X86-S
EXTERNAL ARCHITECTURAL SPECIFICATION

Rev. 1.0
April 2023

Document Number: 351407-001

Notice: This document contains information on products in the design phase of development. The information here is subject to change without notice. Do not finalize a design with this information.

Intel technologies may require enabled hardware, software or service activation.

No computer system can be absolutely secure. Intel does not assume any liability for lost or stolen data or systems or any damages resulting from such losses.

Your costs and results may vary.

You may not use or facilitate the use of this document in connection with any infringement or other legal analysis concerning Intel products described herein. You agree to grant Intel a non-exclusive, royalty-free license to any patent claim thereafter drafted which includes subject matter disclosed herein.

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

All product plans and roadmaps are subject to change without notice.

The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

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.

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

Figures

Tables

(This page intentionally left blank)

1 About This Document

1.1 Audience

This document is intended for software development for the X86-S ISA.

To provide feedback, email x86s_feedback@intel.com

2 Introduction

X86-S is a legacy-reduced-OS ISA that removes outdated execution modes and operating system ISA.

The presence of the X86-S ISA is enumerated by a single, main CPUID feature bit LEGACY_REDUCED_OS_ISA in a future CPUID field. The bit implies all the ISA removals described in this document. Some of the additional features, like 64bit SIPI or the 5 Level page table switch, have separate CPUID feature flags and can be implemented independently of X86-S.

Changes in X86-S ISA consist of:

- restricting the CPU to be always in paged mode
- removing 32-bit ring 0, as well as vm86 mode.
- removing ring 1 and ring 2
- removing 16-bit real and protected modes
- removing 16-bit addressing
- removing fixed MTRRs
- removing user-level I/O and string I/O
- removing CR0 Write-Through mode
- removing legacy FPU control bits in CR0
- removing ring 3 interrupt flag control
- removing obsolete CR access instruction
- rearchitecting INIT/SIPI
- adding a new mechanism to switch between 4- and 5-level page tables
- removing XAPIC and only supporting x2APIC
- removing APIC support for 8529
- removing the disabling of NX or SYSCALL or long mode in the EFER MSR
- removing the #SS and #NP exceptions
- supporting a subset of segmentation architecture, with the following conditions:
    - restricted to a subset of IDT event delivery
    - base only for FS, GS
    - base and limit for GDT, IDT, and TSS
    - no limit on data or code fetches in 32-bit mode (similar to 64-bit)
    - no AR or unusable selector checking on CS, DS, ES, FS, and GS on data or code fetches in any mode
    - restricted support for far call, far return, far jump, and IRET (like FRED).

(This page intentionally left blank)

3 Architectural Changes

3.1 Removal of 32-Bit Ring 0

32-bit ring 0 is not supported anymore and cannot be entered.

3.2 Removal of Ring 1 and Ring 2

Ring 1 and 2 are not supported anymore and cannot be entered.

3.3 Removal of 16-Bit and 32-Bit Protected Mode

16-bit and 32-bit protected mode are not supported anymore and cannot be entered. The CPU always operates in long mode. The 32-bit submode of Intel64 (compatibility mode) still exists. An attempt to load a descriptor into CS that has CS.L==0 and CS.D==0 will generate a #GP(sel) exception.

3.4 Removal of 16-Bit Addressing and Address Size Overrides

For 32-bit compatibility mode, the 16-bit address mode override prefix (0x67) triggers a #GP(0) exception when it leads to a memory reference and the instruction is not a jump.

Jumps with 0x66 prefix that previously did truncate RIP to 16bit (Jump Short 0x7*, Jump Near 0x0f 8*, LOOP 0xE0-2, JECZ 0xE3, JMP near 0xE9 and 0xEB, CALL rel 0xE8, JMP near 0xFF/4, CALL indirect near 0xff/2, RET near 0xC2-3, JMP far 0xEA and 0xFF/5, CALL indirect far 0xFF/3, CALL far 0x9A, RET far 0xCA-B) will now #UD.

Jumps with 0x67 prefix that previously did truncate to 16bit (CALL indirect near mem 0xff/2 mem, JMP far 0xea and 0xff /5, CALL indirect far 0xff /3) will now #GP(0)

Note that there is no fault for operations which do not modify memory or jump, like LEA or NOPs.

An attempt to load from SS that has SS.B==0 (16bit data segment) in compatibility mode will generate a #GP(sel) exception.

3.5 CPUID

The LEGACY_REDUCED_OS_ISA feature bit in a future CPUID field indicates all the ISA removals described in this document.

SIPI64 in a future CPUID field indicates support for 64-bit SIPI.

LA57SWITCH in a future CPUID field indicates support for 5-level paging switch.

3.6 Restricted Subset of Segmentation

X86-S supports a subset of segmentation:

- No gates are supported in the GDT/LDT; it only supports data segments, non conforming code segments, and TSSs.
- Bases are supported for FS, GS, GDT, IDT, LDT, and TSS registers; the base for CS, DS, ES, and SS is ignored for 32-bit mode, same as 64-bit mode (treated as zero).

- Limits are supported for GDT, IDT, LDT, and TSS; the limit for CS, DS, ES, FS, GS, and SS is treated as infinite.
- Expand down, conforming, and unusable segments are not supported.
- The descriptor.DPL field must be 0 or 3, and the selector.RPL must be 0 or 3; otherwise, access rights on segment loads are identical.
- On loads/stores, R/W access rights and NULL are ignored.
- IRET can switch rings from 0 to 3, or stay within a ring, but cannot cause a task switch or enter into VM86 mode.
- Descriptor accessed bits will not be set in memory, but appear to be set when accessed through the LAR instruction.
- Segment override prefixes (other than FS/GS) are ignored when they refer to segments. The segment override prioritization is not changed; any group of overrides that resolves to something other than FS or GS is forced to use DS.
- #SS exceptions are removed and will signal a #GP(0) instead.
- #NP exceptions are removed and will signal a #GP(0) instead.
- The LMSW instruction is removed and will signal a #UD exception.

The three operating modes shown in Table 1 are supported.

Table 1. Supported Operating Modes

	CPL=0	**CPL=3**
LMA=1 CS.L=0	Unsupported	User32 (restricted ring 3 compatibility mode)
LMA=1 CS.L=1	Supervisor (ring 0 64-bit mode)	User64 (ring 3 64-bit mode)

3.7 New Checks When Loading Segment Registers

When loading segment registers, the following conditions are checked:

- The descriptor.DPL field must be either 0 or 3.
- Code descriptors must not be conforming, or be 16-bit, or be 32-bit when DPL==0.
- Data descriptors must not be expand down.

A #GP(sel) exception is signaled if these conditions are not met.

No #GP exception will be generated when loading a segment with a non-infinite limit or non-zero base.

Pseudocode for the modified instructions can be found in Chapter 4.

3.8 Removal of #SS and #NP Exceptions

Any faulting stack segment references, both explicit and implicit, do not cause #SS exceptions

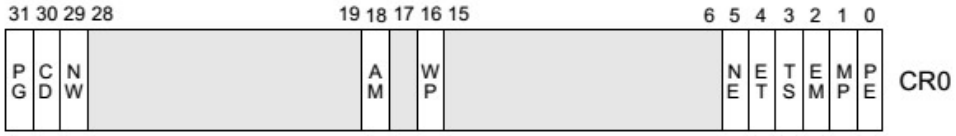
anymore. Instead, #GP exceptions will be generated. All descriptor loads with desc.P==0 will generate a #GP(sel) exception.

3.9 Fixed Mode Bits

The CPU is always running in the 64-bit submode of Intel64. Real mode, protected mode, or VM86 modes cannot be enabled.

3.9.1 Fixed CR0 Bits

All bits in the CR0 register, shown in Figure 1, except for the TS, WP, AM, and CD bits, are fixed. ET is fixed to 1 but ignored on input. An incorrect value in a fixed bit will produce a #GP(0) exception, but only after causing a VM exit from CR0 exiting if configured. Reading will always return the fixed value with the current value of the flexible bits, unless changed by a VM exit from CR0 exiting.

