABLED = IA32_U_CET.SH_STK_EN;
            ELSE
                (* Obtain the shadow stack enable from MSR 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
```

2.3 Near CALL and RET Behavior with Shadow Stacks Enabled

Near CALL, when shadow stack is enabled 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 address popped from the two stacks are not same then the processor causes a #CP(near-ret) exception.

2.4 Far CALL and RET

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, 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.

The shadow stack behavior on a far CALL to higher privilege level 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
    - o IA32_PL2_SSP is transitioning to ring 2
    - o IA32_PL1_SSP is transitioning to ring 1
    - o IA32_PL0_SSP is 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.

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 – 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.

The following figure illustrates a supervisor shadow stack with a “supervisor shadow stack token” located at its base:

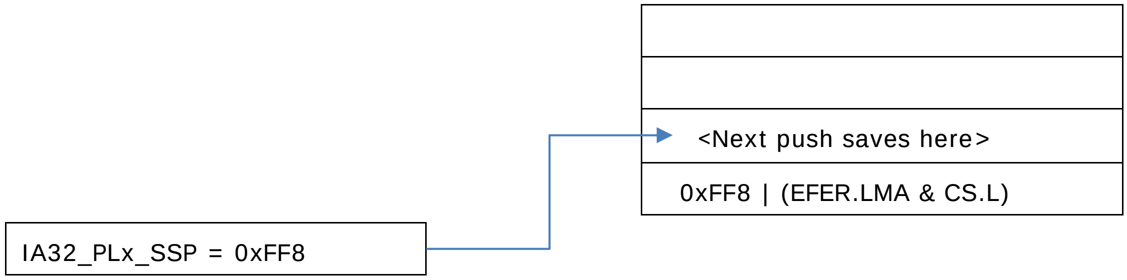


Figure 1 Supervisor Shadow Stack with a Kernel Shadow Stack Token

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

1. Load the supervisor shadow stack token from the address specified in IA32_PLx_SSP using a shadow_stack_load.
2. The busy bit in the token must be 0.
3. The address programmed in the MSR must match the address in the “supervisor shadow stack token”.
4. If the 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.
5. If the 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:

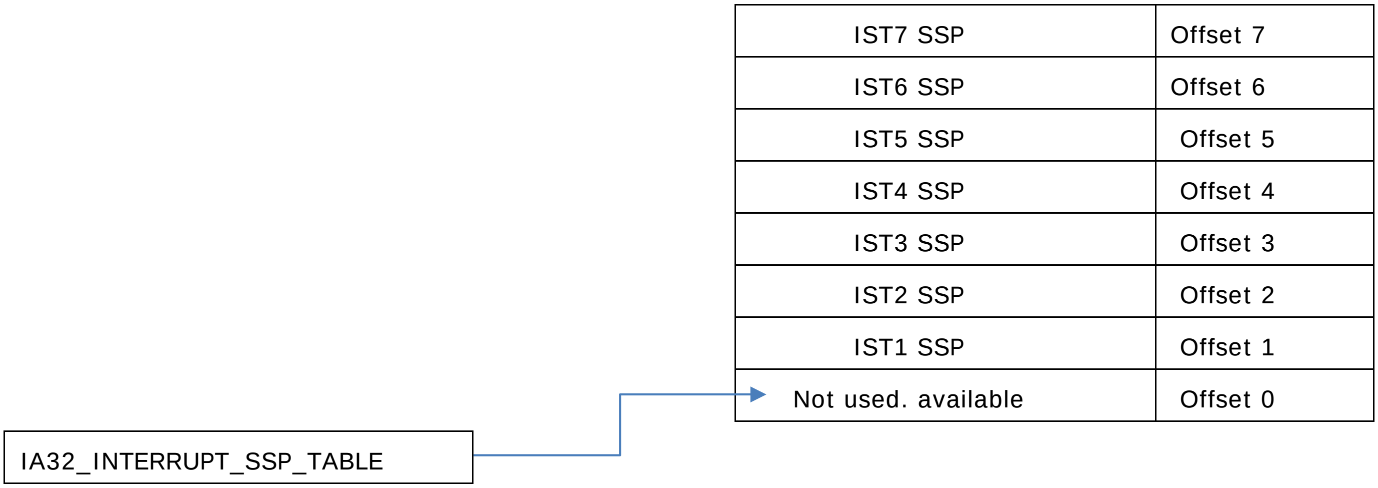
1. Load the kernel shadow stack token from the SSP using a shadow_stack_load.
2. Check if the busy bit is 1.
3. Check if the address programmed “supervisor shadow stack token” matches SSP.
4. If the checks 2 and 3 are successful then clear the busy bit in the token using a shadow_stack_store else continue 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 7 data stack pointers in the 64-bit TSS when there is no privilege change involved as part of the call. 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, shadow stack feature provides an MSR, IA32_INTERRUPT_SSP_TABLE, to program the linear address of a table of 7 shadow stack pointers. When a non-zero IST value is specified and there is no privilege change involved as part of the call, the MSR points to a 64 byte table in memory that is indexed using the IST index.



IST7 SSP	Offset 7
IST6 SSP	Offset 6
IST5 SSP	Offset 5
IST4 SSP	Offset 4
IST3 SSP	Offset 3
IST2 SSP	Offset 2
IST1 SSP	Offset 1
Not used. available	Offset 0

Figure 2 Interrupt Shadow Stack Table

2.6 Shadow Stack Usage on Task Switch

A task switch 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 nested task switch initiated by a CALL instruction, the SSP of the old task is not saved to the old task TSS. Instead 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. Likewise on a non-nested 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.

3 Indirect Branch Tracking

When the indirect branch tracking feature is active the indirect jmp/call instruction behavior changes as below:

- 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 #CP fault. 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, r/m32, r/m64
    - jmp m16:16, m16:32, 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 #CP fault. Below call instructions are tracked to enforce an endbranch. Note that relative and zero displacement form 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, r/m32, r/m64
    - call m16:16, m16:32, 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 still do not change register state (NOP-like). 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 perspective.

The processor implements a two-state state machine 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.

When instructions other than indirect call/jmp retire the state machine stays in the IDLE state.

On an indirect call or jmp instruction retirement, the state machine transitions to WAIT_FOR_ENDBRANCH state.

In the WAIT_FOR_ENDBRANCH state, the indirect branch tracking stat machine will:

