

# AArch64 ThinHypervisor 開発記録

2023/12/02(土) Kernel/VM探検隊@北陸 Part 6

産業技術総合研究所超分散コンピューティング研究チーム森 真誠



# 本発表の内容

本発表では、AArch64向けThinHypervisor: MilvusVisor( <https://github.com/RIKEN-RCCS/MilvusVisor> )  
の開発の中で遭遇したArm CPUの低レイヤー特有の問題や解決方法、それに至る経緯を説明

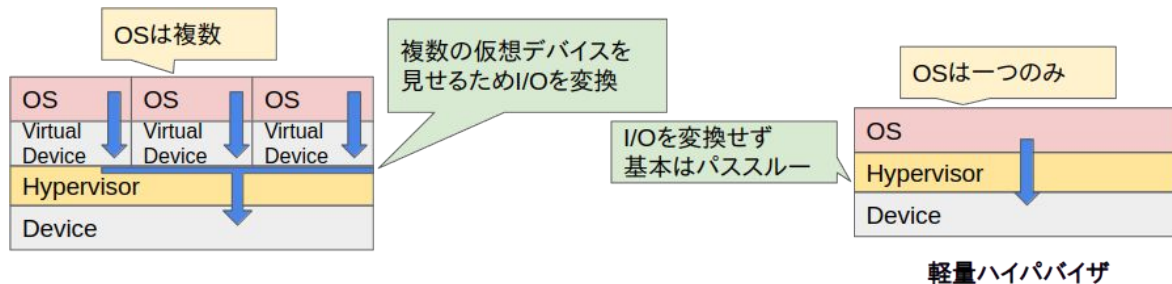
今回発表するのは以下の三本

- Armでプログラミングする際はメモリキャッシュに気をつけよう
- PCI Base Address Registerは絶対アドレスじゃ無かった話
- Arm用の割り込みコントローラGICv3とPCIのITS-MSI割り込みについて

注意: 本発表での意見・感想は個人的なものであり、組織を代表するものではありません

# ThinHypervisorとは

- Hypervisorは通常、複数のOSを動かすためのソフトウェア
  - 複数のOSを動作させるために仮想デバイスでI/Oを変換
- Thin HypervisorはOSを一つしか動かさない代わりにデバイスの仮想化をしない
- そのため動作が軽量
- 性能劣化が少ない





## 何故仮想化するのか

- HypervisorはOSを高い権限レベルで動作する
- OSの動作を制限できる
- 実デバイスの中に仮想デバイスを紛れ込ませることができる
- デバイスのファームウェアなどをOSからアクセスできないようにする
  - MilvusVisorではIntel I210 NICのEEPROMの書き込みを禁止している
- OSの動作をトレースすることができる
  - メモリの書き込みを制限して書き込もうとしている内容をデバッグすることが可能
  - 裏で何をしているかをデバッグすることができる

Thin Hypervisorにも色々使い道はある

---

それでは本編に入ります



Armでプログラミングする際は  
メモリキャッシュに気をつけよう



## 背景

- Armマシンで手頃なものといえば、**Raspberry Pi**だがMilvusVisorは対応してなかった
- MilvusVisorはブートローダーがUEFIで、ファームウェアがACPI or DTBの場合にのみ対応

2023/06/01に@garasubo さんにより、U-Bootを使用して1コアのみ(nosmp)で動作するようになった

<https://github.com/RIKEN-RCCS/MilvusVisor/pull/3>



## MilvusVisorとマルチコア

- CPUのコアには仮想化に関する設定レジスタが存在
- CPUの初期化は権限の高いレベルから順に初期化
- MilvusVisorではOSがCPUを起動した際に割り込んで先に初期化

Armのコア起動方法は

- PSCI( Power State Coordination Interface): Firmwareを呼び出してコアを起動
- Spin Table: jump loopで同じ場所をぐるぐる回っているコアのジャンプ先を変更

MilvusVisorはPSCIによるコア起動のみ対応

Raspberry Pi 3/4はSpin Tableによるコア起動





# Spin Table

- ブートローダなどで全部のコアが起動され、メモリのどこかにあるspin Tableを回り続ける
- “cpu\_release\_addr”にジャンプしてほしいアドレスを書くとそのアドレスにジャンプ
- “cpu\_release\_addr”のアドレスはDTB(Device Tree Blob)に記載

[https://github.com/u-boot/u-boot/blob/u-boot-2023.07.y/arch/arm/cpu/armv8/spin\\_table\\_v8.S](https://github.com/u-boot/u-boot/blob/u-boot-2023.07.y/arch/arm/cpu/armv8/spin_table_v8.S)

```
spin_table_reserve_begin:
0:    wfe
      ldr    x0, spin_table_cpu_release_addr
      cbz    x0, 0b
      br     x0

.globl spin_table_cpu_release_addr
      .align 3
spin_table_cpu_release_addr:
      .quad  0
```



