MIPS Virtualization: Processor and SOC Support Overview

Document Number: MD01171
Revision 01.00
April 1, 2016

Introduction

MIPS has implemented the MIPS Virtualization Architecture across all classes of its MIPS processors, from the area and power efficient M-class, to the performance-oriented P-series, to the I-class, which balances performance and cost-effectiveness with an emphasis on multi-threading. Thus, the MIPS Virtualization extensions scale across the entire range of processor technology with a focus specific to each processor class and its respective applications.

This document is a *guideline* for understanding the hardware support and software requirements to enable an SOC design—compatible with the MIPS Virtualization Specification—to support a secure platform. Omnishield™ technology provides a comprehensive security solution that is based on the MIPS Virtualization Architecture, and is enhanced with other MIPS IP.

1 MIPS Virtualization: A Brief Overview

This document uses the terms *root* and *guest*. The MIPS Virtualization Architecture requires the implementation of two privileged Coprocessor0; one Coprocessor0 is root context, the other is guest context. The hypervisor operates in root context and hosts the guest software in guest context. These two independent contexts enable a hypervisor to retain its privileged state physically isolated from guest software (which is typically an unmodified OS). While the OS provides a level of abstraction between applications and hardware, the hypervisor provides another level of abstraction between the OS (or multiple OSs) and hardware.

With this second layer of abstraction, a hypervisor can *monitor* and set restrictions on guest access to hardware. The restrictions are applied through memory management techniques, and by hardware preventing the guest from using sensitive instructions. In implementations with simpler requirements, the hypervisor reduces to an entity that may be generically called a *root monitor.*

This root monitor has minimal functionality: it services guest exceptions and interrupts, provides an API for secure services, and enables context switching between guests, if applicable. Thus, the hypervisor ranges from simple and application-specific to a generic off-the-shelf hypervisor capable of supporting multiple OSs simultaneously. The remainder of this document refers to privileged software operating in root context as a hypervisor.

The guest OS continues to manage its virtual memory independently of root. The guest MMU translates guest virtual address (GVA) to guest physical address (GPA). The GPA is translated to root physical address (RPA), or the system physical address through an additional level of address translation managed by the hypervisor. In this manner, the guest OS is actively prevented from interacting with hardware directly, or from accessing memory of root or other guests.

While the terms root and guest have so far been used in the context of hardware, the terms also identify software running in each of the two contexts.

GuestID is a hypervisor-programmable value that is used to tag root and guest operations. Support for GuestID is optional—but highly recommended—to provide a consistent hardware-based method to distinguish operating contexts. Otherwise, hypervisor must intervene to maintain consistent state. (This document assumes the presence of GuestID.)

While root operations are always tagged with a GuestID value of zero, all guest operations are tagged with a non-zero value. The existence of a GuestID allows all operations, whether root or guest, to be clearly distinguished anywhere in the SOC, assuming that the SOC is selectively modified to recognize the GuestID. See "Section 4 "Extending Virtualization from the Core Across the SOC". A virtualized platform thus allows independently operating software domains to co-exist and share hardware resources on the platform in a secure manner.

The MIPS Virtualization Architecture has additional extensions for embedded systems:

- A Root Protection Unit (RPU), a reduced form of the Root MMU, which validates all guest addresses without the overhead of translating to a final system physical address. See Section 2 "Memory Management Support in MIPS Virtualized Processors".
- Shadow Register Sets, which may be partitioned among root and multiple guests. These register sets allow context-switching between root and multiple guests to occur securely with minimal penalty because GPR sets are physically distinct.
- A Virtualized External Interrupt Controller Interface. Root Interrupts are processed independently of guest interrupts. Root may service high-priority interrupts without being blocked by concurrent servicing of a guest interrupt.

2 Memory Management Support in MIPS Virtualized Processors

The following Memory Management support is currently implemented in all classes of MIPS processors. Guest virtual addresses are mapped in a two-step process through one of the following three methods:

