CLUI — Clear User Interrupt Flag

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 01 EE CLUI	ZO	V/I	UINTR	Clear user interrupt flag; user interrupts blocked when user interrupt flag cleared.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA	NA

Description

CLUI clears the user interrupt flag (UIF). Its effect takes place immediately: a user interrupt cannot be delivered on the instruction boundary following CLUI.

An execution of CLUI inside a transactional region causes a transactional abort; the abort loads EAX as it would have had it been caused due to an execution of CLI.

Operation

```
UIF := 0;
```

Flags Affected

None.

Protected Mode Exceptions

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

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

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

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used. If executed inside an enclave. If CR4.UINTR = 0. If CPUID.07H.0H:EDX.UINTR[bit 5] = 0.

ENQCMD — Enqueue Command

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 38 F8 !(11):rrr:bbb ENQCMD r32/r64, m512	A	V/V	ENQCMD	Atomically enqueue 64-byte user command with PASID from source memory operand to destination offset in ES segment specified in register operand as offset in ES segment.

Instruction Operand Encoding

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

Description

The ENQCMD instruction allows software to write commands to **enqueue registers**, which are special device registers accessed using memory-mapped I/O (MMIO).

Enqueue registers expect writes to have the following format:

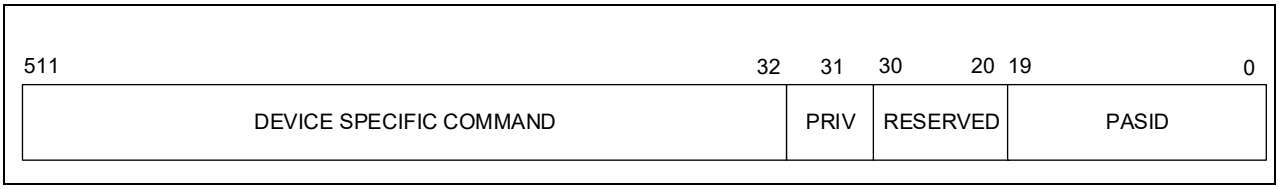

Figure 2-1. 64-Byte Data Written to Enqueue Registers

Bits 19:0 convey the process address space identifier (PASID), a value which system software may assign to individual software threads. Bit 31 contains privilege identification (0 = user; 1 = supervisor). Devices implementing enqueue registers may use these two values along with a device-specific command in the upper 60 bytes. Chapter 4 provides more details regarding how ENQCMD uses PASIDs.

The ENQCMD instruction begins by reading 64 bytes of command data from its source memory operand. This is an ordinary load with cacheability and memory ordering implied normally by the memory type. The source operand need not be aligned, and there is no guarantee that all 64 bytes are loaded atomically.

The instruction then formats those 64 bytes into **command data** with a format consistent with that given in Figure 2-1:

- Command[19:0] get IA32_PASID[19:0].[1]
- Command[30:20] are zero.
- Command[31] is 0 (indicating user).
- Command[511:32] get bits 511:32 of the source operand that was read from memory.

(The instruction ignores bits 31:0 of the source operand.)

The ENQCMD instruction uses an **enqueue store** (defined below) to write this command data to the destination operand. The address of the destination operand is specified in a general-purpose register as an offset into the ES segment (the segment cannot be overridden).[2] The destination linear address must be 64-byte aligned. The operation of an enqueue store disregards the memory type of the destination memory address.

1. It is expected that system software will load the IA32_PASID MSR so that bits 19:0 contain the PASID of the current software thread. The MSR's valid bit, IA32_PASID[31], must be 1. The PASID MSR is discussed in more detail in Section 4.1.
2. In 64-bit mode, the width of the register operand is 64 bits (32 bits with a 67H prefix). Outside 64-bit mode when CS.D = 1, the width is 32 bits (16 bits with a 67H prefix). Outside 64-bit mode when CS.D=0, the width is 16 bits (32 bits with a 67H prefix).

An enqueue store is not ordered relative to older stores to WB or WC memory (including non-temporal stores) or to executions of the CLFLUSHOPT or CLWB (when applied to addresses other than that of the enqueue store). Software can enforce such ordering by executing a fencing instruction such as SFENCE or MFENCE before the enqueue store.

An enqueue store does not write the data into the cache hierarchy, nor does it fetch any data into the cache hierarchy. An enqueue store's command data is never combined with that of any other store to the same address.

Unlike other stores, an enqueue store returns a status, which the ENQCMD instruction loads into the ZF flag in the RFLAGS register:

- ZF = 0 (success) reports that the 64-byte command data was written atomically to a device's enqueue register and has been accepted by the device. (It does not guarantee that the device has acted on the command; it may have queued it for later execution.)
- ZF = 1 (retry) reports that the command data was not accepted. This status is returned if the destination address is an enqueue register but the command was not accepted due to capacity or other temporal reasons. This status is also returned if the destination address was not an enqueue register (including the case of a memory address); in these cases, the store is dropped and is written neither to MMIO nor to memory.

Availability of the ENQCMD instruction is indicated by the presence of the CPUID feature flag ENQCMD (CPUID.(EAX=07H, ECX=0H):ECX[bit 29]).

Operation

```
IF IA32_PASID[31] = 0
    THEN #GP;
ELSE
    COMMAND := (SRC & ~FFFFFFFFH) | (IA32_PASID & FFFFFH);
    DEST := COMMAND;
FI;
```

Intel C/C++ Compiler Intrinsic Equivalent

ENQCMD int_enqcmd(void *dst, const void *src)

Flags Affected

The ZF flag is set if the enqueue-store completion returns the retry status; otherwise it is cleared. All other flags are cleared.

SIMD Floating-Point Exceptions

None.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If destination linear address is not aligned to a 64-byte boundary.
	If the PASID Valid field (bit 31) is 0 in IA32_PASID MSR.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#UD	If CPUID.07H.0H:ECX.ENQCMD[bit 29] = 0.
	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside the effective address space from 0 to FFFFH.
	If destination linear address is not aligned to a 64-byte boundary.
	If the PASID Valid field (bit 31) is 0 in IA32_PASID MSR.
#UD	If CPUID.07H.0H:ECX.ENQCMD[bit 29] = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

Same exceptions as in real-address mode. Additionally:

#PF(fault-code)	For a page fault.

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in non-canonical form.
#GP(0)	If the memory address is in non-canonical form. If destination linear address is not aligned to a 64-byte boundary. If the PASID Valid field (bit 31) is 0 in IA32_PASID MSR.
#PF(fault-code)	For a page fault.
#UD	If CPUID.07H.0H:ECX.ENQCMD[bit 29]. If the LOCK prefix is used.

ENQCMDS — Enqueue Command Supervisor

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 38 F8 !(11):rrr:bbb ENQCMDS r32/r64, m512	A	V/V	ENQCMD	Atomically enqueue 64-byte command from source memory operand to destination offset in ES segment specified in register operand as offset in ES segment.

Instruction Operand Encoding

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

Description

The ENQCMDS instruction allows system software to write commands to **enqueue registers**, which are special device registers accessed using memory-mapped I/O (MMIO).

Enqueue registers expect writes to have the format given in Figure 2-1 and explained in the section on "ENQCMD — Enqueue Command."

The ENQCMDS instruction begins by reading 64 bytes of command data from its source memory operand. This is an ordinary load with cacheability and memory ordering implied normally by the memory type. The source operand need not be aligned, and there is no guarantee that all 64 bytes are loaded atomically.

ENQCMDS formats its source data differently from ENQCMD. Specifically, it formats them into **command data** as follows:

- Command[19:0] get bits 19:0 of the source operand that was read from memory. These 20 bits communicate a process address-space identifier (PASID). Chapter 4 provides more details regarding how ENQCMDS uses PASIDs.
- Command[30:20] are zero.
- Command[511:31] get bits 511:31 of the source operand that was read from memory. Bit 31 communicates a privilege identification (0 = user; 1 = supervisor).

(The instruction ignores bits 30:20 of the source operand.)

The ENQCMDS instruction then uses an **enqueue store** (defined below) to write this command data to the destination operand. The address of the destination operand is specified in a general-purpose register as an offset into the ES segment (the segment cannot be overridden).[1] The destination linear address must be 64-byte aligned. The operation of an enqueue store disregards the memory type of the destination memory address.

An enqueue store is not ordered relative to older stores to WB or WC memory (including non-temporal stores) or to executions of the CLFLUSHOPT or CLWB (when applied to addresses other than that of the enqueue store). Software can enforce such ordering by executing a fencing instruction such as SFENCE or MFENCE before the enqueue store.