## MilvusVisorのSpin Table戦略

1. システム起動時にDTBをパースしてcpu\_release\_addrを特定
2. OSからのcpu\_release\_addrへのアクセスをMilvusVisorでトラップするように設定
3. OSからcpu\_release\_addrへの書き込みがあった際その値を保存し、  
MilvusVisorのコア初期化用アドレスをcpu\_release\_addrに書き込む
4. MilvusVisorで仮想化関連のレジスタを初期化
5. 保存していたOSが書き込もうとしていたアドレスにジャンプ

## 実装初期は動いたのだが ...

デバッグコードを混ぜながら、いざ実装してみるとすんなり動いた！

**が** デバッグコードを消してリリースできる状態にすると動いたり動かなかったり

逆黒髭危機一髪デバッグをしたところ  
println!を消すと動かないことが発覚

...キャッシュが問題🤔？

```
0.009477] CPU1: Booted secondary processor 0x0000000001 [0x410fd083]
0.010653] Detected PIPT I-cache on CPU2
0.010780] CPU2: Booted secondary processor 0x0000000002 [0x410fd083]
0.011915] Detected PIPT I-cache on CPU3
0.012047] CPU3: Booted secondary processor 0x0000000003 [0x410fd083]
0.012195] smp: Brought up 1 node, 4 CPUs
0.012285] SMP: Total of 4 processors activated.
0.012307] CPU features: detected: 32-bit EL0 Support
0.012326] CPU features: detected: 32-bit EL1 Support
0.012348] CPU features: detected: CRC32 instructions
0.012499] CPU: All CPU(s) started at EL1
```



## Armのキャッシュ

- Arm CPUは明示的にキャッシュが存在
- キャッシュから追い出されない限りメインメモリには反映されない(らしい)
- データキャッシュを飛ばすための命令(DC命令)が存在
  - Invalidate: キャッシュラインを破棄する(WriteBackされてないデータは消える)
  - Clean: キャッシュを書き戻し、同期する
  - Flush: キャッシュを書き戻し、キャッシュを無効化する
- コア間でメモリ内容を共有する際は明示的にキャッシュのクリアが必要

cpu\_release\_addrの部分のメモリキャッシュをCleanすればいいのでは？

(補足: MilvusVisorで他のコアとデータを共有する部分もCleanする必要あり)

# DC CVAC: 指定した仮想アドレスのキャッシュを Clean

DC CVAC, `cpu_release_addr`などをすれば動くのでは？

## Arm A-profile Architecture Registers

Version: 2023-09

Download

Subscribe

Search within this document

### DOCUMENT TABLE OF CONTENTS

- DC CIVAC: Data or unified Cache line Clean and...
- DC CSW: Data or unified Cache line Clean by Set/Way
- DC CVAC: Data or unified Cache line Clean by VA to PoC**
- DC CVADP: Data or unified Cache line Clean by VA to...
- DC CVAP: Data or unified

### ← Previous Section

## DC CVAC, Data or unified Cache line Clean by VA to PoC

The DC CVAC characteristics are:

### Purpose

Clean data cache by address to Point of Coherency.

---

だがしかし ...  
うごかなかった ...



## 苦肉の策として ...

MilvusVisor 1.3.0では、コア起動ごとに“The initialization completed.”と出るお間抜けな実装に

```
cpu::isb();
cpu::clean_and_invalidate_data_cache(accessing_memory_address);
/* FIXME: The current implementation probabilistically fails to boot a guest on Raspberry Pi 4B
 * with multiple cores. This message printing is a workaround to reduce the probability of
 * boot failure. A bug related to cache coherency, exclusive control, or other non-deterministic
 * event is suspected (See #10 for details). */
println!("The initialization completed.");
//pr_debug!("The initialization completed.");
```

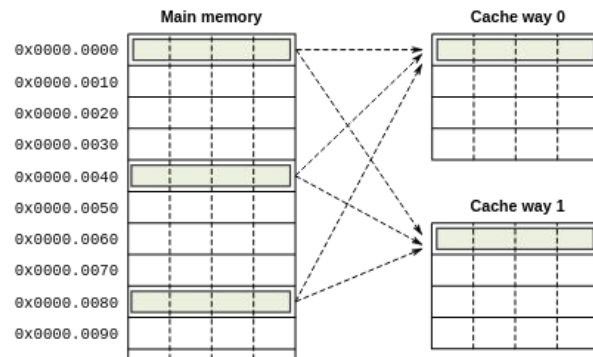
# Arm Architecture Reference Manualを読んでいると

DC C1SW, Data or unified Cache line Clean and Invalidate by Set/Way  
Set/Way/CacheLevelを指定してキャッシュをCleanできる命令

- Set: キャッシュラインの集まり
- Way: メモリアドレスから生成されるインデックス値が  
同じキャッシュラインの集まり
- CacheLevel: キャッシュラインのレベル(L1/L2/L3 Cache)

いくつかのソフトウェアはこれを使って全キャッシュクリア  
する関数を用意している

<https://developer.arm.com/documentation/den0013/d/Caches/Cache-architecture/Set-associative-caches>







