**IOPMP Proposal**

Contend

[1 Introduction 2](#_heading=h.gjdgxs)

[2 Background and Terminology 2](#_heading=h.30j0zll)

[3 Source-ID and Transaction 2](#_heading=h.1fob9te)

[4 Source-Enforcement 3](#_heading=h.3znysh7)

[5 Master Port and Slave Port 3](#_heading=h.2et92p0)

[6 Memory Domain 3](#_heading=h.tyjcwt)

[7 IOPMP Entry and IOPMP Entry Array 4](#_heading=h.3dy6vkm)

[8 Models 4](#_heading=h.1t3h5sf)

[9 The Full Model 5](#_heading=h.4d34og8)

[10 Configuration Protection 5](#_heading=h.2s8eyo1)

[10.1 Configuration Protection by a Memory Domain 6](#_heading=h.17dp8vu)

[10.2 The Other Protection Mechanism 6](#_heading=h.3rdcrjn)

[11 Tables Reduction and Detection 7](#_heading=h.lnxbz9)

[12 The Other Models 8](#_heading=h.35nkun2)

[12.1 The Rapid-](#_heading=h.1ksv4uv)*k* Model 9

[12.2 The Dynamic-](#_heading=h.44sinio)*k* Model 9

[12.3 The Isolation Model 10](#_heading=h.2jxsxqh)

[12.4 The Compact-](#_heading=h.z337ya)*k* Model 10

[12.5 Model Detections 10](#_heading=h.3j2qqm3)

[13 Registers 11](#_heading=h.1y810tw)

[14 Reset 11](#_heading=h.4i7ojhp)

[15 Program IOPMPs 11](#_heading=h.2xcytpi)

[Appendix A1: Reference Data Path 11](#_heading=h.1ci93xb)

[Appendix A2: Cascading IOPMPs 11](#_heading=h.3whwml4)

[Appendix A3: Permission on Memory Domains 11](#_heading=h.2bn6wsx)

[Appendix A4: Security Monitor Protection 12](#_heading=h.qsh70q)

# Introduction

This document describes a mechanism to improve the security of a platform. In a platform, the bus masters on it can access the slave devices, just like a RISC-V hart. While a RISC-V hart has PMP, Physical Memory Protection Unit to regulate its access, the bus masters also need a mechanism to regulate their accesses. Here we propose the Physical Memory Protection Unit of Input/Output Devices, IOPMP for short, to regulate the accesses issued from the bus masters.

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, when the requirement is not as simple as a DMA transfer, the check could take a long time. A GPU, for example, can take a piece of program to run. Generally, examining whether a program violates any 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.

# Background and Terminology

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 confined 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*.

# Source-ID and Transaction

Source-ID, SID for short, is a unique ID to identify a bus master or a group of bus masters with the same permission. When a bus master wants to access a piece of memory, it issues a transaction. A transaction should carry an SID indicating which bus master issued it when needed. We will discuss the exception in the next section.

The uniqueness of SID is within the scope of an IOPMP. The scope of an IOPMP is a set of bus masters, and the IOPMP through which the transactions issued from the set of bus masters go is the first IOPMP. The uniqueness will be emphasized when cascading IOPMPs is mentioned. In the document, the SID is assumed to exist when needed, but is not a part of an IOPMP. Assignment and/or configuration SID is not a part of an IOPMP, either. The number of bits of an SID is implementation-defined. 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 case when more flexibility is preferred, 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 master has multiple channels and every channel is granted with different access permissions, each channel should have its own SID. If a bus master 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.

How to carry a SID for a transaction is implementation-defined.

# Source-Enforcement

If the scope of an IOPMP contains only one bus master or a set of bus masters with the same permission, the Source-ID can be ignored on the bus master side as well as the transactions going through the IOPMP. In this case, we denote it as the IOPMP Source Enforced, IOPMP/SE, for short. In the rest of the cases, we still need different SIDs to distinguish the transaction issuers.

# Master Port and Slave Port

An IOPMP has at least a master port and at least a slave port. A slave port is where a transaction goes into the IOPMP, and a master port is where a transaction leaves the IOPMP if the transaction passes all the checks.

# 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.

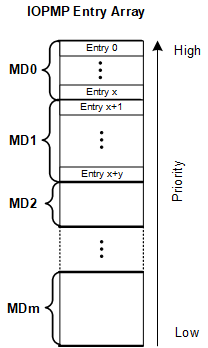
It should be noted: one SID may be associated with more than one MD, and one MD may be associated with more than one SID. However, some models may limit the flexibility due to different requirements.