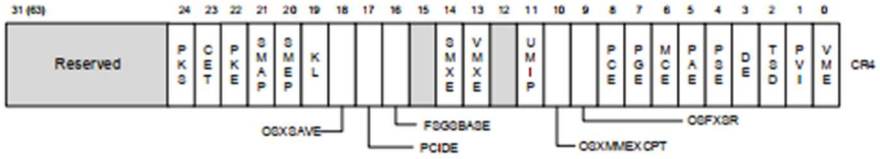


CR0 Bit	Fixed Value	Bit	Implication
PE	1	0	Protection enable: always in protected mode.
MP	1	1	Monitor coprocessor: always enabled.
EM	0	2	FP emulation.
TS	-	3	Task switch. Disable FPU. This bit is still flexible.
ET	1	4	Extension type (ignored on input).
NE	1	5	Numeric error.
WP	-	16	Write protect page tables. This bit is still flexible.
AM	-	18	Enable alignment checks with RFLAGS.AC.
NW	0	29	Write through. Always disabled.
CD	-	30	Cache disable. This bit is still flexible.
PG	1	31	Paging is always enabled.

Figure 1. CR0 Register

3.9.2 Fixed CR4 Bits

The following bits are fixed in the CR4 register, shown in Figure 2. Writing any other value (except for any value of VME) for them will produce a #GP(0) exception if not resulting in a VM-exit from CR4 exiting.



CR4 Bit	Fixed Value	Bit	Implication
VME	0	0	No support for VM86 mode (ignored on input).
PVI	0	1	No support for Protected-mode virtual interrupts.
PAE	1	5	8-byte PTEs. Always enabled in 64-bit mode.

Figure 2. CR4 Register

3.9.3 Fixed EFER Bits

The bits listed in Table 2 are fixed in the EFER MSR. Writing other values to EFER will produce a #GP exception, except for LMA, which is ignored.

Table 2. Fixed EFER Bits

EFER Bit	Fixed Value	Bit	Implication
SCE	1	0	Syscall is always enabled.
LMA	1	8	Always in long mode but changes ignored.
LME	1	10	Always in long mode.
NXE	1	11	NX bit for page tables is always enabled.

3.9.4 Obsolete RFLAGS

Figure 3 shows the bits in the RFLAG register. The IOPL, VM, VIF, and VIP bits are always zero. The rules in Table 3 apply.

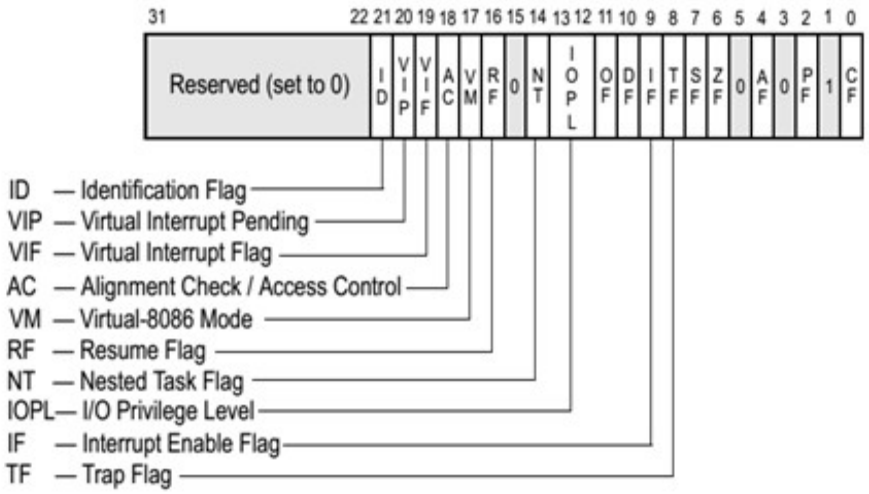


Figure 3. RFLAGS Register

Table 3. Behavior of Obsolete RFLAGS

Action	Action on newIOPL != 0	Action on newVIF != 0 or newVIP != 0	Action on newVM!=0
POPF CPL3	Ignored	Ignored	Ignored
POPF CPL0	Ignored	Ignored	Ignored
SYSRET	#GP(0)	#GP(0)	N/A (always cleared)
IRET CPL3->CPL3	Ignored	Ignored	Ignored
IRET CPL0	#GP(0)	#GP(0)	Ignored
ERETU	#GP(0)	#GP(0)	#GP(0)
ERETS	#GP(0)	#GP(0)	#GP(0)
VMEntry	Bad Guest State error	Bad Guest State error	Bad Guest State error
SEAMRET	Bad Guest State error	Bad Guest State error	Bad Guest State error
RSM	Forced to 0	Forced to 0	Forced to 0

3.9.5 Obsolete Status Register Instruction

The LMSW instruction is removed and will result in a #UD fault.

3.9.6 Removal of Ring 3 I/O Port Instructions

There is no concept of user mode I/O port accesses anymore, and using INB/INW/INL/INQ/OUTB/OUTW/OUTL/OUTQ in ring 3 always leads to a #GP(0) exception. That #GP(0) check will be before VM execution checks or I/O permission bitmap checks. This implies there will be no loads from the I/O permission bitmap.

3.9.7 Removal of String I/O

INS/OUTS are not supported and will result in a #UD exception. This includes the REP variants of the INS/OUTS instructions as well.

3.10 64-Bit SIPI

64-bit SIPI defines an architectural package scope IA32_SIPI_ENTRY_STRUCT_PTR MSR that contains a physical pointer to an entry structure in memory.

To boot APs in paged 64-bit mode, a 64-bit startup sequence is used. Booting CPUs is triggered by triggering INIT or SIPI messages through the X2APIC ICR register after 64-bit SIPI is enabled in the IA32_SIPI_ENTRY_STRUCT_PTR MSR, as well as in the in-memory entry structure features bit pointed to by the MSR. Legacy INIT/SIPI are not supported. The presence of 64bit SIPI is enumerated with the SIPI64 CPUID feature bit.

3.10.1 IA32_SIPI_ENTRY_STRUCT_PTR

The IA32_SIPI_ENTRY_STRUCT_PTR package scope MSR, shown in Table 4, defines the execution context of the target CPU after receiving a SIPI message. It points to an entry structure in memory.The MSR is read only after the package BIOS_DONE MSR bit is set.

Table 4. IA32_SIPI_ENTRY_STRUCT_PTR MSR

Bits	Field	Attr	Reset Value	Description
63:MAXPA	Reserved	NA	0	-
MAXPA:12	SIPI_ENTRY_STRUCT_PTR	RW	0	Physical pointer to SIPI_ENTRY_STRUCT.
11:1	Reserved	NA	0	-
0	ENABLED	RW	0	Enable 64-bit SIPI.

After INIT, NMIs are blocked until explicitly unblocked by ERETS/ERETU/IRET.

On receiving a SIPI, the target CPU loads the register state from the entry struct and starts executing at the specified RIP. The vectors delivered in the vector field of the INIT/SIPI IPI messages are ignored. Any consistency check failures in the SIPI_ENTRY_STRUCT will also lead to a shutdown on the target CPU.

3.10.2 The SIPI_ENTRY_STRUCT Definition

The entry struct memory table, shown in Table 5, defines the execution context of a CPU receiving a SIPI.

Table 5. SIPI_ENTRY_STRUCT Structure in Memory

Offset	Size	RW	Name	Description
0	8	R	FEATURES	Bit 0 - enable bit (0 - shutdown). Other bits are reserved.
8	8	R	RIP	New instruction pointer to execute after SIPI. Valid values depend on new CR4.
16	8	R	CR3	New CR3 value. Must be consistent with new CR4.PCIDE and no reserved bits set.
24	8	R	CR0	New CR0 value. Non-flexible bits must match fixed values and no reserved bits set.
32	8	R	CR4	New CR4 value. Non-flexible bits must match fixed value. Must be consistent with new CR3, new RIP, new CR0 and no reserved bits set.