- Only allow the next instruction retirement to be an ENDBRANCH instruction (i.e. ENDBR64 if EFER.LMA=1 & CS.L=1, ENDBR32 if EFER.LMA=0 || (EFER.LMA=1 & CS.L=0)), or
- Allow next instruction retirement to be compatible with non-CET code generation if legacy compatibility configuration allows (see 3.6).

When a #CP(ENDBRANCH) exception is thrown, the priority of the exception is as follows:

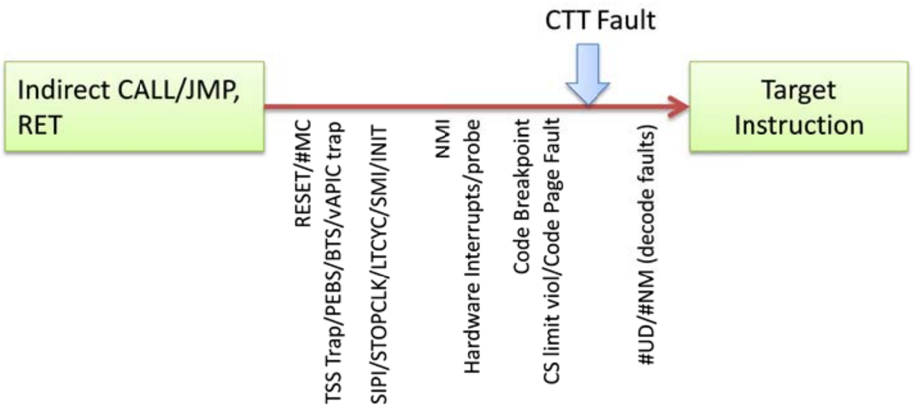

CTT Fault
Indirect CALL/JMP, RET
Target Instruction
RESET/#MC
TSS Trap/PEBS/BTS/vAPIC trap
SIPI/STOPCLK/LTCYC/SMI/INIT
NMI
Hardware Interrupts/probe
Code Breakpoint
CS limit viol/Code Page Fault
#UD/#NM (decode faults)

Figure 3 Priority of Control Protection Exception on Missing Endbranch

Thus higher priority faults/traps/events that occur at the end of an indirect call/jmp or ret 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 CET 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

Near indirect call and jmp instructions of "reg" form when prefixed with 3EH are termed non-tracked indirect control transfer instructions and do not modify the CET indirect branch tracker. Far call and jmp are always tracked and ignore the 3EH prefix. Near indirect call and jmp of the "mem" form are always tracked and ignore the 3E prefix.

3.2 Terminology

The pseudo-code for the instruction operations use a notation EndbranchEnabled(CPL) as a test of whether endbranch 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
    (* Endbranch 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
    (* Endbranch 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. Thus when a user space program is executing the CPL 3 CET indirect branch tracker is active and when supervisor mode software is executing the CPL < 3 tracker is active. This sections 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 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 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. Thus if the user mode was interrupted by a higher priority event, like an interrupt at the end of the indirect call/jmp or ret, 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 enabled, active indirect branch tracker is unsuppressed and goes to WAIT_FOR_ENDBRANCH.
    - Near indirect call/jmp: if indirect branch tracking 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 enabled, the active indirect branch tracker is unsuppressed and goes to WAIT_FOR_ENDBRANCH.
- Case 3: iret
    - If indirect branch tracking enabled, the active indirect branch tracker keeps its state.

3.4 ENDBRANCH 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 of “reg” form	TRACKER=IDLE, SUPPRESS=0, ENDBR_EN=1
	Indirect call/jmp of “mem” form Far call/jmp	TRACKER=WAIT_FOR_ENDBRANCH, SUPPRESS=0, ENDBR_EN=1

TRACKER= WAIT_FOR_ENDBRANCH, SUPPRESS=0, ENDBR_EN=1	INT3	TRACKER= WAIT_FOR_ENDBRANCH, SUPPRESS=0, ENDBR_EN=1
	Endbranch instruction	TRACKER=IDLE, SUPPRESS=0, ENDBR_EN=1
	ENCLU[ERESUME]	TRACKER=IDLE, SUPPRESS=0, ENDBR_EN=1
	Instructions other than endbranch, ENCLU[ERESUME] or int3	If legacy interwork not enabled or if not allowed by legacy code page bit-map: • No state change and deliver #CP (ENDBRANCH) If legacy interwork enabled and transfer allowed by legacy code page bitmap: • TRACKER=IDLE, SUPRESS=1, 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	TRACKER=WAIT_FOR_ENDBRANCH, SUPPRESS=0, ENDBR_EN=1
	Endbranch instruction 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/RSM)	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.

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 to 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 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_linaddr. Osize8(BITMAP_BASE + LA[31:15]]
    ELSE
      BITMAP_BYTE = load.Asize_linaddr. 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 = 1
    ELSE
        IA32_S_CET.TRACKER = IDLE
        IA32_S_CET.SUPPRESS = 1
    ENDIF
    Restart the instruction (handle all arch. consistency around MOV SS state machines, STI etc.)
    without opening up interrupt/trap window (like other inserted flows)
ELSE
    Deliver #CP(ENDBRANCH) Fault
ENDIF
```