1. Guest Fixed Mapping Table (FMT) and Root RPU
2. Guest TLB and Root RPU
3. Guest TLB and Root TLB

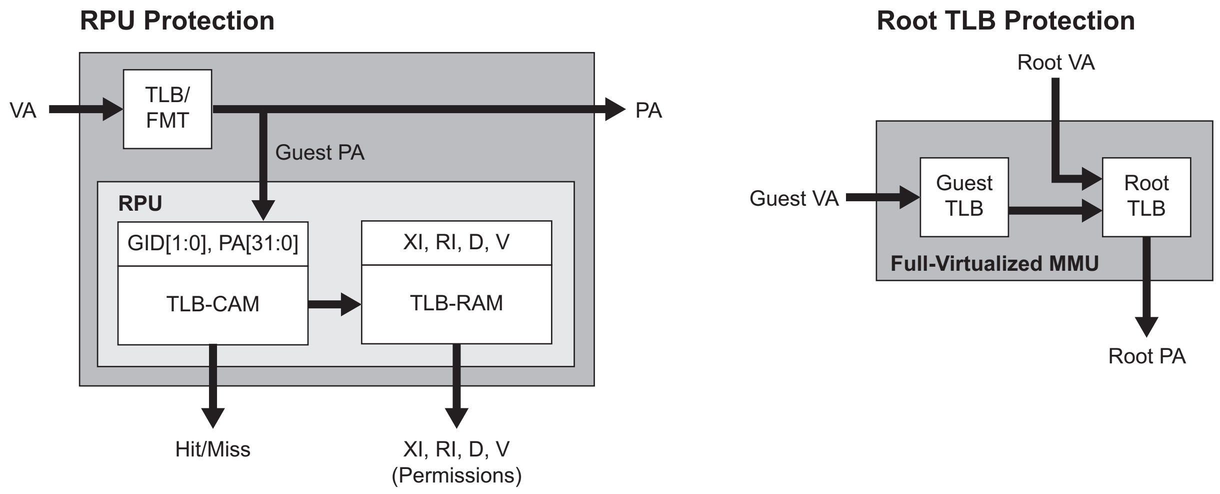
Within the M-class processors, the M515x supports all three configurations while the M510x supports only option 1. The I-class and P-class processors support only option 1.

The RPU is unique to M-class processors. It provides guest-specific protection (read-inhibit (RI), write (D), execute-inhibit (XI), and miss) while reducing area by eliminating storage for the additional address translation step. The RPU only supplies these page-level protection bits and does not provide address re-mapping.

In contrast, the Root TLB supplies both address re-mapping and page-level protection bits.

Figure 1 compares the RPU and Root TLB methods for guarding guest access to memory and MMIO.

Figure 1 Comparing RPU and Root TLB Protection



3 Virtualization Use Cases in Embedded Applications

Virtualization is commonly associated with server-class processor technology. For example, a virtualized platform provides the means to host multiple OSs in a single system in a manner that is transparent to each OS. This arrangement consolidates multiple workloads in data centers to lower hardware costs and reduce power consumption.

The focus of virtualization in an embedded platform is entirely different; the primary purpose is to ensure secure operation. Security, loosely defined, is the capability to isolate, police, and enforce behaviors on guest software. The scope of such secure operation is large, ranging from providing simple software isolation to more complex digital media management. With embedded applications growing increasingly sophisticated and connected, the scope of risks to which an embedded platform is exposed is growing proportionally.

The use cases envisioned for such platforms are:

- Preventing piracy
- Preventing corruption
- Establishing and maintaining trust
- Fast interrupt context switching

3.1 Preventing Piracy

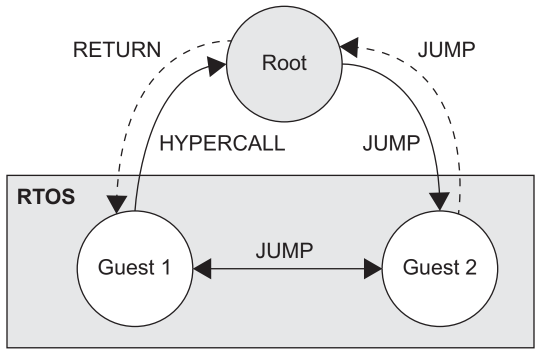
Code in such an environment is typically not single source. For example, the company that designs the SOC provides the boot code and the OEM provides its own application-specific code. The software stack may also contain third-party proprietary libraries.

In the simplest environments, all code is executed by an RTOS at kernel privilege with unmapped access to memory. Without process isolation, nothing prevents copying of proprietary code from memory by an ill-intentioned party that also hosts code in this closed environment.

A virtualized environment based on the MIPS Virtualized Architecture can prevent IP theft by isolating code segments to specified guests. Access from any segment to all other segments is guarded through a page-based memory protection scheme that prevents illegal access. Typically, the RPU used for this purpose is programmed during the secure boot sequence. The hypervisor, whose integrity should be validated through the Secure Boot protocol, is the only entity that can program the RPU, and thus the RPU remains secure throughout runtime operation.