3.10.3 Pseudocode on Receiving INIT When Not Blocked

```
IF in guest mode THEN
      Trigger exit
FI
IF IA32_SIPI_ENTRY_STRUCT_PTR.ENABLED = 0 THEN
   Shutdown // Non X86-S falls back to legacy INIT
```

```
FI
RFLAGS = 2 # clear all modifiable bits in RFLAGS
Clear flexible bits in CR2/CR3/CR4
Clear TS and AM bits in CR0, preserve CR0.CD # rest is fixed values
Set CS/SS/DS/ES/FS/GS/GDTR/IDTR/LDTR/TS selector to 0, with P=1, W=1
Set IDTR/TS/GDTR/LDTR limits to 0
Clear FS/GS bases
Clear KERNEL_GS MSR
Set RDX to 0x000n06xxx, where n is extended model value and x is a stepping number
Clear all other GPRs
Clear DR0/DR1/DR2/DR3
Set DR6 to 0xffff0ff0
Set DR7 to 0x400
Set x87 FPU control word to 0x37f
Set x87 FPU status word to 0
Set x87 FPU tag word to 0xffff
Flush all TLBs
IF IA32_APICBASE.BSP = 1 THEN
   Force 64bit supervisor mode as in reset
   Execute 64bit reset vector using CR3 value from reset
ELSE
      Enter wait for SIPI state
FI
```

3.10.4 Pseudocode on Receiving SIPI

```
IF IA32_SIPI_ENTRY_STRUCT_PTR.ENABLED = 0 THEN
   Shutdown // On non X86-S fall back to legacy SIPI
FI
// following memory reads are done physically with normal ring 0 rights honoring range registers and allowing
MKTME keys but not TDX
ENTRY_STRUCT = IA32_SIPI_ENTRY_STRUCT_PTR[12:MAXPA]
IF ENTRY_STRUCT->FEATURES != 1 THEN
   Non triple fault Shutdown // On non X86-S fall back to legacy SIPI
FI
# note the order of these checks is not defined
newCR4 = ENTRY_STRUCT->CR4 # read entry_struct.CR4
newRIP = ENTRY_STRUCT->RIP # read entry_struct.RIP
newCR0 = ENTRY_STRUCT->CR0 # read entry_struct.CR0
newCR3 = ENTRY_STRUCT->CR3 # read entry_struct.CR3
IF newCR4.VME != 0 OR newCR4.PVI != 0 OR newCR4.DE != 1 OR newCR4.PSE != 0
  OR newCR4.PAE != 1 OR newCR4.UMIP != 1 OR
```

newCR4 has reserved bits set OR

newCR0.PE != 1 OR newCR0.MP != 1 OR newCR0.EM != 0 OR

newCR0.NE != 1 or newCR0.NW != 0 OR

newCR0.PG != 1 OR

newCR3 has reserved bits set depending on newCR4.PCIDE OR

newRIP is not canonical depending on newCR4.LA57 THEN

Unbreakable Shutdown

FI

CR4 = newCR4 ; CR3 = newCR3 ; CR0 = newCR0

Move received SIPI vector zero extended to R10

Force 64-bit supervisor mode

NMIs are blocked

RIP = newRIP

3.11 64-Bit Reset Through the Firmware Interface Table

The CPU starts executing in 64-bit paged mode after reset. The Firmware Interface Table (FIT) contains a reset state structure containing RIP and CR3 that defines the initial execution state of the CPU.

3.12 Removal of Fixed MTRRs

There is no support for fixed MTRRs. The FIX bit, bit[8] in the IA32_MTRRCAP register, is cleared and all the MTRR_FIX_* MSRs are not implemented. MTRR_DEF_TYPE bit[10] is reserved.

Table 6 lists the fixed MTRR MSRs removed.

Table 6. Removed MTRR MSRs

Name
IA32_MTRR_FIX64_00000
IA32_MTRR_FIX16_80000
IA32_MTRR_FIX16_a0000
IA32_MTRR_FIX4_c0000
IA32_MTRR_FIX4_c8000
IA32_MTRR_FIX4_d0000
IA32_MTRR_FIX4_d8000
IA32_MTRR_FIX4_e0000
IA32_MTRR_FIX4_e8000
IA32_MTRR_FIX4_f0000
IA32_MTRR_FIX4_f8000

3.13 Removal of XAPIC and ExtInt

The only way to access the X2APIC is through MSR accesses. Virtual XAPIC through VMX is still supported.

The CPU is always in x2APIC mode (IA32_APIC_BASE[EXTD] is 1) and is enabled. Attempts to write IA32_APIC_BASE to disable the APIC or leave x2APIC mode will cause a #GP(0) exception.

This will be enumerated to software through the IA32_XAPIC_DISABLE_STATUS[LEGACY_XAPIC_DISABLED] MSR bit being 1.

For more details, see https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/cpuid-enumeration-and-architectural-msrs.html

The ExtINT decoding in the local APIC is removed. If an interrupt is received with ExtInt encoding, it will trigger a Receive Error with bit[6] set in the APIC error status register. This slightly redefines the meaning of bit[6].

3.14 4L/5L Paging Switch

Switching between 4L and 5L paging uses two thread-scope MSRs: CR3_LOAD_4LPGTBL and CR3_LOAD_5LPGTBL. They act like a CR3 load and have the same layout as CR3, but also set or clear the CR4.5L bit as a side effect. When read, the MSRs always return the CR3 value. The MSRs are forbidden in VMX load/store lists or MSR list instructions.

Both CR3_LOAD_4LPGTBL and CR3_LOAD_5LPGTBL are only available when the CPU supports 5-level page tables as indicated by the LA57 CPUID feature bit and reports a LA57SWITCH CPUID feature bit in a future CPUID field.

3.15 Virtualization Changes

This section describes changes to the virtualization state.

"Fixed" fields are consistency checked and VM entry will fail if they do not match the fixed value.

3.15.1 VMCS Guest State

Guest VMCS field changes are listed in Table 7.

Table 7. VMCS Fields Changed (Guest State)

VMCS Field	Change	Reason
WFS encoding in Guest activity state	Fixed 0	64-bit SIPI does not support.

3.15.2 VMCS Exit Controls

VM exit controls that are changed are listed in Table 8.

Table 8. VMCS Exit Control Changes

VMCS Field	Change	Reason
Host Address Space Size (HASS)	Fixed 1	Host is always in 64-bit supervisor mode.
IA32 mode guest	Fixed 1	Guest is always in long mode.

3.15.3 VMCS Secondary Processor-Based Execution Controls

Changes are listed in Table 9.

Table 9. Secondary Processor-Based Execution Control Changes

VMCS Field	Change	Reason
Unrestricted guest	Fixed 0	Unrestricted guest not supported.

3.15.4 VMX Enumeration

Table 10 describes changes in VMX enumeration.

Table 10. VMX Enumeration Changes

MSR	Bit(s)	Corresponding Field	Value	Notes
IA32_VMX_EXIT_CTLS	9, 41	Host address space size	1	EFER LME and LMA are fixed to 1.
IA32_VMX_TRUE_EXIT_CTLS				
IA32_VMX_PROCBASED_CTLS2	39	Unrestricted guest	0	No unrestricted guest.
IA32_VMX_MISC	8	Supports activity state: wait-for-SIPI	0	Unsupported.
IA32_VMX_CR0_FIXED0	0	PE: Protected mode enable	1 (legacy)	Always long mode, no legacy FPU modes. Fixed to 0.
	1	MP: Monitor coprocessor	1	
	5	NE: Numeric Error	1 (legacy)	
	31	PG: Paging Enabled	1 (legacy)	
IA32_VMX_CR0_FIXED1	2	EM: FP emulation	0	

MSR	Bit(s)	Corresponding Field	Value	Notes
	29	NW: Not write through	0	These CR0 bits are fixed to 0
IA32_VMX_CR4_FIXED0				No changes
IA32_VMX_CR4_FIXED1	1	PVI: Protected-mode Virtual Interrupts	0	No support for Protected-mode virtual interrupts.

3.16 Summary of Removals

A summary of removals is given in Table 11.