# 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. Since there is no SID in an IOPMP/SE, when selecting the matching IOPMP entry, an IOPMP/SE ignores the SID comparison.



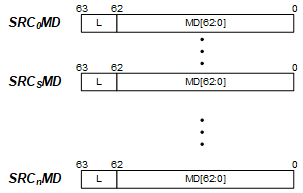
# Models

Since the types of platforms are diverse, we don’t have one size to fit all. Thus, we provide several models to different requirements. We will begin with the full model that equips almost all the features described in this document. We will then mention optional features. By combining these optional features, we will come out with models other than the full model. These models help users select and optimize their designs for different requirements, such as low area, low power, low latency, high throughput, high portability, and so on.

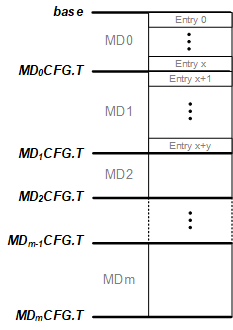
As to IOPMP/SE, we don’t have models, and the IOPMP entry array is mandatory.

# The Full Model

When a full model IOPMP receives a transaction with SID *s*, it firstly lookups a SRCMD table to find out all the memory domains associated with source *s*. The size of the table is implementation-defined. In the table, the register SRC*s*MD is for SID *s*, occupies a 64-bit space, and has two fields, SRC*s*MD.L and SRC*s*MD.MD. SRC*s*MD.L is a sticky lock to this register. In the model, SRC*s*MD.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*s*MD.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. The MDCFG table has an array of registers where the register MD*m*CFG is for the memory domain *m*. One field, MD*m*CFG.T, indicates the bottom index of the IOPMP entry belonging to the memory domain *m*. An IOPMP entry with index *j* belongs to MD *m* if MD*m*-1CFG.T ≤ *j* < MD*m*CFG.T, where *m*>0. The MD 0 owns the IOPMP entries with index *j*<MD0CFG.T. Each MD*m*CFG register occupies a 32-bit space and the field MD*m*CFG.T occupies the lowest 16 bits. The number of implemented bits is implement-defined.



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.

# Configuration Protection

Let’s see the below platform.

<<adding a figure: SoC with MMIO-space IOPMP>>

The idea of the figure is that in the platform, there is an IOPMP also in charge of the bridge to the MMIO bus. Here is the IOPMP-2. The control paths to all IOPMPs (including IOPMP-2 itself) go through it. Thus, if we want to enforce that the control operations from a specific SID can control desired IOPMPs only, we can impose rules in the IOPMP-2. However, if we think about when the security monitor could be breached during the run-time and then these rules could be changed maliciously, we need some mechanisms to prevent such rules from unwanted modifications. We will depict such mechanisms in this section.

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.

# Configuration Protection by a Memory Domain

To lock the IOPMP-2 in the figure-???, you can do the following steps. Firstly, you program IOPMP to the desired setting but reserve MD 0 and IOPMP entry 0. Let all SIDs associate MD 0, and MD 0 has one IOPMP entry, which must be IOPMP entry 0. Then, let the region of the entry contain all control registers of the IOPMP-2. The final step is to enable the entry with no-write permission. A lock is created to stop all future control operations to the IOPMP-2. If you want to reserve some control registers’ programmability, use more IOPMP entries in MD 0.

To lock more IOPMPs, you can put more IOPMP entries in MD 0 to block out all control operations to these IOPMPs.

However, the protection granularity of this method is as fine as 4 bytes. If you want a more fine-grained lock, please check the following subsection.

# The Other Protection Mechanism

In the above method, 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*s*MD[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 SRCsMD.MD[0]=1, and then let MDMSK.MD[*0*]=1 and MDMSK.L=1. The rest of the 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 SRCsMD.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 SRC0MD.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.

Subsequently, the belonging of IOPMP entries, that is MDCFG table, can be locked. The register MDLCK is designed for the purpose, which has two fields: MDLCK.L and MDLCK.F. Please note that if the top index of MD *m* is locked, that of DM *m*-1 should be locked, too. Otherwise, the IOPMP entries of MD *m* can be added or removed by modifying MD*m*-1CFG.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*<MDLCK.F. MDLCK.F is initialized to 0 after reset, and can be increased only when written. MDLCK.L is the lock of MDLCK. If MDLCK is hardwired, MDLCK.L should be wired to 1.