Faults/traps in pseudo-code 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).

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

Once CET has been suppressed, subsequent execution of endbranch instructions will do the following – see section 7 for details:

```
IF EndbranchEnabled(CPL)
    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

3.7 Other Considerations

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

There are no direct architectural interactions between CET and TSX. 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.

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. Thus #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.

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)
                          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)
                          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)
                          (* 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 of “reg” form AND no 3EH prefix) OR
       (Call near indirect, absolute indirect of “mem” form)
        IF EndbranchEnabledAndNotSuppressed(CPL)
            IF CPL = 3
                THEN
                    IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH
                ELSE
                    IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
                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 + IP;
                    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;
        SSP = SSP & 0xFFFFFFFFFFFFFFF8H
        ShadowStackPush8B(tempPushCS);          (* Padded with 48 high-order bits of 0 *)
        ShadowStackPush8B(tempPushLIP);        (* EIP padded with 32 high-order bits of 0 *)
        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 + IP;
                    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;
    SSP = SSP & 0xFFFFFFFFFFFFFFF8H
    ShadowStackPush8B(tempPushCS); (* Padded with 48 high-order 0 bits *)
    ShadowStackPush8B(tempPushLIP);
    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 *)
                ShadowStackPush8B(oldCSBASE+oldRIP); (* Padded with 32 high-order bits *)
                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;
         SSP = SSP & 0xFFFFFFFFFFFFFFF8H;
         (* push cs:lip:ssp on shadow stack *)
         ShadowStackPush8B(oldCS); (* Padded with 48 high-order bits *)
         ShadowStackPush8B(oldCSBASE + oldRIP);
         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 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);
          ShadowStackPush8B(oldCSBASE + oldRIP);
          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);
             ShadowStackPush8B(oldCSBASE + oldRIP);
             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;
(* 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
                     IF SSP & 0x07 != 0
                         THEN #GP(0); FI;
                    Fault = 0
                     Atomic Start
                         SSPToken = 8 bytes 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 with shadow stack semantics to SSP;
                      Atomic End
                      If fault = 1
                          THEN #GP(0); FI;
            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)
    (* Align to next 8 byte boundary *)
    tempSSP = SSP;
    SSP = newSSP & 0xFFFFFFFFFFFFFFF8H;
    (* push cs:lip:ssp on shadow stack *)
    ShadowStackPush8B(oldCS);
    ShadowStackPush8B(oldCSBASE + oldRIP);
    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 a 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 Lbit 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 of “reg” form AND no 3EH prefix) OR
           (JMP near indirect, absolute indirect of “mem” form)
            IF EndbranchEnabledAndNotSuppressed(CPL)
                   IF CPL = 3
                          THEN
                                IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH
                          ELSE
                                IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
                   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;
        prevSSP = PopShadowStack8B();
        tempSsLIP = PopShadowStack8B();
        tempSsCS = PopShadowStack8B();
        (* do a 64 bit-compare to check if any bits beyond bit 15 are set *)
        tempCS = CS; (* zero extended to 64 bit *)
        IF tempCS != tempSsCS
            THEN #CP(FAR-RET/IRET); FI;
        (* do a 64 bit-compare *)
        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
              tempSSP = PopShadowStack8B();
              tempSsEIP = PopShadowStack8B();
              tempSsCS = PopShadowStack8B();
              (* 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 *)
              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 <- 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;
    tempSSP = PopShadowStack8B();
    tempSsLIP = PopShadowStack8B();
    tempSsCS = PopShadowStack8B();
    tempCS = CS; (* zero extended 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
            tempSSP = PopShadowStack8B();
            tempSsLIP = PopShadowStack8B();
            tempSsCS = PopShadowStack8B();
            (* 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 *)
                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.

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 <-IA32_PL0_SSP;
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-Bi t 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.

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 <-IA32_PL0_SSP;
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.

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	Instructio n	Op/ En	64-Bi t 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*).
- 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:

[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.

— 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.

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	Instructio n	Op/ En	64-Bi t 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 unblocks 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 EIP not within CS limit
        THEN #GP(0); FI;
    (* If shadow stack or endbranch enabled at CPL3 then #GP(0) *)
```

```
     IF ShadowStackEnabled(CPL3) OR EndbranchEnabled (CPL3)
         THEN #GP(0); FI;
    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 *)
    CPL <- 3;
    (* Resume execution in Virtual-8086 mode *)
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:
    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
        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 *)
             ESP <- tempESP;
             SS <- tempSS;
         ELSE
             IF OperandSize = 16
                 THEN
```

```
                    tempESP <- Pop();
                    tempSS <- Pop(); (* 16-bit pop; segment descriptor loaded *)
                    ESP <- tempESP;
                    SS <- tempSS;
                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
            tempSSP = PopShadowStack8B();
            tempSsLIP = PopShadowStack8B();
            tempSsCS = PopShadowStack8B();
            (* 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 *)
            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 *)
RSP <- tempRSP;
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 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;
          tempSSP = PopShadowStack8B();
          tempSsLIP = PopShadowStack8B();
          tempSsCS = PopShadowStack8B();
          tempCS = CS; (* zero extended 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; EIP zero padded to 64 bit *)
             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;
         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 are 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 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. 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;
tempPushCsEIPSsp = 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
                        tempPushCsLIPSsp = 1 (* no privilege change; supv->supv; supv->user *)
                        tempSSP = SSP
                        tempSsLIP =CSBASE + EIP
                        tempSsCS = CS
        FI;
    FI
FI
tempVerifyCsEIP = 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 *)
                IF SSP not aligned to 8B then #GP(0)
                tempSSP = ShadowStackPop8B()
                tempSsLIP = ShadowStackPop8B()
                tempSsCS = ShadowStackPop8B()
                tempVerifyCsEIP = 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
    FI
FI
```

