CALL—Call Procedure

Opcode	Instruction	Op/En	64-bit Mode	Compat/Leg Mode	Description
E8 cw	CALL rel16	D	N.S.	Valid	Call near, relative, displacement relative to next instruction.
E8 cd	CALL rel32	D	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) onto 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 far address is obtained at the 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 a different privilege level, no stack switch occurs.) On the new stack, the processor pushes the parameter selector and stack pointer for the calling procedure's stack, an optional set of parameters from the calling procedure's 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 resume 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 a 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 can not 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 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.
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 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 can not 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.

Refer to Chapter 6, "Procedure Calls, Interrupts, and Exceptions" and Chapter 18, "Control-Flow Enforcement Technology (CET)" in the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1* for CET details.

Instruction ordering. Instructions following a far call may be fetched from memory before earlier instructions complete execution, but they will not execute (even speculatively) until all instructions prior to the far call have completed execution (the later instructions may execute before data stored by the earlier instructions have become globally visible).
Certain situations may lead to the next sequential instruction after a near indirect CALL being speculatively executed. Software can prevent this (e.g., in order to prevent a speculative execution side channel), then an LFENCE instruction opcode can be placed after the near indirect CALL in order to block speculative execution.

Operation

```
IF near call
    THEN near relative call
        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)
                    ShadowStackPush8(RIP);
                FI;
                RIP = tempRIP;
            FI;
        IF OperandSize = 32
            THEN
                tempEIP = 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)
                    ShadowStackPush4(EIP);
                FI;
                EIP = tempEIP;
            FI;
        IF OperandSize = 16
            THEN
                tempEIP = 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(EIP);
                IF ShadowStackEnabled(CPL)
                    ShadowStackPush2(EIP);
                FI;
                EIP = tempEIP;
            FI;
        FI;
    FI;
    IF near indirect, absolute indirect
        IF EndbranchEnableAndNotSuppressed(CPL)
            IF CPL = 3
                THEN
                    (* no 3EH prefix OR IA32_U_CET.NO_TRACK_EN == 0 )
                    THEN
                        IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH
                    FI;
                ELSE
                    (* no 3EH prefix OR IA32_S_CET.NO_TRACK_EN == 0 )
                    THEN
                        IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
                    FI;
                FI;
            FI;
        FI;
    FI; near
```



```
    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)
            ShadowStackPush8(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)
            ShadowStackPush4(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(EIP);
        IF ShadowStackEnabled(CPL)
            (* IP is zero extended and pushed as a 32 bit value on shadow stack *)
            ShadowStackPush4(EIP);
        FI;
        EIP = tempEIP;
    FI;
ELSE (* Near absolute call *)
    IF OperandSize = 64
```



```
        THEN #UD; FI;
    tempEIP = DEST(Offset)
    IF target mode = Compatibility mode
        THEN tempEIP = tempEIP AND 00000000_FFFFFFFFH; FI;
    IF OperandSize = 16
        THEN
            tempEIP = tempEIP AND 0000FFFFH; (* Clear upper 16 bits *)
    IF (IA32_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)
        ShadowStackPush8(CS:tempEIP); (* Pushed with 48 high-order bits of 0 *)
        ShadowStackPush8(SSP+8*(IsEnabled(PL) ? PL:0)); (* Pushed with 32 high-order bits of 0 for 32 bit LIP*)
        ShadowStackPush8(SSP);
    FI;
    IF EndbranchEnabled(CPL)
        IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
        IA32_S_CET.SUPRESS = 0
    FI;
    RIP = tempEIP;
ELSE
    IF OperandSize = 16
        THEN
            Push(CS);
            Push(RIP);
            CS = DEST(CodeSegmentSelector);
            (* Segment descriptor information also loaded *)
            CS[RPL] = CPL;
            RIP = tempEIP;
        ELSE (* OperandSize = 64 *)
            Push(CS); (* Pushed with 48 high-order bits *)
            Push(RIP);
            CS = DEST(CodeSegmentSelector);
            (* Segment descriptor information also loaded *)
            CS[RPL] = CPL;
            RIP = tempEIP;
        FI;
    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 target mode = Compatibility mode
        THEN tempEIP = tempEIP AND 00000000_FFFFFFFFH; FI;
    IF OperandSize = 16