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 security monitor is compromised, one unlocked entry in higher priority can overwrite all the other locked entries in lower priority. To enforce it, we use MD*m*CFG.F, a 15-bit field in MD*m*CFG, to define the number of nonprogrammable entries in the memory domain *m*; that is, the first MD*m*CFG.F IOPMP entries belonging to MD *m* is not programmable. MD*m*CFG.F is initialized to 0 and can be increased only when written. Besides, MD*m*CFG.L is the lock to MD*m*CFG.F and itself. If MD*m*CFG.F is not implemented, MD*m*CFG.L and MD*m*CFG.F should be wired to 1 and 0, respectively. Please note that MD*m*CFG.L does not control whether or not MD*m*CFG.T is programmable. MDLCK.F does.

// Locking partial of the MD cannot prevent privilege attack, such as m-mode attack. For example, if an IOPMP entry is locked, all its precedent IOPMP entries shall be locked. Otherwise, a privileged attacker may tamper a precedent IOPMP entry which effectively invalid the protection enforced by the locked IOPMP entry.

//An alternative solution might be: Use an IOPMPLCK register which defines the number of IOPMP entries to be locked (starting from index 0). And the number should only allow a one way increment in a reset cycle.

# 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 semantic 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.

# The Other Models

We showed two table reductions and several options in the previous section. This section will introduce several common combinations of these reductions and options, the so-called models.

# The Rapid-*k* Model

To shorten the latency, the rapid-*k* model replaces the lookup of the MDCFG table by simple logic. 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 SRCsMD.MD directly maps to a selected IOPMP entry for SID=*s*.

To make it semantically compatible with the full model, the original meaning should be reflected when the related fields are read. MDLCK.F should be the same as the number of implemented MDs and MDLCK.L should be 1. MD*m*CFG.T should be (*m*+1)\**k*. That is, one can read MD0CFG.T to retrieve the value *k*.

MD*m*CFG.F and MD*m*CFG.L can still be programmable or hardwired. The two fields do not typically 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.

# 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. It may happen in a multiple-purpose chip: one purpose needs a larger number of IOPMP entries, and the other needs fewer.

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

MD0CFG.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 shorter latency. MD*m*CFG.T is read-only and equals to (*m*+1)\**k* when it is read. By updating MD0CFG.T and then examining MD1CFG.T’s change, one can know the IOPMP is the dynamic-*k* model.

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

# The Isolation Model

One of the benefits of the two-table design in the full model is to share common memory regions (by memory domains) among different SIDs. However, when the design is not the case, it becomes a burden. The isolation model is for the case of no or a few shared regions. The model asks SID *s* exactly associate with MD *s*. Thus, no table-lookup is needed to retrieve the associated MD. It is good for 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 wired to 1. MDMSK.L should also be wired to 1.

There is no constrain on the MDCFG table and the MDLCK register.

# The Compact-*k* Model

The compact-*k* model is used for even lower latency and smaller area than the isolation model. Besides asking SID *s* should only associate with MD *s*, 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. MDLCK 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.

# 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 SRC0MD.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 MD0CFG.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 MDLCK.L is 1 and MDLCK.F is non-zero, it is the rapid-*k* model. Then, if MD0CFG.T accepts zero, it is the full model; otherwise, the dynamic-*k* model.

# Registers

TBD

# Reset

TBD

# Program IOPMPs

Programming the IOPMP usually consists of a series operation. There exists a potential race condition while the security monitor is programming the IOPMP, some in-flight transactions are checked or under checking but have not been completed yet. Such race conditions may lead to vulnerabilities, for example, another entity starts to use the re-allocated region for confidential information but the entity who had the access is still able to retrieve the data in that region. To mitigate such vulnerabilities, the series of IOPMP updates should be atomic in the view of other transactions that go through the IOPMP.

A straightforward thinking is: all transactions should be checked by either (1) the previous setting before any IOPMP updates or (2) the new setting after all IOPMP updates take effect. However, for some systems, stalling/blocking all transactions on the bus may impose a large impact on the performance; and in certain use cases, it is even not allowed to stall transactions from some specific sources. As an alternative, instead of stalling all transactions, one may implement the IOPMP atomic update in a way that only stalls transactions which are impacted.

An IOPMP update could be any of the following cases:

1. Update a SRC*s*MD register – the impacted scope is transactions from source *s*targeting MDs that are to be updated.
2. Update a MD*m*CFG register – the potential impacted scope is transactions targeting MD*m* and MD*m+1*.
3. Update an IOPMP entry – the impacted scope is transactions targeting address range as specified by the IOPMP entry.

