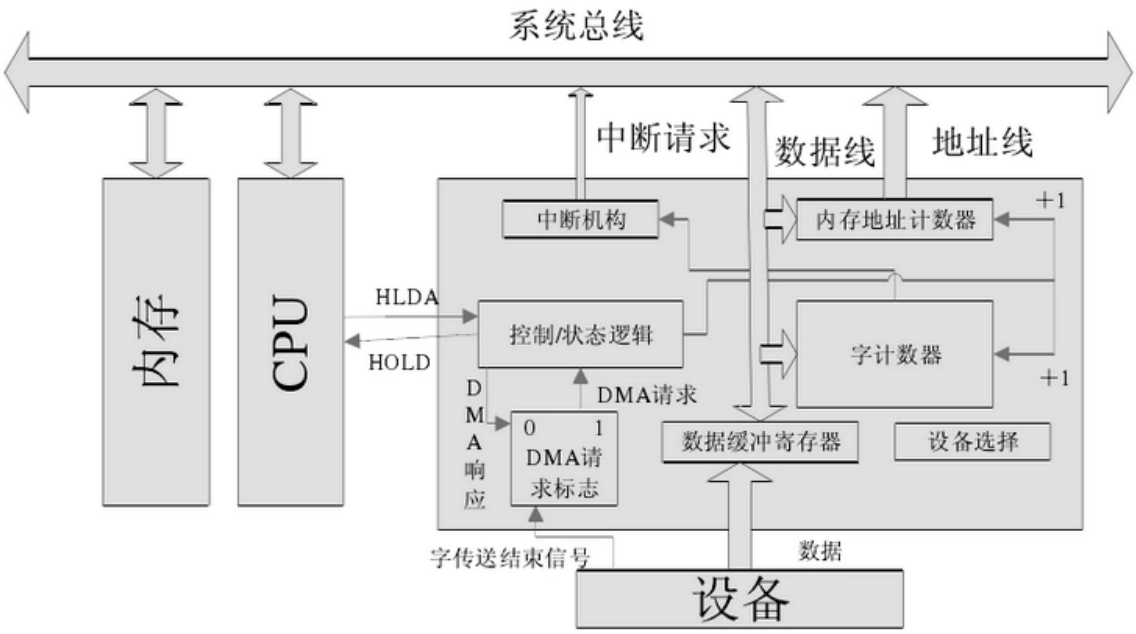
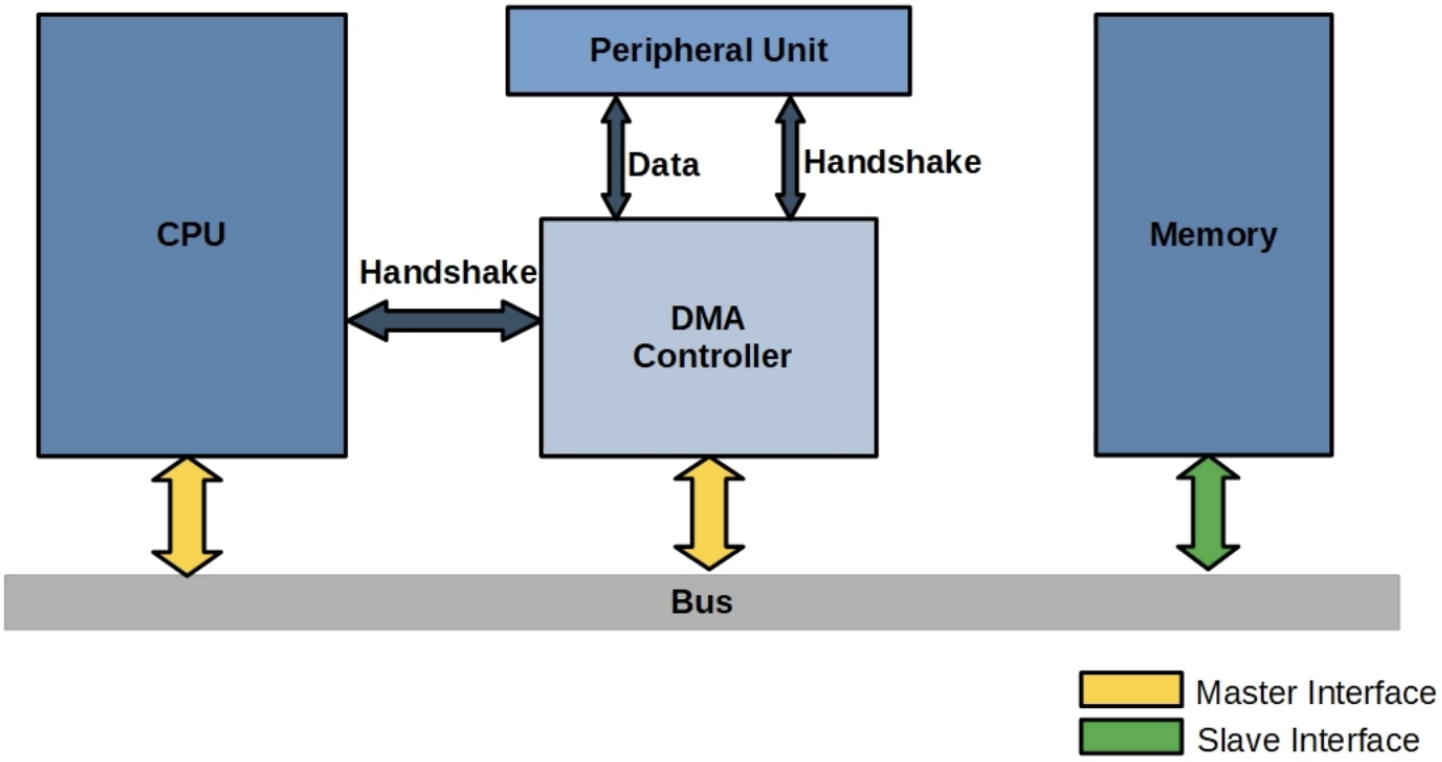
