[] | *risc-v_logo.svg*

RISC-V IOPMP Architecture Specification

RISC-V IOPMP Task Group

Version 1.0.0-draft6, April, 2024: This document is in development. Assume everything can change.
See http://riscv.org/spec-state for details.

Table of Contents

Preamble

This document is in the Development state

Assume everything can change. This draft specification will change before being accepted as standard, so implementations made to this draft specification will likely not conform to the future standard.

Copyright and license information

Contributors

This RISC-V specification has been contributed to directly or indirectly by:

Many…

Chapter 1. Introduction

This document describes a mechanism to improve the security of a platform. In a platform, the bus initiators on it can access the target devices, just like a RISC-V hart. The introduction of I/O agents like the DMA (Direct Memory Access Unit) to systems improves performance but exposes the system to vulnerabilities such as DMA attacks. In the RISC-V eco-system, there already exists the PMP/ePMP which provides standard protection scheme for accesses from a RISC-V hart to the physical address space, but there is not a likewise standard for safeguarding non-CPU initiators. Here we propose the Physical Memory Protection Unit of Input/Output Devices, IOPMP for short, to control the accesses issued from the bus initiators.

IOPMP is considered a hardware component in a bus fabric. But why is a pure-software solution not enough? For a RISC-V-based platform, a software solution mainly refers to the security monitor, a program running on the M-mode in charge of handling security-related requests. Once a requirement from another mode asks for a DMA transfer, for example, the security monitor checks if the requirement satisfies all the security rules and then decides whether the requirement is legal. Only a legal requirement will be performed. However, the check could take a long time when the requirement is not as simple as a DMA transfer. A GPU, for example, can take a piece of program to run. Generally, examining whether a program violates access rules is an NP-hard problem. Even though we only consider the average execution time, the latency is not tolerable in most cases. A hardware component that can check accesses on the fly becomes a reasonable solution. That is the subject of this document, IOPMP.

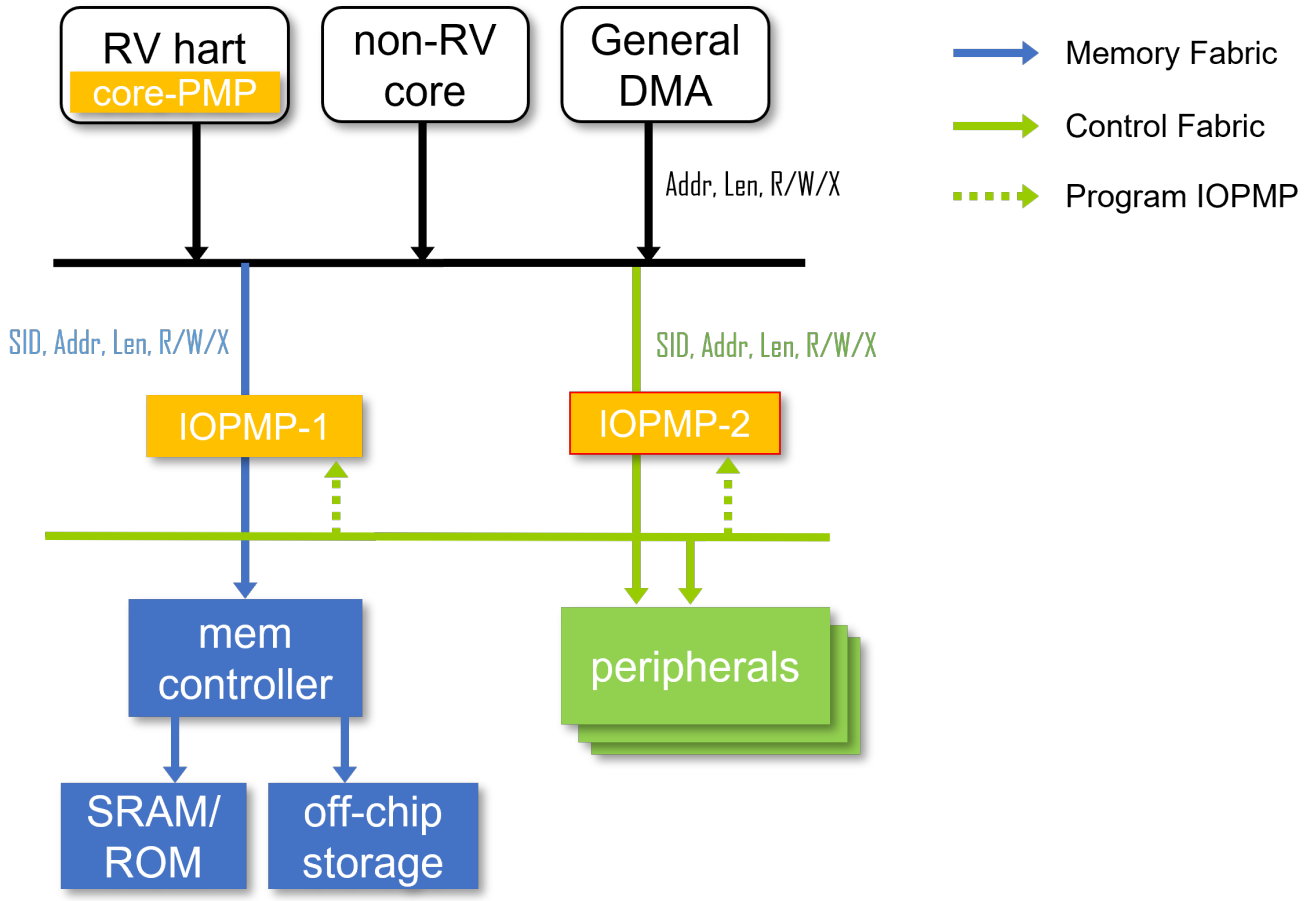

Figure 1. Examplary Integration of IOPMP(s) in System.

Chapter 2. Terminology and Concepts

This document refers to the term “secure monitor” as the software responsible for managing security-related tasks, including the programming of IOPMPs. The secure monitor is not restricted to operating on a single CPU or hart; instead, it has the flexibility to be distributed across multiple CPUs.

Glossary/ Acronyms	
Term	**Description**
DC	don’t care
IMP	implementation-dependent
MMIO	memory mapped input/output devices
NA4	naturally aligned four-byte region, one of the address matching mode used in RISC-V PMP and IOPMP
NAPOT	naturally aligned power-of-2 region, one of the address matching mode used in RISC-V PMP and IOPMP
N/A	not available
RX	receiver
TOR	top boundary of an arbitrary range, one of the address matching mode used in RISC-V PMP and IOPMP
TX	transmitter
WARL	write any read legal
W1C	write '1' clear
W1CS	write '1' clear and sticky to 0
W1S	write '1' set
W1SS	write '1' set and sticky to 1
X(n)	the n-th register in the register array X, which starts from 0.
X[n]	the n-th bit of a register X or register field X
X[n : m]	the n-th to m-th bits of a register X or register field X.

2.1. Request-Role-ID and Transaction

Request Role ID, RRID for short, is a unique ID to identify a system-defined security context. For example, a unique RRID can be a bus initiator or a group of bus initiators with the same permission. When a bus initiator wants to access a piece of memory, it issues a transaction. A transaction should be tagged with an RRID to identify the issuing bus initiator. We will discuss about the exception in the next section. Tagging bus initiators with RRID could be implementation-dependent. The number of bits of an RRID is implementation-dependent as well. If different channels or modes of a bus initiator could be granted different access permissions, they can have

its own RRID.

2.2. Source-Enforcement

If all transactions going through the IOPMP are issued by the same bus initiator or a set of bus initiators with the same permission, the Request-Role-ID can be ignored on the bus initiator side and the above transactions. In the case, we denote the IOPMP performs source enforcement, IOPMP-SE for short.

2.3. Initiator Port, Receiver Port and Control Port

An IOPMP has at least an initiator port, at least a receiver port and one control port. A receiver port is where a transaction goes into the IOPMP, and a initiator port is where a transaction leaves it if the transaction passes all the checks. The control port is used to program the IOPMP.

