Intel® Advanced Performance Extensions (Intel® APX) Software Enabling Introduction

July 2023

Revision 1.0

Document Number: 356112-001US

Notices & Disclaimers

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 product or component can be absolutely secure.

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.

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.

Code names are used by Intel to identify products, technologies, or services that are in development and not publicly available. These are not "commercial" names and not intended to function as trademarks.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document, with the sole exception that a) you may publish an unmodified copy and b) code included in this document is licensed subject to the Zero-Clause BSD open source license (0BSD), https://opensource.org/licenses/0BSD. You may create software implementations based on this document and in compliance with the foregoing that are intended to execute on the Intel product(s) referenced in this document. No rights are granted to create modifications or derivatives of this document.

Revision History

Revision	Description	Date
1.0	• Initial draft	July 2023

REVISION HISTORY

TABLE OF CONTENTS

. 1-51

TABLE OF TABLES

CHAPTER 1: INTRODUCTION TO INTEL® ADVANCED PERFORMANCE EXTENSIONS

1.1 ABOUT THIS DOCUMENT

This document introduces the Intel ® Advanced Performance Extensions (Intel® APX) and the accompanying software changes that are potentially required to enable and use it, including facets that cover operating system enabling, hypervisor enabling, and application-facing conventions, such as Application Binary Interface (ABI).

1.2 INTRODUCTION

Intel® Advanced Performance Extensions (Intel® APX) expands the Intel® 64 instruction set architecture with access to more registers and adds various new features that improve general-purpose performance. The extensions are designed to provide efficient performance gains across a variety of workloads without significantly increasing silicon area or power consumption of the core.

The main features of Intel® APX include:

- 16 additional general-purpose registers (GPRs) R16–R31, also referred to as Extended GPRs (EGPRs) in this document;
- Three-operand instruction formats with a new data destination (NDD) register for many integer instructions;
- Conditional ISA improvements: New conditional load, store and compare instructions, combined with an option for the compiler to suppress the status flags writes of common instructions;
- Optimized register state save/restore operations;
- A new 64-bit absolute direct jump instruction.

This guide will focus on introductory software enabling aspects of the Intel® APX feature, itself, across the typical software stack of both client and server computing platforms.

CHAPTER 2: SOFTWARE VIEW OF INTEL® ADVANCED PERFORMANCE EXTENSIONS

2.1 SOFTWARE STACK

Intel ® Advanced Performance Extensions (Intel® APX) is an XSAVE-enabled feature that must be enabled through a combination of software support spanning the Operating System (OS), Hypervisor (Virtual Machine Monitor, or VMM), and code-generators (compilers, JITs) and any supporting run-time environments, and their adherence to application binary interfaces (ABIs). These components, which make up the base of many software stacks, require adjustments and enabling to account for the EGPR state introduced by Intel® APX.

2.2 COMPILERS AND APPLICATION BINARY INTERFACE (ABI)

The Application Binary Interface (ABI) defines the conventions that allow software components, down to the function level, to interact with one another in a functional, predictable, and composable way. ABIs define register usage conventions that both describe:

1) The general usage of specific sets of CPU state, including general purpose registers and how datatypes map to underlying architectural storage elements
2) The usage of specific sets of CPU state to provide function linkage, including calling conventions that describe:
    a. How parameters are passed into functions.
    b. How return values are passed back to callers of functions.

The rules and behavior of an ABI are defined by a function of hardware/software components, and may be platform- and/or vendor-specific:

- Hardware architecture and ISA
- Operating system type and version
- Compiler/language type and version

As CPUs evolve, so do their corresponding ABI definitions. Stateful extensions, which augment the x86 CPU architecture with new states, must define the usage model of the new states to provide a common usage model across software components. New CPU extensions, and their resulting ABI changes, can be viewed through a lens as being either legacy ABI compatible or incompatible:

- *Compatible* ABI changes, or ABI extensions, provide backward compatibility; making sure that software components that were compiled to, or *abide by*, the former ABI definition can transparently interact with newer software components without breakage.
- *Incompatible* ABI changes may introduce concepts and rules that are not backwards compatible with older software components.

