INT *n*/INTO/INT 3—Call to Interrupt Procedure

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

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 an interrupt vector number from 0 to 255, encoded as an 8-bit unsigned intermediate value. Each interrupt vector number provides an index to a gate descriptor in the IDT. The first 32 interrupt vector numbers are reserved by Intel for system use. Some of these interrupts 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), interrupt vector number 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 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 interrupt vector number 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).

Table 3-64. 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
TRAP-OR-INTERRUPT-GATE				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 and exceptions.

```
IF PE = 0
   THEN
       GOTO REAL-ADDRESS-MODE;
   ELSE (* PE = 1 *)
       IF (VM = 1 and IOPL < 3 AND INT n)
           THEN
                #GP(0);
           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 ∗ 4) + 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 *)
```

```
    CS ← IDT(Descriptor (vector_number ∗ 4), selector));
    EIP ← IDT(Descriptor (vector_number ∗ 4), offset)); (* 16 bit offset AND 0000FFFFH *)
END;
PROTECTED-MODE:
    IF ((vector_number ∗ 8) + 7) is not within IDT limits
    or selected IDT descriptor is not an interrupt-, trap-, or task-gate type
        THEN #GP((vector_number ∗ 8) + 2 + EXT); FI;
        (* EXT is bit 0 in error code *)
    IF software interrupt (* Generated by INT n, INT 3, or INTO *)
        THEN
            IF gate descriptor DPL < CPL
                THEN #GP((vector_number ∗ 8) + 2 ); FI;
                (* PE = 1, DPL<CPL, software interrupt *)
    FI;
    IF gate not present
        THEN #NP((vector_number ∗ 8) + 2 + EXT); FI;
    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 ((vector_number ∗ 16) + 15) is not in IDT limits
    or selected IDT descriptor is not an interrupt-, or trap-gate type
        THEN #GP((vector_number ∗ 16) + 2 + EXT); FI;
        (* EXT is bit 0 in error code *)
    IF software interrupt (* Generated by INT n, INT 3, but not INTO *)
        THEN
            IF gate descriptor DPL < CPL
                THEN #GP((vector_number ∗ 16) + 2 ); FI;
                (* PE = 1, DPL < CPL, software interrupt *)
        ELSE (* Generated by INTO *)
            THEN #UD;
    FI;
    IF gate not present
        THEN #NP((vector_number ∗ 16) + 2 + EXT); FI;
    IF ((vector_number * 16)[IST] ≠ 0)
        NewRSP ← TSS[ISTx]; FI;
    GOTO TRAP-OR-INTERRUPT-GATE; (* Trap/interrupt gate *)
END;
TASK-GATE: (* PE = 1, task gate *)
    Read segment selector in task gate (IDT descriptor);
        IF 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 interrupt caused by fault with error code
        THEN
            IF stack limit does not allow push of error code
                THEN #SS(0); FI;
            Push(error code);
    FI;
    IF EIP not within code segment limit
        THEN #GP(0); FI;
END;
TRAP-OR-INTERRUPT-GATE:
    Read segment selector for trap or interrupt gate (IDT descriptor);
    IF segment selector for code segment is NULL
        THEN #GP(0H + EXT); FI; (* NULL selector with EXT flag set *)
    IF segment selector is not within its descriptor table limits
        THEN #GP(selector + EXT); FI;
    Read trap or interrupt handler descriptor;
    IF descriptor does not indicate a code segment
    or code segment descriptor DPL > CPL
        THEN #GP(selector + EXT); FI;
    IF trap or interrupt gate segment is not present,
        THEN #NP(selector + EXT); FI;
    IF code segment is non-conforming and DPL < CPL
        THEN
            IF VM = 0
                THEN
                    GOTO INTER-PRIVILEGE-LEVEL-INTERRUPT;
                    (* PE = 1, interrupt or trap gate, nonconforming
                    code segment, DPL < CPL, VM = 0 *)
                ELSE (* VM = 1 *)
                    IF code segment DPL ≠ 0
                        THEN #GP; (new code segment selector);
                    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(new code segment selector); FI;
                IF code segment is conforming or code segment DPL = CPL
                    THEN
                        GOTO INTRA-PRIVILEGE-LEVEL-INTERRUPT;
                    ELSE
                        #GP(CodeSegmentSelector + EXT);
                        (* PE = 1, interrupt or trap gate, nonconforming
                        code segment, DPL > CPL *)
                FI;
        FI;
    END;
    INTER-PRIVILEGE-LEVEL-INTERRUPT:
        (* PE = 1, interrupt or trap gate, non-conforming code segment, DPL < CPL *)
