# RISC-V IOPMP Specification Document

Paul Ku, Channing Tang, RISC-V IOPMP Task Group

Version 1.0.0-draft2, 05/2023: 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:


- Paul Shan-Chyun Ku <scku@andestech.com>
- Channing Tang <channingt@nvidia.com>

# 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 IO peripherals like the DMA (Direct Memory Access Unit) to systems improves performance but exposes the system to vulnerabilities such as DMA attacks. In the RISCV eco-system, there already exists the core-PMP and MPU which provide standard protection scheme for accesses from RISCV core to the physical address space, but there is not a likewise standard for safeguarding non-core initiators. Here we propose the Physical Memory Protection Unit of Input/Output Devices, IOPMP for short, to regulate 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 software solution on a RISC-V-based platform, it's generally the security monitor, a program running on the M-mode takes care of security-related requests. However, it's impratical for the security monitor to check the legality of each request as the overhead and latency of trap-check-ret is non-negligible. And the latency becomes even worse when there are multipe inflight transactions in the system, for example, an DMA and a Crypto Accelerator with different privileges are making requests at the same time. A hardware component that can check accesses on the fly becomes a reasonable solution. That is the subject of this document, IOPMP.

[iopmp system position] | *iopmp_system_position.png*

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

# Chapter 2. Concepts

This document uses the term security monitor to refer to the software in charge of security-related tasks. The security monitor also programs the IOPMPs. The security monitor is not confided to run on a core or a hart. It could be distributed on more than one core, virtualized on a virtual platform, or cascaded within a nested sub-platform.

For a register or a field X, X[n] represents the bit-n of X and X[n:m] for the bit-n to bit-m of X.

## 2.1. Source-ID and Transaction

Source-ID, SID for short, is a unique ID to identify 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 SID indicating which bus initiator issued it. We will discuss about the exception in the next section. Associating bus initiators with SID could be implementation-dependent and is out of the scope of this document, but the IOPMP does have requirement on the uniqueness of SIDs which will be discussed when we talking about the cascading IOPMPs. The number of bits of an SID is implementation-dependent. However, there are some suggestions in this document if SID is programmable. SID could be a vulnerability because a malicious program can gain extra permission by forging SID. Hardwired SID can avoid such a risk. However, in the cases of requiring more flexibility, locking SID before entering REE can be a good choice. If a system provides programmable SIDs during the runtime, the write permission of SIDs should be controlled very carefully.

If a bus initiator has multiple channels and every channel is granted different access permissions, every channel should have its own SID. If a bus initiator runs in different privilege modes, such as a processor, every privilege mode should have a different SID if the system is designed to use IOPMPs to regulate its access. The usage of multiple SIDs is also applied to multiple virtual machines with different permissions.

## 2.2. Source-Enforcement

If the scope of an IOPMP contains only one bus initiator or a set of bus initiators with the same permission, the Source-ID can be ignored in bus initiator side as well as the transactions going through the IOPMP. In the case, we denote the IOPMP performs source enforcement, IOPMP-SE for short. In the rest of the cases, we still need SID to distinct the transaction issuers.

## 2.3. Initiator Port and Target Port

An IOPMP has at least a initiator port and at least a target port. A target 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.

## 2.4. Memory Domain

A memory domain, MD for short, is a concept inside an IOPMP. It is used to group a set of memory regions for a specific purpose and is 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 associate with this MD. However, associating the memory domain doesn't mean having full permissions on all memory regions. The permission of each region is defined in the corresponding IOPMP entry. However, there is an extension to adhere the permission to the MD that will be introduced in the appendix.

One thing should be noted: one SID may associate with more than one MD, and one MD may be associated with more than one SID. However, some models may limit the flexibility due to different requirements.

## 2.5. IOPMP Entry and IOPMP Entry Array

IOPMP entry array is the most fundamental structure of an IOPMP and is a list of ordered IOPMP entries. An IOPMP entry is indexed from zero and defines a rule when checking a transaction: including a memory region and the read/write permission. Each IOPMP entry belongs to exactly one memory domain, and a memory domain may have multiple IOPMP entries. An SID associating with an MD also means it associates with all IOPMP entries belonging to the MD.

IOPMP entries are statically prioritized. The lowest-numbered IOPMP entry that (1) matches any byte of the in-coming transaction and (2) is associated with the SID carried by the transaction determines whether the transaction is legal. The matching IOPMP entry must match all bytes of a transaction, or the transaction is illegal, irrespective of its permission.

As to an IOPMP-SE, the only structure of it is the IOPMP entry array. Due to no SID, when selecting the matching IOPMP entry, an IOPMP-SE ignores the SID comparison.

[iopmp unit block diagram] | *iopmp_unit_block_diagram.png*

*Figure 2. IOPMP Block Diagram.*

# Chapter 3. IOPMP Models

To fit in various needs of the different platforms, we provide several IOPMP configuration models. We will begin with the full model which comes with practically all of the features listed in this document. Next, we'll discuss optional features and moving to the other models. These models assist users in choosing and refining their designs to meet various needs, including those for low area, low power, low latency, high throughput, high portability, and other criteria.