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

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.). If shadow stacks are enabled at the CPL of the new task then the processor performs following checks:

```
IF shadow stack enabled at current CPL
    If task switch initiated by CALL instruction, JMP instruction, interrupt or exception (* switch stack *)
          Load the 4 byte SSP from offset 104 in the TSS
        // Verify new SSP to be legal
        IF SSP & 0x07 != 0
            THEN #TSS(New-Task-TSS); FI;
         FI;
    FI;

FI;
IF shadow stack enabled at current CPL OR endbranch enabled at current CPL
    THEN
        IF EFLAGS.VM = 1
            THEN #TSS(new-Task-TSS);FI;
FI;
```

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
    IF task switch initiated by CALL instruction, JMP instruction, interrupt or exception (* switch stack *)
        Fault = 0
        Atomic Start
            SSPToken = 8 bytes 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 with shadow stack semantics to SSP;
        Atomic End
        IF fault = 1
            THEN GP(0); FI;
        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
        If tempSsCS and tempSsLIP do not match CS and CSBASE+EIP
            THEN #CP(FAR-RET/IRET); FI;
     FI
     IF ShadowStackEnabled(CPL)
        THEN
            IF (CPL of current Task = 3) 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;
```

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.

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

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.

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.

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	New Task's 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 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 – SSP – by 4 bytes in 32-bit/compatibility mode and by 4 or 8 bytes in 64-bit mode.
- **RDSSP** – instruction used to read the contents of the SSP register into a GPR.
- **SAVESSP** – this instruction aligns the SSP to the next 8 byte boundary and pushes an 8 byte “stack context token” on the shadow stack. The stack context token contains the SSP at the time of invoking the SAVESSP instruction along with the mode of the machine. The format of this token is as follows:
  - Bit 63:3 – SSP at the time of invoking this instruction
  - Bit 2:1 – reserved. Must be zero
  - Bit 0 – L flag; if 1 indicates this shadow stack is for use in 64-bit mode
- **RSTORSSP** – this instruction is used to restore a shadow stack context previously saved on the shadow stack. This instruction loads the shadow stack context token from the memory operand specified in the instruction is pointing to a valid shadow stack context token. This instruction clears the shadow stack context token and establishes the SSP recorded in the shadow stack context token as the new SSP.
- **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 takes a memory operand to a kernel shadow stack token and if the token is a valid token sets it busy.
- **CLRSSBSY** – This instruction takes a memory operand to a kernel 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-Bi t Mode	Compat/ Leg Mode	Description
F3 0F 01 E9	INCSSPD	NP	Valid	Valid	Increment SSP by 4
F3 REX.W 0F 01 E9	INCSSPQ	NP	Valid	N.E.	Increment SSP by 8

Instruction Operand Encoding

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

Description

This instruction can be used to increment the current shadow stack pointer by operand size of the instruction. The instruction performs an operand-sized load on the shadow stack prior to incrementing the shadow stack pointer.

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
        TMP = ShadowStackLoad8B(SSP)
       SSP <- SSP + 8;
   ELSE
        TMP = ShadowStackLoad4B(SSP)
       SSP <- SSP + 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

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

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

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 SAVESSP —Save Shadow Stack Pointer

Opcode	Instruction	Op/ En	64-Bi t Mode	Compat/ Leg Mode	Description
F3 0F 01 EA (mod=11, /5, RM=010)	SAVESSP	NP	Valid	Valid	Save 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 current SSP and state of EFER.LMA & CS.L to the current shadow stack after aligning to next 8 byte boundary.

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

temp <- (SSP | (EFER.LMA & CS.L));
SSP <- SSP & ~0x07;
ShadowStackPush8B(temp);
```

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 SAVESSP instruction is not recognized in virtual-8086 mode.

Virtual-8086 Mode Exceptions

#UD The SAVESSP 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.

6.4 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 token previously created on shadow stack by SAVESSP.

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;
Atomic Start
SSP_Tmp = Locked shadow_Stack_Load with store intent 8 bytes from SSP_LA
Fault = 0
IF ((SSP_Tmp & 0x01) != (EFER.LMA & CS.L))
    THEN fault = 1; FI; // If L flag in token does not match EFER.LMA & CS.L
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) ? 0 : 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_Tmp & ~0x01;
```

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 linear address of memory operand not 8 byte aligned.
If the DS, ES, FS, or GS register is used to access memory and it contains a NULL segment selector.

#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.

#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.

#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.

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)
(* Destination not 4B aligned *)
IF DEST_LA[1:0]
    THEN GP(0); FI;
IF (operand size is 64 bit)
    THEN
        Shadow_stack_store 8 bytes of SRC to DEST_LA;
    ELSE
        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.

#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 though normally a supervisor-mode shadow stack store to user-mode shadow stack pages is disallowed. This instruction also disallows shadow stack stores to supervisor-mode shadow stack pages.

Operation

```
IF CR4.CET = 0
    THEN #UD; FI;
IF CPL > 0
    THEN #GP(0); FI;
DEST_LA = Linear_Address(mem operand)
(* Destination not 4B aligned *)
IF DEST_LA[1:0]
    THEN GP(0); FI;
Setup mode to allow CPL <3 write to user shadow stack pages and disallow to supervisor pages
IF (operand size is 64 bit)
    THEN
        Shadow_stack_store 8 bytes of SRC to DEST_LA;
    ELSE
        Shadow_stack_store 4 bytes of SRC[31:0] to DEST_LA;
FI;
Clear mode to allow CPL <3 write to user shadow stack pages and disallow to supervisor pages
```

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.

#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.

#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 AE /5	SETSSBSY	M64	Valid	Valid	Mark shadow stack pointed by m64 as 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 as busy

Operation