Figure 2 illustrates the simple case of an RPU providing guarded access to two guest address spaces. The RPU setting determines whether one guest can access the address space of another. Guest-to-guest and guest-to-root calls are possible where permitted. One guest's code can jump to another guest's code for a function call only if the RPU allows it. Otherwise, a guest can call root software to obtain permission to access another region in real time. This method also scales to a greater number of guests.

Figure 2 RPU Guarding Guest Address Space



The RPU protects all transitions between guests within the same operating context (RTOS) or transitions from guest to root.

3.2 Preventing Corruption

Memory is partitioned among different guests either statically at boot or dynamically during runtime. The hypervisor-managed front-end TLB maps each guest's memory and thus ensures that both memory and the physically-tagged caches remain isolated throughout operation. This arrangement prevents accidental or intentional corrupting modification of any single entities' data due to another. In particular, root software's integrity is retained throughout operation because guests cannot modify root code or data. In this scenario, it is assumed root software is trusted and follows Secure Boot protocol. Due to this top-down application of isolation and guarantee of integrity, the platform may be considered to be *trusted*, if that is the intent of the platform's environment.

Secure Boot is a modified form of the boot process whereby certain code from boot to privileged software is signed with a digital signature using public key cryptography. The signature is verified by a combination of hardware and firmware running from ROM prior to use.

3.3 Establishing and Maintaining Trust

A secure platform requires establishment of a Root of Trust. (The word Root in Root of Trust is not to be confused with root context in MIPS Virtualization.) Software operating in root context only becomes part of the Chain of Trust if it is designed and validated for secure functionality and its integrity continues to be guaranteed after following a Secure Boot protocol.

Once the Chain of Trust is established, a secure hypervisor with a compact footprint can be run in root context, managing multiple guests. With the flexibility provided by virtualization, multiple secure applications can be run in isolation in each guest. For example, for Digital Rights Media (DRM), the DRM software can run isolated from the host guest OS to decrypt or encrypt copyrighted media safely without concern of decrypted media theft.

By itself, virtualization software (i.e., the hypervisor) does not guarantee security. A secure platform must be provisioned with other hardware and software components. These components may include a security co-processor, a fabric that has the means to distinguish and gate transactions belonging to different contexts, and firewalls to validate access to different address-mapped regions. (A firewall specifies address ranges, with permissions, that each context has rights to access.) Firmware running within the cryptographic boundary would support at least Secure Boot and potentially Secure Update protocols. The features of a security co-processor vary, depending on the security requirements, but typically support verifying digital signatures using Secure Hash, block ciphers such as AES for bulk encryption and decryption, ROM, and OTP for key storage.

3.4 Fast Interrupt Context Switching

Embedded SOCs typically contain numerous devices. Drivers program the devices and service the device interrupts. The latency of these interrupts must be kept to a minimum by delivering the interrupt to the core in the shortest possible time, and minimizing the context-switch penalty.

In a single-core, non-virtualized environment, servicing one interrupt blocks the servicing of others (unless the handler is re-entrant). With virtualization, high-priority interrupts can be assigned to root. Because root context is independent of guest context, it is possible to service the high-priority interrupt without disrupting the state of guest context and avoid the overhead of a context switch. That is, guest interrupt handling does not block high-priority interrupts assigned to root.

4 Extending Virtualization from the Core Across the SOC

MIPS processors are fully compatible with the MIPS Virtualization specification. Resources outside the core but within the SOC ideally must be able to distinguish accesses by different guests and root, that is, virtualization shoul extend throughout the SOC. The extent of hardware change is dependent on the system requirements, because in many cases, trap and emulate by the hypervisor may be sufficient. The following sections describe the various SOC components and the virtualization requirements using this specific context.

4.1 Cache and DRAM

Data from any guest or root can co-reside in the caches without additional tagging to distinguish cache lines for guests and root. The Root MMU in the core and I/O Memory Management Unit (IOMMU) (or fabric firewall) external to the core ensure that guest physical addresses are strictly limited to the guest's system physical addresses.