## DC CISWを用いて clean\_data\_cache\_all関数を実装

Setの数とWayの数は”clidr\_el1”というシステムレジスタに入っているため、これを元に全てのSet/Wayの組み合わせに対しDC CISW命令を発行

実装内容: <https://github.com/RIKEN-RCCS/MilvusVisor/blob/v1.3.1/src/common/src/cpu.rs#L623>


これをメモリにデータを書き込んだ後に呼び出すように変更しprintlnを削除

## 結果

安定して動作するようになり、Raspberry Pi 4 SMP対応の正式リリースが出来た🎉

# MilvusVisor v1.3.1

Latest

 fukai-t released this Sep 14  v1.3.1  aab5da6

---

## Changes from version 1.3.0

---

Resolve [#10](#)

This version allows you to run MilvusVisor hypervisor with SMP on raspberry pi 4 B !



## まとめ

- Armマシンでコアを起動するには、PSCIとSpin Tableがある
- Arm CPUのコア間でメモリを介してデータをやり取りする際はキャッシュに気を付ける  
必要あり
- 仮想アドレスを指定してキャッシュを飛ばす方法はうまくいかないことあり
  - なぜかはよくわからない
- Set/Wayを指定してデータキャッシュを飛ばす方法はうまくいった

Armのキャッシュはよくわからないこともあるけど、気をつけないといけないことが多い



## (頂いた指摘)

Q: spin tableのwfe命令を抜けるためにSEV命令は発行しなくていいのか？

A: release\_addrに関しては書き込みを行うゲストOSが適切にSEV命令を発行するはずなので、MilvusVisorでは発行していない。



# PCI Base Address Registerは 絶対アドレスじゃ無かった話

# PCIデバイスの操作の仕方

PCIデバイスはBus:Device.Function で表されるツリー構造を取っていて、それぞれのデバイスの情報はメモリ空間にマップされているPCI Configuration Spaceにアクセスすることで取得可能

PCI Configuration SpaceはLinuxで「lspci -vvxx」コマンドなどを使用すると眺めることが可能

```
-[0000:00]--00.0 Intel Corporation Coffee Lake HOST and DRAM Controller
+02.0 Intel Corporation WhiskeyLake-U GT2 [UHD Graphics 620]
+12.0 Intel Corporation Cannon Point-LP Thermal Controller
+14.0 Intel Corporation Cannon Point-LP USB 3.1 xHCI Controller
+14.2 Intel Corporation Cannon Point-LP Shared SRAM
+15.0 Intel Corporation Cannon Point-LP Serial IO I2C Controller #0
+15.1 Intel Corporation Cannon Point-LP Serial IO I2C Controller #1
+16.0 Intel Corporation Cannon Point-LP MEI Controller #1
+17.0 Intel Corporation Cannon Point-LP SATA Controller [AHCI Mode]
+1d.0-[01]--00.0 Realtek Semiconductor Co., Ltd. RTL8411B PCI Express Card Reader
| \-00.1 Realtek Semiconductor Co., Ltd. RTL8111/8168/8411 PCI Express Gigabit Ethernet Controller
+1d.1-[02]----00.0 Intel Corporation Wi-Fi 6 AX200
+1d.4-[03]----00.0 Samsung Electronics Co Ltd NVMe SSD Controller SM981/PM981/PM983
+1f.0 Intel Corporation Cannon Point-LP LPC Controller
+1f.3 Intel Corporation Cannon Point-LP High Definition Audio Controller
+1f.4 Intel Corporation Cannon Point-LP SMBus Controller
\~1f.5 Intel Corporation Cannon Point-LP SPI Controller
```

# PCI Configuration Space

- メモリ空間にテーブル上に並んでいて  
BusとDeviceとFunctionを指定してアクセス可能
- 各エントリには、Device IDやデバイスの状態・設定が存在
- Base Address Registerが0~5までの6つあり、  
ここにかかっているアドレスを元にデバイスを操作する

PCI Configuration Spaceでは基本的な設定や割り込みの設定を行い、実質的なデバイス操作は、BARに書かれていたアドレスでMMIOを介して行う

<https://wiki.osdev.org/PCI>

Register	Offset	Bits 31-24	Bits 23-16	Bits 15-8	Bits 7-0
0x0	0x0	Device ID		Vendor ID	
0x1	0x4	Status		Command	
0x2	0x8	Class code	Subclass	Prog IF	Revision ID
0x3	0xC	BIST	Header type	Latency Timer	Cache Line Size
0x4	0x10	Base address #0 (BAR0)			
0x5	0x14	Base address #1 (BAR1)			
0x6	0x18	Base address #2 (BAR2)			
0x7	0x1C	Base address #3 (BAR3)			
0x8	0x20	Base address #4 (BAR4)			
0x9	0x24	Base address #5 (BAR5)			
0xA	0x28	Cardbus CIS Pointer			
0xB	0x2C	Subsystem ID		Subsystem Vendor ID	
0xC	0x30	Expansion ROM base address			
0xD	0x34	Reserved			Capabilities Pointer
0xE	0x38	Reserved			
0xF	0x3C	Max latency	Min Grant	Interrupt PIN	Interrupt Line