```
IF CPL = 0
        IF (CR4.CET & IA32_S_CET.SH_STK_EN) = 0
                THEN #UD; FI;
ENDIF
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;
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 #GP(0); FI;    (* If invalid token then fault *)
```

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 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 linear address of destination points to a busy token.

If the token pointed by the destination has SSP address beyond 4G in 32 bit and compatibility mode.

If the address in the token does not match the memory operand.

If CPL is not 0.

#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 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.

#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 memory operand linear address not aligned to 8 bytes.

If the memory address is in a non-canonical form.

If linear address of destination points to a busy token.

If CPL is not 0.

If the token pointed by the destination has SSP address beyond 4G in 32 bit and compatibility mode.

If the address in the token does not match the memory operand.

#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 as not busy.

Operation

```
IF CPL = 0
    IF (CR4.CET & IA32_S_CET.SH_STK_EN) = 0
        THEN #UD; FI;
ENDIF
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
IF invalid_token == 1
    THEN #GP(0); 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.

#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.
	If the token is invalid.
#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 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.
#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 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.
#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-Bi t 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 Shadow Stacks Exception, Enumeration, Enabling, Context Switches

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.

Exception Error Code

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

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

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 Enumeration

CET Feature flag – if CPUID.(EAX=7, ECX=0):ECX.CET[bit 7(tentatively)] is 1, the processor supports CET, including the MSR described in Section 9.5.

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:
  - o EBX
    - o If CET_U bit (tentatively bit 11) set in IA32_XSS then reports additional 16 bytes to save CET user state.
    - o If CET_S bit (tentatively bit 12) set in IA32_XSS then reports additional 24 bytes to save CET supervisor state.
  - o ECX
    - IA32_XSS[CET_U] bit (tentatively bit 11) is supported if 1.
    - IA32_XSS[CET_S] bit (tentatively bit 12) is supported if 1.

- CET user mode state XSAVES sub-leaf (tentatively sub-leaf 11).
  Processor Extended State Enumeration Sub-leaf (EAX = 0DH, ECX = 11) returns:
  - EAX – 16 bytes
  - EBX – 0
  - ECX – 1 (supervisory state)
  - EDX – 0
- CET supervisor mode state XSAVES sub-leaf (tentatively sub-leaf 12).
  Processor Extended State Enumeration Sub-leaf (EAX = 0DH, ECX = 12) returns:
  - EAX – 24 bytes
  - EBX – 0
  - ECX – 1 (supervisory state)
  - EDX – 0

8.3 Master Enable

- CR4.CET bit (tentatively bit 23) defined as master enable for CET.
- Master enable allows working with hosted VMM which are not CET aware.

8.4 XSAVE State

CET has two pieces 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].

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.

8.5 CET MSRs

- IA32_U_CET
  - Bit 0 - ENDBR_EN – when set to 1 enables tracking of indirect call/jmp targets to be ENDBRANCH instruction.
  - Bit 1 - LEG_IW_EN – Enable legacy compatibility treatment for indirect call/jmp tracking.
  - Bit 2 - SH_STK_EN – when set to 1 enable shadow stacks at CPL3 when paging is enabled.
  - Bit 3 - WR_SHSTK_EN – when set to 1 enables the WRSS{D,Q}W instructions.
  - Bit 9:4 – 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 endbranch 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 does not land on ENDBRANCH when CET is enabled and not

suppressed. Valid when ENDBR_EN is 1. Must be machine canonical when written in 64 bit mode. On parts that do not support 64 bit mode, the bits 63:32 are reserved and must be 0.

- IA32_S_CET – similar format as IA32_U_CET – configures supervisor mode CET.
- IA32_PL3_SSP – 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 in 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_PL2_SSP - 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 in 64 bit mode. On parts that do not support 64 bit mode, the bits 63:32 are reserved and must be 0.
- IA32_PL1_SSP - 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 in 64 bit mode. On parts that do not support 64 bit mode, the bits 63:32 are reserved and must be 0.
- IA32_PL0_SSP - 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 in 64 bit mode. On parts that do not support 64 bit mode, the bits 63:32 are reserved and must be 0.
- IA32_INTERRUPT_SSP_TABLE_ADDR – 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 in 64 bit mode. On parts that do not support 64 bit mode, this MSR is not present.

9 IA Paging and EPT Extensions

9.1 Shadow Stack Marking in IA Page Tables

Linear data access from control transfer instructions (CALL, RET and IRET), flows that cause control transfers (interrupts/exceptions) and instructions that operate on the shadow stack is a shadow-stack access. Shadow-stack access are allowed only to linear addresses that are mapped such that the logical-AND of the R/W flags in the non-leaf paging structure entries is 1 and in the leaf paging structure entry has R/W flag set to 0 and the dirty flag is 1.