It should be noted that, as there exists memory domain overlap, only fencing transactions targeting the to-be-updated MDs is not sufficient. For example, if MDi and MDj (j > i) are overlapped, an illegal transaction may pass MDj security check if only MDi is fenced. Based on the above observation, without loss of genericity, we introduce two ways to enforce an IOPMP atomic update:

1. Per-SID fence
   * **SIDFEN**: The register has one bitmapped field SIDFEN.SID and a field SIDSTALLEN.M
     1. where each SIDFEN.SID bit corresponding to one source. Setting any SIDFEN.SID bit, e.g. *s0*, indicates new requests from source *s0* shall be stalled/faulted.
     2. SIDFEN.M(mode): indicates if the pending is implemented as ‘fence-as-stall’ or ‘fence-as-fault’.
   * **SIDFENACK**: The register has one bitmapped field SIDFENACK.SID, where each bit corresponding to one source. When any SIDFENACK.SID bit is set, e.g. *s0*, it indicates source *s0* is idle, here 'idle' means:
     1. there is no new request from the source *s0*
     2. all *s0*’soutstanding requests are accomplished
   * Programming Sequence:
     1. Set SIDFEN.SID corresponding bits, *s0, s1, ....,* to indicate the sources that needs to be stalled
     2. Wait until SIDFENACK.SID bits *s0, s1, ....,* are set
     3. Update IOPMP: updating SRCsMD is only allowed when both SIDFEN.SID and SIDFEN.SID bits are set
     4. Unset SIDSTALLEN.SID corresponding bits, *s0, s1, ....,* to indicate the atomic updates finished.
2. **Per-MD fence**

The per-MD stall scheme requires to stall/fault all SIDs that is associated with MD*i* whenever there needs an update on MD*i*. Implement below registers to enforce the atomic update:

* + **MDFEN**: The register has a bitmapped field MDFEN.MD and and a field MDFEN.M,
    1. where each MDFEN.MD bit corresponds to one memory domain. Setting any SIDFEN.MD bit, e.g. *m0*, indicates new requests from any sources, e.g. *s0, s1, ....,* that is associated with *m0*, shall be fenced.
  + **MDFENACK**: The register has one bitmapped field MDFENACK.MD, where each bit corresponding to one MD. When any MDFENACK.MD bit is set, e.g. *m0*, it indicates all *m0* associated sources, e.g. *s0, s1, ....,* are idle, here 'idle' means:
    1. there is no new request from the source *s0, s1, ....,*
    2. all outstanding requests are accomplished
  + **FENBYPASS**: The register has three fields
    1. FENBYPASS.S, which specifies the source (ID) that should bypass the IOPMP stall. This is to prevent the deadlock scenario where the security monitor who should program the IOPMP is stalled.
  + The expected programming sequence are:
    1. Set FENBYPASS.S = SID(security monitor)
    2. Set MDFEN.MD[*m0*] if there is any change on MD *m0.*
    3. Wait until MDFENACK.MD bits m*0* is set
    4. IOPMP blocks any succeeding requests from source *s0, s1, ....,*
    5. Update IOPMP  
       1. Allows to update MDCFG*m0*
       2. Allows to update IOPMP entries that belongs to MD *m0*
       3. Allows to update SRC*s*MD, *s = s0, s1, ....,*
    6. Unset MDFENACK.SID
    7. Unset MDFEN.MD[m0]

# Appendix A1: Reference Data Path

//TBD

# Appendix A2: Cascading IOPMPs

//TBD

# Appendix 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*. Currently, each SPS[*s, m*] has defined 3 fields:

1. SPS[*s, m*].R indicates if the transaction (from source *s*) is allowed to read from MD *m*
2. SPS[*s, m*].W indicates if the transaction (from source *s*) is allowed to write to MD *m*
3. SPS[*s, m*].X is optional, it indicates if he transaction (from source *s*) has executability on MD *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.

Since an IOPMP/PMD has two sets of permission settings, the permission setting in an IOPMP entry could be hardwired to ‘allow-read-and-write’ and we could use the second permission setting only.

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.

The programmability of the SPS table is the same as the programmability of the SRCMD table. SPS extension also supports up to 63 memory domains. The LSB of SPS[*s*, 63] is reserved for locking all second permission settings for SID=*s*.

// Register definition of SPS: TBD

# Appendix A4: Security Monitor Protection

//TBD