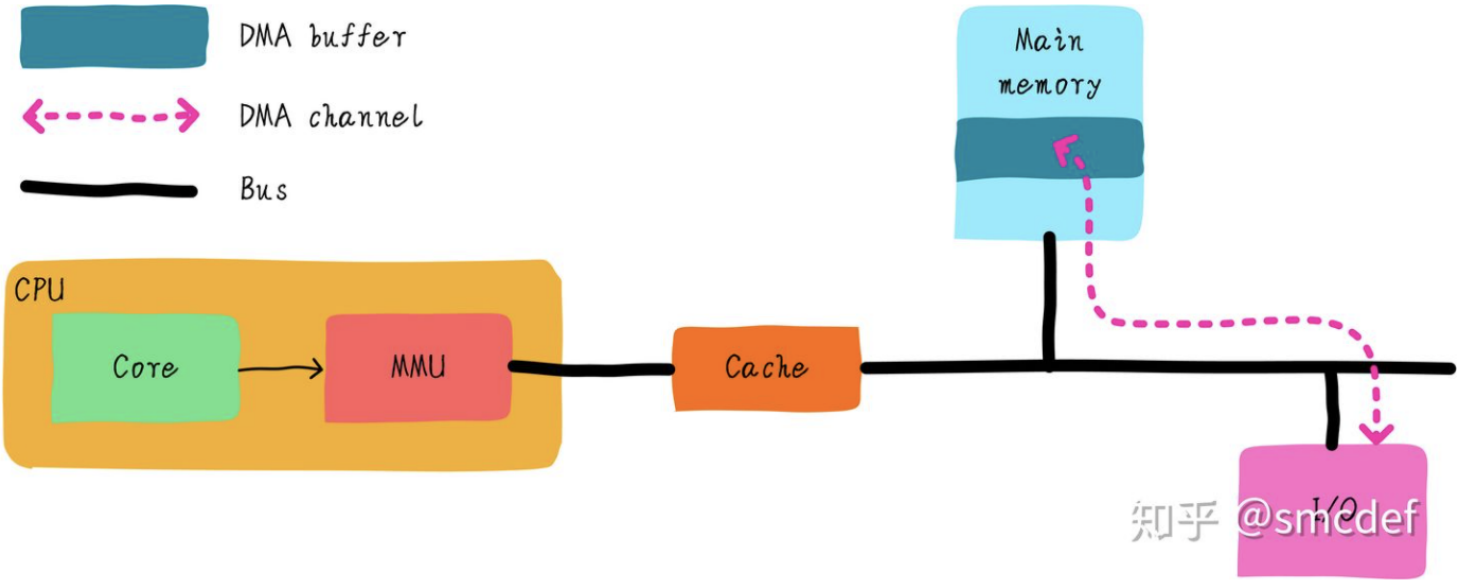
DMA