From an x86 perspective, which has a long history of instruction-set evolutions, backward compatibility is seen as an important asset that allows new programs, which may make use of new states/instructions, to function in a

world that is full of legacy binaries (applications, libraries, other software components, etc.) that may have been created before such extensions were released and known. As such, Intel® APX is another evolutionary change within the x86 ecosystem, and any programs that are to make use of Intel® APX features, should be able to transparently interact with programs that might not make use of Intel® APX features, and pre-date Intel® APX-featured platforms.

2.2.1 LEGACY-COMPLIANT APPLICATION BINARY INTERFACE (ABI)

An Intel® APX-enabled legacy compatible ABI introduces new architectural state in such a way that allows transparent interactions and co-existence with software that pre-dates the ABI change.

From a register usage perspective, registers are often categorized in one of approximately three usage categories:

1) Callee-Saved Register (Non-Volatile)
2) Caller-Saved Register (Volatile)
3) Scratch Register (Volatile)

In addition, ABIs define “calling conventions” that precisely describe which architectural registers are used for parameter passing between functions (function inputs and outputs), and the rules of usage with regards to register usage and memory (stack) usage.

When introducing new architectural state, a high-confidence strategy for building a legacy-compatible ABI definition that ensures backward compatibility involves defining new architectural state with the following gules:

1) Keeping the ABI’s parameter passing definition constant.
2) Defining all new state (Intel® APX’s EGPRs) as volatile (caller-saved or scratch).
3) Defining the OS/VMM as providing full context switch support for the new states.

From a *typical* application-perspective, this type of definition does not put any burden on EGPR state preservation onto older/legacy x86 code, nor does it introduce any incompatibility in function linkage. Additionally, this definition does not put an undue burden on Intel® APX code, as it only requires Intel® APX code to preserve the EGPRs that it uses within a given function context.

The cost of a legacy ABI may preclude certain performance optimizations (i.e., enhanced parameter passing of arguments within registers), but the benefit (i.e., backwards compatibility) allows for APX programs and libraries to co-exist with legacy binaries in a transparent manner.

As such, at Intel® APX launch time, software platforms (Operating System + toolchain varieties) are expected to prefer ABIs where all EGPRs are considered caller-saved (volatile).

In addition to the ABI’s register definition extensions, the x86-64 psABI will be extended with new relocation types used by the linker. The new relocation types account for the new types of instructions, namely Intel® APX-prefixed instructions. Intel® APX-enabled compilers and assemblers will generate these new relocation types, primarily when instructions are using EGPRs, and an updated Intel® APX-enabled linker will be able to process these relocations.

2.2.2 EXAMPLE: SETJMP AND LONGJMP

The POSIX C standard has a concept of setjmp/longjmp, which are APIs used to perform *non-local* goto operations, which can be used for context switching and other forms of “complex”, non-local control flow.

Today's implementations of setjmp/longjmp are ABI-optimized in that these functions do not save the complete architectural state of the machine within their state save container (also known as jmp_buf). Instead, they perform partial state save and state restore by focusing only on the architectural machine state that is required to "maintain the current environment". For x86-64, this means only saving the smaller subset of registers that are not already preserved naturally via the x86-64 ABI that is in use on the platform – which is defined by callee-saved registers and registers/state that are preserved across function calls.

According to the System V ABI for x86-64 this includes:

- Required:
  - RBX
  - RSP
  - R12
  - R13
  - R14
  - R15
  - RIP
- Optional (system *dependent*):
  - RFLAGS
  - MXCSR
  - X87 FP CW

The remaining architectural state, while important, is not *actively* managed by setjmp/longjmp, as caller-saved state is automatically preserved by the ***caller*** of setjmp itself.

Concretely:

- Any usages of EGPRs in a call stack are guaranteed to be preserved on the stack when a function call is made, thus pre-preserving them, before a call to setjmp is made.
- Post longjmp, the state held within the jmpbuf is restored. This is a partial state restoration, when it comes to caller-saved registers, which will be iteratively restored at later points in time as the call stack is unwound through the natural flow of function returns

By defining all new EGPR state as caller-saved, Intel® APX-containing binaries can work transparently with standard x86-64 libraries, system infrastructure, and have no adverse impacts to the layout and size of the jmpbuf structure.