9.1.1 Page Faulting Behavior

IA cumulative fault checking causes the following page faults with shadow stacks enabled:

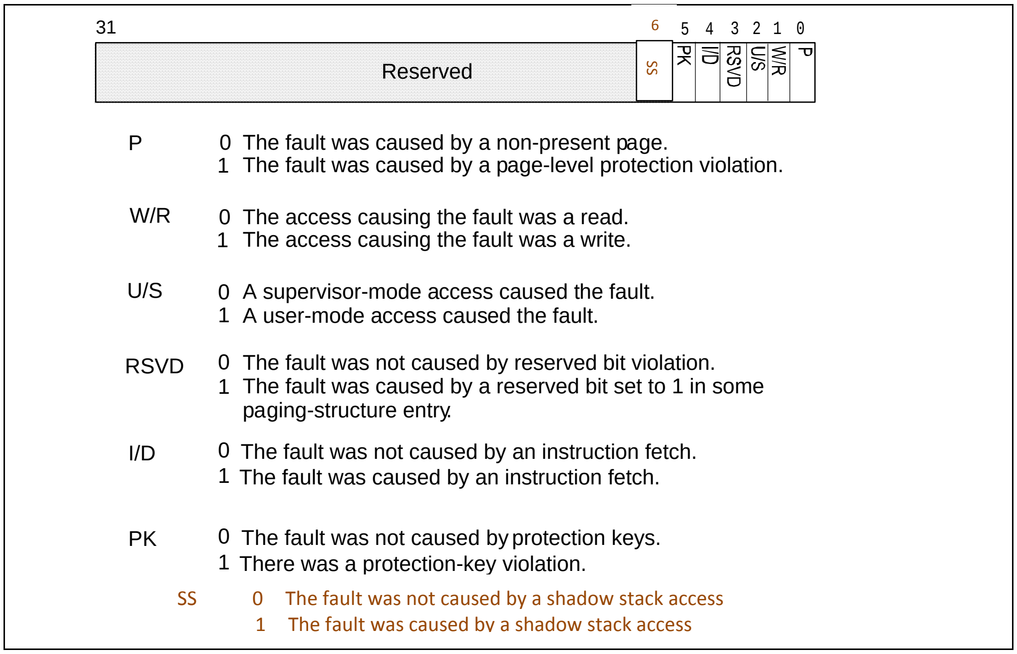
- Page entry is not writeable (W=0) for a user-level regular-store or regular-store-intent access.
- Page entry is not writeable (W=0) for a user-level shadow-stack-store or shadow-stack-store-intent when enclave mode is 1.
- Page entry is not writeable (W=0) for a supervisor-level regular-store or regular-store-intent access when CR0.WP=1.
- Page entry has supervisor privilege (U=0) for a user-level access (no change from current).
- Page entry has user privilege (U=1) for a supervisor-level instruction fetch when CR4.SMEP=1(no change from current).
- Page entry is not executable (NX=1) for an instruction fetch (no change from current).
- Page entry has user privilege (U=1) for a supervisor-level access when CR4.SMAP=1 and EFLAGS.AC=1 (no change from current).
- Page entry is writeable (W=1) or not dirty (D=0) i.e. not a shadow stack page for shadow-stack-store or shadow-stack-store-intent or shadow-stack-load access when enclave mode is 0.
- Page entry is not writeable (W=0) in any non-leaf paging structure i.e. not a shadow stack page for shadow-stack-store or shadow-stack-store-intent or shadow-stack-load access when enclave mode is 0.
- Page entry has user privilege (U=1) for a supervisor-level shadow-stack-load, shadow-stack-store-intent or shadow-stack-store access except those that originate from the WRUSS instruction.

9.1.2 Page-Fault Exceptions

Accesses using linear addresses may cause **page-fault exceptions** (#PF; exception 14). An access to a linear address may cause page-fault exception for either of two reasons: (1) there is no translation for the linear address; or (2) there is a translation for the linear address, but its access rights do not permit the access.

There is no translation for a linear address if the translation process for that address would use a paging-structure entry in which the P flag (bit 0) is 0 or one that sets a reserved bit.

Figure 4 Page-Fault Error Code illustrates the error code that the processor provides on delivery of a page-fault exception. The following items explain how the bits in the error code describe the nature of the page-fault exception:



P 0 The fault was caused by a non-present page.
1 The fault was caused by a page-level protection violation.

W/R 0 The access causing the fault was a read.
1 The access causing the fault was a write.

U/S 0 A supervisor-mode access caused the fault.
1 A user-mode access caused the fault.

RSVD 0 The fault was not caused by reserved bit violation.
1 The fault was caused by a reserved bit set to 1 in some paging-structure entry.

I/D 0 The fault was not caused by an instruction fetch.
1 The fault was caused by an instruction fetch.

PK 0 The fault was not caused by protection keys.
1 There was a protection-key violation.

SS 0 The fault was not caused by a shadow stack access
1 The fault was caused by a shadow stack access

Figure 4 Page-Fault Error Code

- P flag (bit 0).
  This flag is 0 if there is no translation for the linear address because the P flag was 0 in one of the paging-structure entries used to translate that address.

- W/R (bit 1).
  If the access causing the page-fault exception was a write, this flag is 1; otherwise, it is 0. This flag describes the access causing the page-fault exception, not the access rights specified by paging.

- U/S (bit 2).
  If a user-mode access caused the page-fault exception, this flag is 1; it is 0 if a supervisor-mode access did so. This flag describes the access causing the page-fault exception, not the access rights specified by paging.

- RSVD flag (bit 3).
  This flag is 1 if there is no translation for the linear address because a reserved bit was set in one of the paging-structure entries used to translate that address. (Because reserved bits are not checked in a paging-structure entry whose P flag is 0, bit 3 of the error code can be set only if bit 0 is also set.[2])

  Bits reserved in the paging-structure entries are reserved for future functionality. Software developers should be aware that such bits may be used in the future and that a paging-structure entry that causes a page-fault exception on one processor might not do so in the future.