## 3.1. The Full Model

When a full model IOPMP receives a transaction with SID *s*, it first lookups the SRCMD table to find out all the memory domains associated to *s*. The size of the table is implementation-dependent. In the table, the register $SRC_sMD$ is defined per SID *s*, occupies a 64-bit space, and has two fields, $SRC_sMD.L$ and $SRC_sMD.MD$. $SRC_sMD.L$ is a sticky lock to this register. In the model, $SRC_sMD.MD$ is a bitmapped field and has up to 63 bits. Each bit here indicates if a MD is associated with the SID *s*. Not all bits should be implemented, but implemented bits should be right justified. For unimplemented memory domains, the corresponding bits in $SRC_sMD.MD$ should be WARZ. A full model IOPMP supports up to 63 memory domains. For a system requiring more memory domains than 63, one could cascade IOPMPs. Cascading IOPMPs is described in the Appendix.

Once an IOPMP retrieves all associated MDs for a transaction with SID *s*, it looks up the corresponding IOPMP entries belonging to these MDs. The MDCFG table defines the relation between MDs and IOPMP entries. The MDCFG table has an array of registers where the register $MD_mCFG$ is for the memory domain *m*. One field, $MD_mCFG.T$, indicates the top index of the IOPMP entry belonging to the memory domain *m*. An IOPMP entry with index *j* belongs to MD *m* if $MD_{m-1}CFG.T_j < MD_mCFG.T$, where $m>0$. The MD *0* owns the IOPMP entries with index $j<MD_0CFG.T$. Each $MD_mCFG$ register occupies a 32-bit space and the field $MD_mCFG.T$ occupies the lowest 16 bits. The number of implemented bits is implement-dependent.

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

Appendix provides a reference implementation for a full model IOPMP.

## 3.2. Configuration Protection

A hardwire behavior that makes an IOPMP fully or partially nonprogrammable unless resetting the IOPMP is the so-called "lock." A lock in an IOPMP is similar to that in a PMP. It can ensure critical settings are unchanged even though the security monitor is compromised. To lock the programmability of an IOPMP completely, we may not really need a new mechanism. If you have a platform similar to the above example, you could create a memory domain to stop any future IOPMP control operations. We will go through the idea first. If you want a fine-grained method that reserves partial programmability, the subsequent optional features are designed for it.

### 3.2.1. Protect the SRCMD Table

In order to enforce that all SIDs associate MD 0, you should lock the whole the SRCMD table due to the granularity. You are not able to lock all $SRC_sMD[0]=1$ (for all s) but leave the rest of the bits programmable. Thus, you lose all programmability of the mapping from SID to MD.

Here, there are two optional mechanisms to lock the SRCMD table partially. The register MDMSK can lock certain bits for each $SRC_s$MD.MD. MDMSK has two fields: a 63-bit MDMSK.MD and a 1-bit MDMSK.L. On MDMSK.MD[*m*]=1, all $SRC_s$MD.MD[*m*] are not programmable for all SID s. MDMSK.L is the lock bit for the MDMSK. In above example, when you want to enforce every SID to associate MD 0, you can first set all $SRC_s$MD.MD[0]=1, and then let MDMSK.MD[0]=1 and MDMSK.L=1. The rest mappings are still programmable. If MDMSK is not implemented, it should be WARZ.

For unimplemented memory domains, the corresponding bits of MDMSK.MD should be WARZ. The bits for implemented memory domains in MDMSK.MD can be also hardwired. However, in this case, the corresponding bits in $SRC_s$MD.MD should be well initialized during reset process. If whole MDMSK.MD is hardwired, MDMSK.L should be wired to 1. To figure out which memory domains are implemented, you can do the following steps: (1) set all ones (0x7fffffff_ffffffff) to $SRC_0$MD.MD, (2) read back the field, and (3) OR it with MDMSK.MD. The corresponding bits for implemented memory domains in the result will be 1's.

Besides, every $SRC_s$MD register has an optional bit, L, which is used to lock this register. It is a convenient way to lock the MD mapping of an SID without consuming any IOPMP entry. If a programmable $SRC_s$MD.L is implemented, $SRC_s$MD.L should be initialized to zero after reset. If $SRC_s$MD.L is not implemented, it can be hardwired to 0 or 1. If it is wired to 1, $SRC_s$MD.MD should be hardwired properly.