An enqueue store does not write the data into the cache hierarchy, nor does it fetch any data into the cache hierarchy. An enqueue store's command data is never combined with that of any other store to the same address.

Unlike other stores, an enqueue store returns a status, which the ENQCMDS instruction loads into the ZF flag in the RFLAGS register:

- ZF = 0 (success) reports that the 64-byte command data was written atomically to a device's enqueue register and has been accepted by the device. (It does not guarantee that the device has acted on the command; it may have queued it for later execution.)
- ZF = 1 (retry) reports that the command data was not accepted. This status is returned if the destination address is an enqueue register but the command was not accepted due to capacity or other temporal reasons.

1. In 64-bit mode, the width of the register operand is 64 bits (32 bits with a 67H prefix). Outside 64-bit mode when CS.D = 1, the width is 32 bits (16 bits with a 67H prefix). Outside 64-bit mode when CS.D=0, the width is 16 bits (32 bits with a 67H prefix).

This status is also returned if the destination address was not an enqueue register (including the case of a memory address); in these cases, the store is dropped and is written neither to MMIO nor to memory.

The ENQCMDS instruction may be executed only if CPL = 0. Availability of the ENQCMDS instruction is indicated by the presence of the CPUID feature flag ENQCMD (CPUID.(EAX=07H, ECX=0H):ECX[bit 29]).

Operation

```
DEST := SRC & ~7FF00000H;          // clear bits 30:20
```

Intel C/C++ Compiler Intrinsic Equivalent

ENQCMDS int_enqcmds(void *dst, const void *src)

Flags Affected

The ZF flag is set if the enqueue-store completion returns the retry status; otherwise it is cleared. All other flags are cleared.

SIMD Floating-Point Exceptions

None.

Protected Mode Exceptions

#GP(0)	For an illegal memory operand effective address in the CS, DS, ES, FS or GS segments.
	If destination linear address is not aligned to a 64-byte boundary.
	If the current privilege level is not 0.
#SS(0)	For an illegal address in the SS segment.
#PF(fault-code)	For a page fault.
#UD	If CPUID.07H.0H:ECX.ENQCMD[bit 29] = 0.
	If the LOCK prefix is used.

Real-Address Mode Exceptions

#GP	If any part of the operand lies outside the effective address space from 0 to FFFFH.
	If destination linear address is not aligned to a 64-byte boundary.
#UD	If CPUID.07H.0H:ECX.ENQCMD[bit 29] = 0.
	If the LOCK prefix is used.

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#SS(0)	If a memory address referencing the SS segment is in non-canonical form.
#GP(0)	If the memory address is in non-canonical form.
	If destination linear address is not aligned to a 64-byte boundary.
	If the current privilege level is not 0.
#PF(fault-code)	For a page fault.
#UD	If CPUID.07H.0H:ECX.ENQCMD[bit 29].
	If the LOCK prefix is used.

HRESET — History Reset

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 3A F0 C0 /ib HRESET imm8, <EAX>	A	V/V	HRESET	Processor history reset request. Controlled by the EAX implicit operand.

Instruction Operand Encoding

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

Description

Provides a hint to the processor to selectively reset the prediction history of the current logical processor. HRESET operation is controlled by the implicit EAX operand. The value of the explicit imm8 operand is ignored.

CPUID.07H.01H:EAX.HRESET[bit 22] indicates support of the HRESET instruction. This instruction can only be executed at CPL 0.

The HRESET instruction is capable of providing a reset hint for multiple predictions. Prior to the execution of HRESET, the system software must take the following steps:

1. Enumerate the HRESET capabilities via CPUID.20H.0H:EBX, which indicates what predictions can be reset.
2. Opt-in to reset a subset of the available capabilities by setting the respective bits in the IA32_HRESET_ENABLE MSR. The opt-in bits in the IA32_HRESET_ENABLE MSR are aligned with the HRESET capabilities CPUID bits.

The implicit EAX operand must contain set bits that are a subset of those set in the IA32_HRESET_ENABLE MSR. Otherwise, HRESET generates #GP(0). When EAX=0 this instruction is interpreted as NOP.

Any attempt to execute the HRESET instruction inside a transactional region will result in a transaction abort.

Operation

```
IF EAX = 0
   THEN NOP
   ELSE
      FOREACH i such that EAX[i] = 1
         Reset prediction history for feature i
FI
```

Flags Affected

None.

Protected Mode Exceptions

#GP(0)	If CPL > 0 or (EAX AND NOT IA32_HRESET_ENABLE) ≠0.
#UD	If CPUID.07H.01H:EAX.HRESET[bit 22] = 0.

Real-Address Mode Exceptions

Same exceptions as in protected mode.

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

Same exceptions as in protected mode.

PCONFIG — Platform Configuration

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 01 C5 PCONFIG	A	V/V	PCONFIG	This instruction is used to execute functions for configuring platform features.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	NA	NA	NA	NA

Description

PCONFIG allows software to configure certain platform features. PCONFIG supports multiple leaf functions, with a leaf function identified by the value in EAX. The registers RBX, RCX, and RDX may provide input information for certain leaves. All leaves write status information to EAX but do not modify RBX, RCX, or RDX.

Each PCONFIG leaf function applies to a specific hardware block called a PCONFIG target, and each PCONFIG target is associated with a numerical identifier. The identifiers of the PCONFIG targets supported by the CPU (which imply the supported leaf functions) are enumerated in the sub-leaves of the PCONFIG-information leaf of CPUID (EAX = 1BH). An attempt to execute an undefined leaf function results in a general-protection exception (#GP).

Addresses and operands are 32 bits outside 64-bit mode and are 64 bits in 64-bit mode. The value of CS.D does not affect operand size or address size.

Table 2-1 shows the leaf encodings for PCONFIG.

Table 2-1. PCONFIG Leaf Encodings

Leaf	Encoding	Description
MKTME_KEY_PROGRAM	00000000H	This leaf is used to program the key and encryption mode associated with a KeyID.
RESERVED	00000001H - FFFFFFFFH	Reserved for future use (#GP(0) if used).

The MKTME_KEY_PROGRAM leaf of PCONFIG pertains to the MKTME target, which has target identifier 1. It is used by software to manage the key associated with a KeyID. The leaf function is invoked by setting the leaf value of 0 in EAX and the address of MKTME_KEY_PROGRAM_STRUCT in RBX. Successful execution of the leaf clears RAX (set to zero) and ZF, CF, PF, AF, OF, and SF are cleared. In case of failure, the failure reason is indicated in RAX with ZF set to 1 and CF, PF, AF, OF, and SF are cleared. The MKTME_KEY_PROGRAM leaf uses the MKTME_KEY_PROGRAM_-STRUCT in memory shown in Table 2-2.

Table 2-2. MKTME_KEY_PROGRAM_STRUCT Format

Field	Offset (bytes)	Size (bytes)	Comments
KEYID	0	2	Key Identifier.
KEYID_CTRL	2	4	KeyID control: ▪ Bits [7:0]: COMMAND. ▪ Bits [23:8]: ENC_ALG. ▪ Bits [31:24]: Reserved, must be zero.
RESERVED	6	58	Reserved, must be zero.
KEY_FIELD_1	64	64	Software supplied KeyID data key or entropy for KeyID data key.
KEY_FIELD_2	128	64	Software supplied KeyID tweak key or entropy for KeyID tweak key.

A description of each of the fields in MKTME_KEY_PROGRAM_STRUCT is provided below:

- **KEYID:** Key Identifier being programmed to the MKTME engine.
- **KEYID_CTRL:** The KEYID_CTRL field carries two sub-fields used by software to control the behavior of a KeyID: Command and KeyID encryption algorithm.

  The command used controls the encryption mode for a KeyID. Table 2-3 provides a summary of the commands supported.

Table 2-3. Supported Key Programming Commands

Command	Encoding	Description
KEYID_SET_KEY_DIRECT	0	Software uses this mode to directly program a key for use with KeyID.
KEYID_SET_KEY_RANDOM	1	CPU generates and assigns an ephemeral key for use with a KeyID. Each time the instruction is executed, the CPU generates a new key using a hardware random number generator and the keys are discarded on reset.
KEYID_CLEAR_KEY	2	Clear the (software programmed) key associated with the KeyID. On execution of this command, the KeyID gets TME behavior (encrypt with platform TME key).
KEYID_NO_ENCRYPT	3	Do not encrypt memory when this KeyID is in use.