- I/D flag (bit 4).
  This flag is 1 if (1) the access causing the page-fault exception was an instruction fetch; and (2) either (a) CR4.SMEP = 1; or (b) both (i) CR4.PAE = 1 (either PAE paging or IA-32e paging is in use); and

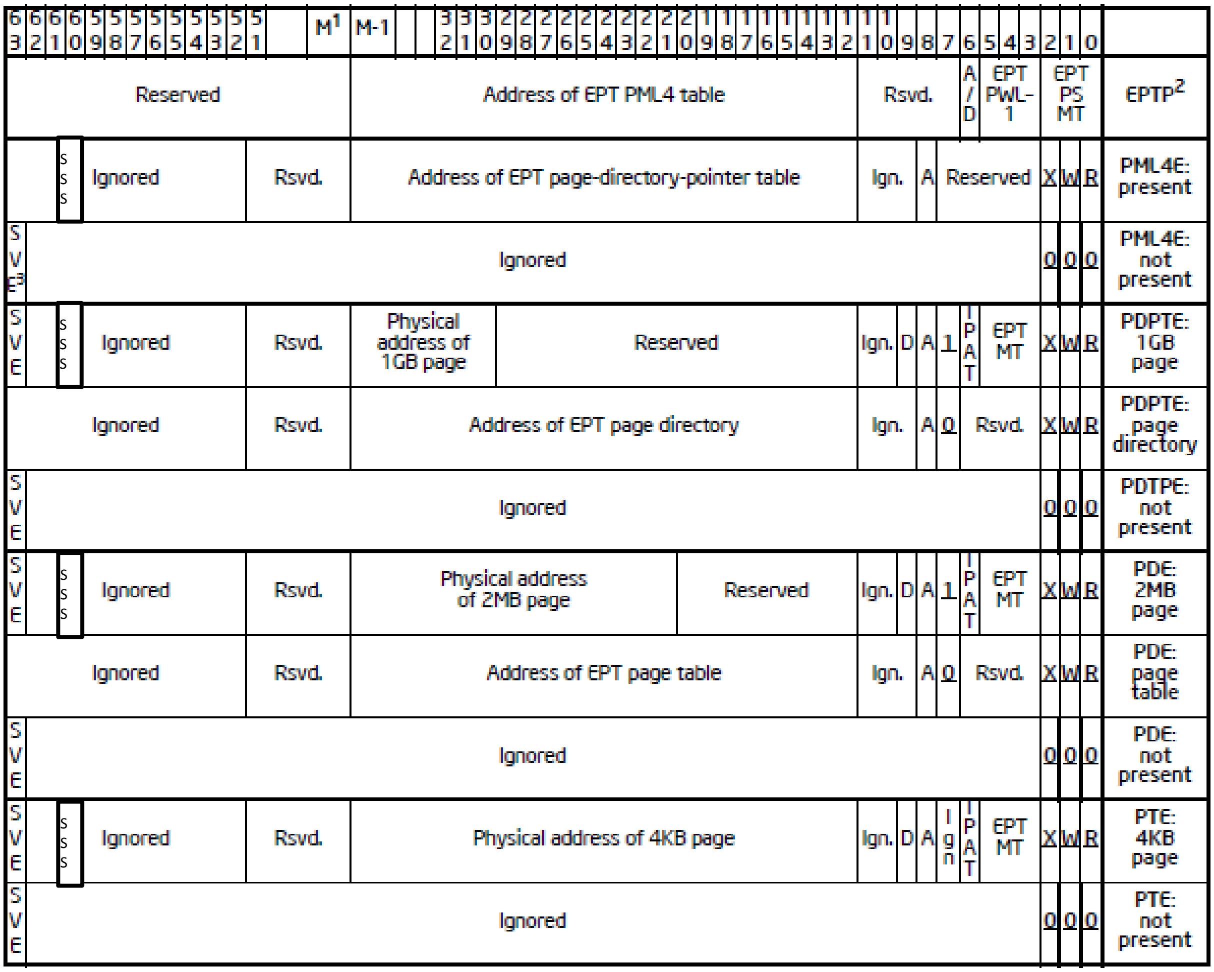
[2] Some past processors had errata for some page faults that occur when there is no translation for the linear address because the P flag was 0 in one of the paging-structure entries used to translate that address. Due to these errata, some such page faults produced error codes that cleared bit 0 (P flag) and set bit 3 (RSVD flag).

(ii) IA32_EFER.NXE = 1. Otherwise, the flag is 0. This flag describes the access causing the page-fault exception, not the access rights specified by paging.

- PK flag (bit 5).
  This flag is 1 if (1) IA32_EFER.LMA = CR4.PKE = 1; (2) the access causing the page-fault exception was a data access; (3) the linear address was a user-mode address with protection key *i*; and (5) the PKRU register is such that either (a) AD*i* = 1; or (b) the following all hold: (i) WD*i* = 1; (ii) the access is a write access; and (iii) either CR0.WP = 1 or the access causing the page-fault exception was a user-mode access.
- SGX flag (bit 15).
  This flag is 1 if the page-fault exception was induced due to fault checks in the EPCM
- SS flag (bit 6).
  This flag is 1 if (1) CR4.CET = 1; (2) the access causing the page-fault exception was a shadow-stack data access.

Page-fault exceptions occur only due to an attempt to use a linear address. Failures to load the PDPTE registers with PAE paging cause general-protection exceptions (#GP(0)) and not page-fault exceptions.