        THEN tempEIP = tempEIP AND 0000FFFFH; (* Clear upper 16 bits *)
    IF (IA32_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 IA32_EFER.LMA & CS.L
            ShadowStackPush8(CS:RIP); (* Pushed with 32 high-order bits of 0 for 32 bit LIP*)
            ShadowStackPush8(SSP);
        ELSE
            ShadowStackPush4(CS:EIP);
        FI;
    FI;
    IF EndbranchEnabled(CPL)
        IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
        IA32_S_CET.SUPRESS = 0
    FI;
    RIP = tempEIP;
    IF (SSP & 0xFFFFFFFF00000000!= 0)
        THEN #GP(0); FI;
    IF
    (* align to 8 byte boundary if not already aligned *)
    tempSSP = SSP;
    Shadow_stack_store 4 bytes of 0 to (SSP - 4)
    SSP = SSP & 0xFFFFFFFF00000FFF;
    ShadowStackPush8(tempPushCS); (* Pushed with 48 high-order 0 bits *)
    ShadowStackPush8(tempPushSSP); (* Pushed 32 high-order bits of 0 for 32 bit LIP*)
    ShadowStackPush8(tempSSP);
    FI;
    IF EndbranchEnabled(CPL)
        IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
        IA32_S_CET.SUPRESS = 0
    FI;
    RIP = tempEIP;
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;
```



```
        (* 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); (* Pushed with 48 high-order bits *)
                Push(RIP);
                CS = DEST(CodeSegmentSelector);
                (* Segment descriptor information also loaded *)
                CS[RPL] = CPL;
                RIP = tempRIP;
        FI;
    FI;
IF ShadowStackEnabled(CPL)
    IF (IA32_EFER.LMA and DEST(CodeSegmentSelector)(L) = 0
        (* If target is legacy or compatibility mode then the SSP must be in low 4GB *)
        IF (SSP & 0xFFFFFFFF00000000 != 0)
            THEN #GP(0); FI;
    FI;
    (* align to 8 byte boundary if not already aligned *)
    tempSSP = SSP;
    Shadow_stack_store 4 bytes of 0 to (SSP - 4)
    SSP = SSP & 0xFFFFFFFF00000FFF;
    ShadowStackPush8(tempPushCS); (* Pushed with 48 high-order 0 bits *)
    ShadowStackPush8(tempPushSSP); (* Pushed 32 high-order bits of 0 for 32 bit LIP*)
    ShadowStackPush8(tempSSP);
    FI;
    IF EndbranchEnabled(CPL)
        IF CPL = 3
            THEN
                IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH
                IA32_U_CET.SUPRESS = 0
            ELSE
                IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
                IA32_S_CET.SUPRESS = 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;
        FI;
    IF IA32_EFER.LMA = 0 and NewSS is NULL
        THEN #TS(NewSS); FI;
    Read 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 (* 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:EIP = 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) AND CPL = 3
        THEN
            IF IA32_EFER.LMA = 0
                THEN IA32_PL3_SSP := TSS;
                ELSE (* adjust bits 63:N get the value of bit N-1, where N is the CPU's maximum linear-address width *)
                    IA32_PL3_SSP := LA_adjust(SSP);
            FI;
    FI;
    CPL := CodeSegment[RPL]
    CLEAR_BUSY(oldCS);
    IF ShadowStackEnabled(CPL)
        oldSSP := SSP
        SSP := IA32_PL3_SSP; (* where is the CPL *)
        IF SSP & 0x07 != 0 (* if SSP not aligned to 8 bytes then #GP *)
            THEN #GP(0); FI;
        IF (IA32_EFER.LMA and CS.L) = 0 AND SSP[63:32] != 0)
```



```
            THEN #GP(0); FI;
        expected_token_value = SSP
        new_token_value = SSP | BUSY_BIT          (* busy bit - bit position 0 - must be clear *)
        IF #relax_stack_token_unpaired(oldSSP, new_token_value, expected_token_value) != expected_token_value
            THEN #GP(0); FI;
        IF oldSS.DPL != 3
            ShadowStackPush8(oldCS); (* Padded with 48 high-order bits of 0 *)
            ShadowStackPush8(oldSSP+8*(IsEnabled(PL) ? PL:0)); (* Padded with 32 high-order bits of 0 for 32 bit LIP*)
            ShadowStackPush8(oldSSP);
        FI;
    FI;
    IF EndbranchEnabled(CPL)
        IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH
        IA32_S_CET.SUPRESS = 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 IsCallGateSize = 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:EIP = 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:oldIP); (* Return address to calling procedure *)
                FI;
            FI;
    CS[RPL] := CPL
    IF ShadowStackEnabled(CPL)
        (* Align to next 8 byte boundary *)
        tempSSP = SSP;
        Shadow_stack_store 4 bytes of 0 to (SSP - 4)
        SSP = SSP & 0xFFFFFFFF00000FFF;
        (* push cs/lpsw on shadow stack *)
        ShadowStackPush8(oldCS); (* Padded with 48 high-order bits of 0 *)
```



```
        ShadowStackPush8(oldSS:oldSP); (* Padded with 32 high-order bits of 0 for 32 bit LIP*)
        ShadowStackPush8(tempSSP);
    FI;
    IF EndbranchEnabled(CPL)
        IF CPL = 3
            THEN
                IA32_U_CET.TRACKER = WAIT_FOR_ENDBRANCH;
                IA32_U_CET.SUPRESS = 0
            ELSE
                IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH;
                IA32_S_CET.SUPRESS = 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 descriptor is not a TSS segment
        THEN #GP(TSS selector); FI;
    IF TSS descriptor specifies that the TSS is busy
        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