2.4. Memory Domain

An RRID is an abstract representation of a transaction source. It encompasses one or more transaction initiators that are granted identical permissions. On the other hand, a Memory Domain, MD for short, is an abstract representation of a transaction destination that groups a set of memory regions for a specific purpose. MDs are indexed from zero. For example, a network interface controller, NIC, may have three memory regions: an RX region, a TX region, and a region of control registers. We could group them into one MD. If a processor can fully control the NIC, it can be associated with the MD. An RRID associated with a MD doesn't mean having full permissions on all memory regions of the MD. The permission of each region is defined in the corresponding IOPMP entry. Additionally, there is an extension to adhere the permission to the MD that will be introduced in the Appendix A3.

It's important to note that, generally speaking, a single RRID can be associated with multiple Memory Domains (MDs), and vice versa. However, certain models may impose restrictions on this flexibility, which will be discussed in the following chapter.

2.5. IOPMP Entry and IOPMP Entry Array

The IOPMP entry array, a fundamental structure of an IOPMP, contains IOPMP entries. Each entry, starting from an index of zero, defines how to check a transaction. An entry includes a specified memory region and the corresponding read/write permissions.

IOPMP entry encodes the memory region in the same way as the RISC-V PMP, which are OFF, NA4, NAPOT, and TOR. Please refer to the RISC-V unprivileged spec for the details of the encoding schemes.

Since the address encoding scheme of TOR refers to the previous entry's memory region, which is not in the same memory domain, it would cause two kinds of unexpected results. If the first entry of a memory domain selects TOR, the entry refers to the previous memory domain. When the previous memory domain may not change unexpectedly, the region of this entry will be altered. To prevent the

> unexpected change of memory region, one should avoid adopting TOR for the first entry of a memory domain. The second issue is that the memory region of the last entry is referred by the next memory domain. To avoid it, one can set an OFF for the last entry of a memory domain with the maximal address.

Memory domains are a way of dividing the entry array into different subarrays. Each entry in the array can belong to at most one memory domain, while a memory domain could have multiple entries.

When an RRID is associated with a Memory Domain, it is also inherently associated with all the entries that belong to that MD. An RRID could be associated with multiple Memory Domains, and one Memory Domain could be associated with multiple RRIDs.

2.6. Priority and Matching Logic

IOPMP entries exhibit partial prioritization. Entries with indices smaller than **HWCFG2.prio_entry** are prioritized according to their index, with smaller indices having higher priority. These entries are referred to as priority entries. Conversely, entries with indices greater than or equal to **prio_entry** are treated equally and assigned the lowest priority. These entries are referred to as non-prioritized entries. The value of **prio_entry** is implement-dependent.

> The specification incorporates both priority and non-priority entries due to considerations of security, latency, and area. Priority entries, which are locked, safeguard the most sensitive data, even in the event of secure software being compromised. However, implementing a large number of these priority entries results in higher latency and increased area usage. On the other hand, non-priority entries are treated equally and can be cached in smaller numbers. This approach reduces the amortized latency, power consumption, and area when the locality is sufficiently high. Thus, the mix of entry types in the specification allows for a balance between security and performance.

The entry with the highest priority that (1) matches any byte of the incoming transaction and (2) is associated with the RRID carried by the transaction determines whether the transaction is legal. If the matching entry is priority entry, the matching entry must match all bytes of a transaction, or the transaction is illegal, irrespective of its permission.

Let’s consider a non-priority entry matching all bytes of a transaction. It is legal if the entry grants the transaction permission to operate. If the entry doesn’t grant it permission but suppresses to trigger the corresponding interrupt or to respond by an error, this entry is also considered a hit. If no such above entry exists, the transaction is illegal with error code = "not hit" (0x05).

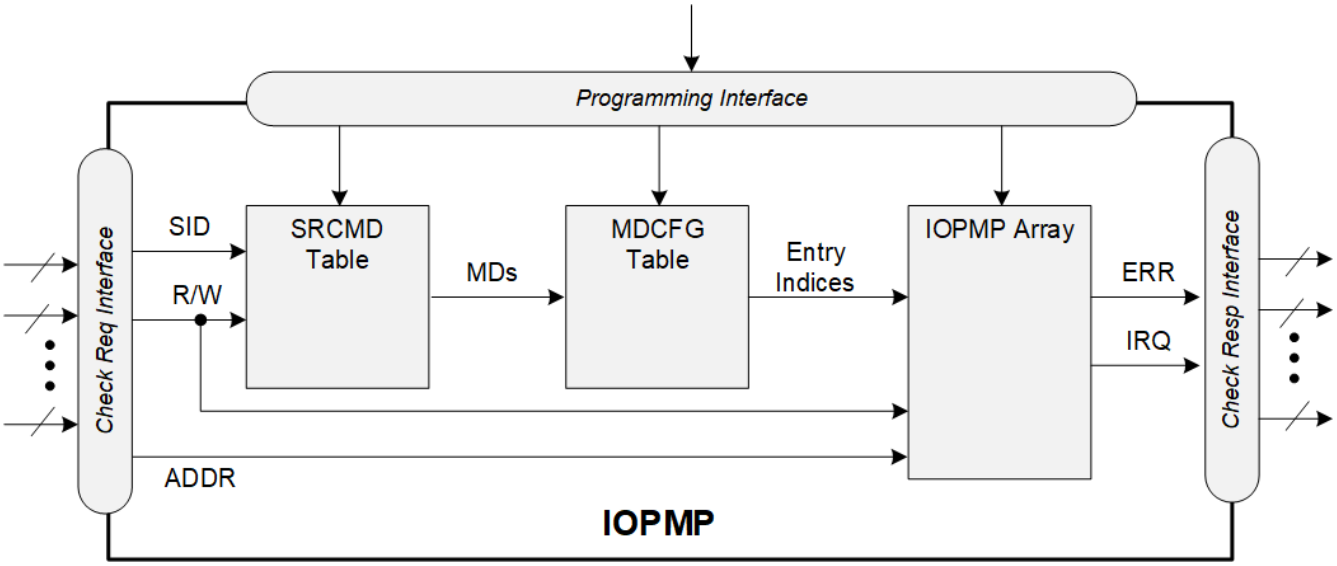

Figure 2. IOPMP Block Diagram.

2.7. Error Reactions

When an IOPMP detects an illegal transaction, it could initiate three of the following actions. Firstly, the IOPMP can return a bus error (or a decode error) or a success with an implementation-defined value. Secondly, it could trigger an interrupt. Lastly, it could record the error. They are defined in the register **ERR_CFG**. In addition, an IOPMP has the capability to trigger an interrupt when it detects an illegal access. Specifically, if **ERR_CFG.ie** is enabled and **ERR_CFG.ire** is set to 1, an interrupt is triggered for an illegal read access. Similarly, if **ie** is enabled and **ERR_CFG.iwe** is set to 1, an interrupt is triggered for an illegal write access. Furthermore, an optional **ERR_CFG.ixe** bit is for illegal instruction fetch and is implemented only for **HWCFG0.chk_x** = 1. **ire** is no longer to trigger an interrupt for an illegal instruction fetch when **chk_x** = 1. If **ie** is enabled and **ixe** is set to 1, an interrupt is triggered for an illegal instruction fetch. Regardless of whether **ie** is set to 1, **ERR_REQINFO.ip** will be set to 1 for an illegal read with **ire** = 1, an illegal write with **iwe** = 1 or an illegal instruction fetch with **ixe** = 1. When **ip** is set, no new interrupt will be triggered, and the error capture record (registers **ERR_XXX**) is kept valid and unchanged until the bit is cleared.

Compared to the local interrupt controls, **ire**, **iwe**, and **ixe** are considered global. Every entry *i* has three optional interrupt-suppressing bits in register **ENTRY_CFG(*i*)**: **sire**, **siwe**, and **sixe** to suppress interrupts due to illegal reads, illegal writes, and illegal instruction fetches caught by the entry, respectively. **HWCFG0.peis** is 1 if an implementation supports **sire**, **siwe**, or **sixe**. For an access that is not granted enough permission by any entries but there is a non-priority entry matching all bytes of the access and suppressing the corresponding interrupt, the interrupt won't be triggered. There is no interrupt suppression for an access matching no entry. The above local interrupt-suppressing bits are optional, and any unimplemented interrupt-suppressing bits should be wired to zero. An illegal access triggers an interrupt only when no suppressing bits regarding the access are set.