9.2 EPT Extensions



63	62	61	60 59 58 57 56 55 54 53 52	51 M[1]	M-1 … 32 31 30 … 12		11 10 9		8	7	6	5 4 3	2	1	0	
Reserved					Address of EPT PML4 table		Rsvd.				A/D	EPT PWL-1	EPT PS MT			EPTP[2]
		SSS	Ignored	Rsvd.	Address of EPT page-directory-pointer table		Ign.		A	Reserved			X	W	R	PML4E: present
SVE[3]	Ignored												0	0	0	PML4E: not present
SVE		SSS	Ignored	Rsvd.	Physical address of 1GB page	Reserved	Ign.	D	A	1	IPAT	EPT MT	X	W	R	PDPTE: 1GB page
Ignored				Rsvd.	Address of EPT page directory		Ign.		A	0	Rsvd.		X	W	R	PDPTE: page directory
SVE	Ignored												0	0	0	PDTPE: not present
SVE		SSS	Ignored	Rsvd.	Physical address of 2MB page	Reserved	Ign.	D	A	1	IPAT	EPT MT	X	W	R	PDE: 2MB page
Ignored				Rsvd.	Address of EPT page table		Ign.		A	0	Rsvd.		X	W	R	PDE: page table
SVE	Ignored												0	0	0	PDE: not present
SVE		SSS	Ignored	Rsvd.	Physical address of 4KB page		Ign.	D	A	Ign	IPAT	EPT MT	X	W	R	PTE: 4KB page
SVE	Ignored												0	0	0	PTE: not present

Figure 5 Formats of EPTP and EPT Paging-Structure Entries

- SSS (bit 59) - If the “Enable EPT kernel Shadow Stack Control” VM-execution control is 0, this bit is ignored. When “Enable EPT kernel Shadow Stack Control” VM-execution control is 1, this bit instead of the W bit indicates whether shadow-stack-store accesses using supervisor guest linear addresses are allowed to the region controlled by this entry or the page referenced by this entry.

9.2.1 EPT Misconfiguration

An EPT misconfiguration occurs if “Enable EPT kernel shadow stack control” VM execution control is 1, the EPT paging structure entry is present and SSS bit in the paging structure entry is 1 and the R bit in the paging structure entry is 0.

9.2.2 EPT Violations

Once the ultimate physical address is determined, the privileges determined by the EPT paging structure entries are evaluated. The EPT cumulative fault checking is extended as follows to cause an EPT violation if:

- EPT permission is not readable (RWX=’001) for a regular-load or shadow-stack-load access.
- EPT permission is not writeable (RWX=’100 or ‘101) for a regular-store or regular-store-intent access.
- EPT permission is not writeable (RWX=’100 or ‘101) for a shadow-stack-store or shadow-stack-store-intent access when “Enable EPT kernel shadow stack control”=0.
- EPT permission is not readable or writeable (RSssX=`100 or `101) for shadow-stack-load, shadow-stack-store or shadow-stack-store intent access using a supervisor linear address (U/S=0) when “Enable EPT kernel shadow stack control” is 1.
- EPT permission is not writeable (RWX=`100 or `101) for shadow-stack-store or shadow-stack-store intent access using a user linear address (U/S=1) when “Enable EPT kernel shadow stack control” is 1.
- EPT permission is not executable (RWX=’100 or ‘110) for an instruction fetch when “mode-based execution control” is 0.
- EPT permission is not executable (RWX=’100 or ‘110) for an instruction fetch using a supervisor linear address (U=0) when “mode-based execution control” is 1.
- EPT permission is not executable (RWXu=’100 or ‘110) for an instruction fetch using a user linear address (U=1) when “mode-based execution control” is 1.

9.2.3 EPT Violations EXIT QUALIFICATION

A new bit at position TBD in the EPT exit qualification is used to indicate that the access causing the EPT violation was a shadow-stack access.
A new bit at position TBD in the EPT exit qualification is used to report the state of the accumulated SSS bit when “Enable shadow stack control” is 1. When “enable shadow stack control” is 0 this bit reports 0.

9.3 Paging Disabled Behavior

When paging is disabled (CR0.PG=0), the shadow stack load and shadow stack store accesses are allowed to complete always. Shadow stack memory protections are disabled when paging is disabled.

All shadow-stack-stores and shadow-stack-load accesses are treated as accesses using a user linear address (U=1) when paging is disabled and the EPT SSS is not consulted for permission checks.

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 see section 11. 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	Encoding	Size (bits)	
VMX_GUEST_IA32_S_CET_FULL		64	Guest IA32_S_CET MSR
VMX_GUEST_IA32_S_CET_HIGH		32	
VMX_GUEST_SSP_FULL		64	Guest Shadow Stack Pointer (SSP)
VMX_GUEST_SSP_HIGH		32	
VMX_GUEST_IA32_INTERRUPT_SSP_TABLE_ADDR_FULL		64	Guest IA32_INTERRUPT_SSP_TABLE_ADDR MSR
VMX_GUEST_IA32_INTERRUPT_SSP_TABLE_ADDR_HIGH		32	

10.2 VMCS Host State Area Extensions

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

Field	Encoding	Size (bits)	
VMX_HOST_IA32_S_CET_FULL		64	Host IA32_S_CET MSR
VMX_HOST_IA32_S_CET_HIGH		32	
VMX_HOST_SSP_FULL		64	Host Shadow Stack Pointer (SSP)
VMX_HOST_SSP_HIGH		32	
VMX_HOST_IA32_INTERRUPT_SSP_TABLE_ADDR_FULL		64	Host IA32_INTERRUPT_SSP_TABLE_ADDR MSR
VMX_HOST_IA32_INTERRUPT_SSP_TABLE_ADDR_HIGH		32	

10.3 VMCS VM-Exit Controls Extensions

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

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

10.5 Secondary Processor Based VM Execution Control

The secondary processor based VM execution controls are extended with as follows:

Bit Position(s)	Name	Description
	Enable kernel shadow stack control.	Enable kernel shadow stack control bit in EPT.

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.
- 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 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.

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 kernel shadow stack control” execution control bit must be 0 if secondary execution controls field 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 9:4 must be 0. If host address space size is 0 then bits 63:32 must be 0 else EB_LEG_BITMAP_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 9:4 must be 0. 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.
- 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.

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

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?
TBD	TBD	SSP	Yes

11.2 SMI Handler Execution Environment

Processors that support CET, 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.

11.3 RSM

The RSM on processors that support CET 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.

12 TXT Interactions

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

13 SGX Interactions

This section will be updated in a future release of the document.