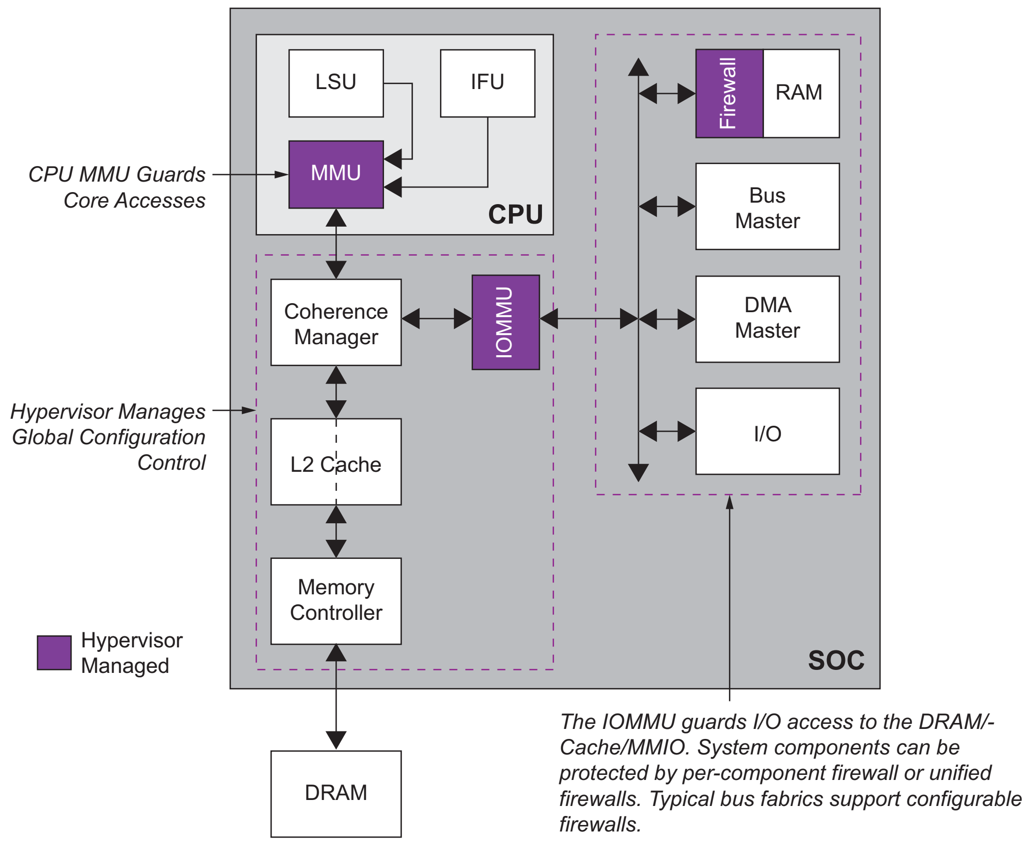
Competitor's security schemes, which provide endpoint validation of accesses prior to accessing memory, require cache-line tagging because the MMUs operate independently in both secure and non-secure contexts. Only the tag bits distinguish between the two data types. It is to be strongly noted that the tagging scheme does not scale to multiple guests and multiple levels of cache hierarchy.

The IOMMU and firewalls protect memory from access by devices with DMA (more generally, addressing) capability. IOMMU and firewalls are discussed subsequently.

A virtual MMU also allows root software to manipulate the Cache and Coherency Attributes (CCA) of any access to memory or memory-mapped I/O (MMIO). The CCA, in it simplest form, specifies whether the access is cacheable or uncacheable. The root MMU may allow guest CCAs to propagate through unmodified, or modify the guest CCA to be cacheable or uncacheable regardless of the guest CCA. For example, a guest may intentionally cache data with an I/O address, which should be non-cacheable, later causing an eviction and writeback to the I/O address while another guest (or root) is in operation.

Figure 3 shows a simplified, logical description of how a hypervisor-managed core, MMU, I/O MMU, and firewalls should be positioned in the system to provide complete isolation of guests and root data.

Figure 3 Hypervisor-Managed MMU, I/O MMU, and Firewall in System


LSU
IFU
CPU MMU Guards Core Accesses
MMU
CPU
Firewall
RAM
Bus Master
DMA Master
I/O
Coherence Manager
IOMMU
Hypervisor Manages Global Configuration Control
L2 Cache
Memory Controller
Hypervisor Managed
SOC
DRAM
The IOMMU guards I/O access to the DRAM/-Cache/MMIO. System components can be protected by per-component firewall or unified firewalls. Typical bus fabrics support configurable firewalls.

4.2 I/O Devices

The following sections describe several models that allow access to I/O devices from the core.

4.2.1 Static Guest Access

The static guess access model assumes that each guest has predefined access to a set of devices that is fixed at boot time or changes in a limited manner with intervention from root software. Guests are considered *well-behaved*, and are limited, with preliminary access permissions specified during boot time. Guest software can make explicit calls to root software through an API to obtain additional permissions.