Additionally, the bus error response behavior on an IOPMP violation can be optionally defined globally via **ERR_CFG** register or locally through each **ENTRY_CFG** register. There defines three bits, **ERR_CFG.rre**, **ERR_CFG.rwe**, **ERR_CFG.rxe**, to hint if a bus shall suppress the bus error on any read access, write access, or instruction fetch that violates IOPMP rules. Transactions that violates the IOPMP rule will by default yield a bus error. The IOPMP will signal the bus to the

presence of a read violation but will suppress the bus error if **ERR_CFG.rre** is implemented and set to 1 on a violation. User-defined suppression behavior allows, for example, a read response of 0x0. Likewise, the bus error response on an illegal write or instruction fetch.

Similarly, the bus error response behavior can be configured per each IOPMP entry. Every entry *i* has three optional bus error-suppressing bits in register **ENTRY_CFG(*i*)**, **esre**, **eswe**, and **esxe** to suppress bus error response due to illegal reads, illegal writes and illegal instruction fetchs on the corresponding the entry, respectively. **HWCFG0.pees** is 1 if an implementation supports **esre**, **eswe**, and **esxe**. Regardless of the value on **ERR_CFG.rre**, IOPMP will indicate a "bus error suppression" when **esre** on an entry is set to 1. The suppression behavior is also user defined. Like wise for the bus error response for a write violation or an illegal instruction fetch.

The error capture record maintains the specifics of the first illegal access detected, except the following two conditions are held: (1) any interrupt-suppress bit regarding the access is set, and (2) no bus error is returned. New error capture only occurs when there is no currently pending error, namely **ERR_REQINFO.ip** is '0'. If there exists an pending error (**ip** is '1'□, the record will not be updated, even if a new illegal access is detected. In other words, **ip** indicates whether the content of the capture record is valid and should be intentionally cleared in order to capture subsequent illegal accesses. All fields in the error capture record are optional. If a field is not implemented, it should be wired to zero.

Chapter 3. IOPMP Models and Configuration Protection

The spec offers several IOPMP configuration models to accommodate varied platforms. Users can choose one of the models that best fits the use cases, including those for low area, low power, low latency, high throughput, high portability, and other criteria.

3.1. Full Model

When a Full model IOPMP receives a transaction with RRID *s*, IOPMP first lookups SRCMD table to find out all the memory domains associated to requestor *s*. An IOPMP instance can support up to 65,535 requestor, the actual number of requestor can be implementation-defined and is indicated in **HWCFG1** register. Each entry in SRCMD table defines the mapping of MDs to a specific requestor with RRID *s*. An SRCMD entry must impelment an **SRCMD_EN(*s*)** register. If SPS extension described in Appendix A3 is supported, **SRCMD_R(*s*)** and **SRCMD_W(*s*)** must be implemented. If the number of MDs is more than 31, **SRCMD_ENH(*s*)** must be implemented, same for **SRCMD_RH(*s*)** and **SRCMD_WH(*s*)** if SPS extension is implemented.

For easier description, **SRCMD(*s*)** is a 64-bit register representing the concatenation of **SRCMD_ENH(*s*)** for the higher word and **SRCMD_EN(*s*)** for the lower word. Field **SRCMD(*s*).md** is the concatenation of **SRCMD_ENH(*s*).mdh** and **SRCMD_EN(*s*).md**, and bit **SRCMD(*s*).l** is bit **SRCMD_EN(*s*).l**.

Field **SRCMD(*s*).md** is a bitmapped field and has up to 63 bits. Bit **md[*j*]** in **SRCMD(*s*)** indicates if MD *j* is associated with RRID *s*. For unimplemented memory domains, the corresponding bits should be zero. A Full model IOPMP supports up to 63 memory domains. For a system requiring more memory domains than 63, please refer to Appendix A2.

When a transaction with RRID *s* arrives at an IOPMP, the IOPMP retrieves all associated MDs with RRID *s* by looking up SRCMD table. Then, by using MDCFG table, the IOPMP can obtain all entries for a MD. MDCFG table, viewed as a partition of the entries in the IOPMP, contains an array of registers. Each register in this array, denoted as **MDCFG(*m*)**, corresponds to a specific memory domain *m*. Field **MDCFG(*m*).t** indicates the top index of IOPMP entry belonging to the memory domain *m*. An IOPMP entry with index *j* belongs to MD *m* if **MDCFG(*m*-1).t** ≤ *j* < **MDCFG(*m*).t**, where *m* > 0. MD 0 owns the IOPMP entries with index *j* < **MDCFG(0).t**.

After retrieving all associated IOPMP entries, a Full model IOPMP checks the transaction according to these entries.

3.2. Configuration Protection

The term 'lock' refers to a hardware feature that renders one or more fields or registers nonprogrammable until the IOPMP is reset. This feature serves to maintain the integrity of essential configurations in the event of a compromise of secure software. In cases where a lock bit is programmable, it is expected to be reset to '0' and sticky to '1' upon receiving a write of '1'.

3.2.1. SRCMD Table Protection

The associations between a specific MD *j* and all RRIDs can be effectively locked to prevent any subsequent modifications, ensuring that **SRCMD(*s*).md[*j*]** remains nonprogrammable for all *s*. The registers MDLCK and MDLCKH are specifically to secure these associations. To lock MD *j*, one can set **MDLCK.md[*j*]** for *j*<31 or set **MDLCKH.mdh[*j*-31]** for *j* ≥ 31.

Bit **MDLCK.l** is a stickly to 1 and indicates if **MDLCK** is locked.

MDLCK.md is optional, if not implemented, **MDLCK.md** should be wired to 0 and **MDLCK.l** should be wired to 1.

Besides, every **SRCMD_EN(*s*)** register has a bit **l**, which is used to lock registers **SRCMD_EN(*s*)**, **SRCMD_ENH(*s*)**, **SRCMD_R(*s*)**, **SRCMD_RH(*s*)**, **SRCMD_W(*s*)**, and **SRCMD_WH(*s*)** if any.

Locking SRCMD table in either way can prevent the table from being altered accidentally or maliciously. By locking the association of the MD containing the configuration regions of a component, one can prevent the component from being configured by unwanted RRIDs. To make it more secure, one can use another high-priority MD containing the same regions but no permission, let it be associated with all unwanted RRIDs, and then lock the two MDs' associations by **MDLCK /MDLCKH**. By adopting this approach, it is possible to safeguard the configuration from direct access by potentially compromised security software.

3.2.2. MDCFG Table Protection

Register **MDCFGLCK** is designed to partially or fully lock MDCFG table. **MDCFGLCK** is consisted of two fields: **MDCFGLCK.l** and **MDCFGLCK.f**. **MDCFG(*j*)** is locked if *j*< **MDCFGLCK.f**. **MDCFGLCK.f** is incremental-only. Any smaller value can not be written into it. Bit **MDCFGLCK.l** is used to lock **MDCFGLCK**.

If a MD is locked, while its preceding MD is not locked, it could lead to the potential addition or removal of unexpected entries within the locked MD. This can occur by manipulating the top index of the preceding unlocked MD. Thus, the specification asks that one MD is locked, all its preceding MDs should be locked.

3.2.3. Entry Protection

IOPMP entry protection is also related to the other IOPMP entries belonging to the same memory domain. For a MD, locked entries should be placed in the higher priority. Otherwise, when the secure monitor is compromised, one unlocked entry in higher priority can overwrite all the other locked or non-locked entries in lower priority. A register **ENTRYLCK** is define to indicate the number of nonprogrammable entries. **ENTRYLCK** register has two fields: **ENTRYLCK.l** and **ENTRYLCK.f**. Any IOPMP entry with index *i* ≤ **ENTRYLCK.f** is not programmable. **ENTRYLCK.f** is initialized to 0 and can be increased only when written. Besides, **ENTRYLCK.l** is the lock to **ENTRYLCK.f** and itself. If **ENTRYLCK** is hardwired, **ENTRYLCK.l** should be wired to 1.