  The encryption algorithm field (ENC_ALG) allows software to select one of the activated encryption algorithms for the KeyID. The BIOS can activate a set of algorithms to allow for use when programming keys using the IA32_TME_ACTIVATE MSR (does not apply to KeyID 0 which uses TME policy). The processor checks to ensure that the algorithm selected by software is one of the algorithms that has been activated by the BIOS.

- **KEY_FIELD_1:** This field carries the software supplied data key to be used for the KeyID if the direct key programming option is used (KEYID_SET_KEY_DIRECT). When the random key programming option is used (KEYID_SET_KEY_RANDOM), this field carries the software supplied entropy to be mixed in the CPU generated random data key. It is software's responsibility to ensure that the key supplied for the direct programming option or the entropy supplied for the random programming option does not result in weak keys. There are no explicit checks in the instruction to detect or prevent weak keys. When AES XTS-128 is used, the upper 48B are treated as reserved and must be zeroed out by software before executing the instruction.
- **KEY_FIELD_2:** This field carries the software supplied tweak key to be used for the KeyID if the direct key programming option is used (KEYID_SET_KEY_DIRECT). When the random key programming option is used (KEYID_SET_KEY_RANDOM), this field carries the software supplied entropy to be mixed in the CPU generated random tweak key. It is software's responsibility to ensure that the key supplied for the direct programming option or the entropy supplied for the random programming option does not result in weak keys. There are no explicit checks in the instruction to detect or prevent weak keys. When AES XTS-128 is used, the upper 48B are treated as reserved and must be zeroed out by software before executing the instruction.

  All KeyIDs use the TME key on MKTME activation. Software can at any point decide to change the key for a KeyID using the PCONFIG instruction. Change of keys for a KeyID does NOT change the state of the TLB caches or memory pipeline. It is software's responsibility to take appropriate actions to ensure correct behavior.

  Table 2-4 shows the return values associated with the MKTME_KEY_PROGRAM leaf of PCONFIG. On instruction execution, RAX is populated with the return value.

Table 2-4. Supported Key Error Codes

Return Value	Encoding	Description
PROG_SUCCESS	0	KeyID was successfully programmed.
INVALID_PROG_CMD	1	Invalid KeyID programming command.
ENTROPY_ERROR	2	Insufficient entropy.
INVALID_KEYID	3	KeyID not valid.
INVALID_ENC_ALG	4	Invalid encryption algorithm chosen (not supported).
DEVICE_BUSY	5	Failure to access key table.

PCONFIG Virtualization

Software in VMX root operation can control the execution of PCONFIG in VMX non-root operation using the following VM-execution controls introduced for PCONFIG:

- PCONFIG_ENABLE: This control is a single bit control and enables the PCONFIG instruction in VMX non-root operation. If 0, the execution of PCONFIG in VMX non-root operation causes #UD. Otherwise, execution of PCONFIG works according to PCONFIG_EXITING.
- PCONFIG_EXITING: This is a 64b control and allows VMX root operation to cause a VM-exit for various leaf functions of PCONFIG. This control does not have any effect if the PCONFIG_ENABLE control is clear. It is recommended that VMMs intercept execution of any PCONFIG leaves with which they are not familiar and convert such executions into #GP(0).

PCONFIG Concurrency

In a scenario where the MKTME_KEY_PROGRAM leaf of PCONFIG is executed concurrently on multiple logical processors, only one logical processor will succeed in updating the key table. PCONFIG execution will return with an error code (DEVICE_BUSY) on other logical processors and software must retry. In cases where the instruction execution fails with a DEVICE_BUSY error code, the key table is not updated, thereby ensuring that either the key table is updated in its entirety with the information for a KeyID, or it is not updated at all. In order to accomplish this, the MKTME_KEY_PROGRAM leaf of PCONFIG maintains a writer lock for updating the key table. This lock is referred to as the Key table lock and denoted in the instruction flows as KEY_TABLE_LOCK. The lock can either be unlocked, when no logical processor is holding the lock (also the initial state of the lock) or be in an exclusive state where a logical processor is trying to update the key table. There can be only one logical processor holding the lock in exclusive state. The lock, being exclusive, can only be acquired when the lock is in unlocked state.

PCONFIG uses the following syntax to acquire KEY_TABLE_LOCK in exclusive mode and release the lock:

- KEY_TABLE_LOCK.ACQUIRE(WRITE)
- KEY_TABLE_LOCK.RELEASE()

Operation

Table 2-5. PCONFIG Operation Variables

Variable Name	Type	Size (Bytes)	Description
TMP_KEY_PROGRAM_STRUCT	MKTME_KEY_PROGRAM_STRUCT	192	Structure holding the key programming structure.
TMP_RND_DATA_KEY	UINT128	16	Random data key generated for random key programming option.
TMP_RND_TWEAK_KEY	UINT128	16	Random tweak key generated for random key programming option.

```
(* #UD if PCONFIG is not enumerated or CPL>0 *)
IF (CPUID.7.0:EDX[18] == 0 OR CPL > 0) #UD;

IF (in VMX non-root mode)
{
    IF (VMCS.PCONFIG_ENABLE == 1)
    {
        IF ((EAX > 62 AND VMCS.PCONFIG_EXITING[63] ==1) OR
            (EAX < 63 AND VMCS.PCONFIG_EXITING[EAX] == 1))
        {
            Set VMCS.EXIT_REASON = PCONFIG; //No Exit qualification
            Deliver VMEXIT;
        }
    }
    ELSE
    {
        #UD
    }
}

(* #GP(0) for an unsupported leaf *)
IF (EAX != 0) #GP(0)

(* KEY_PROGRAM leaf flow *)
IF (EAX == 0)
{
    (* #GP(0) if TME_ACTIVATE MSR is not locked or does not enable TME or multiple keys are not enabled *)
    IF (IA32_TME_ACTIVATE.LOCK != 1 OR IA32_TME_ACTIVATE.ENABLE != 1 OR IA32_TME_ACTIVATE.MK_TME_KEYID_BITS == 0)
#GP(0)

    (* Check MKTME_KEY_PROGRAM_STRUCT is 256B aligned *)
    IF (DS:RBX is not 256B aligned) #GP(0);

    (* Check that MKTME_KEY_PROGRAM_STRUCT is read accessible *)
    <<DS: RBX should be read accessible>>

    (* Copy MKTME_KEY_PROGRAM_STRUCT to a temporary variable *)
    TMP_KEY_PROGRAM_STRUCT = DS:RBX.*;

    (* RSVD field check *)
    IF (TMP_KEY_PROGRAM_STRUCT.RSVD != 0) #GP(0);

    IF (TMP_KEY_PROGRAM_STRUCT.KEYID_CTRL.RSVD !=0) #GP(0);

    IF (TMP_KEY_PROGRAM_STRUCT.KEY_FIELD_1.BYTES[63:16] != 0) #GP(0);

    IF (TMP_KEY_PROGRAM_STRUCT.KEY_FIELD_2.BYTES[63:16] != 0) #GP(0);

    (* Check for a valid command *)
    IF (TMP_KEY_PROGRAM_STRUCT. KEYID_CTRL.COMMAND is not a valid command)
    {
        RFLAGS.ZF = 1;
        RAX = INVALID_PROG_CMD;
        goto EXIT;
```

```
}
(* Check that the KEYID being operated upon is a valid KEYID *)
IF (TMP_KEY_PROGRAM_STRUCT.KEYID >
            2^IA32_TME_ACTIVATE.MK_TME_KEYID_BITS - 1
    OR TMP_KEY_PROGRAM_STRUCT.KEYID >
            IA32_TME_CAPABILITY.MK_TME_MAX_KEYS
    OR TMP_KEY_PROGRAM_STRUCT.KEYID == 0)
{
    RFLAGS.ZF = 1;
    RAX = INVALID_KEYID;
    goto EXIT;
}

(* Check that only one algorithm is requested for the KeyID and it is one of the activated algorithms *)
IF (NUM_BITS(TMP_KEY_PROGRAM_STRUCT.KEYID_CTRL.ENC_ALG) != 1 ||
    (TMP_KEY_PROGRAM_STRUCT.KEYID_CTRL.ENC_ALG &
        IA32_TME_ACTIVATE. MK_TME_CRYPTO_ALGS == 0))
{
    RFLAGS.ZF = 1;
    RAX = INVALID_ENC_ALG;
    goto EXIT;
}
(* Try to acquire exclusive lock *)
IF (NOT KEY_TABLE_LOCK.ACQUIRE(WRITE))
{
    //PCONFIG failure
    RFLAGS.ZF = 1;
    RAX = DEVICE_BUSY;
    goto EXIT;
}

(* Lock is acquired and key table will be updated as per the command
    Before this point no changes to the key table are made *)

switch(TMP_KEY_PROGRAM_STRUCT.KEYID_CTRL.COMMAND)
{
case KEYID_SET_KEY_DIRECT:
    <<Write
        DATA_KEY=TMP_KEY_PROGRAM_STRUCT.KEY_FIELD_1,
        TWEAK_KEY=TMP_KEY_PROGRAM_STRUCT.KEY_FIELD_2,
        ENCRYPTION_MODE=ENCRYPT_WITH_KEYID_KEY,
        to MKTME Key table at index TMP_KEY_PROGRAM_STRUCT.KEYID
    >>
    break;

case KEYID_SET_KEY_RANDOM:
    TMP_RND_DATA_KEY = <<Generate a random key using hardware RNG>>
    IF (NOT ENOUGH ENTROPY)
    {
        RFLAGS.ZF = 1;
        RAX = ENTROPY_ERROR;
        goto EXIT;
    }
    TMP_RND_TWEAK_KEY = <<Generate a random key using hardware RNG>>
```

```
        IF (NOT ENOUGH ENTROPY)
        {
            RFLAGS.ZF = 1;
            RAX = ENTROPY_ERROR;
            goto EXIT;
        }
        (* Mix user supplied entropy to the data key and tweak key *)
        TMP_RND_DATA_KEY = TMP_RND_KEY XOR
            TMP_KEY_PROGRAM_STRUCT.KEY_FIELD_1.BYTES[15:0];
        TMP_RND_TWEAK_KEY = TMP_RND_TWEAK_KEY XOR
            TMP_KEY_PROGRAM_STRUCT.KEY_FIELD_2.BYTES[15:0];

        <<Write
            DATA_KEY=TMP_RND_DATA_KEY,
            TWEAK_KEY=TMP_RND_TWEAK_KEY,
            ENCRYPTION_MODE=ENCRYPT_WITH_KEYID_KEY,
            to MKTME_KEY_TABLE at index TMP_KEY_PROGRAM_STRUCT.KEYID
        >>
        break;

    case KEYID_CLEAR_KEY:
        <<Write
        DATA_KEY='0,
        TWEAK_KEY='0,
        ENCRYPTION_MODE = ENCRYPT_WITH_TME_KEY,
        to MKTME_KEY_TABLE at index TMP_KEY_PROGRAM_STRUCT.KEYID
        >>

        break;
    case KD_NO_ENCRYPT:
        <<Write
        ENCRYPTION_MODE=NO_ENCRYPTION,
        to MKTME_KEY_TABLE at index TMP_KEY_PROGRAM_STRUCT.KEYID
        >>
        break;
    }

    RAX = 0;
    RFLAGS.ZF = 0;

    //Release Lock
    KEY_TABLE_LOCK(RELEASE);

    EXIT:
    RFLAGS.CF=0;
    RFLAGS.PF=0;
    RFLAGS.AF=0;
    RFLAGS.OF=0;
    RFLAGS.SF=0;
}

end_of_flow
```