Table 11. Summary of Removals

Removal of	Replacement	Implied by
Segment bases (except FS/GS/GDT/IDT/LDT/TSS), limits (except GDT/IDT/TSS/LDT) in 32-bit mode	-	Limited segmentation
Real mode (big and 16-bit)	64-bit paged mode, 64-bit SIPI	-
16-bit protected mode	-	-
16-bit address override in other modes when address is referenced	-	-
32-bit ring 0, including 2- and 3-level paging modes	64-bit ring 0	-
Disabling FPU through CR0.MP	-	-
Legacy numeric error handling	-	-
VM86 mode	-	16-bit mode removal
Protected-mode virtual interrupts (PVI)	-	-
Clearing EFER.NXE bit to disable presence of NX bit in page table entries	-	-
Disabling SYSCALL through EFER.SCE	-	-
FAR jumps changing rings	SYSCALL, INT	Limited segmentation
IRET/SYSCALL/SYSRET entering 16-bit mode, vm86 mode or conforming segments	-	16-bit mode removal, Limited segmentation
Fixed MTRRs	Variable MTRRs , PAT in page tables	-
MMIO-based XAPIC access	X2APIC access through MSRs	-
APIC ExtInt removal	-	-
Ring 1, ring 2 removal	-	-

Removal of	Replacement	Implied by
Ring 3 I/O port access (IOPL, I/O bitmap)	Ring 0 I/O port access	-
INS and OUTS instructions	IN, OUT instructions in loops	-
#SS exception	#GP(0) exception	Limited segmentation
#NP exception	#GP(0) exception	Limited segmentation
Support for INIT/SIPI on entry in VMCS	-	64-bit SIPI
Support for unrestricted guest in VMCS	-	16-bit mode removal, paging always enabled
VMCS support for 32-bit ring 0	-	32-bit ring 0 removal

3.17 Summary of Additions

New additions in the architecture are given in Table 12.

Table 12. Summary of Additions

Addition of	Reason	Needed by
64-bit SIPI and INIT	Boot APs in paged 64-bit mode	Real mode removal
4L/5L paging switch MSRs	No real mode	Real mode removal

3.18 Changed Instructions

The following descriptions pertain only to new behavior of the instructions. A longer list of segmentation-related instructions with changed behavior (if any) is shown in Section 4.3. Some instruction with trivial changes are only documented in the Summaries.

For legacy behavior, please refer to *Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4*.

3.18.1 SYSRET

SYSRET will generate a #GP(0) exception if a non-zero value is loaded into RFLAGS.IOPL, RFLAGS.VIP, or RFLAGS.VIF.

3.18.2 IRET

IRET cannot jump to 16-bit mode, task gates or conforming segments. IRET will generate a #GP(0) exception if a non-zero value is loaded into RFLAGS.IOPL, RFLAGS.VIP, or RFLAGS.VIF when in ring 0. The details of the IRET instruction are shown in the pseudocode in Section 4.2.6.

3.18.3 POPF – Pop Stack Into RFLAGS Register

Opcode	Instruction	Op/ En	64-Bit Mode	Compatibility/ Legacy Mode	Description
9D	POPFD	ZO	N.E.	Valid	Pop top of stack into EFLAGS.
9D	POPFQ	ZO	Valid	N.E.	Pop top of stack and zero-extend into RFLAGS.

POPF pops a doubleword (POPFD) from the top of the stack (if the current operand size attribute is 32) and stores the value in the EFLAGS register, or pops a word from the top of the stack (if the operand size attribute is 16) and stores it in the lower 16 bits of the EFLAGS register (that is, the FLAGS register). These instructions reverse the operation of the PUSHF/PUSHFD/PUSHFQ instructions.

The IOPL, VM, VIP, and VIF flags are always zero and are ignored on POPF.

The POPF instruction never raises an #SS exception, but only a #GP(0) or a #PF exception.

It changes RFLAGS according to Table 13.

Table 13. RFLAGS Changes with the POPF Instruction

Mode	Operand Size	CPL	Flags															
			21	20	19	18	17	16	14	13:12	11	10	9	8	7	6	4	2	0
			ID	VIP	VIF	AC	VM	RF	NT	IOPL	OF	DF	IF	TF	SF	ZF	AF	PF	CF
User32, User64 and Supervisor modes	32, 64	*	S	N	N	S	N	0	S	N	S	S	N	S	S	S	S	S	S

Key	
S	Updated from stack
N	No change in value
0	Value is cleared

Pseudocode:

```
tempFlags = POP // according to 32/64 operand size
IF CPL = 0 THEN
   // modify non reserved flags with “tempFlags” except RF, IOPL, VIP, VIF, VM.
   // RF is cleared.
   // Do not modify flags not popped due to operand size.
ELSE
   // modify non reserved flags with “tempFlags” except RF, IOPL, VIP, VIF, VM, IF.
   // RF is cleared.
   // Do not modify flags not popped due to operand size.
FI
```

Table 14 shows a summary of removed instructions.

Table 14. Removed Instructions

Instruction	Possible Legacy Usage in Rings	Replacement
INS / OUTS	Ring 3, 1, 2, 0	IN, OUT
64-bit indirect far jump with 0x67 prefix will #UD 32-bit near ret, far call, far ret, far jmp, with 0x67 prefix will #UD 32-bit near jmp, jCC, JECX*, near ret, near call, loop*, far jmp with 0x67 prefix will #UD	Ring 3, 1, 2, 0	32-bit/64-bit memory references

32-bit any instruction that references memory and is not a jump with 0x67 prefix will #GP		
LMSW	Ring 3, 1, 2, 0	Mov CR0

3.19 Summary of Changed Instructions

Table 15 shows a summary of changed instructions.

Table 15. Changed Instructions

Instruction	Rings	Change
ERETU	0	No support for 16-bit mode or VM86 or gates. Limited segmentation architecture. RFLAGS handling changes.
ERETS	0	RFLAGS handling changes.
IRET	3, 0	No support for 16-bit mode or vm86 or gates. Limited segmentation architecture. RFLAGS handling changes.
SYSRET	0	RFLAGS handling changes
SYSEXIT	0	RFLAGS handling changes
FAR CALL	3, 0	Limited segmentation architecture. #UD on 0x66 prefix. #GP on 0x67 in 32bit mode and indirect.
FAR JMP	3, 0	Limited segmentation architecture. #UD on 0x66 prefix. #GP on 0x67 in 32bit mode.
POPF	3, 0	RFLAGS handling changes.
FAR RET	3, 0	Limited segmentation architecture. #UD with 0x66 prefix
STI	3, 0	No support for ring 3 changes through VM86/PVI/IOPL.
CLI	3, 0	No support for ring 3 changes through VM86/PVI/IOPL.
ARPL, VERW, VERR, MOV to sel, MOV from sel, PUSH sel, POP sel, LAR, LGS, LFS, LES, LFS, LGS, LSS, LDS, LKGS	3, 0	Limited segmentation architecture
IN*, OUT*	3	Removed support for ring 3 port I/O
JMP short, JMP, LOOP, JECX, CALL, RET, JMP	3, 0	#UD with 0x66 prefix in 32bit mode

3.20 Software Compatibility Notes

3.20.1 Emulation of Ring 3 I/O Port Access

If there are legacy uses of ring 3 I/O port accesses using the TSS I/O port bitmap or IOPL, it is possible to emulate this case through a #GP(0) handler that executes IN/OUT in the kernel. INS/OUTS can be emulated in a #UD handler with an appropriate emulation routine.

3.20.2 64-Bit SIPI

The BIOS should always disable 64-bit SIPI in the SIPI_ENTRY_STRUCT ENABLES field before passing control to the OS. On a legacy architecture, this will ensure that a legacy OS can use legacy SIPI. A 64-bit-SIPI-aware OS can enable it. On X86-S it is not possible to use legacy SIPI, but the OS owns the enabling of 64-bit SIPI, too, for consistency.

The BIOS is always expected to point the IA32_SIPI_ENTRY_STRUCT_PTR to the SIPI_ENTRY_STRUCT on all packages before passing control to the OS.

3.20.3 Legacy OS Virtualization

The VMM is responsible for setting up the system state and VMCS appropriately so that the necessary VMexits and faults occur for cases where emulation of legacy behavior by the VMM is required. There are also cases where the VMM should not attempt to perform a VM entry, but instead emulate until a supported guest state is reached.