Each device is associated with guest (or root) explicitly at any given time by programming the device with an identity that corresponds to the guest with permission to access. Illegal accesses to the device are terminated. The system determines whether such errors are reported to root software. Errors of this nature are considered fatal and non-recoverable for the violating guest. They are not construed as an attempt to trap and emulate access through root. To prevent guest hangs of this nature, a root-enabled watchdog timer could interrupt root context as a result of guest inactivity for a preset period of time.

The following pseudo-code indicates the minor enhancement required to support virtualized device identity. This checking can be a wrapper on ingress to the device. Additional checks may be done to qualify read and write access permissions. Such checking may be part of a firewall that may be configured with off-the-shelf bus fabric IP. The pseudo-code can be extended to support more than one permitted guest, as shown.

```
input rGID        // GuestID of request. e.g., - MReqInfo of OCP

var oGID          // GuestID of owner; hypervisor programmed
                  // Alternate : multi-hot vector of permitted owners.
var permit        // Permit access

   if (rGID == oGID | rGID == 0)
                  // Alternate : match multi-hot vector of permitted
                  // Owners against decoded request GuestID
                  // Optional - root always allowed access
      permit=1
   else   permit=0
   endif

   if permit then
      grant access as in non-virtualized case
   else           // not permitted
      terminate access
   endif
endfunction
```

4.2.2 Root Only

In this method, only root software (for example, the hypervisor) has direct access to devices. The root MMU in the core prevents guest software from accessing devices. If guest requires access to devices, it may obtain it through explicit calls to root software. Alternatively, guest device drivers can be para-virtualized—modifications to guest software are limited to the device drivers, or root software can trap and emulate guest access to devices. This method is less efficient and is typically employed in para-virtualized systems.

4.2.3 Dynamic Guest Access

This method is similar to the Static Guest Access model except it supports a larger number of guests. Guests may be granted access on demand. This method is considered a fully-virtualized model.

4.3 GPU

For the most part, a non-virtualized GPU is managed as another system I/O device. A guest interacts with the GPU through a hypervisor API and the GPU driver runs with hypervisor privilege. The GPU requires DMA access to memory. Because the hypervisor programs the GPU, the system could be designed to allow the GPU to bypass an IOMMU to avoid additional latency. However, at a minimum, GPU access to memory should be guarded by a firewall in the on-chip fabric.

Alternatively, if the GPU itself is virtualized, it may support direct guest access.

4.4 DMA

Any device that may require DMA should be guarded by a hypervisor-managed firewall or an IOMMU. Root software always performs the firewall setup. The firewalls can be integrated into fabric as part of the fabric configuration process. The IOMMU can be shared among different requestors, providing the IOMMU can distinguish between the different request streams.

4.5 Firewalls

A firewall describes a system logic block whose input is a request to access an address space, whether it is associated with memory or a Control and Status Register (CSR) (or alternatively MMIO). The firewall checks whether the requestor has the right to access the address space, and if so, allows the request to pass through. Otherwise, the firewall terminates access with system appropriate error reporting.

A firewall can be placed between a device and the fabric that distributes the requests at an ingress point to the fabric or at an egress point of the fabric where requests with a common destination are routed. Typically, firewalls can be integrated into the fabric/NOC to minimize latency.

A firewall does not contain address translation capability, and as such, assumes any request address is already mapped to a system physical address.

The following pseudo-code describes the logic required to satisfy the minimum requirements of a firewall. Additional system-dependent checks may be added. The pseudo-code can be extended to support more than one permitted guest, as shown.

```
input rGID                    // GuestID of request. E.g., MReqInfo of OCP
input Addr                    // Transaction addr - Root Physical Address

var [] oGID                   // Owner’s GuestID for each Addr range
var [] AddrHi                 // AddrHi of bounds register
var [] Addr Lo                // AddrLo of bounds register
                              // - All hypervisor programmed
var permit                    // Permit access

   permit=0
   for i = 1 to max
      if rGID == oGID[i]
         if (AddrHi[i]<= Addr <= AddrLo[i])
            permit =1
      if permit then break    // One match sufficient
   endfor

   if permit then
      grant access as in non-virtualized case
   else                       // Not permitted
      terminate access
      signal error in system-dependent manner
   endif

endfunction
```

4.6 I/O MMU

An I/O MMU has the same capabilities as a firewall except it can also translate from a guest physical address to the system physical address. An IOMMU is typically required for the following reasons:

- For legacy devices that are programmed with physical addresses. If guest is allowed access, this address is a guest physical address. The IOMMU provides the additional level of translation to the root physical address.
- For legacy devices that are programmed with virtual addresses and have an MMU that translates to a guest physical address. In this case, the internal MMU is required because the memory allocation is dynamic. The IOMMU provides the additional level of translation to the root physical address.
- Where DMA access is on demand such as for PCIe. In this case, the IOMMU populates its internal TLBs on demand through an internal hardware page-table walker.