        (* Check segment selector and descriptor for stack of new privilege level in current TSS *)
        IF current TSS is 32-bit TSS
            THEN
                TSSstackAddress ← (new code segment DPL ∗ 8) + 4;
                IF (TSSstackAddress + 7) > TSS limit
                    THEN #TS(current TSS selector); FI;
                NewSS ← TSSstackAddress + 4;
                NewESP ← stack address;
            ELSE
                IF current TSS is 16-bit TSS
                    THEN(* TSS is 16-bit *)
                        TSSstackAddress ← (new code segment DPL ∗ 4) + 2
                        IF (TSSstackAddress + 4) > TSS limit
                            THEN #TS(current TSS selector); FI;
                        NewESP ← TSSstackAddress;
                        NewSS ← TSSstackAddress + 2;
                    ELSE (* TSS is 64-bit *)
                        NewESP ← TSS[RSP FOR NEW TARGET DPL];
                        NewSS ← 0;
                FI;
        FI;
        IF segment selector is NULL
            THEN #TS(EXT); FI;
        IF segment selector index is not within its descriptor table limits
        or segment selector’s RPL ≠ DPL of code segment,
            THEN #TS(SS selector + EXT); FI;
        IF (IA32_EFER.LMA = 0) (* Not IA-32e mode *)
            Read segment descriptor for stack segment in GDT or LDT;
            IF stack segment DPL ≠ DPL of code segment,
        or stack segment does not indicate writable data segment
            THEN #TS(SS selector + EXT); FI;
        IF stack segment not present
            THEN #SS(SS selector + EXT); FI;
    FI
    IF 32-bit gate
            THEN
                IF new stack does not have room for 24 bytes (error code pushed)
                or 20 bytes (no error code pushed)
                    THEN #SS(segment selector + EXT); FI;
            FI
        ELSE
            IF 16-bit gate
                THEN
                    IF new stack does not have room for 12 bytes (error code pushed)
                    or 10 bytes (no error code pushed);
                    THEN #SS(segment selector + EXT); FI;
            ELSE (* 64-bit gate*)
                IF StackAddress is non-canonical
                    THEN #SS(0);FI;
        FI;
    FI;
    IF (IA32_EFER.LMA = 0) (* Not IA-32e mode *)
        THEN
            IF instruction pointer is not within code segment limits
                THEN #GP(0); FI;
            SS:ESP ← TSS(NewSS:NewESP);
                (* Segment descriptor information also loaded *)
        ELSE
            IF instruction pointer points to non-canonical address
                THEN #GP(0); FI:
    FI;
    IF 32-bit gate
        THEN
            CS:EIP ← Gate(CS:EIP); (* Segment descriptor information also loaded *)
        ELSE
            IF 16-bit gate
                THEN
                    CS:IP ← Gate(CS:IP);
                    (* Segment descriptor information also loaded *)
                ELSE (* 64-bit gate *)
                    CS:RIP ← Gate(CS:RIP);
                    (* Segment descriptor information also loaded *)
            FI;
    FI;
    IF 32-bit gate
            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 16-bit gate
                    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 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;
    CPL ← CodeSegmentDescriptor(DPL);
    CS(RPL) ← CPL;
    IF interrupt gate
        THEN IF ← 0 (* Interrupt flag set to 0: disabled *); FI;
    TF ← 0;
    VM ← 0;
    RF ← 0;
    NT ← 0;
END;
INTERRUPT-FROM-VIRTUAL-8086-MODE:
    (* Check segment selector and descriptor for privilege level 0 stack in current TSS *)
    IF current TSS is 32-bit TSS
        THEN
            TSSstackAddress ← (new code segment DPL ∗ 8) + 4;
            IF (TSSstackAddress + 7) > TSS limit
                THEN #TS(current TSS selector); FI;
            NewSS ← TSSstackAddress + 4;
```

```
                NewESP ← stack address;
            ELSE (* TSS is 16-bit *)
                TSSstackAddress ← (new code segment DPL ∗ 4) + 2;
                IF (TSSstackAddress + 4) > TSS limit
                    THEN #TS(current TSS selector); FI;
                NewESP ← TSSstackAddress;
                NewSS ← TSSstackAddress + 2;
        FI;
        IF segment selector is NULL
            THEN #TS(EXT); FI;
        IF segment selector index is not within its descriptor table limits
        or segment selector’s RPL ≠ DPL of code segment
            THEN #TS(SS selector + EXT); FI;
        Access segment descriptor for stack segment in GDT or LDT;
        IF stack segment DPL ≠ DPL of code segment,
        or stack segment does not indicate writable data segment
            THEN #TS(SS selector + EXT); FI;
        IF stack segment not present
            THEN #SS(SS selector + EXT); FI;