- System and VMCS state
    - Set CR0 and CR4 masks to ones for all fields corresponding to fields that have fixed values.
    - Unconditional I/O exiting set to 1 due to IOPL changes.
    - Unrestricted guest set to 0
- VM entry requirements
    - Guest state
        - Activity state: WFS should not be set. This is consistency checked on VM entry.
        - CR0: The following fields must be set as described. Emulation or running with shadow page table is required otherwise.
            - PE=1
            - EM=0
            - NE=1
            - PG=1
        - CR4: VME must be cleared. Emulation is required for all 16-bit modes.
        - EFER: LME/LMA must be set to one.
        - RFLAGS: VM must be cleared.
    - Secondary processor based execution controls. Unrestricted guest = 0 (unrestricted guest is not supported)
    - Exit controls. Host address space size = 1 (host is always 64-bit CPL0)
- Emulation
    - Non-flat segments (except for FS/GS base)
    - Real and virtual 8086 modes

If required for guest compatibility, the VMM is responsible for (a) setting the exception bitmap such that #UD and #GP cause a VMexit and then (b) emulating to determine the cause of the exception and appropriate response. Some examples:

- Some variants of CLI will spuriously #GP(0), for example, if a legacy guest tried to execute a CLI in ring 3 and RFLAGS.IOPL==3. Since RFLAGS.IOPL is always 0, this ring 3 CLI will always #GP(0). If the guest requires these IOPL semantics, it is up to the VMM to emulate this instruction with the emulated legacy guest RFLAGS.IOPL value. Note that that are un-virtualizable aspects of non-zero IOPL that are discussed later.
- #SS and #NP are converted to #GP. If the guest expects to see the #SS/#NP, the VMM will need to detect cases where a #GP would have been an #SS or #NP and inject them to the guest.

Some guest CR values are ignored on VMENTRY (they retain the fixed values and are not consistency checked). If required by the guest, the VMM can virtualize differences, some of which are described below. In general these are expected to be uncommon in Intel64 guests.

- CR0.MP is fixed to one.
  - VMM should diagnose and emulate spurious faulting cases.
- CR4.PVI is fixed to zero.
  - VMM can diagnose #GPs from STI/CLI and emulate expected guest behavior.
- CR4.DE is fixed to one.
  - VMM can diagnose and emulate spurious faulting cases.
- CR4.PSE and CR4.PAE are fixed. Legacy paging modes require shadow paging or emulation.
- EFER.LME: LME is fixed to one. If the guest is in 32-bit CPL0 mode and VMM wants to do a VMentry, it should use emulation.
- RFLAGS:
  - IOPL is fixed 0
  - VIF, VIP are fixed 0. Some CLI/STI may #GP(0) and can be emulated to handle these appropriately if the guest requires this functionality.

A VMM can choose to emulate legacy functionality as required:

1. VMM changes required for mainstream Intel64 guest using legacy SIPI or non-64-bit boot
   a. Emulate 16-bit modes (real mode, virtual 8086 mode)
   b. Emulate unpaged modes
   c. Emulate legacy INIT/SIPI
2. Optional VMM changes for handling uncommon cases
   a. IOPL != 0 (if guest wants ring 3 I/O port access or ring 3 CLI/STI)
      i. Catch CLI #GP in CPL3 and emulate
      ii. Catch STI #GP in CPL3 and emulate
      iii. Catch IN/OUT #GP in CPL3 and emulate
      iv. IRET in CPL0 will #GP if attempting to change IOPL, catch and emulate
      v. Note that that are un-virtualizable aspects of non-zero IOPL in the next section.
   b. INS/OUTS instructions are removed: Catch #UD and emulate
   c. Call gates: VMM needs to catch relevant #GPs and emulate
   d. #SS removal: VMM can catch relevant #GPs and report #SS back to guest
   e. #NP removal: VMM can catch relevant #GPs and report #NP back to guest
   f. CR4.PVI catch and emulate associated #GPs
   g. Emulate 16-bit addressing by catching #GPs/#UDs
   h. CR4.VME, RFLAGS.VM: Emulate v8086 mode
   i. Emulate 32-bit ring 0 and run 32-bit ring 3 with shadow paging in legacy paging modes
   j. Support for unsupported obscure segmentation features like expand down or non conforming code segments: Can be emulated by catching #GPs
3. Uncommon cases with expensive SW solutions:
   a. CPL1/2 requires partial emulation
   b. Non-flat CS/DS/ES/SS segments or setting access bits in descriptors in memory requires full emulation trigged by Descriptor Table Exiting and then setting GDT/LDT limit to zero (or read/write protect GDT/LDT) to catch segmentation instructions
   c. When EFER.NXE is cleared, a set NX bit in PTE requires shadow paging
   d. Segmentation permission checking on load/store/execute: Would require full emulation
4. Cases that are un-virtualizable

a. RFLAGS.IOPL != 0: When IOPL is non-zero most cases where behavior would typically change will instead #GP, which the VMM can catch/emulate (i.e., many cases are virtualizable). The problematic Intel64 cases are as follows.
    i. Ring 0 privileged SW sets IOPL to 3 and changes to ring 3. If ring 3 SW runs PUSHF or SYSCALL, the value with IOPL=3 should go into the memory or register destination. When this sequence runs in a VM, the ring 0 instruction that sets IOPL to 3 would cause #GP and trigger a VMExit. If the VMM resumes the VM with the "wrong" IOPL, i.e., IOPL==0, the ring 3 PUSHF or SYSCALL would expose this incorrect IOPL through memory or the register. Also, ring3 POPF will not update IF. The preferred scheme is for VMM to emulate the guest till IOPL is changed back to 0. This case is not expected on a modern OS.
    ii. If the guest attempts to set IOPL to a value greater than zero using a POPF instruction in ring 0, this will be silently ignored. The IOPL value will not be updated and the VMM will be unaware that this occurred. Some subsequent consumers of this value will #GP (e.g., CLI/STI/IN/OUT) but others will silently continue with different semantics (e.g., IF updating POPF, memory written by PUSHF, flags stored by SYSCALL, etc.)
b. #UD behavior on SYSCALL/SYSEXIT when EFER.SCE is cleared.

4 Appendix

This appendix gives further details on limited segmentation and exception compatibility.

4.1 Segmentation Instruction Behavior

Note the descriptions only describe the new behavior of the instructions. For legacy behavior please refer to the SDM. The pseudocode might not have the final fault ordering or error codes. If something is not changed from baseline, it will not be mentioned.

```
Check_selector(selector): // do not do segment load if its NULL
  IF CS AND selector is NULL THEN
    #GP(0); // OR VMEntry Bad Guest State
  FI
  IF (selector.TI == 0 AND selector exceeds GDT limit) OR
    (selector.TI == 1 AND selector exceeds LDT limit) OR
    Descriptor address in table is non canonical THEN
    #GP(selector); // OR VMEntry Bad Guest State OR ZF := 0
  FI
END

Check_CS_desc(selector, Descriptor, newCPL):
  IF Descriptor is not non-conforming code segment
  OR (Descriptor.L == 0 AND Descriptor.D == 0)  // prevents 16b size
  OR Descriptor.DPL == 1
  OR Descriptor.DPL == 2
  OR Descriptor.DPL != selector.RPL
  OR Descriptor.DPL != newCPL
  OR Descriptor.L == 0 AND Descriptor.DPL == 0 THEN // prevents SUP32
    #GP(selector);
  FI
  IF descriptor.P == 0 THEN
    #GP(selector);
  FI
  Descriptor.Accessed := 1 // not in memory
  Save_descriptor_for_VMX(selector, Descriptor);
END

Check_CS_desc_for_IRET(selector, Descriptor):
  IF Descriptor is not non-conforming code segment
  OR (Descriptor.L == 0 AND Descriptor.D == 0)  // prevents 16b size
```

```
   OR Descriptor.DPL == 1
   OR Descriptor.DPL == 2
   OR Descriptor.DPL != selector.RPL
   OR selector.RPL <  CPL
   OR Descriptor.L == 0 AND Descriptor.DPL == 0 THEN // prevents SUP32
      #GP(selector);
   FI
   IF descriptor.P == 0 THEN
      #GP(selector);
   FI
END

// newMode is current mode or mode from VMCS/SMM state
Check_Data_desc(selector, Descriptor, newMode):
   IF selector is not NULL THEN
      IF selector exceeds GDT/LDT limit // does not apply to VMEntry/RSM
      OR Descriptor is not data or readable non-conforming code segment
      OR Descriptor.DPL == 1
      OR Descriptor.DPL == 2
      OR Descriptor.DPL < selector.RPL
      OR Descriptor.DPL < CPL  THEN
         #GP(selector); // OR VMEntry Bad Guest State OR ZF := 0
      FI
      IF descriptor.P == 0 THEN
         #GP(selector); // OR VMEntry Bad Guest State OR ZF := 0
      FI
   FI
   Descriptor.Accessed := 1 // not in memory
   Save_descriptor_for_VMX(selector, Descriptor);
END

// newCPL is current CPL or newCPL from VMCS/SMM/IRET state
Check_SS_desc(selector, Descriptor, newCPL):
   IF NOT (selector is NULL AND newCPL != 3) THEN
      IF selector exceeds GDT/LDT limit
      OR ((selector is NULL) AND newCPL == 3)
      OR Descriptor is not writeable data segment
      OR Descriptor.DPL == 1
      OR Descriptor.DPL == 2
      // Descriptor.B=1 leads to #GP(0) deferred to first SS reference in 32bit mode
      OR Descriptor.DPL != selector.RPL
```

```
      OR Descriptor.DPL != CPL THEN
        #GP(selector); // OR VMEntry Bad Guest State OR ZF := 0
      FI
      IF descriptor.P == 0 THEN
        #GP(selector); // OR VMEntry Bad Guest State OR ZF := 0
      FI
    FI
    Descriptor.Accessed := 1 // not in memory
    Save_descriptor_for_VMX(selector, Descriptor);
END

Load_descriptor_from_GDT_LDT(selector):
    IF (selector & 0xFFF8) != 0x0 THEN
      IF selector.TI == 1 THEN BASE := LDT Base;
      ELSE BASE := GDT Base; FI;
      Desc := load_physical_sup(BASE + (selector & 0xFFF8));
      Set accessed bit in Descriptor copy, not in memory;
      Return Desc;
    ELSE
      Return 0;
    FI
END

Load_descriptor_from_IDT(vector):
    Desc := load physical_sup(IDT base + vector << 4);
    Return Desc;
END
```