Protected Mode Exceptions

#GP(0)	If input value in EAX encodes an unsupported leaf. If IA32_TME_ACTIVATE MSR is not locked. If TME and MKTME capability are not enabled in IA32_TME_ACTIVATE MSR. If the memory operand is not 256B aligned. If any of the reserved bits in MKTME_KEY_PROGRAM_STRUCT are set. If a memory operand effective address is outside the DS segment limit.
#PF(fault-code)	If a page fault occurs in accessing memory operands.
#UD	If any of the LOCK/REP/OSIZE/VEX prefixes are used. If current privilege level is not 0. If CPUID.7.0:EDX[bit 18] = 0 If in VMX non-root mode and VMCS.PCONFIG_ENABLE = 0.

Real-Address Mode Exceptions

#GP	If input value in EAX encodes an unsupported leaf. If IA32_TME_ACTIVATE MSR is not locked. If TME and MKTME capability is not enabled in IA32_TME_ACTIVATE MSR. If a memory operand is not 256B aligned. If any of the reserved bits in MKTME_KEY_PROGRAM_STRUCT are set.
#UD	If any of the LOCK/REP/OSIZE/VEX prefixes are used. If current privilege level is not 0. If CPUID.7.0:EDX.PCONFIG[bit 18] = 0 If in VMX non-root mode and VMCS.PCONFIG_ENABLE = 0.

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

Same exceptions as in protected mode.

64-Bit Mode Exceptions

#GP(0)	If input value in EAX encodes an unsupported leaf. If IA32_TME_ACTIVATE MSR is not locked. If TME and MKTME capability is not enabled in IA32_TME_ACTIVATE MSR. If a memory operand is not 256B aligned. If any of the reserved bits in MKTME_KEY_PROGRAM_STRUCT are set. If a memory operand is non-canonical form.
#PF(fault-code)	If a page fault occurs in accessing memory operands.
#UD	If any of the LOCK/REP/OSIZE/VEX prefixes are used. If the current privilege level is not 0. If CPUID.7.0:EDX.PCONFIG[bit 18] = 0. If in VMX non-root mode and VMCS.PCONFIG_ENABLE = 0.

SENDUIPI — Send User Interprocessor Interrupts

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F C7 /6 SENDUIPI reg	A	V/I	UINTR	Send interprocessor user interrupt.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	NA	NA	NA

Description

The SENDUIPI instruction takes a single register operand. The operand always has 64 bits; operand-size overrides (e.g., the prefix 66) are ignored.

Although SENDUIPI may be executed at any privilege level, all of the instruction's memory accesses are performed with supervisor privilege.

Virtualization of the SENDUIPI instruction (in particular, that of the sending of the notification interrupt) is discussed in Section 11.9.2.5.

The Operation section refers to the values UITTADDR and UITTSZ. The values are defined in Section 11.3.1. It also includes operations on a user posted-interrupt descriptor (UPID). The format of a UPID is defined in Section 11.5.

Operation

```
IF reg > UITTSZ;
    THEN #GP(0);
FI;
read tempUITTE from 16 bytes at UITTADDR+ (reg « 4);
IF tempUITTE.V = 0 or tempUITTE sets any reserved bit (see Section 11.7.1)
    THEN #GP(0);
FI;
read tempUPID from 16 bytes at tempUITTE.UPIDADDR;// under lock
IF tempUPID sets any reserved bits or bits that must be zero (see Table 11-1)
    THEN #GP(0); // release lock
FI;
tempUPID.PIR[tempUITTE.UV] := 1;
IF tempUPID.SN = tempUPID.ON = 0
    THEN
        tempUPID.ON := 1;
        sendNotify := 1;
    ELSE sendNotify := 0;
FI;
write tempUPID to 16 bytes at tempUITTE.UPIDADDR;// release lock
IF sendNotify = 1
    THEN
        IF local APIC is in x2APIC mode
            THEN send ordinary IPI with vector tempUPID.NV
                to 32-bit physical APIC ID tempUPID.NDST;
            ELSE send ordinary IPI with vector tempUPID.NV
                to 8-bit physical APIC ID tempUPID.NDST[15:8];
        FI;
FI;
```

Flags Affected

None.

Protected Mode Exceptions

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

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

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

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used.
	If executed inside an enclave.
	If CR4.UINTR = 0.
	If CPUID.07H.0H:EDX.UINTR[bit 5] = 0.
#PF	If a page fault occurs.
#GP	If the value of the register operand exceeds UITTSZ.
	If the selected UITTE is not valid or sets any reserved bits.
	If the selected UPID sets any reserved bits.
	If there is an attempt to access memory using a linear address that is not canonical relative to the current paging mode.

SERIALIZE — Serialize Instruction Execution

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
NP 0F 01 E8 SERIALIZE	ZO	V/V	SERIALIZE	Serialize instruction fetch and execution.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA	NA

Description

Serializes instruction execution. Before the next instruction is fetched and executed, the SERIALIZE instruction ensures that all modifications to flags, registers, and memory by previous instructions are completed, draining all buffered writes to memory. This instruction is also a serializing instruction as defined in the section "Serializing Instructions" in Chapter 8 of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3A*.

SERIALIZE does not modify registers, arithmetic flags or memory.