# Base Address Register(BAR)

- PCI Configuration Spaceの主なレジスタと言っても過言ではないレジスタ
- PCIデバイスとやりとりする際は、このBARレジスタからIO空間を読み取る必要がある
- IO Port(x86でのIN/OUT命令用の16bitアドレス空間)とMMIO(32/64bitアドレス空間)の二種類
- 最近のデバイスは、Memory Spaceでやり取りすることが多い
- BARの内容は変更可能

今回はこのBARの値をそのまま使用してはいけない  
というお話をします

Memory Space BAR Layout

Bits 31-4	Bit 3	Bits 2-1	Bit 0
16-Byte Aligned Base Address	Prefetchable	Type	Always 0

I/O Space BAR Layout

Bits 31-2	Bit 1	Bit 0
4-Byte Aligned Base Address	Reserved	Always 1

<https://wiki.osdev.org/PCI>



# — 背景



## MilvusVisorで実際のPCIバスにVirtIOを実装してた

- PCI Busなど既存のデバイスはできる限りそのまま使用して追加の仮想デバイスを作成
- PCI Configuration Spaceへのアクセスを一部トラップし、実際にはデバイスが存在しないのにあたかもVirtIO PCIが存在しているかのように振る舞ってた
  - BitVisorを知っている人は**Virtual VirtIO Net**を想像してもらえると  
("drivers/net/virtio\_net.c"と"drivers/net/virtual\_virtio\_net.c")
- PCI Configuration SpaceのBARを監視してOSが書き換えたらそれに追従してMMIO空間のアドレスを変更する必要あり



## そして事件は起こった ...

LinuxはPCIのBARを整理して変更してくるが、その際に書き込まれるアドレスがlinuxの主張するものと食い違った

以下は、起動メッセージ“Hypervisor: “から始まる行はハイパーバイザが出力したもの

```
[ 1.454562] pci 0000:00:01.0: BAR 0: assigned [mem 0xe000200000-0xe000200fff 64bit]
Hypervisor: BAR0: 0x200004
Hypervisor: BAR1: 0x0
Hypervisor: New Base Address: 0x200000
```

Linuxが主張するBase Address: 0xe000200000  
BARに書き込まれたアドレス      0x200000

---

この挙動を理解するために  
Linux Kernel 5.4を読んてみた



# PCI Base Addressの割り当て

PCI Base Address RegisterにMMIOのアドレスを書き込んでいるのは、  
drivers/pci/setup-res.c の pci\_std\_update\_resource

pci\_std\_update\_resource(struct pci\_dev \*dev, int resno)

概要:

- res = dev->resource[resno];
- **pcibios\_resource\_to\_bus**(dev->bus, &region, res);
- new = **region.start**;
- pci\_write\_config\_dword(dev, reg, new);

```
static void pci_std_update_resource(struct pci_dev *dev, int resno)
{
    struct pci_bus_region region;
    bool disable;
    u16 cmd;
    u32 new, check, mask;
    int reg;
    struct resource *res = dev->resource + resno;

    /* Per SR-IOV spec 3.4.1.11, VF BARs are RO zero */
    if (dev->is_virtfn)
        return;

    /*
     * Ignore resources for unimplemented BARs and unused resource slots
     * for 64 bit BARs.
     */
    if (!res->flags)
        return;

    if (res->flags & IORESOURCE_UNSET)
        return;

    /*
     * Ignore non-moveable resources. This might be legacy resources
     * which no functional BAR register exists or another important
     * system resource we shouldn't move around.
     */
    if (res->flags & IORESOURCE_PCI_FIXED)
        return;

    pcibios_resource_to_bus(dev->bus, &region, res);
    new = region.start;

    if (res->flags & IORESOURCE_IO) {
```



## pcibios\_resource\_to\_bus の中身

region->start = res->start - offset;

offsetはbridge->windowsを探して  
resを含むwindow->resがあれば、セットされる

予想:

res->start: 0xe000200000

offset: 0xe000000000

region->start: 0x200000 (BARに書き込まれる)

```
void pcibios_resource_to_bus(struct pci_bus *bus, struct pci_bus_region *region,
                           struct resource *res)
{
    struct pci_host_bridge *bridge = pci_find_host_bridge(bus);
    struct resource_entry *window;
    resource_size_t offset = 0;

    resource_list_for_each_entry(window, &bridge->windows) {
        if (resource_contains(window->res, res)) {
            offset = window->offset;
            break;
        }
    }

    region->start = res->start - offset;
    region->end = res->end - offset;
}
```

## window->offsetがセットされるのは？

```
static acpi_status acpi_dev_new_resource_entry(struct resource_win *win,
                                              struct res_proc_context *c)
{
    struct resource_entry *rentry;

    reentry = resource_list_create_entry(NULL, 0);
    if (!rentry) {
        c->error = -ENOMEM;
        return AE_NO_MEMORY;
    }
    *rentry->res = win->res;
    reentry->offset = win->offset;
    resource_list_add_tail(reentry, c->list);
    c->count++;
    return AE_OK;
}
```

---

ACPIかあ...