4.2 Segmentation Instruction Pseudocode

4.2.1 CALL Far

Far CALLs are intra-level only. Mode restrictions are enforced. The selector must point to a non-conforming code descriptor in the GDT/LDT. The CS.accessed bit is not set. The new descriptor is saved for use in VMX. With 0x66 prefix instruction #UDs. With 0x67 prefix and indirect and in 32bit mode instruction #GP(0)s. #NP and #SS are replaced with #GP.

```
IF 0x66 prefix THEN #UD ; FI

IF 0x67 prefix AND indirect AND 32bit mode THEN #GP(0); FI

Check_selector(newCS);

newCSdesc := Load_descriptor_from_GDT_LDT(tempCS);

Check_CS_desc(tempCS, newCSdesc, CPL);

IF newRIP is non-cannonical THEN
```

```
   #GP(0)
FI
Push CS;
Push RIP;
CS  := newCS;
RIP := newRIP;
Save newCSdesc;
Do shadow stack pushes if enabled
Do end branch state transition if enabled
```

4.2.2 ERETU

Enforces RFLAGS restrictions. Mode restrictions are enforced, as well as limits on code selector types. No access bits for descriptors are set. #NP and #SS are replaced with #GP.

```
Beginning of flow the same as Intel64
// Intel64 FRED checks for CS/SS in STAR
ELSE IF newCS not in STAR OR newSS not in STAR THEN
   Check_selector(newCS);
   newCSdesc :=    Load_descriptor_from_GDT_LDT(newCS);
   // do modified selector checks
   Check_CS_desc_for_IRET(tempCS, newCSdesc, newCPL);
   // no limit check for CS even with CS.L = 0
   newSSdesc := Load_descriptor_from_GDT_LDT(newSS);
   Check_SS_selector(new_SS, newSSdesc, newCPL);
   Load new CS and new SS
FI
Rest of flow is same as Intel64
```

4.2.3 ERETS

Enforces RFLAGS restrictions.

4.2.4 FRED ENTRY FLOW

Enforces RFLAGS restrictions.

4.2.5 Int n, INT3, INTO, External Interrupt, Exceptions with CR4.FRED == 0

Mode restrictions and descriptor type restrictions are enforced. Access bits for descriptors. are not set. #NP is replaced with #GP.

```
IF INTO and CS.L = 1 THEN
   #UD;
FI;
IF ((vector_number « 4) + 15) is not in IDT.limit THEN
   #GP(error_code(vector_number,1,EXT));
```

```
FI;
gate := Read_descriptor_from_IDT(vector_number);
IF gate.type not in {intGate64, trapGate64} THEN
   #GP(error_code(vector_number,1,EXT));
FI;
IF software interrupt (* does not apply to INT1 *) THEN
   IF gate.DPL < CPL THEN
      #GP(error_code(vector_number,1,0));
   FI;
FI;
IF gate.P == 0 THEN
   #GP(error_code(vector_number,1,EXT));
FI
newCS := gate.selector;
IF newCS is NULL THEN
   #GP(EXT); (* Error code contains NULL selector *)
FI;
Check_selector(newCS);
newCSdesc := Load_descriptor_from_GDT_LDT(newCS);
Check_CS_desc(newCS, newCSdesc, 0);
IF newCSdesc.DPL < CPL THEN
   GOTO INTER-PRIVILEGE-LEVEL-INTERRUPT;
ELSIF newCSdesc.DPL = CPL THEN
   GOTO INTRA-PRIVILEGE-LEVEL-INTERRUPT;
ELSE
   #GP(error_code(new code-segment selector,0,EXT));
FI
END;
INTER-PRIVILEGE-LEVEL-INTERRUPT:
   IF gate.IST == 0 THEN
      TSSstackAddress := (newCSdesc.DPL « 3) + 4;
   ELSE
      TSSstackAddress := (gate.IST « 3) + 28;
   FI;
   IF (TSSstackAddress + 7) > TSS.limit THEN
      #TS(error_code(TSS.selector,0,EXT);
   FI;
   NewRSP := 8 bytes loaded from (TSS.base + TSSstackAddress);
   NewSS  := newCSdesc.DPL; (* NULL selector with RPL = new CPL *)
   IF gate.IST = 0 THEN
      NewSSP := IA32_PLi_SSP; (* where i = newCSdesc.DPL *)
```

```
ELSE
    NewSSPAddress := IA32_INTERRUPT_SSP_TABLE_ADDR + (gate.IST « 3);
    IF ShadowStackEnabled(CPL0) THEN
            NewSSP := 8 bytes loaded from NewSSPAddress;
    FI;
FI;
IF NewRSP is non-canonical THEN
    #GP(EXT); (* Error code contains NULL selector *)
FI;
IF gate.IP is non-canonical THEN
    #GP(EXT);  (* Error code contains NULL selector *)
FI;
RSP  := NewRSP & FFFFFFFFFFFFFF0H;
SS   := NewSS;
SSdesc := const;
Push(SS);
Push(RSP);
Push(RFLAGS); (* 8-byte push *)
Push(CS);
PUSH(RIP);
Push(ErrorCode); (* If needed, 8-bytes *)
RIP := gate.RIP;
CS  := newCS;
IF ShadowStackEnabled(CPL) AND CPL == 3 THEN
    IA32_PL3_SSP := LA_adjust(SSP);
FI;
CPL     := newCSdesc.DPL;
CS.RPL  := CPL;
IF ShadowStackEnabled(CPL) THEN
    oldSSP := SSP
    SSP    := NewSSP
    IF (SSP & 0x07 != 0) THEN
            #GP(0);
    FI
    IF (CS.L = 0 AND SSP[63:32] != 0) THEN
       #GP(0);
    FI
     FI;
expected_token_value := SSP;          (* busy bit- must be clear *)
new_token_value      := SSP | BUSY_BIT; (* Set the busy bit *)
IF (shadow_stack_lock_cmpxchg8b(SSP, new_token_value,
```

```
                        expected_token_value) !=
      expected_token_value) THEN
      #GP(0);
   FI;
   IF oldSS.DPL != 3
      ShadowStackPush8B(oldCS);
      ShadowStackPush8B(oldRIP);
      ShadowStackPush8B(oldSSP);
   FI;
   IF EndbranchEnabled (CPL)
      IA32_S_CET.TRACKER = WAIT_FOR_ENDBRANCH;
      IA32_S_CET.SUPPRESS = 0
   FI;
   IF gate.type is intGate64 THEN
      RFLAGS.IF := 0 (* Interrupt flag set to 0, interrupts disabled *);
   FI;
   RFLAGS.TF := 0;
   RFLAGS.RF := 0;
   RFLAGS.NT := 0;
END;

INTRA-PRIVILEGE-LEVEL-INTERRUPT:
   NewSSP       := SSP;
   CHECK_SS_TOKEN := 0;
   IF gate.IST != 0 THEN
      TSSstackAddress := (IDT-descriptor IST « 3) + 28;
      IF (TSSstackAddress + 7) > TSS.limit  THEN
         #TS(error_code(current TSS selector,0,EXT));
      FI;
      NewRSP := 8 bytes loaded from (current TSS base +
                           TSSstackAddress);
   ELSE
      NewRSP := RSP;
   FI;
   IF ShadowStackEnabled(CPL) THEN
    NewSSPAddress  := IA32_INTERRUPT_SSP_TABLE_ADDR + (IDT gate IST « 3)
    NewSSP       := 8 bytes loaded from NewSSPAddress
    CHECK_SS_TOKEN := 1
   FI;
   IF NewRSP is non-canonical THEN
      #GP(EXT); (* Error code contains NULL selector *)
```

```
  FI;
  IF gate.RIP is non-canonical THEN
    #GP(EXT); (* Error code contains NULL selector *)
  FI;
  RSP := NewRSP & FFFFFFFFFFFFFFF0H;
  Push(SS);
  Push(RSP);
  Push(RFLAGS); // 8-byte push – including .IF, not affected by IOPL,CPL
  Push(CS);
  PUSH(RIP);
  Push(ErrorCode); (* If needed, 8-bytes *)
  oldCS  := CS;
  oldRIP := RIP;
  RIP := gate.RIP;
  CS  := newCS;
  CS.RPL := CPL;
  IF ShadowStackEnabled(CPL) AND CHECK_SS_TOKEN == 1 THEN
    IF NewSSP & 0x07 != 0 THEN
      #GP(0);
        FI;
    IF (CS.L = 0 AND NewSSP[63:32] != 0) THEN
          #GP(0);
    FI;
    expected_token_value := NewSSP (* busy bit – (0)- must be clear *)
    new_token_value      := NewSSP | BUSY_BIT (* Set the busy bit *)
    IF shadow_stack_lock_cmpxchg8b(NewSSP, new_token_value,
                          expected_token_value) !=
      expected_token_value THEN
           #GP(0);
    FI;
  FI;
  IF ShadowStackEnabled(CPL) THEN
    (* Align to next 8 byte boundary *)
    tempSSP = SSP;
    Shadow_stack_store 4 bytes of 0 to (NewSSP – 4)
    SSP := newSSP & 0xFFFFFFFFFFFFFFF8H;
    ShadowStackPush8B(oldCS);
    ShadowStackPush8B(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
      RFLAGS.IF := 0; (* Interrupt flag set to 0; interrupts disabled *)
   FI;
   RFLAGS.TF := 0;
   RFLAGS.NT := 0;
   RFLAGS.RF := 0;
END;
```