Chapter 4. Other IOPMP Models

4.1. Tables Reduction

Full model comprises two tables and an array, offering substantial flexibility for configuring an IOPMP. However, this comes at the cost of increased latency and area usage. The chapter presents the other models designed to simplify these tables, thereby catering to diverse design requirements.

The IOPMP array functions as the primary repository for IOPMP entries with it's size adjustable. Sharing memory domains among RRIDs can result in shared entries between them. This sharing approach has the potential to decrease the overall footprint of the IOPMP array. Nevertheless, if a design doesn't encompass many shared regions, simplifying the SRCMD table, as demonstrated in the isolation and compact-*k* models, could be a viable consideration.

As to MDCFG table, its primary function is to partition the entries within the IOPMP. Besides programming each **MDCFG(*m*).t** for every MD *m*, an alternative approach involves evenly distributing entries across each MD. This distribution method is implemented in the Rapid-*k*, dynamic-*k*, and compact-*k* models.

4.2. Rapid-k Model

The Rapid-*k* model is based on Full model and aims to reduce latency by eliminating the need to lookup the MDCFG table. In this model, each memory domain has exactly *k* entries where *k* is implementation-dependent and non-programmable. Only MDCFG(0) is utilized, rendering the implementation of MDCFG(j) unnecessary for j greater than 0. **MDCFG(0).t** store the value *k*, while **MDCFGLCK.f** is ignored and **MDCFGLCK.l** should be 1.

4.3. Dynamic-k Model

Dynamic-*k* model is based on rapid-*k* model, except the *k* value is programmable. That is, **MDCFG(0).t** is WARL and accepts a limited set of values. **MDCFGLCK.f** is ignored, and **MDCFGLCK.l** indicates if **MDCFG(0).t** is still programmable or locked.

4.4. Isolation Model

The bitmap implementation of SRCMD table facilitates the sharing of regions between RRIDs. The Isolation model is specifically tailored for scenarios where there are minimal to no shared regions. In this model, each RRID is exactly associated with a signle MD, eliminating the necessity for SRCMD table lookups. Each RRID *i* is directly associated with MD *i*, resulting in advantages in terms of area, latency, and system complexity. However, duplication of entries occurs when shared regions are required, representing a potential drawback. In this model, SRCMD table and **MDLCK(H)** registers are omitted.

The number of RRIDs to support is bounded by the maximal number of MDs, 63.

There is no constraint imposed on MDCFG table and **MDCFGLCK** register.

4.5. Compact-k Model

The Compact-*k* model, being the most compact variant, is based on the Isolation model. It mandates that each MD has exactly *k* entries with *k* being non-programmable. Only **MDCFG(0)** is implemented. **MDCFG(0).t** holds the value *k*, **MDCFGLCK.f** is ignored and **MDCFGLCK.l** is 1.

4.6. Model Detections

To distinguish the above models, the user can read register **HWCFG0.model** to determine the current implemented IOPMP model.

Chapter 5. Registers

OFFSET	Register	Description
0x0000	INFO	
	VERSION	Indicates the specification and the IP vendor.
	IMPLEMENTATION	Indicates the implementation version.
	HWCFG0~2	Indicate the configurations of current IOPMP instance.
	ENTRYOFFSET	Indicates the internal address offsets of each table.
	Programming Protection	
	MDSTALL/MDSTALLH	(Optional) Stall and resume the transaction checks when programming the IOPMP.
	RRIDSCP	
	Configuration Protection	
	MDLCK/MDLCKH	Lock register for SRCMD table.
	MDCFGLCK	Lock register for MDCFG table.
	ENTRYLCK	Lock register for IOPMP entry array.
	Error Reporting	
	ERR_CFG	Indicates the reactions for the violaions
	ERR_REQINFO	Indicate the information regarding the first captured violation.
	ERR_REQID	
	ERR_REQADDR/ERR_REQADDRH	
	ERR_MFR	(Optional) To retrieve which RRIDs make subsequent violations.
	ERR_USER(0~7)	(Optional) User-defined violation information.
0x0800	MDCFG Table, *m* =0…HWCFG0.md_num -1	
	MDCFG(*m*)	MD config register, which is to specify the indices of IOPMP entries belonging to a MD.

OFFSET	Register	Description
0x1000	SRCMD Table, *s* = 0...HWCFG1.rrid_num-1	
	SRCMD_EN(*s*)/SRCMD_ENH(*s*)	The bitmapped MD enabling register of the requestor *s* that **SRCMD_EN(*s*)[*j*]** indicates if the requestor is associated with MD *j* and **SRCMD_ENH(*s*)[*j*]** indicates if the requestor is associated with MD (*j*+31).
	SRCMD_R(*s*)/SRCMD_RH(*s*)	(Optional) bitmapped MD read eanble register, *s* corresponding to number of requestors, it indicates requestor *s* read permission on MDs.
	SRCMD_W(*s*)/SRCMD_WH(*s*)	(Optional) bitmapped MD write eanble register, *s* corresponding to number of requestors, it indicates requestor *s* write permission on MDs.
ENTRYOFFSET	Entry Array, *i* =0...HWCFG1.entry_num-1	
	ENTRY_ADDR(*i*)	The address (region) for entry *i*.
	ENTRY_ADDRH(*i*)	(Optional for 32-bit system).
	ENTRY_CFG(*i*)	The configuration of entry *i*.
	ENTRY_USER_CFG(*i*)	(Optional) extension to support user customized attributes.

If a field is optional and not implemented in an implemented register, it should return zero when read, unless otherwise specified. If an optional register is not implemented, the behavior is implementation-dependent, unless otherwise specified.

5.1. INFO registers

INFO registers are use to indicate the IOPMP instance configuration info.

VERSION				
0x0000				
Field	Bits	R/W	Default	Description
vendor	23:0	R	IMP	The vendor ID
specver	31:24	R	IMP	The specification version

IMPLEMENTATION				
0x0004				
Field	Bits	R/W	Default	Description
impid	31:0	R	IMP	The implementation ID

HWCFG0
0x0008

Field	Bits	R/W	Default	Description
model	3:0	R	IMP	Indicate the iopmp instance model • 0x0: Full model: the number of MDCFG registers is equal to **HWCFG0.md_num**, all MDCFG registers are readable and writable. • 0x1: Rapid-k model: a single MDCFG register to indicate the k value, read only. • 0x2: Dynamic-k model: a single MDCFG register to indicate the k value, readable and writable. • 0x3: Isolation model: the number of MDCFG registers is equal to **HWCFG0.md_num**, all MDCFG registers are readable and writable. • 0x4: Compact-k model: a single MDCFG register to indicate the k value, read only.
tor_en	4:4	R	IMP	Indicate if TOR is supported
sps_en	5:5	R	IMP	Indicate secondary permission settings is supported; which are **SRCMD_R/RH(*i*)** and **SRCMD_W/WH** registers.
user_cfg_en	6:6	R	IMP	Indicate if user customized attributes is supported; which are **ENTRY_USER_CFG(*i*)** registers.
prient_prog	7:7	W1CS	IMP	A write-1-clear bit is sticky to 0 and indicates if **HWCFG2.prio_entry** is programmable. Reset to 1 if the implementation supports programmable **prio_entry**, otherwise, wired to 0.
rrid_transl_en	8:8	R	IMP	Indicate the if tagging a new RRID on the initiator port is supported
rrid_transl_prog	9:9	W1CS	IMP	A write-1-set bit is sticky to 0 and indicate if the field sid_transl is programmable. Support only for **rrid_transl_en**=1, otherwise, wired to 0.
chk_x	10:10	R	IMP	Indicate if the IOPMP implements the check of an instruction fetch. On **chk_x**=0, all fields of illegal instruction fetchs are ignored, including **HWCFG0.no_x**, **ERR_CFG.ixe**, **ERR_CFG.rxe**, **ENTRY_CFG(*i*).sixe**, **ENTRY_CFG(*i*).esxe**, and **ENTRY_CFG(*i*).x**. It should be wired to zero if there is no indication for an instruction fetch.
no_x	11:11	R	IMP	For **chk_x**=1, the IOPMP with **no_x**=1 always fails on an instruction fetch; otherwise, it should depend on **x**-bit in **ENTRY_CFG(*i*)**. For **chk_x**=0, **no_x** has no effect.