The availability of the SERIALIZE instruction is indicated by the presence of the CPUID feature flag SERIALIZE, bit 14 of the EDX register in sub-leaf CPUID:7H.0H.

Operation

```
Wait_On_Fetch_And_Execution_Of_Next_Instruction_Until(preceding_instructions_complete_and_preceding_stores_globally_visible);
```

Intel C/C++ Compiler Intrinsic Equivalent

SERIALIZE void _serialize(void);

SIMD Floating-Point Exceptions

None.

Other Exceptions

#UD	If the LOCK prefix is used.

STUI — Set User Interrupt Flag

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 01 EF STUI	ZO	V/I	UINTR	Set user interrupt flag.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA	NA

Description

STUI sets the user interrupt flag (UIF). Its effect takes place immediately; a user interrupt may be delivered on the instruction boundary following STUI. (This is in contrast with STI, whose effect is delayed by one instruction).

An execution of STUI inside a transactional region causes a transactional abort; the abort loads EAX as it would have had it been due to an execution of STI.

Operation

```
UIF := 1;
```

Flags Affected

None.

Protected Mode Exceptions

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

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

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

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used. If executed inside an enclave. If CR4.UINTR = 0. If CPUID.07H.0H:EDX.UINTR[bit 5] = 0.

TESTUI — Determine User Interrupt Flag

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 01 ED TESTUI	ZO	V/I	UINTR	Copies the current value of UIF into EFLAGS.CF.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA	NA

TESTUI copies the current value of the user interrupt flag (UIF) into EFLAGS.CF. This instruction can be executed regardless of CPL.

TESTUI may be executed normally inside a transactional region.

Operation

```
CF := UIF;
ZF := AF := OF := PF := SF := 0;
```

Flags Affected

The ZF, OF, AF, PF, SF flags are cleared and the CF flags to the value of the user interrupt flag.

Protected Mode Exceptions

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

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

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

64-Bit Mode Exceptions

#UD	If the LOCK prefix is used. If executed inside an enclave. If CR4.UINTR = 0. If CPUID.07H.0H:EDX.UINTR[bit 5] = 0.

UIRET — User-Interrupt Return

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 01 EC UIRET	ZO	V/I	UINTR	Return from handling a user interrupt.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA	NA

Description

UIRET returns from the handling of a user interrupt. It can be executed regardless of CPL.

Execution of UIRET inside a transactional region causes a transactional abort; the abort loads EAX as it would have had it been due to an execution of IRET.

UIRET can be tracked by Architectural Last Branch Records (LBRs), Intel Processor Trace (Intel PT), and Performance Monitoring. For both Intel PT and LBRs, UIRET is recorded in precisely the same manner as IRET. Hence for LBRs, UIRETs fall into the OTHER_BRANCH category, which implies that IA32_LBR_CTL.OTHER_BRANCH[bit 22] must be set to record user-interrupt delivery, and that the IA32_LBR_x_INFO.BR_TYPE field will indicate OTHER_BRANCH for any recorded user interrupt. For Intel PT, control flow tracing must be enabled by setting IA32_RTIT_CTL.BranchEn[bit 13].

UIRET will also increment performance counters for which counting BR_INST_RETIRED.FAR_BRANCH is enabled.

Operation

```
Pop tempRIP;
Pop tempRFLAGS; // see below for how this is used to load RFLAGS
Pop tempRSP;
IF tempRIP is not canonical in current paging mode
    THEN #GP(0);
FI;
IF shadow stack is enabled for CPL = 3
    THEN
        PopShadowStack SSRIP;
        IF SSRIP ≠ tempRIP
            THEN #CP (FAR-RET/IRET);
        FI;
FI;
RIP := tempRIP;
// update in RFLAGS only CF, PF, AF, ZF, SF, TF, DF, OF, NT, RF, AC, and ID
RFLAGS := (RFLAGS & ~254DD5H) | (tempRFLAGS & 254DD5H);
RSP := tempRSP;
UIF := 1;
Clear any cache-line monitoring established by MONITOR or UMONITOR;
```

Flags Affected

See Operation section.

Protected Mode Exceptions

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

Real-Address Mode Exceptions

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

Virtual-8086 Mode Exceptions

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

Compatibility Mode Exceptions

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

64-Bit Mode Exceptions

#GP(0)	If the return instruction pointer is non-canonical.
#SS(0)	If an attempt to pop a value off the stack causes a non-canonical address to be referenced.
#PF(fault-code)	If a page fault occurs.
#AC(0)	If alignment checking is enabled and an unaligned memory reference is made while the current privilege level is 3.
#CP	If return instruction pointer from stack and shadow stack do not match.
#UD	If the LOCK prefix is used.
	If executed inside an enclave.
	If CR4.UINTR = 0.
	If CPUID.07H.0H:EDX.UINTR[bit 5] = 0.

VPDPBUSD — Multiply and Add Unsigned and Signed Bytes

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 50 /r VPDPBUSD xmm1, xmm2, xmm3/m128	A	V/V	AVX_VNNI	Multiply groups of 4 pairs of signed bytes in xmm3/m128 with corresponding unsigned bytes of xmm2, summing those products and adding them to doubleword result in xmm1.
VEX.256.66.0F38.W0 50 /r VPDPBUSD ymm1, ymm2, ymm3/m256	A	V/V	AVX_VNNI	Multiply groups of 4 pairs of signed bytes in ymm3/m256 with corresponding unsigned bytes of ymm2, summing those products and adding them to doubleword result in ymm1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	EVEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Multiplies the individual unsigned bytes of the first source operand by the corresponding signed bytes of the second source operand, producing intermediate signed word results. The word results are then summed and accumulated in the destination dword element size operand.

This instruction supports memory fault suppression.

Operation

VPDPBUSD dest, src1, src2

```
VL=(128, 256)
KL=VL/32

ORIGDEST := DEST
FOR i := 0 TO KL-1:

    // Extending to 16b
    // src1extend := ZERO_EXTEND
    // src2extend := SIGN_EXTEND

    p1word := src1extend(SRC1.byte[4*i+0]) * src2extend(SRC2.byte[4*i+0])
    p2word := src1extend(SRC1.byte[4*i+1]) * src2extend(SRC2.byte[4*i+1])
    p3word := src1extend(SRC1.byte[4*i+2]) * src2extend(SRC2.byte[4*i+2])
    p4word := src1extend(SRC1.byte[4*i+3]) * src2extend(SRC2.byte[4*i+3])

    DEST.dword[i] := ORIGDEST.dword[i] + p1word + p2word + p3word + p4word

DEST[MAX_VL-1:VL] := 0
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4.

VPDPBUSDS — Multiply and Add Unsigned and Signed Bytes with Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 51 /r VPDPBUSDS xmm1, xmm2, xmm3/m128	A	V/V	AVX_VNNI	Multiply groups of 4 pairs signed bytes in xmm3/m128 with corresponding unsigned bytes of xmm2, summing those products and adding them to doubleword result, with signed saturation in xmm1.
VEX.256.66.0F38.W0 51 /r VPDPBUSDS ymm1, ymm2, ymm3/m256	A	V/V	AVX_VNNI	Multiply groups of 4 pairs signed bytes in ymm3/m256 with corresponding unsigned bytes of ymm2, summing those products and adding them to doubleword result, with signed saturation in ymm1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Multiplies the individual unsigned bytes of the first source operand by the corresponding signed bytes of the second source operand, producing intermediate signed word results. The word results are then summed and accumulated in the destination dword element size operand. If the intermediate sum overflows a 32b signed number the result is saturated to either 0x7FFF_FFFF for positive numbers of 0x8000_0000 for negative numbers.

This instruction supports memory fault suppression.

Operation

VPDPBUSDS dest, src1, src2

```
VL=(128, 256)
KL=VL/32

ORIGDEST := DEST
FOR i := 0 TO KL-1:

    // Extending to 16b
    // src1extend := ZERO_EXTEND
    // src2extend := SIGN_EXTEND

    p1word := src1extend(SRC1.byte[4*i+0]) * src2extend(SRC2.byte[4*i+0])
    p2word := src1extend(SRC1.byte[4*i+1]) * src2extend(SRC2.byte[4*i+1])
    p3word := src1extend(SRC1.byte[4*i+2]) * src2extend(SRC2.byte[4*i+2])
    p4word := src1extend(SRC1.byte[4*i+3]) * src2extend(SRC2.byte[4*i+3])

    DEST.dword[i] := SIGNED_DWORD_SATURATE(ORIGDEST.dword[i] + p1word + p2word + p3word + p4word)

DEST[MAX_VL-1:VL] := 0
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4.