### 3.2.2. Protect the MDCFG Table

Subsequently, register MDCFGLCK is designed for locking the MDCFG table, it has two fields: MDCFGLCK.L and MDCFGLCK.F. Please note that if the top index of MD *m* is locked, that of MD *m-1* should be locked, too. Otherwise, the IOPMP entries of MD m can be added or removed by modifying $MD_{m-1}$CFG.T. By introduction, if MD m is locked, MD n should also be locked, where $n < m$. Thus, we define all $MD_m$CFG.T are nonprogrammable where $m <$ MDCFGLCK.F. MDCFGLCK.F is initialized to 0 after reset, and can be increased only when written. MDCFGLCK.L is the lock of MDCFGLCK. If MDCFGLCK is hardwired, MDCFGLCK.L should be wired to 1.

### 3.2.3. Protect the IOPMP Array

Lastly, a register ARRLCK is defined to support setting the lock to the IOPMP entry array. The ARRLCK register has two fields: ARRLCK.L and ARRLCK.F, where ARRLCK.L is the lock to ARRLCK register and ARRLCK.F is to speicify the number of top entries that is locked. Similar to the MDCFGLCK register, programming the ARRLCK register shall follow a monotinically incremental style, where ARRLCK.F can only be written with value greater than current.

## 3.3. The Rapid-*k* Model

To shorten the latency, the rapid-*k* model replaces the lookup of the MDCFG table by simple logics. Every memory domain has exactly *k* IOPMP entries where *k* is implementation-dependent and hardwired. Since *k* is a fixed number, once MDs are retrieved for a transaction, these indexes of selected MDs can quickly transform into the signals to pick up the IOPMP entries. An extreme case is $k$=1 in which every non-zero bit in $SRC_s$MD.MD directly maps to a selected IOPMP entry for SID=*s*.

To make it semantically compatible with the full model, the realated fields should be read with their original meanings. MDCFGLCK.F should be the same as the number of implemented MDs and MDCFGLCK.L should be 1. $MD_m$CFG.T should be $(m+1)*k$. That is, one can read $MD_0$CFG.T to retrieve

the value *k*.

$MD_m$CFG.F and $MD_m$CFG.L can still be programmable or hardwired. The two fields typically do not affect the latency of checking a transaction. They are usually related to writing to IOPMP registers, and writing latency is not a concern in this model.

## 3.4. The Dynamic-*k* Model

The dynamic-*k* model is similar to the rapid-*k* model, except the *k* value is programmable. If you have a fixed number of IOPMP entries, you probably don't need this model. You can simply divide all IOPMP entries evenly to every memory domain and obtain a fixed k. However, if the IOPMP array is not in dedicated storage and could be shared for other purposes, the dynamic-*k* model helps partition these IOPMP entries.

The IOPMP entry reassignment is not suggested during the run time. The boot time is a better choice.

$MD_0$CFG.T stores the value *k* and is WARL. That is, an implementation may accept limited *k*. However, zero should not be a legal value. One should make sure if a written value is legal by reading it back. The *k* is usually considered as a power of 2 for easier hardware implementation. $MD_m$CFG.T is read-only and equals to *(m+1)*k* when it is read. By updating $MD_0$CFG.T and then examining $MD_1$CFG.T's change, one can know the IOPMP is the dynamic-*k* model.

MDCFGLCK.F should be zero right after the IOPMP resets. MDCFGLCK.F and MDCFGLCK.L can be programmable or hardwired. If MDCFGLCK.F is programmable, it can only accept two values: 0 and the number of MDs.

## 3.5. The Isolation Model

One of the benefits of the full model is to share common memory regions (by memory domains) among different SIDs. The isolation model can be implemented for the case of no or a few shared regions. In this model, each SID is exactly associated with one MD. Thus, no table-lookup is needed to retrieve the associated MD. It benefits the area as well as the latency. The penalty is to duplicate IOPMP entries when two SIDs do share regions. Besides, even though MDMSK and all $SRC_s$MD should be read-only, to ensure the semantic compatibility to the full model, we have the following rules. For reading $SRC_s$MD, $SRC_s$MD.MD should be 1<<s, and $SRC_s$MD.L should be 1. As to MDMSK.MD, all implemented MDs should be hardwired to 1. MDMSK.L should also be wired to 1. There is no constrain on the MDCFG table and the MDCFGLCK register.

## 3.6. The Compact-*k* Model

The compact-*k* model can achieve even lower latency and smaller area than the isolation model. Besides having each SID exactly associated with one MD, every MD should have exactly k IOPMP entries. Once SID is known, the IOPMP entries can be selected efficiently. In the model, MDMSK, all $SRC_s$MD, MDCLK, and all $MD_m$CFG.T are read-only. When read, MDMSK and all $SRC_s$MD should be the same as the isolation model. MDCFGLCK and all $MD_m$CFG.T should be the same as the rapid-k model. $MD_m$CFG.L and $MD_m$CFG.F can still be programable or hardwired.

# 3.7. Model Detections

To distinguish the above models, one can follow the below approach.

First, we figure out how many MDs are implemented by (1) writing all ones to $SRC_0$MD.MD and (2) OR-ing the values and MDMSK.MD. Denote the result as *IMD*. The ones in *IMD* mean the implemented MDs.