no_w	12:12	R	IMP	Indicate if the IOPMP always fails write accesses considered as as no rule matched.
stall_en	13:13	R	IMP	Indicate if the IOPMP implements stall-related features, which are **MDSTALL**, **MDSTALLH**, and **RRIDSCP** registers.
peis	14:14	R	IMP	Indicate if the IOPMP implements interrupt suppression per entry, including fields **sire**, **siwe**, and **sixe** in **ENTRY_CFG(*i*)**.
pees	15:15	R	IMP	Indicate if the IOPMP implements the error suppression per entry, including fields **esre**, **eswe**, and **esxe** in **ENTRY_CFG(*i*)**.
mfr_en	16:16	R	IMP	Indicate if the IOPMP implements Multi Faults Record Extension, that is **ERR_MFR** and **ERR_REQINFO.svc**.
rsv	23:16	ZERO	0	Must be zero on write, reserved for future
md_num	30:24	R	IMP	Indicate the supported number of MD in the instance
enable	31:31	W1SS	0	Indicate if the IOPMP checks transactions by default. If it is implemented, it should be initial to 0 and sticky to 1. If it is not implemented, it should be wired to 1.

HWCFG1				
0x000C				
Field	Bits	R/W	Default	Description
rrid_num	15:0	R	IMP	Indicate the supported number of RRID in the instance
entry_num	31:16	R	IMP	Indicate the supported number of entries in the instance

HWCFG2				
0x0010				
Field	Bits	R/W	Default	Description
prio_entry	15:0	WARL	IMP	Indicate the number of entries matched with priority. These rules should be placed in the lowest order. Within these rules, the lower order has a higher priority.
rrid_transl	31:16	WARL	IMP	The RRID tagged to outgoing transactions. Support only for **HWCFG0.rrid_transl_en**=1.

ENTRYOFFSET
0x0014

Field	Bits	R/W	Default	Description
offset	31:0	R	IMP	Indicate the offset address of the IOPMP array from the base of an IOPMP instance, a.k.a. the address of **VERSION**. Note: the offset is a signed number. That is, the IOPMP array can be placed in front of **VERSION**.

5.2. Programming Protection Registers

MDSTALL(H) and **RRIDSCP** registers are all optional and used to support atomicity issue while programming the IOPMP, as the IOPMP rule may not be updated in a single transaction.

MDSTALL				
0x0030				
Field	Bits	R/W	Default	Description
exempt	0:0	W	N/A	Stall transactions with exempt selected MDs, or Stall selected MDs.
is_stalled	0:0	R	0	After the last writing of **MDSTALL** (included) plus any following writing **RRIDSCP**, 1 indicates that all requested stalls take effect; otherwise, 0. After the last writing **MDSTALLH** (if any) and then **MDSTALL** by zero, 0 indicates that all transactions have been resumed; otherwise, 1.
md	31:1	WARL	0	Writting **md[*i*]**=1 selects MD *i*; reading **md[*i*]** = 1 means MD *i* selected.

MDSTALLH				
0x0034				
Field	Bits	R/W	Default	Description
mdh	31:0	WARL	0	Writting **mdh[*i*]**=1 selects MD (*i*+31); reading **mdh[*i*]** = 1 means MD (*i*+31) selected.

RRIDSCP				
0x0038				
Field	Bits	R/W	Default	Description
rrid	15:0	WARL	DC	RRID to select
rsv	29:16	ZERO	0	Must be zero on write, reserved for future

op	31:30	W	N/A	• 0: query • 1: stall transactions associated with selected RRID • 2: don't stall transactions associated with selected RRID • 3: reserved
stat	31:30	R	0	• 0: **RRIDSCP** not implemented • 1: transactions associated with selected RRID are stalled • 2: transactions associated with selected RRID not are stalled • 3: unimplemented or unselectable RRID

5.3. Configuration Protection Registers

MDLCK and **MDLCKH** are optional registers with a bitmap field to indicate which MDs are locked in SRCMD table.

MDLCK				
0x0040				
Field	Bits	R/W	Default	Description
l	0:0	W1SS	0	Lock bit to **MDLCK** and **MDLCKH** register.
md	31:1	WARL	0	**md[*j*]** is stickly to 1 and indicates if **SRCMD_EN(*i*).md[*j*]**, **SRCMD_R(*i*).md[*j*]** and **SRCMD_W(*i*).md[*j*]** are locked for all *i*.

MDLCKH				
0x0044				
Field	Bits	R/W	Default	Description
mdh	31:0	WARL	0	**mdh[*j*]** is stickly to 1 and indicates if **SRCMD_ENH(*i*).mdh[*j*]**, **SRCMD_RH(*i*).mdh[*j*]** and **SRCMD_WH(*i*).mdh[*j*]** are locked for all *i*.

MDCFGLCK is the lock register to MDCFG table.

MDCFGLCK				
0x0048				
Field	Bits	R/W	Default	Description

l	0:0	W1SS	0	Lock bit to **MDCFGLCK** register. For Rapid-k model and Compact-k model, **l** should be 1. For Dynamic-K model, **l** indicates if **MDCFG(0).t** is still programmable or locked.
f	7:1	WARL	IMP	Indicate the number of locked MDCFG entries – **MDCFG(*i*)** is locked for *i* < **f**. For Rapid-k model, Dynamic-k model and Compact-k model, **f** is ignored. For the rest of the models, the field should be monotonically increased only until the next reset cycle.
rsv	31:8	ZERO	0	Must be zero on write, reserved for future

ENTRYLCK is the lock register to entry array.

ENTRYLCK				
0x004C				
Field	Bits	R/W	Default	Description
l	0:0	W1SS	0	Lock bit to **ENTRYLCK** register.
f	16:1	WARL	IMP	Indicate the number of locked IOPMP entries – **ENTRY_ADDR(*i*)**, **ENTRY_ADDRH(*i*)**, **ENTRY_CFG(*i*)**, and **ENTRY_USER_CFG(*i*)** are locked for *i* < **f**. The field should be monotonically increased only until the next reset cycle.
rsv	31:17	ZERO	0	Must be zero on write, reserved for future

5.4. Error Capture Registers

ERR_CFG is a read/write WARL register used to configure the global error reporting behavior on an IOPMP violation.

ERR_CFG				
0x0060				
Field	Bits	R/W	Default	Description
l	0:0	W1SS	0	Lock fields to **ERR_CFG** register
ie	1:1	RW	0	Enable the interrupt of the IOPMP
ire	2:2	WARL	0	To trigger an interrupt on an illegal read access
iwe	3:3	WARL	0	To trigger an interrupt on an illegal write access
ixe	4:4	WARL	0	To trigger an interrupt on an illegal instruction fetch.

rre	5:5	WARL	0	Response on an illegal read accesses • 0x0: respond an implementation-dependent error, such as a bus error • 0x1: respond a success with a pre-defined value to the initiator instead of an error
rwe	6:6	WARL	0	Response on an illegal write access: • 0x0: respond an error if a response is needed • 0x1: respond a success to the initiator instead of an error if a response is needed Implemented only for **HWCFG0.chk_x**=1.
rxe	7:7	WARL	0	Response on an illegal instruction fetch: • 0x0: respond an error • 0x1: respond a success with a pre-defined value to the initiator instead of an error. Implemented only for **HWCFG0.chk_x**=1.
rsv	31:8	ZERO	0	Must be zero on write, reserved for future

An implementation can optionally support the full and partial functions defined in the fields **ree**, **rwe**, and **rxe**. **ERR_REQINFO** captures more detailed error infomation.