4.2.6 IRET

IRET cannot enter 16-bit mode, VM86 mode, task gates or conforming segments. Descriptor access bits are not set. Mode restrictions are enforced. #NP and #SS are replaced with #GP.

```
IF RFLAGS.NT == 1 THEN
   #GP(0);
FI
tempRIP := POP(); // according to operand size
tempCS := POP(); // according to operand size
tempFlags := POP();// according to operand size

IF CPL == 3 THEN
   tempFlags(VIP, VIF, IOPL) := (0, 0, 0);
ELSIF tempFlags(VIF, VIP, IOPL) != (0,0,0) THEN
    #GP(0);
FI

Check_selector(tempCS);
Descriptor := Load_descriptor_from_GDT_LDT(tempCS);
Check_CS_desc_for_IRET(tempCS, Descriptor);
IF tempCS.RPL > CPL THEN
   IF CR4.FRED THEN
      #GP(tempCS);
   ELSE
      GOTO RETURN_TO_OUTER_PRIVLEDGE_LEVEL; // must be level 3
   FI
ELSIF Started_in_64b_mode THEN
   GOTO RETURN_FROM_IA32e;
ELSE
   GOTO RETURN_FROM_SAME_PRIVLEDGE_LEVEL;
FI
```

```
RETURN_FROM_SAME_PRIVLEDGE_LEVEL:
  IF tempRIP is not canonical THEN
    #GP(0); // Restoring RSP
  FI
  IF ShadowStackEnabled(CPL) THEN
    Perform normal Shadow Stack operations as described in the SDM;
  FI
  CS := tempCS;
  RIP := tempRIP;
  Save_descriptor_for_VMX(CS, Descriptor);
  RFLAGS(CF, PF, AF, ZF, SF, TF, DF, OF, NT, RF, AC, IC) := tempFlags;
  IF CPL == 0 THEN RFLAGS(IF) := tempFlags; FI;
  Unmask NMI;
END;

RETURN_FROM_IA32e:
  tempRSP := POP();
  tempSS := POP();
  Check_sel(tempSS);
  tempSSdesc := Load_descriptor_from_GDT_LDT(tempSS);
  check_SS_desc(tempSS, tempSSdesc, newCPL);
  SS  := tempSS;
  RSP := tempRSP;
  GOTO RETURN_FROM_SAME_PRIVLEDGE_LEVEL;

RETURN_TO_OUTER_PRIVLEDGE_LEVEL:
  IF tempCS.RPL != 3 THEN
     #GP(tempCS);
  FI
  tempRSP := POP();
  tempSS := POP();

  IF tempRIP is not canonical THEN
    #GP(0); // Restoring RSP
  FI
  CPL := tempCS.RPL;
  IF ShadowStackEnabled() THEN
     Perform normal Shadow Stack operations as described in the SDM;
  FI
  Check_selector(tempSS);
  tempSSdesc := Load_descriptor_from_GDT_LDT(tempSS);
  check_SS_desc(tempSS, tempSSdesc, newCPL);
  CS  := tempCS;
  RIP := tempRIP;
  SS  := tempSS;
  RSP := tempRSP;
  Save_descriptor_for_VMX(CS, Descriptor);
  Nullify selectors whose descriptor.DPL < CPL;
```

```
  RFLAGS(CF, PF, AF, ZF, SF, TF, DF, OF, NT, RF, AC, IC) := tempFlags;
  IF CPL == 0 THEN RFLAGS(IF) := tempFlags; FI;
  Unmask NMI;
END;
```

4.2.7 JMP Far

Far JMPs are intra-level only. Mode restrictions are enforced. The selector must point to a non-conforming code descriptor in the GDT/LDT. The CS.accessed bit will not be set. The new descriptor is saved for use in VMX. With 0x66 prefix instruction #UDs. With 0x67 prefix in 32bit mode instruction #GP(0).

Pseudocode:

```
IF 0x66 prefix THEN #UD ; FI


IF 0x67 prefix AND 32bit mode THEN #GP(0); FI
Check_selector(newCS);
newCSdesc := Load_descriptor_from_GDT_LDT(tempCS);
Check_CS_desc(tempCS, newCSdesc, CPL);
IF newRIP is non-cannonical THEN
  #GP(0);
FI
CS  := newCS;
RIP := newRIP;
Save newCSdesc;
Do shadow stack pushes if enabled;
Do end branch state transition if enabled;
```

4.2.8 LSL, LAR, VERW, VERR

Uses modified selector load architecture. LAR forces the access bit to 1.

```
Check_Selector(selector);
// If failure return with ZF := 0
Desc := Load_descriptor_from_GDT_LDT(selector);
// If failure return with ZF := 0
Check_Data_Desc(selector, Desc, CPL);
// if failure return with ZF := 0
// For LAR always return Access = 1
// LSL/LAR/VERW/VERR flow to return information from Desc
```

4.2.9 LDS, LES, LFS, LGS, LSS

The Desc.accessed bit will not be set. The new descriptor is saved for use in VMX. Uses modified data segment load architecture.

Pseudocode:

```
If newSel is NULL AND LSS AND NOT (CPL0 AND CS.L) THEN
   #GP(0);
FI
Check_selector(newSel);
newDesc := Load_descriptor_from_GDT_LDT(newSel);
IF LSS THEN
   Check_SS_desc(newSEL, newDesc);
ELSE
   Check_Data_desc(newSel, newDesc);
Dest(sel) := newSel;
Dest(offset) := offest;
Save newDesc;
```

4.2.10 LGDT

Behaves as described in the SDM.

4.2.11 LLDT

Behaves as described in the SDM.

4.2.12 LIDT

Behaves as described in the SDM.

4.2.13 LKGS

Follows modified selector load checks, similar to MOV to segment register below.

4.2.14 LSL

Behaves as described in the SDM.