Then, we test if the SRCMD table is programmable by reading MDMSK. Suppose MDMSK.L=1 and MDMSK.MD = *IMD*, the SRCMD table is read-only, and the IOPMP is either the isolation or the compact-k models. Subsequently, if $MD_0$CFG.T can accept zero, that is, writing zero and reading back a zero, the MDCFG table is programmable, and the IOPMP is the isolation model. Otherwise, it is the compact-k model because you can never have the compact-0 model.

If the SRCMD table is programmable, the IOPMP should be the rapid-*k* model, the dynamic-*k* model, or the full model. If MDCFGLCK.L is 1 and MDCFGLCK.F is non-zero, it is the rapid-*k* model. Then, if $MD_0$CFG.T accepts zero, it is the full model; otherwise, the dynamic-*k* model.

# Chapter 4. Tables Reduction and Detection

The full model has two tables and one array. It provides good flexibility to configure an IOPMP but sacrifices the latency and the area. In this section, we introduce the methods to reduce these tables in order to reach different design requirements. Some of the bits can be hardwired. However, for the sake of software detection and portability, the values read from these hardwired bits should maintain the same sematic as that of the full model.

The latency consideration here is about checking a transaction instead of accessing the IOPMP control registers because programming IOPMPs is considered less frequent. That is, we will not address the latency of updating the tables or the IOPMP entries.

As to the IOPMP array, it is the body of storing the IOPMP entries, so it cannot be omitted. We can only reduce its size. Memory domains can be shared among different SIDs, so the entries belonging to these shared MDs are shared among SIDs. Sharing entries may help to reduce the size of the IOPMP array.

The SRCMD table can be hardwired fully or partially to save area. For some cases, we can farther save the latency. Every $SRC_s$MD.MD should have the same programmable bits, so one can just detect $SRC_t$MD.MD if $SRC_t$MD.L is not wired to 1. If all $SRC_s$MD.L are wired to 1, there is no reason to implement MDMSK. If all bits in $SRC_s$MD.MD are hardwired, $SRC_s$MD.L should be wired to 1, too.

The SRCMD table can be replaced by simple circuits in order to save area or latency. A special reduction makes $SRC_s$MD.MD=(1 << $s$) for all $s$. It replaces a table look by a binary decoder, which shortens the latency to retrieve the corresponding MD. In the case, there is no any shared MD and it supports up to 63 SIDs.

As to the MDCFG table, the $MD_m$CFG.T can also be hardwired to save area and/or shorten the latency in some cases. It means that the ownership of every IOPMP entry is fixed all the time. Hardwiring $MD_m$CFG.F to a non-zero value is not a usual case because it makes some highest priority IOPMP entries nonprogrammable by software at any time. If $MD_m$CFG.F is hardwired, $MD_m$CFG.L should be wired to 1.

A special reduction makes $MD_m$CFG.T=$(m+1)*k$ for all m. It replaces a table lookup by a simple circuit, e.g., $k$ is a power of 2. It can save some area and shorten the latency as well.

Every $MD_m$CFG should have the same programmable bits in an IOPMP except the dynamic-$k$ model. We will describe it in the next section.

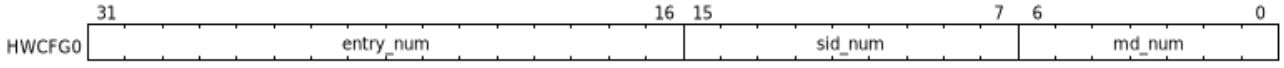
IOPMP-SE is a special case since it only needs the IOPMP entry array. The two tables are not needed, not even hardwired, and do not occupy any address space.