VPDPWSSD — Multiply and Add Signed Word Integers

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 52 /r VPDPWSSD xmm1, xmm2, xmm3/m128	A	V/V	AVX_VNNI	Multiply groups of 2 pairs signed words in xmm3/m128 with corresponding signed words of xmm2, summing those products and adding them to doubleword result in xmm1.
VEX.256.66.0F38.W0 52 /r VPDPWSSD ymm1, ymm2, ymm3/m256	A	V/V	AVX_VNNI	Multiply groups of 2 pairs signed words in ymm3/m256 with corresponding signed words of ymm2, summing those products and adding them to doubleword result in ymm1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	VEX.vvvv (r)	ModRM:r/m (r)	NA

Description

Multiplies the individual signed words of the first source operand by the corresponding signed words of the second source operand, producing intermediate signed, doubleword results. The adjacent doubleword results are then summed and accumulated in the destination operand.

This instruction supports memory fault suppression.

Operation

VPDPWSSD dest, src1, src2

```
VL=(128, 256)
KL=VL/32
ORIGDEST := DEST
FOR i := 0 TO KL-1:
    p1dword := SRC1.word[2*i+0] * t.word[2*i+0]
    p2dword := SRC1.word[2*i+1] * t.word[2*i+1]
    DEST.dword[i] := ORIGDEST.dword[i] + p1dword + p2dword
DEST[MAX_VL-1:VL] := 0
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4.

VPDPWSSDS — Multiply and Add Signed Word Integers with Saturation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 53 /r VPDPWSSDS xmm1, xmm2, xmm3/m128	A	V/V	AVX_VNNI	Multiply groups of 2 pairs of signed words in xmm3/m128 with corresponding signed words of xmm2, summing those products and adding them to doubleword result in xmm1, with signed saturation,.
VEX.256.66.0F38.W0 53 /r VPDPWSSDS ymm1, ymm2, ymm3/m256	A	V/V	AVX_VNNI	Multiply groups of 2 pairs of signed words in ymm3/m256 with corresponding signed words of ymm2, summing those products and adding them to doubleword result in ymm1, with signed saturation.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	VEX.vvvv	ModRM:r/m (r)	NA

Description

Multiplies the individual signed words of the first source operand by the corresponding signed words of the second source operand, producing intermediate signed, doubleword results. The adjacent doubleword results are then summed and accumulated in the destination operand. If the intermediate sum overflows a 32b signed number, the result is saturated to either 0x7FFF_FFFF for positive numbers of 0x8000_0000 for negative numbers.

This instruction supports memory fault suppression.

Operation

VPDPWSSDS dest, src1, src2

```
VL=(128, 256)
KL=VL/32
ORIGDEST := DEST
FOR i := 0 TO KL-1:
    p1dword := SRC1.word[2*i+0] * t.word[2*i+0]
    p2dword := SRC1.word[2*i+1] * t.word[2*i+1]
    DEST.dword[i] := SIGNED_DWORD_SATURATE(ORIGDEST.dword[i] + p1dword + p2dword)
DEST[MAX_VL-1:VL] := 0
```

SIMD Floating-Point Exceptions

None.

Other Exceptions

See Exceptions Type 4.

XRESLDTRK — Resume Tracking Load Addresses

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 01 E9 XRESLDTRK	ZO	V/V	TSXLDTRK	Specifies the end of an Intel TSX suspend read address tracking region.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA	NA

Description

The instruction marks the end of an Intel TSX (RTM) suspend load address tracking region. If the instruction is used inside a suspend load address tracking region it will end the suspend region and all following load addresses will be added to the transaction read set. If this instruction is used inside an active transaction but not in a suspend region it will cause transaction abort.

If the instruction is used outside of a transactional region it behaves like a NOP.

Chapter 5 provides additional information on Intel® TSX Suspend Load Address Tracking.

Operation

XRESLDTRK

```
IF RTM_ACTIVE = 1:
    IF SUSLDTRK_ACTIVE = 1:
        SUSLDTRK_ACTIVE := 0
    ELSE:
        RTM_ABORT
ELSE:
    NOP
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

XRESLDTRK void _xresldtrk(void);

SIMD Floating-Point Exceptions

None.

Other Exceptions

#UD	If CPUID.(EAX=7, ECX=0):EDX.TSXLDTRK[bit 16] = 0.
	If the LOCK prefix is used.

XSUSLDTRK— Suspend Tracking Load Addresses

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
F2 0F 01 E8 XSUSLDTRK	ZO	V/V	TSXLDTRK	Specifies the start of an Intel TSX suspend read address tracking region.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA	NA

Description

The instruction marks the start of an Intel TSX (RTM) suspend load address tracking region. If the instruction is used inside a transactional region, subsequent loads are not added to the read set of the transaction. If the instruction is used inside a suspend load address tracking region it will cause transaction abort.

If the instruction is used outside of a transactional region it behaves like a NOP.

Chapter 5 provides additional information on Intel® TSX Suspend Load Address Tracking.

Operation

XSUSLDTRK

```
IF RTM_ACTIVE = 1:
    IF SUSLDTRK_ACTIVE = 0:
        SUSLDTRK_ACTIVE := 1
    ELSE:
        RTM_ABORT
ELSE:
    NOP
```

Flags Affected

None.

Intel C/C++ Compiler Intrinsic Equivalent

XSUSLDTRK void _xsusldtrk(void);

SIMD Floating-Point Exceptions

None.

Other Exceptions

#UD	If CPUID.(EAX=7, ECX=0):EDX.TSXLDTRK[bit 16] = 0.
	If the LOCK prefix is used.

LDTILECFG — Load Tile Configuration

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.NP.0F38.W0 49 !(11):000:bbb LDTILECFG m512	A	V/N.E.	AMX-TILE	Load tile configuration as specified in m512.

Instruction Operand Encoding

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

Description

The LDTILECFG instruction takes a operand containing a pointer to a 64-byte memory location containing the description of the tiles to be supported. In order to configure the tiles, the AMX-TILE bit in CPUID must be set and the operating system has to have enabled the tiles architecture.

The memory area first describes the number of tiles selected and then selects from the palette of tile types. Requests must be compatible with the restrictions provided by CPUID.

The memory area describes how many tiles are being used and defines each tile in terms of rows and columns; see Table 3-1 below.

Table 3-1. Memory Area Layout

Byte(s)	Field Name	Description
0	palette	Palette selects the supported configuration of the tiles that will be used.
1	start_row	start_row is used for storing the restart values for interrupted operations.
2-15	reserved, must be zero	
16-17	tile0.colsb	Tile 0 bytes per row.
18-19	tile1.colsb	Tile 1 bytes per row.
20-21	tile2.colsb	Tile 2 bytes per row.
...	(sequence continues)	
30-31	tile7.colsb	Tile 7 bytes per row.
32-47	reserved, must be zero	
48	tile0.rows	Tile 0 rows.
49	tile1.rows	Tile 1 rows.
50	tile2.rows	Tile 2 rows.
...	(sequence continues)	
55	tile7.rows	Tile 7 rows.
56-63	reserved, must be zero	

If a tile row and column pair is not used to specify tile parameters, they must have the value zero. All enabled tiles (based on the palette) must be configured. Specifying tile parameters for more tiles than the implementation limit or the palette limit results in a #GP fault.

If the palette_id is zero, that signifies the INIT state for the both XTILECFG and XTILEDATA. Tiles are zeroed in the INIT state. The only legal non-INIT value for palette_id is 1.

Any attempt to execute the LDTILECFG instruction inside an Intel TSX transaction will result in a transaction abort.