2.3 OPERATING SYSTEM INTERFACE

Operating systems provide abstractions and services to applications and drivers, some of which are related to encapsulating and virtualizing the architectural state of the CPU.

In the face of stateful CPU extensions, operating systems often augment their context switching routines for saving/restoring CPU state, which is a critical interface that it used for switching between applications, processes, and threads.

The operating system is responsible for both enumerating and enabling such features, and for managing the CPU context in ways that enable usages of the feature (usages may include one or more of the following: in-kernel usage, driver usage, and application-level usage).

Intel® APX's new state, EGPRs, are encapsulated as XSAVE-enabled state that can be saved/restored through XSAVE*/XRSTOR* interfaces. This design choice was purposefully made to allow operating systems to enable Intel® APX ISA usage at the application-level (and guest-level) without having to use new/extended Intel® APX instructions to do so.

This is important as it is quite common for lower levels of the software stack (namely operating systems and hypervisors) to be compiled using restricted ABIs and instruction/state footprints to save overhead and to maximize compatibility amongst hardware platforms (i.e. compiling to a common subset of ISA).

2.3.1 INTEL® APX CPUID ENUMERATION AND XSAVE DEFINITION

Intel® APX, like all XSAVE-enabled features, is enumerated via CPUID as part of the Processor Extended State Enumeration Main Leaf and Sub-leaves and is controlled/enabled via XCR0.

Intel® APX defines a single set of state that can be managed via XSAVE*/XRSTOR* instructions:

- Intel® APX EGPR state (r16-r31) is save/restore controlled via XCR0[APX_F=19].

Intel® APX's XSAVE footprint, which *re-uses* (via re-definition) the 128B area of the now-deprecated Intel ® Memory Protection Extensions (Intel® MPX). Since Intel® MPX had been previously deprecated, no processor will enumerate support for both Intel® MPX and Intel® APX. The architecture does not re-use any XCR0 control bits and instead only re-purposes the 128-byte XSAVE area that had been previously allocated by Intel® MPX (state component indices 3 and 4, making up a 128-byte area located at an offset of 960 bytes into an un-compacted XSAVE buffer). Intel® APX re-architects the two previous 64-byte state components and uses them as a single state component housing 128-bytes of storage for EGPRs (8-bytes * 16 registers).

Intel® APX uses XCR0 index 19, and as such, the monotonic relationship between XCR0 index and XSAVE buffer offset is altered. The logical ordering of the first 8 entries in the un-compacted XSAVE buffer with regards to XCR0 indices changes in the following manner:

- Before Intel® APX has been introduced:
  - 0, 1, 2, 3, 4, 5, 6, 7, …
- After Intel® APX has been introduced:
  - 0, 1, 2, 19, 5, 6, 7, 9, …

Conversely, in a compacted XSAVE buffer (via XSAVEC), which saves state components in a dynamic, XCR0 index-relative order, Intel® APX state would be placed later with respect to all state components with lesser XCR0 indices. Therefore, the logical order of Intel® APX state differs between un-compacted and compacted forms.

Re-purposing the deprecated state area of Intel® MPX allows for Intel® APX to avoid potential interactions with being placed after large state components, such as Intel® AMX.

2.4 HYPERVISOR INTERFACE

Hypervisors, commonly referred to as Virtual Machine Monitors (VMMs), are like operating systems in that they support the management and virtualization of hardware, but different in that they are designed to host entire sets of virtualized OS and application stack instances, as opposed to just applications. As such, a VMM is responsible for many of the same duties as an operating system, but it may use a richer set of underlying architectural features that are meant to support these increased virtualization duties. These features may include management and configuration and use of both VMX mode and Virtual Machine Control Structures (VMCS's) that are part of the Intel ® Virtual Machine Extensions (VMX) feature-set.