Figure 4 shows a firewall and different forms of IOMMU in a hypothetical system.

Figure 4 Hypothetical Firewall and IOMMU System

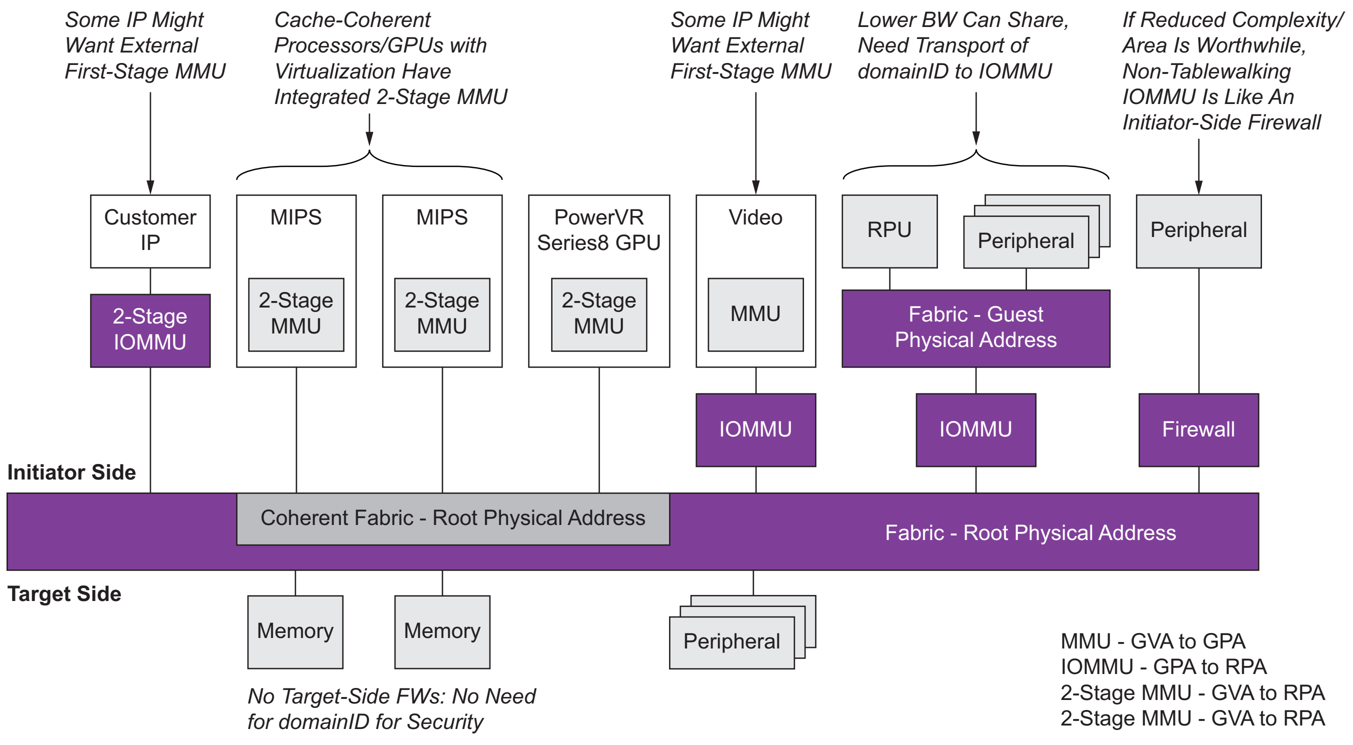
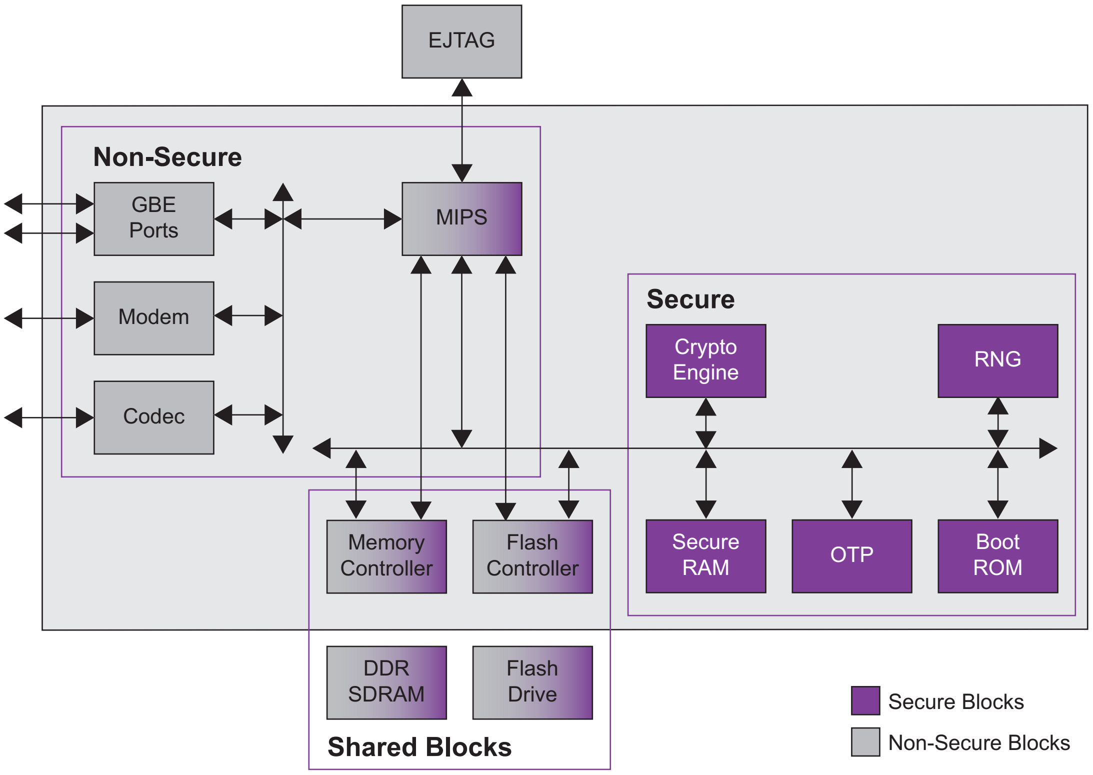