## acpi\_dev\_new\_resource\_entryの呼ばれ方

コードリーディングをして以下のように呼ばれると予想

```
acpi_pci_probe_root_resources    (drivers/acpi/pci_root.c:769)
↳ acpi_dev_get_resources        (drivers/acpi/resource.c:622)
  ↳ __acpi_dev_get_resources     (drivers/acpi/resource.c:569)
    ↳ acpi_walk_resources        (drivers/acpi/acpica/rsxface.c:593)
      ↳ (CallBack)
        acpi_dev_process_resource (drivers/acpi/resource.c:530)
          ↳ if (acpi_dev_resource_address_space(ares, &win))
            return acpi_dev_new_resource_entry(&win, c);
```



## acpi\_dev\_get\_resources

コメント曰く、「与えられたデバイスの\_CRS関数を評価し、リソースを処理する」

```
/**  
 * acpi_dev_get_resources - Get current resources of a device.  
 * @adev: ACPI device node to get the resources for.  
 * @list: Head of the resultant list of resources (must be empty).  
 * @preproc: The caller's preprocessing routine.  
 * @preproc_data: Pointer passed to the caller's preprocessing routine.  
 *  
 * Evaluate the _CRS method for the given device node and process its output
```

---

# ACPI Machine Language(AML) だぁ...



# ACPI Machine Language(AML)とは

- デバイスの接続状況が記述されており  
名前付き数値や制御用の関数が混在
- 専用ツールによりASL(ACPI Source Language)  
に変換可能(AMLはバイナリデータ)
- 名前空間がツリー構造になっている  
(素直に親→子で記述されているのではなく  
右図のようにネストしている場合あり)
- OS/VMMを作っているときにインタープリタ  
を作る素敵な体験ができる
  - 2年前に自作OSでPCの電源を切ろうと  
して苦戦した経験あり

```
Device (LPCB)
{
    Name (_ADR, 0x001F0000) // _ADR: Address
    Scope (\_SB)
    {
        Scope (\_SB)
        {
            OperationRegion (SMI0, SystemIO, 0x0000FE00, 0x00000002)
            Field (SMI0, AnyAcc, NoLock, Preserve)
            {
                SMIC,      8
            }
        }
    }
}
```



## AMLを読み解くには

- Intelが開発しているACPICAというソフトウェアを組み込む
  - C言語製でそこそこ大きい
  - メモリの確保・開放などいくつかの関数を提供する必要あり
  - LinuxはACPICAを組み込んでいる
- 自分でインタプリタなどを組む
  - 必要な機能だけ実装できる
  - 茨の道(登壇者はRustで自作した: Methylenixの”src/kernel/drivers/acpi/aml”)
  - OpenBSDのAMLインタプリタは自前で実装

---

# 閑話休題



## PCIとACPIの\_CRS関数の関係

Linuxにドキュメントがあった <https://docs.kernel.org/PCI/acpi-info.htm>

以下抜粋

There's no standard hardware mechanism for enumerating PCI host bridges, so the ACPI namespace must describe each host bridge, the method for accessing PCI config space below it, the address space windows the host bridge forwards to PCI (using \_CRS)

意識

PCI host bridgeを検出する標準的なハードウェアメカニズムは存在せず、ACPI名前空間は各ホストブリッジについて記述しなければならず、\_CRSはアドレス空間の情報を提供する

## PCI Host Bridgeの\_CRS関数を見える

QWordMemoryの値を返却

```
Method (_CRS, 0, Serialized) // _CRS: Current Resource Settings
{
    Name (RBUF, ResourceTemplate ()
    {
        QWordMemory (ResourceProducer, PosDecode, MinFixed, MaxFixed, Cacheable, ReadWrite
            0x0000000000000000, // Granularity
            0x0000000000000000, // Range Minimum
            0x000000007FFFFFFF, // Range Maximum
            0x00000000E000000000, // Translation Offset
            0x000000000800000000, // Length
            ,, , AddressRangeMemory, TypeStatic)
    })
    Return (RBUF) /* \_SB_.PCI0._CRS.RBUF */
}
```





## Translation Offset...?

[https://uefi.org/htmlspecs/ACPI\\_Spec\\_6\\_4\\_html/06\\_Device\\_Configuration/Device\\_Configuration.html#qword-address-space-descriptor](https://uefi.org/htmlspecs/ACPI_Spec_6_4_html/06_Device_Configuration/Device_Configuration.html#qword-address-space-descriptor) に記述あり

Byte 30	Address Translation offset, _TRA bits[7:0]	For bridges that translate addresses across the bridge, this is the offset that must be added to the address on the secondary side to obtain the address on the primary side. Non-bridge devices must list 0 for all Address Translation offset bits.
---------	--	---