Operation

LDTILECFG mem

```
error := False
buf := read_memory(mem, 64)
temp_tilecfg.palette_id := buf.byte[0]
if temp_tilecfg.palette_id > max_palette:
    error := True
if not xcr0_supports_palette(temp_tilecfg.palette_id):
    error := True
if temp_tilecfg.palette_id !=0:
    temp_tilecfg.start_row := buf.byte[1]
    if buf.byte[2..15] is nonzero:
        error := True
    p := 16
    # configure columns
    for n in 0 ... palette_table[temp_tilecfg.palette_id].max_names-1:
        temp_tilecfg.t[n].colsb:= buf.word[p/2]
        p := p + 2
        if temp_tilecfg.t[n].colsb > palette_table[temp_tilecfg.palette_id].bytes_per_row:
            error := True
    if nonzero(buf[p...47]):
        error := True

    # configure rows
    p := 48
    for n in 0 ... palette_table[temp_tilecfg.palette_id].max_names-1:
        temp_tilecfg.t[n].rows:= buf.byte[p]
        if temp_tilecfg.t[n].rows > palette_table[temp_tilecfg.palette_id].max_rows:
            error := True
        p := p + 1

    if nonzero(buf[p...63]):
        error := True

    # validate each tile's row & col configs are reasonable
    for n in 0 ... palette_table[temp_tilecfg.palette_id].max_names-1:
        if temp_tilecfg.t[n].rows !=0 and temp_tilecfg.t[n].colsb != 0:
            temp_tilecfg.t[n].valid := 1
        elif temp_tilecfg.t[n].rows == 0 and temp_tilecfg.t[n].colsb == 0:
            temp_tilecfg.t[n].valid := 0
        else:
            error := True// one of rows or colsbwas 0 but not both.

if error:
    #GP
elif temp_tilecfg.palette_id == 0:
    TILES_CONFIGURED := 0// init state
    tilecfg := 0// equivalent to 64B of zeros
    zero_all_tile_data()
else:
    tilecfg := temp_tilecfg
    zero_all_tile_data()
    TILES_CONFIGURED := 1
```

Intel C/C++ Compiler Intrinsic Equivalent

LDTILECFG void _tile_loadconfig(const void *);

Flags Affected

None.

Exceptions

AMX-E1; see Section 3.7, “Exception Classes” for details.

STTILECFG — Store Tile Configuration

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.66.0F38.W0 49 !(11):000:bbb STTILECFG m512	A	V/N.E.	AMX-TILE	Store tile configuration in m512.

Instruction Operand Encoding

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

Description

The STTILECFG instruction takes a pointer to a 64-byte memory location (described in Table 3-1) that will, after successful execution of this instruction, contain the description of the tiles that were configured. In order to configure tiles, the AMX-TILE bit in CPUID must be set and the operating system has to have enabled the tiles architecture.

If the tiles are not configured, then STTILECFG stores 64B of zeros to the indicated memory location.

Any attempt to execute the STTILECFG instruction inside an Intel TSX transaction will result in a transaction abort.

Operation

STTILECFG mem

```
if TILES_CONFIGURED == 0:
    //write 64 bytes of zeros at mem pointer
    buf[0..63] := 0
    write_memory(mem, 64, buf)
else:
    buf.byte[0] := tilecfg.palette_id
    buf.byte[1] := tilecfg.start_row
    buf.byte[2..15] := 0

    p := 16
    for n in 0 ... palette_table[tilecfg.palette_id].max_names-1:
        buf.word[p/2] := tilecfg.t[n].colsb
        p := p + 2
    if p < 47:
        buf.byte[p..47] := 0
    p := 48
    for n in 0 ... palette_table[tilecfg.palette_id].max_names-1:
        buf.byte[p++] := tilecfg.t[n].rows
    if p < 63:
        buf.byte[p..63] := 0

    write_memory(mem, 64, buf)
```

Intel C/C++ Compiler Intrinsic Equivalent

STTILECFG void _tile_storeconfig(void *);

Flags Affected

None.

Exceptions

AMX-E2; see Section 3.7, “Exception Classes” for details.

TDPBF16PS — Dot Product of BF16 Tiles Accumulated into Packed Single Precision Tile

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.F3.0F38.W0 5C 11:rrr:bbb TDPBF16PS tmm1, tmm2, tmm3	A	V/N.E.	AMX-BF16	Matrix multiply BF16 elements from tmm2 and tmm3, and accumulate the packed single precision elements in tmm1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	VEX.vvvv (r)	NA

Description

This instruction performs a set of SIMD dot-products of two BF16 elements and accumulates the results into a packed single precision tile. Each dword element in input tiles tmm2 and tmm3 is interpreted as a BF16 pair. For each possible combination of (row of tmm2, column of tmm3), the instruction performs a set of SIMD dot-products on all corresponding BF16 pairs (one pair from tmm2 and one pair from tmm3), adds the results of those dot-products, and then accumulates the result into the corresponding row and column of tmm1.

"Round to nearest even" rounding mode is used when doing each accumulation of the FMA. Output denormals are always flushed to zero and input denormals are always treated as zero. MXCSR is not consulted nor updated.

Any attempt to execute the TDPBF16PS instruction inside a TSX transaction will result in a transaction abort.

Operation

```
define make_fp32(x):
    // The x parameter is bfloat16. Pack it in to upper 16b of a dword.
    // The bit pattern is a legal fp32 value. Return that bit pattern.
    dword: = 0
    dword[31:16] := x
    return dword
```

TDPBF16PS tsrcdest, tsrc1, tsrc2

```
// C = m x n (tsrcdest), A = m x k (tsrc1), B = k x n (tsrc2)

# src1 and src2 elements are pairs of bfloat16
elements_src1 := tsrc1.colsb / 4
elements_src2 := tsrc2.colsb / 4
elements_dest := tsrcdest.colsb / 4
elements_temp := tsrcdest.colsb / 2          // Count is in bfloat16 prior to horizontal

for m in 0 ... tsrcdest.rows-1:
    temp1[ 0 ... elements_temp-1 ] := 0
    for k in 0 ... elements_src1-1:
        for n in 0 ... elements_dest-1:

            // FP32 FMA with DAZ=FTZ=1, RNE rounding.
            // MXCSR is neither consulted nor updated.
            // No exceptions raised or denoted.

            temp1.fp32[2*n+0] += make_fp32(tsrc1.row[m].bfloat16[2*k+0]) * make_fp32(tsrc2.row[k].bfloat16[2*n+0])
            temp1.fp32[2*n+1] += make_fp32(tsrc1.row[m].bfloat16[2*k+1]) * make_fp32(tsrc2.row[k].bfloat16[2*n+1])

    for n in 0 ... elements_dest-1:
        // DAZ=FTZ=1, RNE rounding.
        // MXCSR is neither consulted nor updated.
        // No exceptions raised or denoted.
        tmpf32 := temp1.fp32[2*n] + temp1.fp32[2*n+1]
        tsrcdest.row[m].fp32[n] := tsrcdest.row[m].fp32[n] + tmpf32
    write_row_and_zero(tsrcdest, m, tmp, tsrcdest.colsb)

zero_upper_rows(tsrcdest, tsrcdest.rows)
zero_tilecfg_start()
```

Intel C/C++ Compiler Intrinsic Equivalent

TDPBF16PS void _tile_dpbf16ps(__tile dst, __tile src1, __tile src2);

Flags Affected

None.

Exceptions

AMX-E4; see Section 3.7, “Exception Classes” for details.

TDPBSSD/TDPBSUD/TDPBUSD/TDPBUUD — Dot Product of Signed/Unsigned Bytes with Dword Accumulation

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.F2.0F38.W0 5E 11:rrr:bbb TDPBSSD tmm1, tmm2, tmm3	A	V/N.E.	AMX-INT8	Matrix multiply signed byte elements from tmm2 by signed byte elements from tmm3 and accumulate the dword elements in tmm1.
VEX.128.F3.0F38.W0 5E 11:rrr:bbb TDPBSUD tmm1, tmm2, tmm3	A	V/N.E.	AMX-INT8	Matrix multiply signed byte elements from tmm2 by unsigned byte elements from tmm3 and accumulate the dword elements in tmm1.
VEX.128.66.0F38.W0 5E 11:rrr:bbb TDPBUSD tmm1, tmm2, tmm3	A	V/N.E.	AMX-INT8	Matrix multiply unsigned byte elements from tmm2 by signed byte elements from tmm3 and accumulate the dword elements in tmm1.
VEX.128.NP.0F38.W0 5E 11:rrr:bbb TDPBUUD tmm1, tmm2, tmm3	A	V/N.E.	AMX-INT8	Matrix multiply unsigned byte elements from tmm2 by unsigned byte elements from tmm3 and accumulate the dword elements in tmm1.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (r, w)	ModRM:r/m (r)	VEX.vvvv (r)	NA

Description

For each possible combination of (row of tmm2, column of tmm3), the instruction performs a set of SIMD dot-products on all corresponding four byte elements, one from tmm2 and one from tmm3, adds the results of those dot-products, and then accumulates the result into the corresponding row and column of tmm1. Each dword in input tiles tmm2 and tmm3 is interpreted as four byte elements. These may be signed or unsigned. Each letter in the two-letter pattern SU, US, SS, UU indicates the signed/unsigned nature of the values in tmm2 and tmm3, respectively.