Some IP Might Want External First-Stage MMU
Cache-Coherent Processors/GPUs with Virtualization Have Integrated 2-Stage MMU
Some IP Might Want External First-Stage MMU
Lower BW Can Share, Need Transport of domainID to IOMMU
If Reduced Complexity/ Area Is Worthwhile, Non-Tablewalking IOMMU Is Like An Initiator-Side Firewall
Customer IP
2-Stage IOMMU
MIPS
2-Stage MMU
MIPS
2-Stage MMU
PowerVR Series8 GPU
2-Stage MMU
Video
MMU
IOMMU
RPU
Peripheral
Fabric - Guest Physical Address
IOMMU
Peripheral
Firewall
Initiator Side
Coherent Fabric - Root Physical Address
Fabric - Root Physical Address
Target Side
Memory
Memory
Peripheral
No Target-Side FWs: No Need for domainID for Security
MMU - GVA to GPA
IOMMU - GPA to RPA
2-Stage MMU - GVA to RPA
2-Stage MMU - GVA to RPA

Figure 5 shows a system with strict partitioning of system components into secure and non-secure operation zones. A secure hypervisor running on a single virtualized MIPS core manages access to shared components. Without virtualization, the system requires a separate core for each domain. Shared components for the non-virtualized case would be strictly partitioned without support for dynamically modified allocation.

Figure 5 Strict System Component Partitioning



5 Software Requirements

Privileged software running on a non-virtualized processor architecturally compatible with the MIPS Architecture is portable to guest context with none or minimal modifications. Software operating in root context varies based on the configuration chosen.

5.1 Guest FMT and Root RPU

In this case, the guests are applications running at a common kernel privilege, mapped by the FMT, whose operation is labelled by different values of GuestID as determined by root software. The RPU is programmed with pages allocated to each guest based on a known system address map.

This root software has the following minimal capabilities:

- The RPU detects service guest protection violations or misses with appropriate exception handlers
- Service interrupts assigned directly to root with appropriate handlers, and access to appropriate device drivers
- Service interrupts reported to root but assignable to guests (optional)
    - Directly by root with appropriate device driver
    - Pass-through to guest after root software enables guest context switch

- Provide access to secure services through a pre-defined API

Root software uses an FMT to translate root addresses.

This root software is event-driven with root software triggered by guest and system events or guest calls. The implementation does not require a fully-featured hypervisor with a large memory footprint.

5.2 Guest TLB and Root RPU

The root software requirements described in “Guest FMT and Root RPU” on page 11 apply here, except the guest is an OS.