Intel® APX extends the VMCS definition to include backwards-compatible enhancements to certain VMCS fields used for capturing decoded register identifier information for VM-Exiting instructions. VMCS has two such fields for this today:

- VM Exit Qualification Field
- VM Exit Instruction Info Field

Both fields have register identifier sub-fields that are 4-bits in size, thus requiring extensions to support a change from 16 integer GPRs to the new total of 32 integer GPRs introduced by Intel® APX.

The VM Exit Qualification Field is a 64-bit VMCS field and was architected in such a way as to allow in-place extension to 5-bit register identifiers without re-architecting the field definition itself, where the most significant reserved bit is re-defined to be the 5th register ID bit.

The VM Exit Instruction Info Field is a 32-bit VMCS field, so the virtualization architecture requires slight modification to allow for both legacy VMMs and Intel® APX-enabled VMMs to operate efficiently and correctly. Intel® APX accomplishes this by adding a new VMCS field, entitled VM-Exit Extended Instruction Information Field (EII).

The behavior of instructions which populate VM Exit Instruction Info Field are as follows:

- Any instruction which has a defined VM-Exit Instruction Info field will populate both the VM-Exit Instruction Info field and VM-Exit Extended Instruction Info field. The information in the VM-Exit Instruction Info field is incomplete for use by a VMM that enables Intel® APX for guest usage, since all register ID fields will contain legacy, truncated 4-bit register IDs, instead of full 5-bit register IDs. As such, an Intel® APX-enabled VMM should only use and rely on VM-Exit Extended Instruction Info. A VMM that does not enable Intel® APX for guest usage is free to use the legacy VM-Exit Instruction Info, since it is informationally complete if Intel® APX is not enabled.
- Any instruction which has a defined VM Exit Qualification field which contains register ID info will continue to populate this info in a legacy-compatible way, although the defined format of this field adds an additional register ID bit that had been previously un-defined/reserved. As such, an Intel® APX-enabled VMM should use this field according to the new format, so that it considers a potential 5-bit register ID. A non-Intel® APX enabled VMM is free to continue using the legacy definition of the field, since lack of Intel® APX enabling will guarantee that register IDs are only 4-bits, maximum.

The VMCS is extended with a new 64-bit field (encoding 0x2406/0x2407) called the VM-Exit Extended Instruction-Information (EII) field. The field will have space for a total of 4 register IDs (reg1, reg2, base, index) to match the current capabilities of all the existing register fields in the VM-Exit Instruction-Information field.

Any Intel® APX-aware VMM should only use this new EII field to find the full 5-bit register IDs that correspond to decoded reg operands of existing instructions. A non-Intel® APX-enabled VMM (which is not responsible for managing EGPRs) can continue to use the legacy VM Exit Instruction Info field, as it always has.

The purpose of dual-fields is to keep from perturbing legacy VMMs in any way, while allowing newer, Intel® APX-enabled VMMs to have all functional information to support 32 general purpose registers.

In all VMCS fields, the 5-bit register ID encodings of each reg-field are represented as follows:

Table 1: Intel® APX-Extended Architectural register ID Encodings for VMCS

0. RAX	8. R8	16. R16	24. R24
1. RCX	9. R9	17. R17	25. R25
2. RDX	10. R10	18. R18	26. R26
3. RBX	11. R11	19. R19	27. R27
4. RSP	12. R12	20. R20	28. R28
5. RBP	13. R13	21. R21	29. R29
6. RSI	14. R14	22. R22	30. R30
7. RDI	15. R15	23. R23	31. R31

2.4.1 VM EXIT QUALIFICATION FIELD DEFINITION FOR MOV DR

Table 2: Legacy VM Exit Qualification Field Definition for MOV DR

Bits	Name	Meaning
2:0	DR Number	Debug Register Number
3	RESERVED	Not currently defined
4	Direction	Direction of access: • 0 = MOV to DR • 1 = MOV from DR
7:5	RESERVED	Not currently defined
11:8	GPR	GPR used with MOV DR • 4-bit register ID
63:12	RESERVED	Reserved/un-defined (0's)