ERR_REQINFO				
0x0064				
Field	Bits	R/W	Default	Description
ip	0:0	R	0	Indicate if an interrupt is pending on read. for 1, the illegal capture recorder (**ERR_REQID**, **ERR_REQADDR**, **ERR_REQADDRH**, and fields in this register) has valid content and won't be updated even on subsequent violations.
ip	0:0	W1C	N/A	Write 1 clears the bit and the illegal recorder reactivates. Write 0 causes no effect on the bit.
ttype	2:1	R	0	Indicated the transaction type • 0x00 = reserved • 0x01 = read access • 0x02 = write access • 0x03 = instruction fetch
rsv1	3:3	ZERO	0	Must be zero on write, reserved for future

etype	6:4	R	0	Indicated the type of violation • 0x00 = no error • 0x01 = illegal read access • 0x02 = illegal write access • 0x03 = illegal instruction fetch • 0x04 = partial hit on a priority rule • 0x05 = not hit any rule • 0x06 = unknown RRID • 0x07 = user-defined error
svc	7:7	R	0	Indicate there is a subsequent violation caught in **ERR_MFR**. Implemented only for **HWCFG0.mfr_en**=1, otherwise, ZERO.
rsv2	30:8	ZERO	0	Must be zero on write, reserved for future

When the bus matrix doesn't have a signal to indicate an instruction fetch, the **ttype** and **etype** can never return "instruction fetch" (0x03) and "instruction fetch error" (0x03), respectively.

ERR_REQADDR and **ERR_REQADDRH** indicate the errored request address.

ERR_REQADDR				
0x0068				
Field	Bits	R/W	Default	Description
addr	31:0	R	DC	Indicate the errored address[33:2]

ERR_REQADDRH				
0x006C				
Field	Bits	R/W	Default	Description
addrh	31:0	R	DC	Indicate the errored address[65:34]

ERR_REQID indicates the errored RRID and entry index.

ERR_REQID				
0x0070				
Field	Bits	R/W	Default	Description
rrid	15:0	R	DC	Indicate the errored RRID.
eid	31:16	R	DC	Indicates the index pointing to the entry that catches the violation. If no entry is hit, i.e., **etype**=0x05, the value of this field is invalid.

ERR_MFR is an optional register. If Multi-Faults Record Extension is enabled (**HWCFG0.mfr_en**=1),

ERR_MFR can be used to retrieve which RRIDs make subsequent violations.

ERR_MFR				
0x0074				
Field	Bits	R/W	Default	Description
svw	15:0	R	DC	Subsequent violations in the window indexed by **svi**. **svw[*j*]**=1 for the at lease one subsequent violation issued from RRID= **svi***16 + *j*.
svi	27:16	RW	0	Window's index to search subsequent violations. When read, **svi** moves forward until one subsequent violation is found or **svi** has been rounded back to the same value. After read, the window's content, **svw**, should be clean.
rsv	30:28	ZERO	0	Must be zero on write, reserved for future
svs	31:31	R	DC	the status of this window's content: • 0x0 : no subsequent violation found • 0x1 : subsequent violation found

ERR_USER(0..7) are optional registers to provide users to define their own error capture information.

ERR_USER(*i*)				
0x0080 + **0x04** * *i*, *i* = **0...7**				
Field	Bits	R/W	Default	Description
user	31:0	IMP	IMP	(Optional) user-defined registers

5.5. MDCFG Table

MDCFG table is a lookup to specify the number of IOPMP entries that is associated with each MD. For different models:

1. Full model: the number of MDCFG registers is equal to **HWCFG0.md_num**, all MDCFG registers are readable and writable.
2. Rapid-*k* model: a single MDCFG register to indicate the *k* value, read only. Only **MDCFG(0)** is implemented.
3. Dynamic-*k* model: a single MDCFG register to indicate the *k* value, readable and writable. Only **MDCFG(0)** is implemented.
4. Isolation model: the number of MDCFG registers is equal to **HWCFG0.md_num**, all MDCFG registers are readable and writable.
5. Compact-*k* model: a single MDCFG register to indicate the *k* value, read only. Only **MDCFG(0)** is implemented.

MDCFG(*m*), *m* = 0...HWCFG0.md_num-1, support up to 63 MDs				
0x0800 + (*m*)*4				
Field	Bits	R/W	Default	Description
t	15:0	WARL	DC/IMP	Indicate the top range of memory domain *m*. An IOPMP entry with index *j* belongs to MD *m* • If **MDCFG(*m*-1).t** ≤ *j* < **MDCFG(*m*).t,** where m>0. MD0 owns the IOPMP entries with index *j* < **MDCFG(0).t**. • If **MDCFG(*m*-1).t** >= **MDCFG(*m*).t**, then MD *m* is empty. • For rapid-*k*, dynamic-*k* and compact-*k* models, **MDCFG(0).t** indicates the number of IOPMP entries belongs to each MD, that is, the *k* value. The **MDCFG(*i*)** can be omitted for *i*>0.
rsv	31:16	ZERO	0	Must be zero on write, reserved for future

5.6. SRCMD Table Registers

Only full model, rapid-*k* model and dynamic-*k* model implement SRCMD table.

SRCMD_EN(*s*), *s* = 0...HWCFG1.rrid_num-1				
0x1000 + (*s*)*32				
Field	Bits	R/W	Default	Description
l	0:0	W1SS	0	A sticky lock bit. When set, locks **SRCMD_EN(*s*)**, **SRCMD_ENH(*s*)**, **SRCMD_R(*s*)**, **SRCMD_RH(*s*)**, **SRCMD_W(*s*)**, and **SRCMD_WH(*s*)** if any.
md	31:1	WARL	DC	**md[*j*]** = 1 indicates MD *j* is associated with RRID *s*.

SRCMD_ENH(*s*), *s* = 0...HWCFG1.rrid_num-1				
0x1004 + (*s*)*32				
Field	Bits	R/W	Default	Description
mdh	31:0	WARL	DC	**mdh[*j*]** = 1 indicates MD (*j*+31) is associated with RRID *s*.

SRCMD_R, **SRCMD_RH**, **SRCMD_W** and **SRCMD_WH** are optional registers; When SPS extension is enabled, the IOPMP checks both the R/W and the **ENTRY_CFG.r/w** permission and follows a fail-first rule.

SRCMD_R(*s*), *s* = 0...HWCFG1.rrid_num-1
0x1008 + (*s*)*32

Field	Bits	R/W	Default	Description
rsv	0:0	ZERO	0	Must be zero on write, reserved for future
md	31:1	WARL	DC	**md[*j*]** = 1 indicates RRID *s* has read permission to the corresponding MD *j*.

SRCMD_RH(s), *s* = 0...HWCFG1.rrid_num-1				
0x100C + (*s*)*32				
Field	Bits	R/W	Default	Description
mdh	31:0	WARL	DC	**mdh[*j*]** = 1 indicates RRID *s* has read permission to MD (*j*+31).

SRCMD_W(*s*), *s* = 0...HWCFG1.rrid_num-1				
0x1010 + (*s*)*32				
Field	Bits	R/W	Default	Description
rsv	0:0	ZERO	0	Must be zero on write, reserved for future
md	31:1	WARL	DC	**md[*j*]** = 1 indicates RRID *s* has write permission to the corresponding MD *j*.

SRCMD_WH(*s*), *s* = 0...HWCFG1.sid_num-1				
0x1014 + (*s*)*32				
Field	Bits	R/W	Default	Description
mdh	31:0	WARL	DC	**mdh[*j*]** = 1 indicates RRID *s* has write permission to MD (*j*+31).

5.7. Entry Array Registers

ENTRY_ADDR(*i*), *i* = 0...HWCFG1.entry_num-1				
ENTRYOFFSET + (*i*)*16				
Field	Bits	R/W	Default	Description
addr	31:0	WARL	DC	The physical address[33:2] of protected memory region.

ENTRY_ADDRH(*i*), *i* = 0...HWCFG1.entry_num-1				
ENTRYOFFSET + 0x4 + (*i*)*16				
Field	Bits	R/W	Default	Description
addrh	31:0	WARL	DC	The physical address[65:34] of protected memory region.

A complete 64-bit address consists of these two registers, **ENTRY_ADDR** and **ENTRY_ADDRH**.

However, an IOPMP can only manage a segment of space, so an implementation would have a certain number of the most significant bits that are the same among all entries. These bits are allowed to be hardwired.