このBridgeを跨ぐ際にアドレスに加算されるアドレスらしい



## つまり

- 今回使用したマシンのPCI Host BridgeはMemory Bus→PCI Deviceの時は、**アドレスから0xe000000000を減算**し、逆方向は**加算**される仕様になっていた
- BARに書かれるアドレスはこれを考慮して、PCIデバイス側に通知されるアドレスを格納する必要があった
- 今回はPCI Host Bridgeを経由しない(VirtIOの処理はCPUで行う)ので、このアドレス変換をエミュレートする必要があった



## Linuxにpr\_infoをいれて確認してみる

各関数にpr\_infoを入れて値が伝搬しているか確認

```
[ 24.084991] acpi PNP0A08:00: ECAM area [mem 0xe100000000-0xe10fdfffff] reserved by PNP0C02:00
[ 24.093530] acpi PNP0A08:00: ECAM at [mem 0xe100000000-0xe10fdfffff] for [bus 00-fd]
[ 24.101266] acpi_pci_probe_root_resources: probe PCI Bus 0000:00
[ 24.206568] acpi_dev_new_resource_entry: win->offset: 0xe000000000
[ 24.223076] PCI host bridge to bus 0000:00

...

[ 24.462843] pci 0000:00:01.0: BAR 0: assigned [mem 0xe000200000-0xe000200fff 64bit]
[ 24.470487] pci_std_update_resource: res->start = 0xe000200000
[ 24.476308] pcibios_resource_to_bus: window->offset: 0xe000000000
[ 24.482389] pci_std_update_resource: region.start = 0x200000
```



## まとめ

- PCI Configuration SpaceのBARの値を扱う際にはそれにつながっているBridgeの情報を確認し Translation Offsetの値を取り出す必要あり
- PCI Host Bridgeを検出する際は、ACPIのAMLを解析してPCIデバイスの記述を検索し、\_CRS関数を評価する必要あり
  - ECAMでの”MCFG”テーブルなどはあくまで補助的に利用する迄
  - PCIデバイスの実装には実質AMLインタープリタが必須

今回の対応:

ハイパーバイザでBARのアドレスを該当マシンでだけオフセットを加算するようにハードコートした  
(AMLインタープリタを載せるのは断念)



# Arm用の割り込みコントローラ GICv3とPCIのITS-MSI割り込みに ついて



## 背景

1. 割り込みコントローラなどは仮想化せず、そのまま使用
2. VirtIOの実装も(PCI BARのオフセット問題を解決し)完了し、実際に動作させてみた
3. しかし、データの受信がうまくいかない
4. 調べてみると**割り込みが発生していない**
5. VirtIO PCIの割り込みはMSI-Xを使用しておりこれについて調査

使用するマシンは、GIC(Generic Interrupt Controller)v3を搭載



## MSI(Message Signaled Interrupt)とは

- 旧来の割り込みピンによる割り込みに置き換わり、メモリバスに対して6/32bitのMessage Dataを送ることで割り込みを通知する方式
- データの送信先を割り込みコントローラにすることで自分で決定した割り込み番号に割り込みを起こさせることが可能

x86\_64では

- Local APICのMMIO(通常は0xfec00000~)に書き込むことで割り込みを起こせる
- 複数のコアに対し、順番に割り込みを起こすなどが可能



# Generic Interrupt Controller version 3 とは

- Armマシンのうち、そこそこ高スペックなマシンに搭載される割り込みコントローラ
- x86\_64のAPICと同等以上の機能を持つ

GICv3は以下のコンポーネントから成る

- Distributor: I/O APICに該当するもの、外部割り込みの設定を担う
- Redistributor: Local APICに該当するもの、Local Timerなどコアごとの割り込みの設定を担う
- Interrupt Translation Service: 外部からの割り込みを割り込み番号に変換する**MSIで使用**

今回はInterrupt Translation Service(ITS)が関係している





# Interrupt Translation Service

- 割り込み元を示す**DeviceID**と割り込みの種類を示す**EventID**を元にInterrupt Translation Tableを元に割り込み番号(**INTID**)と割り込み先のRedistributor(≒コア)を決定
- 割り込みの優先度を柔軟に変更したり、割り込みを起こす先のRedistributorを適宜変更可能
- 割り込みに関する設定は、Command Queueに適宜コマンドを発行することで設定

x86\_64では割り込み先のコアID(APIC ID)や割り込み方法、割り込み番号などをMSIのMessage Dataとして持たせていたのに対して、GICv3ではCPUがテーブルにセットし、MSIではDeviceIDとEventIDのみ通知する方式



## なぜVirtIOのMSI-X割り込みが上手く行かないか

ハイパーバイザがやることは、設定されたMessage Addressに設定されたMessage Dataを書き込むだけなのでは？

```
*((uint32_t *)message_address) = message_data; //やることはこれだけでは？
```