# Chapter 5. Registers

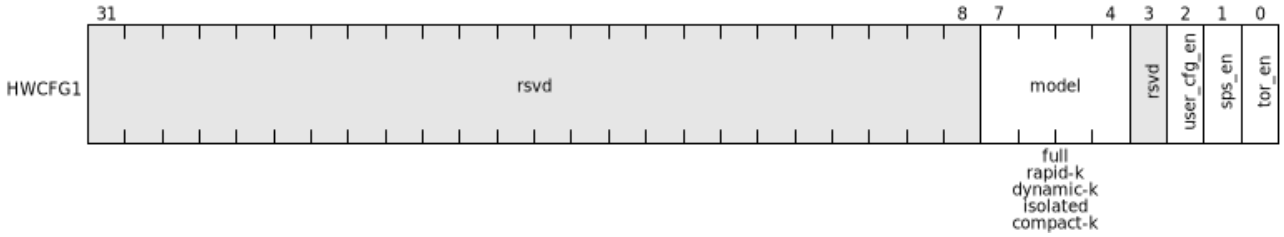
| OFFSET | Register | Description |
|---|---|---|
| 0x0000 | INFO | |
| | HWCFG | Indicates the configurations of current IOPMP instance |
| | RULE_OFFSET | Indicates the internal address offsets of each table. |
| | Programming Protection | |
| | MD_STALL | |
| | SID_SCP | |
| | Error Reporting | |
| | ERR_REQADDR | |
| | ERR_REQID | |
| | ERR_REQINFO | |
| | ERR_IRQSTAT | |
| | ERR_IRQMASK | |
| | Configuration Protection | |
| | MDMSK | Lock Register for SRCMD table. |
| | MDCFGLCK | Lock register for MDCFG table |
| | ARRLCK | Lock register for IOPMP entry array. |
| 0x0800 | MDCFG Table, m = 0...md_num -1 | |
| | MDCFG(m) | MD config register, which is to specify the indices of IOPMP entries belonging to a MD. |
| 0x1000 | SRCMD Table, s = 0...sid_num-1 | |
| | SRCMD_EN (s) | Bitmapped MD enable register, 's' corresponding to number of sources, it indicate source s associated MDs. |
| | SRCMD_R(s) | (Optional)Bitmapped MD read eanble register, 's' corresponding to number of sources, it indicate source s read permission on MDs. |
| | SRCMD_W (s) | (Optional)Bitmapped MD write eanble register, 's' corresponding to number of sources, it indicate source s write permission on MDs. |
| RULE_OFFSET | Entry Array, i = 0...entry_num-1 | |
| | ENTRY_ADDR(i) | |
| | ENTRY_ADDRH(i) | (Optional for 32-bit system) |
| | ENTRY_CFG(i) | |
| | ENTRY_USER_CFG(i) | (Optional) extension to support user customized attributes |

**INFO registers**

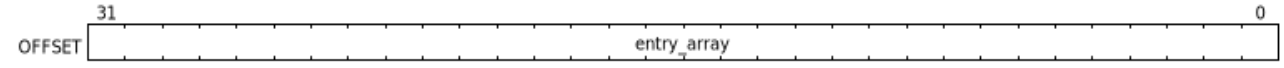
The HWCFG are RO registers, it indicates the IOPMP instance configuration info.




| HWCFG0 | | | |
|---|---|---|---|
| **Field** | Bits | R/W | Description |
| md_num | 6:0 | R | Indicate the supported number of MD in the instance |
| sid_num | 15:7 | R | Indicate the supported number of SID in the instance |
| entry_num | 31:16 | R | Indicate the supported number of entries in the instance |

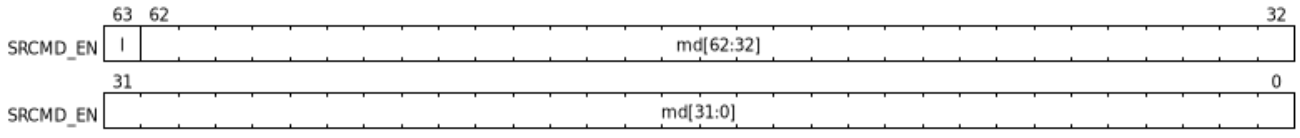



| HWCFG1 | | | |
|---|---|---|---|
| **Field** | Bits | R/W | Description |
| tor_en | 0:0 | R | Indicate if TOR is supported |
| sps_en | 1:1 | R | Indicate the secondary permission settings is supported |
| user_cfg_en | 2:2 | R | Indicate the if user customized attributes is supported |
| model | 7:4 | R | Indicate the iopmp instance model |




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

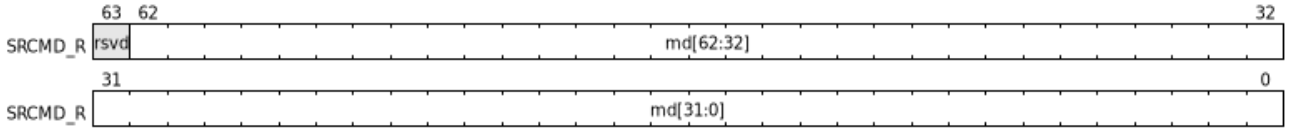
- SRCMD Table Registers



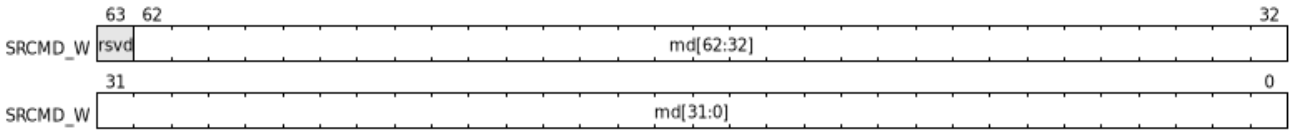

| SRCMD_EN(s), s = 0...sid_num-1 | | | |
|---|---|---|---|
| **Field** | Bits | R/W | Description |
| md | 62:0 | WARL | A bitmap to indicate if the corresponding md is matched with SID i. |

| | | | |
|---|---|---|---|
| l | 63:63 | W1 | A sticky lock bit. When set, locks SRCMD_EN(i), SRCMD_R(i) and SRCMD_W(i) |