ENTRY_CFG(*i*), *i* = 0...HWCFG1.entry_num-1				
ENTRYOFFSET + 0x8 + (*i*)*16				
Field	Bits	R/W	Default	Description
r	0:0	WARL	DC	The read permission to protected memory region
w	1:1			The write permission to the protected memory region
x	2:2			The instruction fetch permission to the protected memory region. Optional field, if unimplemented, write any read the same value as r field.
a	4:3	WARL	DC	The address mode of the IOPMP entry • 0x0: OFF • 0x1: TOR • 0x2: NA4 • 0x3: NAPOT
sire	5:5	WARL	0	To suppress interrupt for an illegal read access caught by the entry
siwe	6:6	WARL	0	Suppress interrupt for write violations caught by the entry
sixe	7:7	WARL	0	Suppress interrupt on an illegal instruction fetch caught by the entry
sere	8:8	WARL	0	Supress the (bus) error on an illegal read access caught by the entry • 0x0: the response by **ERR_CFG.rre** • 0x1: do not respond an error. User to define the behavior, e.g., respond a success with an implementation-dependent value to the initiator.
sewe	9:9	WARL	0	Supress the (bus) error on an illegal write access caught by the entry • 0x0: the response by **ERR_CFG.rwe** • 0x1: do not respond an error. User to define the behavior, e.g., respond a success if response is needed

sexe	10:10	WARL	0	Supress the (bus) error on an illegal instruction fetch caught by the entry • 0x0: the response by **ERR_CFG.rxe** • 0x1: do not respond an error. User to define the behavior, e.g., respond a success with an implementation-dependent value to the initiator.
rsv	31:11	ZERO	0	Must be zero on write, reserved for future

Bits, **r**, **w**, and **x**, grant read, write, or instruction fetch permission, respectively. Not each bit should be programmable. Some or all of them could be wired. Besides, an implementation can optionally impose constraints on their combinations. For example, **x** and **w** can't be 1 simultaneously.

ENTRY_USER_CFG implementation defined registers that allows users to define their own additional IOPMP check rules beside the rules defined in **ENTRY_CFG**.

ENTRY_USER_CFG(*i*), *i* =0...HWCFG1.entry_num-1				
ENTRYOFFSET + 0xC + (*i*)*16				
Field	Bits	R/W	Default	Description
im	31:0	IMP	IMP	User customized field

Chapter 6. Static Rules

TBD

Chapter 7. Programming IOPMPs

At times, it can be difficult or even impossible to configure all IOPMP settings when the system starts, especially before I/O agents are active or connected to the system. As a result, it is necessary to update IOPMP settings during runtime. This may occur when a device is enabled, disabled, added, removed, or when the accessible area of a device changes. When updating, it is important to avoid putting IOPMP in a transient metastable state due to incomplete settings. However, updating IOPMP settings often involves a series of control accesses, and if a transaction check occurs during the update, it can potentially create a vulnerability. It can be difficult for the security software to guarantee that no transactions are in progress from all related initiators. A false alarm could result in significant performance issues. This chapter describes an optional method for updating IOPMP's settings without intervening transaction initiators.

7.1. Atomicity Requirement

The term here "stable" refers to meeting the atomicity requirement. This implies that when updating an IOPMP, all transactions from input ports must be checked either before any changes or after completing all changes. Essentially, using partial settings in an IOPMP should be avoided. The succeeding sections will describe the mechanism to satisfy this requirement.

7.2. Programming Steps

The general approach to the atomicity requirement has three major steps, conceptually described as follows:

- Step 1: Stall related transactions. Before proceeding with any updates, delay checking the transactions that may be impacted.
- Step 2: Update IOPMP's settings.
- Step 3: Resume stalled transactions.

For step 1, it's important to verify if the necessary stalling transactions have taken place since they might not be instantaneous in certain implementations. Following this, execute the IOPMP update as step 2, and finally, resume all stalled transactions in step 3.

In some cases, Step 1 and Step 3 may be skipped as long as no transaction check can interrupt Step 2. Updating MDs associated with a specific RRID to other MDs is an example.

7.3. Stall Transactions

For Step 1, it's possible to postpone all transactions until all updates are finished. However, this could cause unrelated transactions to experience unnecessary delays. This might not be tolerable for devices that require low latency, like a display controller that periodically retrieves a frame from its video buffer. This section explains the mechanism that only stalls specific transactions to prevent the aforementioned scenario and ensure the atomicity requirement. All the features mentioned below are optional.

Since the stalls occur when updating is in progress, determining wheater a transaction's check should wait cannot be based on any IOPMP's configuration about to change. Therefore, the only information that can be relied upon for this decision is the RRID carried by the transaction. To simplify the following description, we use a conceptual signal called rrid_stall[*i*] to indicate whether the transaction with RRID=*i* must wait. Please note that it may not be an actual signal in practice and is not accessible directly for software.

A conceptual internal signal rrid_stall has the same number of bits as the RRIDs in the IOPMP. rrid_stall is generated by the bit **MDSTALL.exempt**. stall_by_md is the concatenation of **MDSTALL.mdh** and **MDSTALL.md**, that is, stall_by_md[30:0] is **MDSTALL.md[31:1]** and stall_by_md[62:31] is **MDSTALLH.mdh[31:0]** if any. When **MDSTALL.exempt** is zero, any non-zero value in stall_by_md[*j*] will cause transactions with RRID=*i* to be stalled for all RRID *i* associated with MD *j*. On the contrary, on **MDSTALL.exempt**=1, checks of all transactions must wait except those with RRID=*i* associated with any MD *j* and stall_by_md[*j*] = 1. This relation can be more precisely described as follows:

```
rrid_stall[i] ⇐ MDSTALL.exempt ^ ( Reduction_OR (SRCMD(i).md & stall_by_md));
```

For any unimplemented memory domain, the corresponding bit in **MDSTALL.md** or **MDSTALLh.mdh** should be wired to 0.

rrid_stall should be captured only when **MDSTALL.exempt** is written, that is, when **MDSTALL** is written. When **MDSTALLH** is written, the only action is to hold the value.

Although rrid_stall is related to SRCMD table, but should be captured only when **MDSTALL.exempt** is written. The behavior of writing **MDSTALL** is used to capture a momentary snapshot of the table because the table may not be stable during the updating.

When writing **MDSTALL**, the specification only defines the transactions with RRID=*i* must wait for all rrid_stall[*i*]=1. It doesn't request the rest of the transactions to stall or not. It only asks that all transactions be resumed when writing **MDSTALL** with a zero.

7.4. Cherry Pick

If **MDSTALL** doesn't stall all the desired transactions, there is an optional method to pick the transaction with specific RRIDs. The **RRIDSCP** register comprises two fields: a 2-bit **RRIDSCP.op** and a field for **RRIDSCP.rrid**. By setting **RRIDSCP.op**=1, the rrid_stall[*i*] is activated for *i* =**RRIDSCP.rrid**. Conversely, by setting **RRIDSCP.op**=2, the rrid_stall[*i*] is deactivated for *i* =**RRIDSCP.rrid**. This register is used to fine-tune the result of writing **MDSTALL**. The value of **RRIDSCP.op**=0 is to query the rrid_stall indirectly, and the value of 3 is reserved.

7.5. Resume Stall

In order to resume all stalled transactions, the IOPMP can be prompted by writing 0 to **MDSTALL**. This corresponds to Step 3 of the "Programming Steps" section. After **MDSTALL** is written by zero, an IOPMP should de-assert **MDSTALL.is_stalled** within some time, at which point all transactions

have been resumed.

7.6. The Order to Stall

In Step 1 of programming IOPMP, **MDSTALL** can be written at most once and before any **RRIDSCP** is written. After a resume, writing a non-zero value to **MDSTALL** multiple times leads to an undefined situation.

RRIDSCP can be written multiple times or not at all. To determine whether all requested stalls take effect, one can read back the bit **MDSTALL.is_stalled**, which is in the same location as **MDSTALL.exempt** on a write. **MDSTALL.is_stalled**=1 indicates all requested stalls taking effect after the last writing **MDSTALL** (included) plus any following writing **RRIDSCP**.