4.2.15 LTR

Behaves as described in the SDM.The TSS busy bit is set.

4.2.16 MOV from Segment Register

Behaves as described in the SDM.

4.2.17 MOV to Segment Register

The Desc.accessed bit is set in internal cache of the descriptor, but not in memory. The new descriptor is saved for use in VMX. Uses modified data segment load procedure.

```
If newSel is NULL AND MOV SS AND NOT (CPL0 AND CS.L) THEN
   #GP(0);
```

```
FI
Check_selector(newSel);
newDesc := Load_descriptor_from_GDT_LDT(newSel);
IF MOV SS THEN
   Check_SS_desc(newSEL, newDesc);
ELSE
   Check_Data_desc(newSel, newDesc);
Dest(sel) := newSel;
Save newDesc;
IF SS THEN MOV SS instruction blocking FI;
```

4.2.18 POP Segment Register

The Desc.accessed bit is set in internal cache of the descriptor, but not in memory. The new descriptor is saved for use in VMX.

```
IF 64b mode and POP DS, POP ES, POP SS THEN
   #UD;
FI
newSel := POP
Check_selector(newSel);
newDesc := Load_descriptor_from_GDT_LDT(newSel);
IF POP SS THEN
   Check_SS_desc(newSEL, newDesc);
ELSE
   Check_Data_desc(newSel, newDesc);
Dest:= newSel;
Save newDesc;
IF SS THEN Do POP SS blocking; FI
```

4.2.19 POPF

Will ignore attemps to change IOPL in CPL0.

4.2.20 PUSH Segment Selector

Behaves as described in the SDM.

4.2.21 PUSHF

Behaves as described in the SDM.

4.2.22 RDFSBASE, RDGSBASE

Behaves as described in the SDM.

4.2.23 RET far

Far RETs are intra-level only. The selector must point to a non-conforming code descriptor in the GDT/LDT. The CS.accssed bit is not set. The new descriptor is saved for use in VMX. With 0x66 prefix instruction #UDs.

Pseudocode:

```
IF 0x66 prefix THEN #UD ; FI
newRIP := POP;
newCS  := POP;
If newCS is NULL THEN
   #GP(0)
FI
Check_selector(newCS);
newCSdesc := Load_descriptor_from_GDT_LDT(tempCS);
Check_CS_desc(tempCS, newCSdesc, CPL);
IF newRIP is non-cannonical THEN
   #GP(0)
FI
CS  := newCS;
RIP := newRIP;
Save newCSdesc;
Do shadow stack if enabled
Do end branch state transition if enabled
```

4.2.24 SGDT

Behaves as described in the SDM.

4.2.25 SLDT

Behaves as described in the SDM.

4.2.26 SIDT

Behaves as described in the SDM.

4.2.27 STR

Behaves as described in the SDM.

4.2.28 SWAPGS

Behaves as described in the SDM.

4.2.29 SYSCALL

Behaves as described in the SDM and FRED EAS, except for enforcing RFLAGS restrictions.

4.2.30 SYSENTER

Behaves as described in the SDM.

4.2.31 SYSEXIT

Behaves as described in the SDM.

4.2.32 SYSRET

Behaves as described in the SDM and FRED EAS, except it faults if incoming RFLAGS (R11) has VIF, VIP, or IOPL != 0.

4.2.33 WRFSBASE, WRGSBASE

Behaves as described in the SDM.

4.2.34 VERR

Behaves as described in the SDM, but follows the modified segment load procedures documented above.

4.2.35 VERW

Behaves as described in the SDM, but follows the modified segment load procedures documented above.

4.2.36 VMEntry

Check_CS_Desc(newCS, newDesc, newCPL)
// If failure report VMEntry entry error
Check_SS_Desc(newSS, newDesc, newCPL)
// If failure report VMEntry entry error
For all of DS/ES/FS/GS:
    Check_Data_Selector(newSelector, newCPL)
    // If failure report VMEntry entry error
Load all descriptors as in 64bit mode.

4.3 List of Segmentation Instructions and Associated Behavior

Table 16 lists the segmentation instructions and associated behavior.

Table 16. List of Segmentation Instructions

Instruction	Behavior
SGDT	No change to Intel64 behavior.

Instruction	Behavior
SIDT	No change to Intel64 behavior.
SLDT	No change to Intel64 behavior.
STR	No change to Intel64 behavior.
LGDT	No change to Intel64 behavior.
LIDT	No change to Intel64 behavior.
LLDT	No change to Intel64 behavior.
LTR	No change to Intel64 behavior.
VERR	Behavior changed to follow the modified segmentation architecture described in Section 4.1.
VERW	Behavior changed to follow the modified segmentation architecture described in Section 4.1.
ARPL	No change to Intel64 behavior.
FAR CALL	Behavior changed to follow the modified segmentation architecture described in Section 4.2. #UD on 0x66 prefix. #GP on 0x67 prefix in 32bit mode and indiret.
FAR JMP	Behavior changed to follow the modified segmentation architecture described in Section 4.2. #UD on 0x66 prefix. #GP on 0x67 prefix in 32bit mode
FAR RET	Behavior changed to follow the modified segmentation architecture described in Section 4.2. #UD on 0x66 prefix.
IRET	The behavior of the IRET instruction is described in Section 4.2.
LDS	Load far pointer in DS with the segment check rules described in Section 4.1.
LES	Load far pointer in ES with the segment check rules described in Section 4.1.
LFS	Load far pointer in FS with the segment check rules described in Section 4.1.
LGS	Load far pointer in GS with the segment check rules described in Section 4.1.
LSS	Load far pointer in SS with the segment check rules described in Section 4.1.
LKGS	Move to Kernel GS Base with the segment check rules described in Section 4.1.
MOV to DS	Move to DS with the segment check rules described in Section 4.1.
MOV to ES	Move to ES with the segment check rules described in Section 4.1.
MOV to SS	Move to SS with the segment check rules described in Section 4.1.
MOV to FS	Move to FS with the segment check rules described in Section 4.1.
MOV to GS	Move to GS with the segment check rules described in Section 4.1.
MOV from DS	No change to Intel64 behavior.
MOV from ES	No change to Intel64 behavior.
MOV from SS	No change to Intel64 behavior.
MOV from FS	No change to Intel64 behavior.
MOV from GS	No change to Intel64 behavior.
POP DS	Pop top of stack into DS with the segment check rules described in Section 4.1.
POP ES	Pop top of stack into ES with the segment check rules described in Section 4.1.
POP SS	Pop top of stack into SS with the segment check rules described in Section 4.1.
POP FS	Pop top of stack into FS with the segment check rules described in Section 4.1.
POP GS	Pop top of stack into GS with the segment check rules described in Section 4.1.
PUSH CS	No change to Intel64 behavior.
PUSH DS	No change to Intel64 behavior.
PUSH ES	No change to Intel64 behavior.
PUSH SS	No change to Intel64 behavior.

Instruction	Behavior
PUSH FS	No change to Intel64 behavior.
PUSH GS	No change to Intel64 behavior.
SWAPGS	No change to Intel64 behavior.
RSM	No changes to segmentation, but enforces other mode and RFLAGS restrictions.
WRFSBASE	No change to Intel64 behavior.
WRGSBASE	No change to Intel64 behavior.
RDFSBASE	No change to Intel64 behavior.
RDGSBASE	No change to Intel64 behavior.
SYSENTER	Cannot enter 16-bit mode or VM86.
SYSEXIT	Cannot enter 16-bit mode or VM86.
SYSCALL	Enforces RFLAGS restrictions.
SYSRET	Enforces RFLAGS restrictions.
ERETU	Modified segment check rules. Enforces RFLAGS restrictions.
FRED entry	No change to Intel64 behavior.
IDT entry	Modified segment check rules and FRED restrictions.

4.4 64-Bit SIPI / 5L Page Switch Without LEGACY_REDUCED_ISA

64bit SIPI can be implemented on systems that don’t set the LEGACY_REDUCED_ISA bit to allow compatibility to X86-S systems. In this case, not enabling 64-bit SIPI in the IA32_SIPI_ENTRY_STRUCT_PTR or in the SIPI_ENTRY_STRUCT FEATURES bit will fall back to legacy INIT/SIPI.

Similarly, the 5-level page switch MSRs can be implemented on non-X86-S systems using its own CPUID bit to allow compatibility.

(This page intentionally left blank)