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




| SRCMD_R(s), s = 0...sid_num-1 | | | |
|---|---|---|---|
| **Field** | Bits | R/W | Description |
| md | 62:0 | WARL | A bitmap to indicate if SID i has read permission to the corresponding MD. |




| SRCMD_W(s), s = 0...sid_num-1 | | | |
|---|---|---|---|
| **Field** | Bits | R/W | Description |
| md | 62:0 | WARL | A bitmap to indicate if SID i has write permission to the corresponding MD. |

- **MDCFG Table**

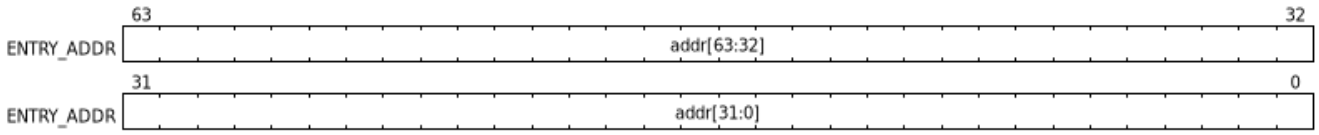
The MDCFG table is a lookup to specify the number of IOPMP entries that is associated with each MD. For different models:

1. Full model: the number of MDCFG registers is equal to md_num, all MDCFG registers are readable and writable.
2. Rapid-k model: a single MDCFG register to indicate the k value, read only.
3. Dyanmic-k model: a single MDCFG register to indicate the k value, readable and writable.
4. isolation model: the number of MDCFG registers is equal to md_num, all MDCFG registers are readable and writable.
5. Compact-k model: a single MDCFG register to indicate the k value, read only.

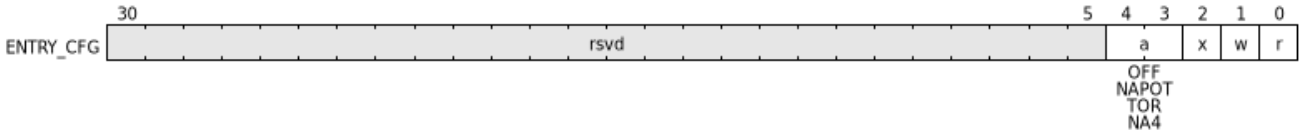

| MDCFG(m), m = 0...md_num-1, support up to 63 MDs | | | |
|---|---|---|---|
| **Field** | Bits | R/W | Description |

| | | | |
|---|---|---|---|
| t | 16 | WARL | • Indicate the top range of memory domain m. An IOPMP entry with index j belongs to MD m<br>• If MDm-1CFG.t ≤ j < MDmCFG.t, where m>0. The MD0 owns the IOPMP entries with index j<MD0CFG.t.<br>• If MDm-1CFG.t >= MDmCFG.t, then MDm is empty.<br>• For rapid-k, dynamic-k and compact-k models, t indicates the number of IOPMP entries belongs to each MD. |

- Entry Array Registers

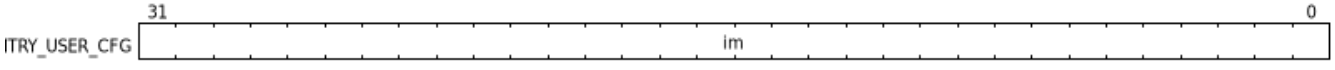



| ENTRY_ADDR(i), i = 0...entry_num-1 | | | |
|---|---|---|---|
| **Field** | Bits | R/W | Description |
| addr | 31(63):0 | WARL | The physical address of protected memory region. |



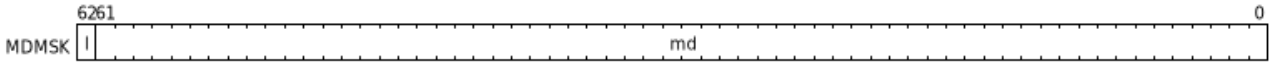

| ENTRY_CFG(i), i = 0...entry_num-1 | | | |
|---|---|---|---|
| **Field** | Bits | R/W | Description |
| r | 0:0 | RW | The read permission to protected memory region |
| w | 1:1 | WARL | The write permission to the protected memory region |
| x | 2:2 | WARL | The executable permission to the protected memory region. Optional field, if undefined, write any read the same value as r field. |
| a | 4:3 | WARL | The address mode of the IOPMP entry |

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




- Configuration Protection Registers

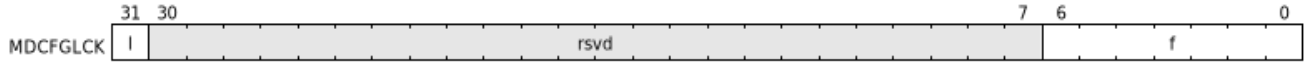
**MDMSK** is a register with a bitmap field to indicate which MDs are locked.