Linuxから設定された値をハイパーバイザで表示させると

- Message Address: 0x12030040
- Message Data: 0x0001

特に問題はないような気がするが..？

# LinuxでMSIを設定している箇所を見る

Linux v5.4: drivers/irqchip/irq-gic-v3-its.c の its\_irq\_compose\_msi\_msg っぽい

```
static void its_irq_compose_msi_msg(struct irq_data *d, struct msi_msg *msg)
{
    struct its_device *its_dev = irq_data_get_irq_chip_data(d);
    struct its_node *its;
    u64 addr;

    its = its_dev->its;
    addr = its->get_msi_base(its_dev);

    msg->address_lo      = lower_32_bits(addr);
    msg->address_hi      = upper_32_bits(addr);
    msg->data            = its_get_event_id(d);

    iommu_dma_compose_msi_msg(irq_data_get_msi_desc(d), msg);
}
```

## pr\_infoを入れ各 IRQに対してどう設定されているか見る

今回はIOMMUをOFFにしているので無視

```
static void its_irq_compose_msi_msg(struct irq_data *d, struct msi_msg *msg)
{
    struct its_device *its_dev = irq_data_get_irq_chip_data(d);
    struct its_node *its;
    u64 addr;

    its = its_dev->its;
    addr = its->get_msi_base(its_dev);
    pr_info("%s: IRQ#%u(HWIRQ %lu): 0x%llx: 0x%x\n", __func__, d->irq, d->hwirq, addr, its_get_event_id(d));
    msg->address_lo = lower_32_bits(addr);
    msg->address_hi = upper_32_bits(addr);
    msg->data = its_get_event_id(d);

    iommu_dma_compose_msi_msg(irq_data_get_msi_desc(d), msg);
}
```

## 動かしてみた結果

IRQが違うのに対して、Message Addressの値は一定、Message DataはEventIDで重複しまくり

```
[ 2.839588] its_irq_compose_msi_msg: IRQ#59(HWIRQ 8192): 0x12030040: 0x0
[ 2.862667] its_irq_compose_msi_msg: IRQ#61(HWIRQ 8194): 0x12030040: 0x2
[ 2.875136] its_irq_compose_msi_msg: IRQ#60(HWIRQ 8193): 0x12030040: 0x1
[ 2.898584] its_irq_compose_msi_msg: IRQ#62(HWIRQ 8224): 0x12030040: 0x0
[ 2.919547] its_irq_compose_msi_msg: IRQ#63(HWIRQ 8225): 0x12030040: 0x0
[ 2.936848] its_irq_compose_msi_msg: IRQ#64(HWIRQ 8226): 0x12030040: 0x1
[ 3.020571] its_irq_compose_msi_msg: IRQ#66(HWIRQ 8227): 0x12030040: 0x0
[ 3.027268] its_irq_compose_msi_msg: IRQ#67(HWIRQ 8228): 0x12030040: 0x1
[ 3.033962] its_irq_compose_msi_msg: IRQ#68(HWIRQ 8229): 0x12030040: 0x2
```



# 0x12030040は一体何者？

メモリマップと照らし合わせて確認すると、GITS\_TRANSLATERというレジスタらしい  
**GITS\_TRANSLATER, ITS Translation Register**

The GITS\_TRANSLATER characteristics are:

## **Purpose**

Written by a requesting a Device to signal an interrupt for translation by the ITS.

## **Usage constraints**

16-bit access to bits [15:0] of this register must be supported. When this register is written by a 16-bit transaction, bits [31:16] are written as zero.

Implementations must ensure that:

- A unique DeviceID is provided for each requesting device, and the DeviceID is presented to the ITS when a write to this register occurs in a manner that cannot be spoofed by any agent capable of performing writes.
- The DeviceID presented corresponds to the DeviceID field in the ITS commands.



# GITS\_TRANSLATERの実装

## GITS\_TRANSLATER, ITS Translation Register

The GITS\_TRANSLATER characteristics are:

### Purpose

Written by a requesting a Device to signal an interrupt for translation by the ITS.

### Usage constraints

16-bit access to bits [15:0] of this register must be supported. When this register is written by a 16-bit transaction, bits [31:16] are written as zero.

Implementations must ensure that:

- A unique DeviceID is provided for each requesting device, and the DeviceID is presented to the ITS when a write to this register occurs in a manner that cannot be spoofed by any agent capable of performing writes.
- The DeviceID presented corresponds to the DeviceID field in the ITS commands.



## GITS\_TRANSLATERに書き込むときは ..?

Implementations must ensure that:

- A unique DeviceID is provided for each requesting device, and the **DeviceID is presented to the ITS when a write to this register occurs** in a manner that **cannot be spoofed** by any agent capable of performing writes.

意識:

実装は以下を保証しなければならない

- このレジスタに書き込みが起きる際は、前もってDeviceIDは詐称できない形で送信される





# つまり？

GITS\_TRANSLATERは**誰が書き込んだか**で挙動が変わる

そして、**書き込み元は詐称できない**

つまり、**CPUから書き込んでも意味がない**

---

ああ、オワタァ ...