The RPU segregates the memory available to root from that of a single guest. This model is not meant for dynamic context switching of multiple guests. Support for more than one guest OS instance may be possible to a limited extent, such as in the case of Linux co-existing with an RTOS. The system address map is then strictly partitioned between Linux and RTOS. This partition is visible to the two guests, whereas root’s partition is not.

In this case, the guest OS manages its own virtual memory.

For single-guest instances, the guest OS is the application OS, such as Android. Software, in root context, may provide secure services to the guest OS.

5.3 Guest TLB and Root TLB

This configuration supports full virtualization. Multiple guest OSs may be supported dynamically with a fully-featured hypervisor in root context managing the multiple guest OSs.

6 Virtualizing the Interrupt Source

A fully-virtualized MIPS core distinguishes root interrupts from guest interrupts, or more specifically the current resident guest on a platform that supports multiple guests (guests may be other OSs or applications that need to be run in isolation).

The MIPS Global Interrupt Controller IP supports full-virtualization. However, any generic Interrupt Controller can support full-virtualization that is compatible with the MIPS Virtualization Specification.

A generic External Interrupt Controller (EIC) typically has a number of input Interrupt Ports that are statically tied to devices in the system. It also has one logical output port to each core in the system, where the port has independent channels for root and guest interrupts. Input port interrupts are routed to the output ports. Logic within the EIC is implementation-dependent, although each slice of logic for the interface to the core can also have a root and guest section to configure interrupts separately for root and guest for the core. The following sections describe the interface virtualization.

6.1 Device to Interrupt Controller Interface

If an EIC interrupt port is tied to either a root or guest-owned device (and not just root), the port should be modified such that it can be programmed with GuestID. In a multi-core system, each port also identifies a core destination. An interrupt can then be routed to a specific core through a specific interrupt channel (root or guest) for the core.

6.2 Interrupt Controller to Core Interface

The logical output port to a core is split into the Root Interrupt Bus and Guess Interrupt Bus. These two independent channels route root and guest interrupts to the core.

The following steps are required to deliver up to two interrupts (one for root, one for guest) in a cycle. The description assumes that interrupts are prioritized and represented by an Interrupt Priority Level.

1. Prioritize all incoming root (GuestID=0) interrupts every cycle based on assigned Interrupt Priority Level (IPL).
    - Deliver highest priority interrupt for the cycle on Root Interrupt Bus.
2. Prioritize all incoming guest (GuestID!=0) interrupts every cycle based on assigned Interrupt Priority Level (IPL).
    - If prioritized GuestID != resident GuestID, deliver interrupt on Root Interrupt Bus. Otherwise, deliver on Guest Interrupt Bus.

      The resident GuestID is established from an input to the EIC from each core.

The External Interrupt Controller may reassign an interrupt from the Root Interrupt Bus to the Guest Interrupt Bus if the guest interrupt on the Root Interrupt Bus can be delivered to the core guest context as a result of a context switch, that is, the guest is now resident. Such handling is optional because root software can accomplish the same task by reprogramming the interrupt controller before switching guest context. The EIC can reassign active interrupts in this way as long as the core has not registered interrupt. This may be established by checking the Interrupt Priority Level that accompanies the interrupt acknowledgment.

7 Secure Debug

Secure systems should protect against situations in which a malicious user can use debug facilities to gain access to components or memory regions that are meant to be protected from guest-level access. In-circuit debugging (such as MIPS EJTAG) with debug probes is especially vulnerable to such attacks.

Current MIPS CPUs supply a mechanism to disable EJTAG access for devices in production. EJTAG can then be re-enabled at the factory for RMA purposes.

MIPS is working on next-generation mechanisms to authorize EJTAG access and make EJTAG usage more flexible for secure systems.

8 Secure Boot

As previously mentioned, components with Hardware Virtualization support, such as CPUs and GPUs, are (by themselves) insufficient to build truly secure systems. Secure systems require a Root of Trust from which all subsequent software component authentication can be created and trusted.

MIPS is working on such Root of Trust products and the secure boot process that is necessary to build secure systems.

9 Conclusion

The MIPS Virtualization Architecture implemented in a MIPS core for embedded applications provides a unique set of features that can be exploited to distinguish a dynamic and secure platform from the competition.

10 References

- MD00846: MIPS32® Architecture for Programmers Volume IV-i: Virtualization Module of the MIPS32® Architecture

11 Revision History

Revision	Date	Description
01.00	April 1, 2016	• First revision.