After writing any non-zero value to **MDSTALL**, **MDSTALL.is_stalled** must be asserted within some time, no matter whether any RRID is stalled. The software polling the status bit doesn't need to consider whether any RRID will be stalled. On the other hand, after writing zero to **MDSTALLH** (if any) and then **MDSTALL**, **MDSTALL.is_stalled** must be de-asserted within some time.

Based on the aforementioned, complete steps to program an IOPMP are suggested.

- Step 1.1: write MDSTALL once // exactly once
- Step 1.2: write RRIDSCP zero or more times
- Step 1.3: poll until MDSTALL.is_stalled == 1 // to ensure all stalls takes effect
- Step 2: update IOPMP's configuration
- Step 3.1: write MDSTALL=0 // resume all transactions
- Step 3.2: poll until MDSTALL.is_stalled == 0 // optional, to ensure all resumes take effect.

Some steps may be skipped according to the actual implementation.

To query if all transactions associated with a specific RRID are stalled, do the following. First, write 0 to **RRIDSCP.op** and the RRID you want to query to **RRIDSCP.rrid**. Then, read back **RRIDSCP**. The readback of **RRIDSCP.stat** = 1 means that transactions with the queried RRID have stalled, that is, the corresponding bit in rrid_stall is 1. If the value is 2, it means they are not stalled. A value of 3 indicates an unimplemented or unselectable RRID in **RRIDSCP.rrid**. **RRIDSCP.stat** is in the same location as **RRIDSCP.op** on a write. **RRIDSCP.rrid** should keep the last written legal RRID and **RRIDSCP.stat** reflects the current state of this RRID. This method is considered an indirect way to read rrid_stall.

7.7. Implementation Options

All registers described in this chapter are optional. Moreover, these features could be partially implemented. In **MDSTALL.md** and **MDSTALLH.mdh**, not every bit should be implemented even though the corresponding MD is implemented. An unimplemented bit means unselectable and should be wired to zero. To test which bits are implemented, one can write all 1's to **MDSTALL.md** and **MDSTALLH.mdh** and then read them back. An implemented bit returns 1.

If an IOPMP implementation has fewer than 32 memory domains, **MDSTALLH** should be wired to zero.

An example of partial implementation of **MDSTALL.md**/**MDSTALLH.mdh** is a system with a display controller, which is a latency-sensitive device. On updating the IOPMP, the transactions initiated from the display controller should not be stalled. Thus, one can always use **MDSTALL.exempt**=1 and **MDSTALL.md[*j*]**=1, where MD *j* is the memory domain for the frame buffer that the display controller keeps accessing. Thus, the system only needs to implement **MDSTALL.md[*j*]**.

If whole **MDSTALL** is not implemented, **MDSTALL** and **MDSTALLH** should always return zero.

If **RRIDSCP** is not implemented, it always returns zero. One can test if it is implemented by writing a zero and then reading it back. Any IOPMP implementing **RRIDSCP** should not return a zero in **RRIDSCP.stat** in this case.

It is unnecessary to allow every implemented RRID to be selectable by **RRIDSCP.rrid**. If an unimplemented or unselectable RRID is written into **RRIDSCP.rrid**, it returns **RRIDSCP.stat** = 3.

A1: Multi-Faults Record Extension

A first violation is one that is detected and logged in the error report. However, since the error report can only accommodate one first violation, any additional violations that are detected but not logged in the error report are termed as subsequent violations. The issue at hand is that these subsequent violations become completely invisible. The Multi-Faults Record Extension is used to record which RRIDs make subsequent violations. The extension maintains a bit, referred to as SV[*s*], for each RRID *s*. When one or more subsequent violations are issued from an RRID, the corresponding bit is set. To retrieve these SVs, a 32-bit register **ERR_MFR** is used. Every 16 contiguous SVs are grouped together into a record window, which is indexed by a 12-bit field, **svi**. When **ERR_MFR** is read, the **svi** sequentially scans all windows from its original position until a violation is found. Once **svi** is overflowed, it rounds to zero. If found, the status bit **svs** is set, and **svi** stops in the window containing the first found set SV. The 16-bit field **svw** reflects the record window indexed by **svi**, where **svw[*j*]**=SV[**svi** * 32 + *j*]. After the register is read out, all bits in the record window are cleared. If not found, **svs** and **svw** return zeros and **svi** keeps the same. Moreover, the bit **svc** in the **ERR_REQINFO** indicates if any subsequent violation is in the log.

Chapter 8. A2: Run Out Memory Domains

In this specification, the support is capped at 63 memory domains. However, this chapter provides pertinent recommendations for situations that necessitate a larger number of memory domains.

8.1. A2.1 Parallel IOPMP

Multiple IOPMPs can be placed in parallel. A transaction should be directed to one of these IOPMPs for its check. The chosen IOPMP then determines its legality. There are two potential methods for routing the transaction: by address or by RRID. Address-based routing divides the address space into multiple disjoint sets, and a transaction is directed to the IOPMP based on its starting address. Similarly, RRID-based routing divides all possible SIDs, and a transaction is directed to the IOPMP based on its RRID.

Placing IOPMPs in parallel can seamlessly enhance the support for an increased number of memory domains since all the IOPMPs are located in the same position. This arrangement may also concurrently increase the checking throughput.

8.2. A2.2 Cascading IOPMP

Cascading multiple IOPMPs allows a transaction to traverse through more than one IOPMP. Each time a transaction goes through an IOPMP, it is tagged a new RRID until it reaches the final IOPMP. This new RRID represents that the transaction has been checked by a specific IOPMP. Subsequent IOPMPs could deem the transaction trustworthy and forward it to their initiator port without further checks, or check it in a higher level view, e.g., a subsystem view. An IOPMP with the above feature of tagging a new RRID is referred to as an IOPMP gateway. Its **HWCFG0.rrid_transl_en** should be set to 1, and **HWCFG2.rrid_transl** is used to store the RRID. **HWCFG0.rrid_transl_prog** indicates whether **HWCFG2.rrid_transl** is programmable or not. To lock **rrid_transl**, write 1 to **rrid_transl_prog**, which clears **rrid_transl_prog** and is sticky to 0.

The integration of several independently developed smaller Systems on a Chip (SoCs) to construct a larger SoC reduces the chip count in a device. This approach also decreases costs by enabling the use of larger and shared memory devices. In such a system, each subsystem upholds its governance through its own secure software, RRID assignment, and security configuration. The cascading approach facilitates this: the secure software manages the IOPMP in the boundary of the subsystem. The boundary IOPMP assigns a new RRID to each outgoing transaction, representing that it has been checked by the IOPMP. The outer IOPMPs are tasked with controlling the transactions from a subsystem perspective by the new subsystem-level RRID. That is, the IOPMP only considers the legality of the transactions initiated from a specific subsystem instead of individual transaction initiators. The boundary IOPMP hides some details of the subsystem good for protecting intellect properties. The development flow becomes more abstract, reusable, and modularized.

A3: Secondary Permission Setting

IOPMP/SPS (Secondary Permission Setting) is an extension to support different sources to share memory domain while allowing each sources to have different R/W/X permission to a single memory domain.

If IOPMP/SPS extension is implemented, each SRCMD table entry shall additionally define read and write permission registers: **SRCMD_R(*s*)** and **SRCMD_W(*s*)**, and **SRCMD_RH(*s*)** and **SRCMD_WH(*s*)** if applicable. Register **SRCMD_R(*s*)** and **SRCMD_W(*s*)** each has a single fields, **SRCMD_R(*s*).md** and **SRCMD_W(*s*).md** respectively representing the read and write permission for each memory domain for source *s*. Setting lock to **SRCMD_EN(*s*).l** also locks **SRCMD_R(*s*)**, **SRCMD_RH(*s*)**, **SRCMD_W(*s*)**, and **SRCMD_WH(*s*)**.

IOPMP/SPS has two sets of permission settings: one from IOPMP entry and the other from **SRCMD_R**/**SRCMD_W**. IOPMP/SPS shall check read and write permission on both the SRCMD table and entries, a transaction fail the IOPMP/SPS check if it violates either of the permission settings.

The IOPMP/SPS register for setting instruction fetch permission on each memory domain is [TBD].

Bibliography