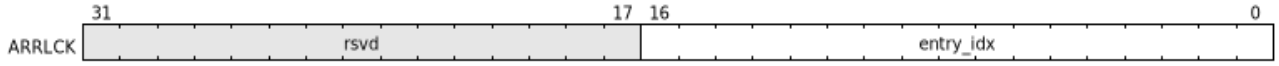

| MDMSK |
|---|

| Field | Bits | R/W | Description |
|---|---|---|---|
| md | 62:0 | WARL | A bitmap to indicate which MDs are locked. |
| md | 63:63 | W1 | Lock bit to MDMSK register. |

**MDCFGLCK** is the lock register to MDCFG table.



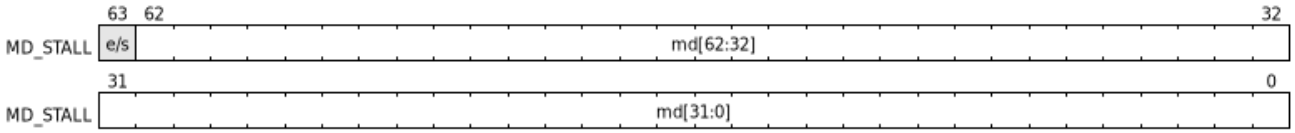

**ARRLCK** is the lock register to entry array.




| ARRLCK | | | |
|---|---|---|---|
| **Field** | Bits | R/W | Description |
| num | 16 | WARL | Indicate the number of locked IOPMP entries – IOPMP_ENTRY[num-1:0] is locked. SW shall write a value that is no smaller than current num. |

- **Programming Protection Registers**

The MD_STALL and SID_SCP registers are implemented to support atomicity issue while programming the IOPMP, as the IOPMP rule may not be updated in a single transaction.



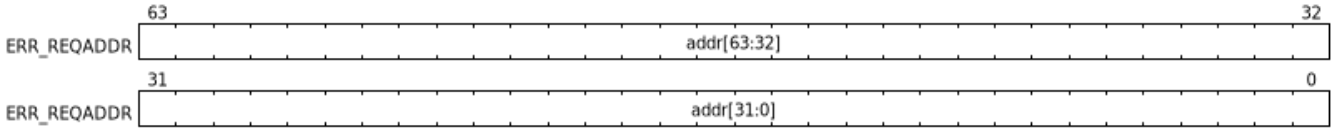

| MD_STALL | | | |
|---|---|---|---|
| **Field** | Bits | R/W | Description |
| EXEMPT | 63:63 | W | Stall transactions with exempt selected MDs, or Stall selected MDs |
| IS_STALLED | 63:63 | R | Indicate if the requested stalls have occured |
| MD | 62:0 | W | a MD[*i*]=1 means selecting MD *i* |
| MD | 62:0 | R | a MD[*i*]=1 indicates MD *i* implemented and selectable in this field |


| SID_SCP | | | |
|---|---|---|---|
| **Field** | Bits | R/W | Description |
| OP | 31:30 | W | 0: query, 1: stall transactions associated with selected SID, 2: don't stall transactions associated with selected SID, and 3: reserved |

| | | | |
|---|---|---|---|
| STAT | 31:30 | R | 0: SID_SCP not implemented, 1: transactions associated with selected SID are stalled, 2: transactions associated with selected SID not are stalled, and 3: unimplemented or unselectable SID |
| SID | 15:0 | WARL | SID to select |

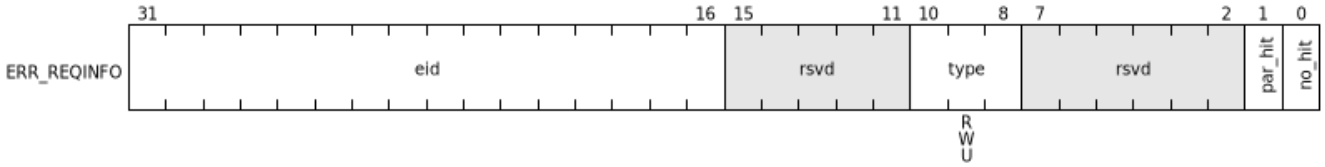
- **Error Capture Registers**

**ERR_REQADDR** indicate the errored request address.




**ERR_REQID** Indicate the errored SID.


**ERR_REQINFO** Captures more detailed error infomation.




| ERR_REQINFO | | | |
|---|---|---|---|
| **Field** | Bits | R/W | Description |
| no_hit | 0:0 | R | Indicate the request hit no entry. |
| par_hit | 1:1 | R | Indicate the request failed due to partial hit. |
| type | 10:8 | R | • Indicated if it's a read, write or user field violation.<br>• 0x0 = read error<br>• 0x1 = write error<br>• 0x3 = user_attr error |
| eid | 31:16 | R | Indicated the errored entry index. |

# Chapter 6. Reset

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 SID 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 SID carried by the transaction. To simplify the following description, we use a conceptual signal called sid_stall[*i*] to indicate whether the transaction with SID=*i* must wait. Please note that it may not be an actual signal in practice and is not accessible directly for software.