## なんとか割り込みを起したい

GICv3の仕様書を血眼になって探していると、ITSのCommand Queueに以下のコマンドを発見

### 6.3.5 INT

This command translates the event defined by EventID and DeviceID into an ICID and pINTID, and instructs the appropriate Redistributor to set the interrupt pending.

Figure 6-8 on page 6-114 shows the format of the INT command.

63				32	31			8	7	0	DW
DeviceID					RES0				0x03		0
RES0					EventID						1
RES0											2
RES0											3

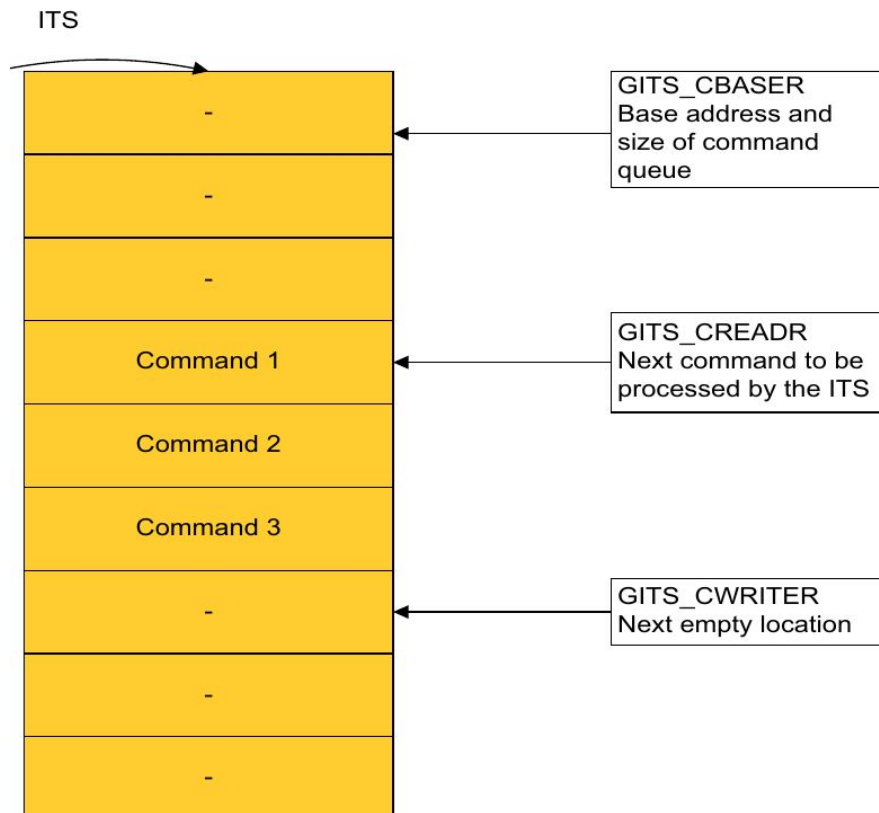
Figure 6-8 INT command format

# ITS Command Queueに INT Commandを投げれば!!

ITS Command Queueにコマンドを投げる方法

1. GITS\_CBASERからキューのアドレスを取得  
(キューは今回はゲストOSが設定済み)
2. GITS\_CWRITERから空いてるエントリを取得
3. INTコマンドにDeviceIDとEventIDをセット  
して、エントリに書き込む
4. GITS\_CWRITERをインクリメント

ゲストOSと競合しないようにする必要あり



# 早速実装

ハイパーバイザで、MSI Message AddressがGITS\_TRANSLATERだった場合、Command Queueに書き込むように実装

動いた!! 🎉🎉🎉

```
$cat /proc/interrupts
```

	CPU0	CPU1	CPU2	CPU3	CPU4	CPU5	CPU6		
66:	0	0	0	0	0	0	0	ITS-MSI 16384 Edge	virtio0-config
67:	3	0	0	0	1	0	0	ITS-MSI 16385 Edge	virtio0-input.0
68:	26	0	0	0	0	0	0	ITS-MSI 16386 Edge	virtio0-output.0



## まとめ

- Armの割り込みコントローラとしてGICv3がある
- ArmのMSI割り込みではInterrupt Translation Service(ITS)がある
- デバイスから割り込みを起こす際はGITS\_TRANSLATERにEventIDを書き込む
- GITS\_TRANSLATERは書き込んだデバイスからDeviceIDを取得する
  - CPUが書き込んでも意味がない
- ITS Command QueueにINTコマンドを入れれば割り込みは起こせる



## おわりに

- ArmはAArch64の命令アーキテクチャは綺麗で使いやすいけど、キャッシュラインや周辺デバイスの実装は拡張性を持たせているので扱いが難しい
- 実際のデバイスに寄生する形のハイパーバイザの仮想デバイス実装は大変
- Armの低レイヤーの情報は案外少ないのでLinux Kernelを読む機会が多かった

MilvusVisorでは皆様のコントリビューションをお待ちしております!