        IF 32-bit gate
            THEN
                    IF new stack does not have room for 40 bytes (error code pushed)
                    or 36 bytes (no error code pushed)
                        THEN #SS(segment selector + EXT); FI;
            ELSE IF 16-bit gate
                THEN
                    IF new stack does not have room for 20 bytes (error code pushed)
                    or 18 bytes (no error code pushed)
                        THEN #SS(segment selector + EXT); FI;
                ELSE (* 64-bit gate*)
                    IF StackAddress is non-canonical
                        THEN #SS(0);
                FI;
        FI;
        IF instruction pointer is not within code segment limits
            THEN #GP(0); FI;
        tempEFLAGS ← EFLAGS;
        VM ← 0;
        TF ← 0;
        RF ← 0;
        NT ← 0;
        IF service through interrupt gate
            THEN IF = 0; FI;
```

```
    TempSS ← SS;
    TempESP ← ESP;
    SS:ESP ← TSS(SS0:ESP0); (* Change to level 0 stack segment *)
    (* Following pushes are 16 bits for 16-bit gate and 32 bits for 32-bit 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 NULLified, invalid in protected mode *)
    FS ← 0;
    DS ← 0;
    ES ← 0;
    CS ← Gate(CS);
    IF OperandSize = 32
        THEN
            EIP ← Gate(instruction pointer);
        ELSE (* OperandSize is 16 *)
            EIP ← Gate(instruction pointer) AND 0000FFFFH;
    FI;
    (* Start execution of new routine in Protected Mode *)
END;
INTRA-PRIVILEGE-LEVEL-INTERRUPT:
    (* PE = 1, DPL = CPL or conforming segment *)
    IF 32-bit gate and 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(0); FI;
        ELSE IF 16-bit gate
            IF current stack does not have room for 8 bytes (error code pushed)
            or 6 bytes (no error code pushed)
                THEN #SS(0); FI;
        ELSE (* 64-bit gate*)
                IF StackAddress is non-canonical
                    THEN #SS(0);
        FI;
    FI;
    IF instruction pointer not within code segment limit
        THEN #GP(0); FI;
    IF 32-bit gate
        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 16-bit gate
                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 (* 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 interrupt gate
        THEN IF ← 0; FI; (* Interrupt flag set to 0: 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(0)	If the instruction pointer in the IDT or in the interrupt-, trap-, or task gate is beyond the code segment limits.
#GP(selector)	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 interrupt vector number is 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.
#SS(0)	If pushing the return address, flags, or error code onto the stack exceeds the bounds of the stack segment and no stack switch occurs.
#SS(selector)	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(selector)	If code segment, interrupt-, trap-, or task gate, or TSS is not present.
#TS(selector)	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.

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(0)	(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.
#GP(selector)	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 interrupt vector number is 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(selector)	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(selector)	If code segment, interrupt-, trap-, or task gate, or TSS is not present.
#TS(selector)	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.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If the instruction pointer in the 64-bit interrupt gate or 64-bit trap gate is non-canonical.
#GP(selector)	If the segment selector in the 64-bit interrupt or trap gate is NULL.
	If the interrupt vector number is outside the IDT limits.
	If the interrupt vector number points to a gate which is in non-canonical space.
	If the interrupt vector number points to a descriptor which is not a 64-bit interrupt gate or 64-bit trap gate.
	If the descriptor pointed to by the gate selector is outside the descriptor table limit.
	If the descriptor pointed to by the gate selector is in non-canonical space.
	If the descriptor pointed to by the gate selector is not a code segment.
	If the descriptor pointed to by the gate selector doesn’t have the L-bit set, or has both the L-bit and D-bit set.
	If the descriptor pointed to by the gate selector has DPL > CPL.
#SS(0)	If a push of the old EFLAGS, CS selector, EIP, or error code is in non-canonical space with no stack switch.
#SS(selector)	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(selector)	If the 64-bit interrupt-gate, 64-bit trap-gate, or code segment is not present.

#TS(selector)	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.