MD_STALL is an optional register used to generate the conceptual signal sid_stall. MD_STALL has two fields, a bit MD_STALL.EXEMPT and bit array MD_STALL.MD[62:0]. When MD_STALL.EXEMPT is set to zero, any non-zero value in MD_STALL.MD[*j*] will cause transactions with SID=*i* to be stalled for all the SID *i* associated with the MD *j*. On the contrary, on MD_STALL.EXEMPT=1, checks of all transactions must wait except those with SID=*i* associated with any MD *j* and MD_STALL.MD[*j*] = 1. This relation can be more precisely described as follows:

```
sid_stall[i] ⇐ MD_STALL.EXEMPT ^ ( | (SRCiMD & MD_STALL.MD));
```

sid_stall should be captured only when MD_STALL is written.

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

*When it comes to specifying transactions to wait, we use memory domains instead of SIDs. The reason for this is that there is a limit of 63 memory domains, making it easy to request stalls with a single write in most cases. Furthermore, the mapping from SID to MD changes more frequently than the rules and MD's configuration. If one needs to move an MD from one SID to another, it's crucial to stall any ongoing transactions from the old SID. With MD_STALL, this can easily be accomplished with just one register write.*

## 7.4. Cherry Pick

If MD_STALL doesn't stall all the desired transactions, there is an optional method to pick the transaction with specific SIDs. The SID_SCP register comprises two fields: a 2-bit SID_SCP.OP and a field for SID_SCP.SID. By setting SID_SCP.OP=1, the sid_stall[*i*] is activated for *i*=SID_SCP.SID. Conversely, by setting SID_SCP.OP=2, the sid_stall[*i*] is deactivated for *i*=SID_SCP.SID. This register can be considered as the fine-tuning sid_stall after MD_STALL. The value of SID_SCP.OP=0 is to query the sid_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 MD_STALL. This corresponds to Step 3 of the "Programming Steps" section.

## 7.6. The Order to Stall

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

SID_SCP can be written multiple times or not at all. To determine whether all requested stalls take effect, one can read back the bit MD_STALL.IS_STALLED, which is in the same location as MD_STALL.EXEMPT on a write. MD_STALL.IS_STALLED=1 indicates all requested stalls taking effect.

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

## 7.7. Implementation-Dependecy

Both registers described in this chapter are optional. Moreover, these features could be partially implemented. In MD_STALL.MD, not every bit should be implemented. The unimplemented bits mean unselectable and are wired zero. To test which bits are implemented, one can read MD_STALL back and an implemented bit in MD_STALL.MD returns 1.

*An example of partial implementation of MD_STALL.MD 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 MD_STALL.EXEMPT=1 and MD_STALL.MD=j, where j is the memory domain for the frame buffer of the display controller. Thus, the system only needs to implement MD_STALL.MD[j].*

If the whole MD_STALL is not implemented, it returns zero, meaning no bit implemented in MD_STALL.MD.

If SID_SCP 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 SID_SCP should not return a zero in SID_SCP.STAT in this case.

It is unnecessary to allow every implemented SID to be selectable by SID_SCP.SID. If an unimplemented or unselectable SID is written into SID_SCP.SID, it will return SID_SCP.STAT = 3 to respond to any defined SID_SCP.OP.

# A1: Reference Data Path

TBD

# A2: Cascading IOPMPs

TBD

# A3: Permission on Memory Domains

In the models mentioned so far, an IOPMP entry is a pair of a memory region and its corresponding permission setting. The bits used to store the memory region are much more than its permission setting. If different memory domains want to share these regions but not their permission settings, the IOPMP/PMD described in this appendix can help.

IOPMP/PMD extends every association bit in the SRCMD table to 4 bits of a second permission setting. We denote SPS[*s*, *m*] as the second permission setting for SID=*s* and MID=*m*. When a transaction arrives at an IOPMP/PMD, it looks up the corresponding memory domains as usual. Then, it also follows the original way to find the IOPMP entry matching the transaction. When checking the permission, IOPMP/PMD has two sets of permission settings: one from IOPMP entry and the other from the second permission setting that is retrieved from SPS. For either read or write operation, only if both permission settings allow, the transaction can do such operation.

Besides, SPS can offer the control of execution permission. If the signal indicating an instruction fetch is carried by a transaction, the second permission setting can control instruction fetches.

As the programmability of every second permission setting, it is the same as the programmability of the bit in the same location in the SRCMD table.

SPS also supports up to 63 memory domains. The LSB of SPS[*s*, 63] is reserved for locking all second permission settings for SID = *s*.

# Index

# Bibliography