Any attempt to execute the TDPBSSD/TDPBSUD/TDPBUSD/TDPBUUD instructions inside an Intel TSX transaction will result in a transaction abort.

Operation

```
define DPBD(c,x,y):// arguments are dwords
    if *x operand is signed*:
        extend_src1 := SIGN_EXTEND
    else:
        extend_src1 := ZERO_EXTEND

    if *y operand is signed*:
        extend_src2 := SIGN_EXTEND
    else:
        extend_src2 := ZERO_EXTEND

    p0dword := extend_src1(x.byte[0]) * extend_src2(y.byte[0])
    p1dword := extend_src1(x.byte[1]) * extend_src2(y.byte[1])
    p2dword := extend_src1(x.byte[2]) * extend_src2(y.byte[2])
    p3dword := extend_src1(x.byte[3]) * extend_src2(y.byte[3])

    c := c + p0dword + p1dword + p2dword + p3dword
```

TDPBSSD, TDPBSUD, TDPBUSD, TDPBUUD tsrcdest, tsrc1, tsrc2 (Register Only Version)

```
// C = m x n (tsrcdest), A = m x k (tsrc1), B = k x n (tsrc2)

tsrc1_elements_per_row := tsrc1.colsb / 4
tsrc2_elements_per_row := tsrc2.colsb / 4
tsrcdest_elements_per_row := tsrcdest.colsb / 4

for m in 0 ... tsrcdest.rows-1:
    tmp := tsrcdest.row[m]
    for k in 0 ... tsrc1_elements_per_row-1:
        for n in 0 ... tsrcdest_elements_per_row-1:
            DPBD( tmp.dword[n], tsrc1.row[m].dword[k], tsrc2.row[k].dword[n] )
    write_row_and_zero(tsrcdest, m, tmp, tsrcdest.colsb)

zero_upper_rows(tsrcdest, tsrcdest.rows)
zero_tilecfg_start()
```

Intel C/C++ Compiler Intrinsic Equivalent

TDPBSSD	void _tile_dpbssd(__tile dst, __tile src1, __tile src2);
TDPBSUD	void _tile_dpbsud(__tile dst, __tile src1, __tile src2);
TDPBUSD	void _tile_dpbusd(__tile dst, __tile src1, __tile src2);
TDPBUUD	void _tile_dpbuud(__tile dst, __tile src1, __tile src2);

Flags Affected

None.

Exceptions

AMX-E4; see Section 3.7, “Exception Classes” for details.

TILELOADD/TILELOADDT1 — Load Tile

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.F2.0F38.W0 4B !(11):rrr:100 TILELOADD tmm1, sibmem	A	V/N.E.	AMX-TILE	Load data into tmm1 as specified by information in sibmem.
VEX.128.66.0F38.W0 4B !(11):rrr:100 TILELOADDT1 tmm1, sibmem	A	V/N.E.	AMX-TILE	Load data into tmm1 as specified by information in sibmem with hint to optimize data caching.

Instruction Operand Encoding

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

Description

This instruction is required to use SIB addressing. The index register serves as a stride indicator. If the SIB encoding omits an index register, the value zero is assumed for the content of the index register.

This instruction loads a tile destination with rows and columns as specified by the tile configuration. The "T1" version provides a hint to the implementation that the data will likely not be reused in the near future and the data caching can be optimized accordingly.

The TILECFG.start_row in the XTILECFG data should be initialized to '0' in order to load the entire tile and is set to zero on successful completion of the TILELOADD instruction. TILELOADD is a restartable instruction and the TILECFG.start_row will be non-zero when restartable events occur during the instruction execution.

Only memory operands are supported and they can only be accessed using a SIB addressing mode, similar to the V[P]GATHER*/V[P]SCATTER* instructions.

Any attempt to execute the TILELOADD/TILELOADDT1 instructions inside an Intel TSX transaction will result in a transaction abort.

Operation

```
TILELOADD[,T1] tdest, tsib

start := tilecfg.start_row

zero_upper_rows(tdest,start)

membegin := tsib.base + displacement
// if no index register in the SIB encoding, the value zero is used.
stride := tsib.index << tsib.scale
nbytes := tdest.colsb
while start < tdest.rows:
    memptr := membegin + start * stride
    write_row_and_zero(tdest, start, read_memory(memptr, nbytes), nbytes)
    start := start + 1
zero_tilecfg_start()
// In the case of a memory fault in the middle of an instruction, the tilecfg.start_row := start
```

Intel C/C++ Compiler Intrinsic Equivalent

TILELOADD void _tile_loadd(__tile dst, const void *base, int stride);
TILELOADDT1 void _tile_stream_loadd(__tile dst, const void *base, int stride);

Flags Affected

None.

Exceptions

AMX-E3; see Section 3.7, “Exception Classes” for details.

TILERELEASE — Release Tile

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.NP.0F38.W0 49 C0 TILERELEASE	A	V/N.E.	AMX-TILE	Initialize TILECFG and TILEDATA.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	NA	NA	NA	NA

Description

This instruction returns TILECFG and TILEDATA to the INIT state.

Any attempt to execute the TILERELEASE instruction inside an Intel TSX transaction will result in a transaction abort.

Operation

```
zero_all_tile_data()
tilecfg := 0// equivalent to 64B of zeros
TILES_CONFIGURED := 0
```

Intel C/C++ Compiler Intrinsic Equivalent

TILERELEASE void _tile_release(void);

Flags Affected

None.

Exceptions

AMX-E6; see Section 3.7, “Exception Classes” for details.

TILESTORED — Store Tile

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.F3.0F38.W0 4B !(11):rrr:100 TILESTORED sibmem, tmm1	A	V/N.E.	AMX-TILE	Store a tile in sibmem as specified in tmm1.

Instruction Operand Encoding

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

Description

This instruction is required to use SIB addressing. The index register serves as a stride indicator. If the SIB encoding omits an index register, the value zero is assumed for the content of the index register.

This instruction stores a tile source of rows and columns as specified by the tile configuration.

The TILECFG.start_row in the XTILECFG data should be initialized to '0' in order to store the entire tile and are set to zero on successful completion of the TILESTORED instruction. TILESTORED is a restartable instruction and the TILECFG.start_row will be non-zero when restartable events occur during the instruction execution.

Only memory operands are supported and they can only be accessed using a SIB addressing mode, similar to the V[P]GATHER*/V[P]SCATTER* instructions.

Any attempt to execute the TILESTORED instruction inside an Intel TSX transaction will result in a transaction abort.

Operation

```
TILESTORED tsib, tsrc

start := tilecfg.start_row

membegin := tsib.base + displacement
// if no index register in the SIB encoding, the value zero is used.
stride := tsib.index << tsib.scale

while start < tdest.rows:
    memptr := membegin + start * stride
    write_memory(memptr, tsrc.colsb, tsrc.row[start])
    start := start + 1
zero_tilecfg_start()
// In the case of a memory fault in the middle of an instruction, the tilecfg.start_row := start
```

Intel C/C++ Compiler Intrinsic Equivalent

TILESTORED void _tile_stored(__tile src, void *base, int stride);

Flags Affected

None.

Exceptions

AMX-E3; see Section 3.7, “Exception Classes” for details.

TILEZERO — Zero Tile

Opcode/ Instruction	Op/ En	64/32 bit Mode Support	CPUID Feature Flag	Description
VEX.128.F2.0F38.W0 49 11:rrr:000 TILEZERO tmm1	A	V/N.E.	AMX-TILE	Zero the destination tile.

Instruction Operand Encoding

Op/En	Tuple	Operand 1	Operand 2	Operand 3	Operand 4
A	NA	ModRM:reg (w)	NA	NA	NA

Description

This instruction zeroes the destination tile.

Any attempt to execute the TILEZERO instruction inside an Intel TSX transaction will result in a transaction abort.

Operation

```
TILEZERO tdest

nbytes := palette_table[palette_id].bytes_per_row

for i in 0 ... palette_table[palette_id].max_rows-1:
    for j in 0 ... nbytes-1:
        tdest.row[i].byte[j] := 0
zero_tilecfg_start()
```

Intel C/C++ Compiler Intrinsic Equivalent

TILEZERO void _tile_zero(__tile dst);

Flags Affected

None.

Exceptions

AMX-E5; see Section 3.7, “Exception Classes” for details.