Table 3: Intel® APX-Extended VM Exit Qualification Field Definition for MOV DR

Bits	Name	Meaning
2:0	DR Number	Debug Register Number
3	RESERVED	Not currently defined

4	Direction	Direction of access: • 0 = MOV to DR • 1 = MOV from DR
7:5	RESERVED	Not currently defined
12:8	GPR	GPR used with MOV DR • 5-bit register ID
63:13	RESERVED	Reserved/un-defined (0's)

2.4.2 VM EXIT QUALIFICATION FIELD DEFINITION FOR MOV CR, LMSW, CLTS

Table 4: Legacy VM Exit Qualification Field Definition for MOV CR, LMSW, CLTS

Bits	Name	Meaning
3:0	CR Number	Control Register Number
5:4	Access Type	Access type: • 0 = MOV to CR • 1 = MOV from CR • 2 = CLTS • 3 = LMSW
6	LMSW Operand Type	Mem/Reg Indicator: • 0 = Register • 1 = Memory For CLTS and MOV CR, always 0
7	RESERVED	Not currently defined
11:8	GPR	GPR used with MOV DR • 4-bit register ID For CLTS/LMSW, cleared to 0
15:12	RESERVED	Reserved/un-defined (0's)
31:16	Source Data	Source Data: • LMSW: The LMSW source data • CLTS: Cleared to 0 • MOV CR: Cleared to 0
63:32	RESERVED	Reserved/un-defined (0's)

Table 5: Intel® APX-Enabled VM Exit Qualification Field Definition for MOV CR, LMSW, CLTS

Bits	Name	Meaning
3:0	CR Number	Control Register Number
5:4	Access Type	Access type: • 0 = MOV to CR • 1 = MOV from CR • 2 = CLTS

		• 3 = LMSW
6	LMSW Operand Type	Mem/Reg Indicator: • 0 = Register • 1 = Memory For CLTS and MOV CR, always 0
7	RESERVED	Not currently defined
12:8	GPR	GPR used with MOV DR • 5-bit register ID For CLTS/LMSW, cleared to 0
15:13	RESERVED	Reserved/un-defined (0's)
31:16	Source Data	Source Data: • LMSW: The LMSW source data • CLTS: Cleared to 0 • MOV CR: Cleared to 0
63:32	RESERVED	Reserved/un-defined (0's)

2.4.3 VMCS VM-EXIT EXTENDED INSTRUCTION INFO FIELD (EII)

Table 6: VMCS VM-Exit Extended Instruction Info Field (EII)

Bits	Name	Meaning
1:0	Scale	Scale: • 0: No scaling • 1: Scale by 2 • 2: Scale by 4 • 3: Scale by 8 (64-bit CPUs only) Undefined for instructions with no index register.
3:2	ASIZE	Address size (ASIZE): • 0: 16-bit • 1: 32-bit • 2: 64-bit (64-bit CPUs only) Others values not used/defined.
4	Mem/Reg	Mem/Reg Indicator: • 0 = Memory • 1 = Register
6:5	OSIZE	Operand size (OSIZE): • 0: 16-bit

		• 1: 32-bit • 2: 64-bit (64-bit CPUs only) Others values not used/defined.
9:7	Segment	Segment register: • 0: ES • 1: CS • 2: SS • 3: DS • 4: FS • 5: GS Other values not used/defined.
10	IndexInvalid	Index reg invalid indicator: • 0: valid • 1: invalid
11	BaseInvalid	Base reg invalid indicator: • 0: valid • 1: invalid
15:12	RESERVED	Reserved/un-defined (0's)
20:16	Reg1	5-bit register ID for Reg1, if applicable
23:21	RESERVED	Reserved/un-defined (0's)
28:24	Index	5-bit register ID for Index, if applicable
31:29	RESERVED	Reserved/un-defined (0's)
36:32	Base	5-bit register ID for Base, if applicable
39:37	RESERVED	Reserved/un-defined (0's)
44:40	Reg2	5-bit register ID for Reg2, if applicable
47:45	RESERVED	Reserved/un-defined (0's)
63:48	RESERVED	Reserved/un-defined (0's)