- ref
  - https://en.wikipedia.org/wiki/Direct_memory_access#Third-party && https://members.tripod.com/~Eagle_Planet/dma__direct_memory_access.html
    - 总结概括非常好，在这里首次得知了**first-part DMA**与**third-prat DMA**的概念
  - https://elinux.org/images/6/65/An_Overview_of_the_Kernel_DMAEngine_Subsystem.pdf
  - https://scholarworks.calstate.edu/downloads/vm40xv787
    - 一个硕士关于DMA的毕业论文，有时间可以看看

1. DMA介绍（重点是 Third-Party and First-Party）







- As you know, the processor is the **brain** of the machine, and in many ways it can also be likened to the **conductor of an orchestra（乐队的编曲）**. In early machines the processor really **did almost everything**. In addition to running programs it was **also responsible for transferring data** to and from peripherals. Unfortunately, having the processor perform these transfers is very **inefficient**, because it then is **unable to do anything else**
- The **invention of DMA** enabled the devices to **cut out the middle man**, allowing the processor to do other work and the peripherals to transfer data themselves, leading to increased performance. Special channels were created, along with circuitry to control them, that allowed the transfer of information without the processor controlling every aspect of the transfer. **This circuitry is normally part of the system chipset on the motherboard**
  - **这个是老的，ISA才这样弄，PCIe完全是另一套**
- Direct memory access (DMA) is a **feature** of computer systems that **allows certain hardware subsystems to access main system memory** (random-access memory) independently of the central processing unit (CPU)

- **Many hardware systems use DMA**, including **disk drive controllers**, graphics cards, network cards and sound cards
  - **这个use在这里应该只能理解为利用DMA机制**
  - 负责**内存**与**外设**之间的**数据传输**，**解放CPU**
- DMA can also be used for **memory to memory** copying or moving of data within memory. DMA can offload expensive memory operations, such as large copies or scatter-gather operations, **from the CPU to a dedicated DMA engine**
  - An implementation example is the **I/O Acceleration Technology**
  - **这类DMA引擎是可以通过lspci看到的，Intel的IOAT加速技术**

1.1 Third-Party and First-Party DMA (Bus Mastering)

- **DMA技术**也是在不断发展的，**自己曾今对DMA控制器的理解就是host chipset上一部分电路**，所以PCIe DMA这部分始终无法理解，最终才发现DMA也是分很多种的，**PCIe设备均有DMA引擎，且PCIe设备不会使用chipset上的DMA引擎**
- 对**第一方DMA**理解的关键在于**以下ref**，**PCIe设备**使用**自身的DMA引擎**，因为DMA实际上是**接口电路**，位于**host**与**设备**都很合理
  - http://news.eeworld.com.cn/qrs/2012/1015/article_12304.html —— **ISA总线的DMA技术**
    1. DMA技术产生时**正是ISA总线在PC中流行**的时侯，ISA我貌似是没有见过
    2. 这种**DMA机制**也称为**标准DMA（standard DMA）**
       - 标准DMA有时也称为**第三方DMA（third-party DMA）**
       - 这是因为系统DMA完成实际的传输过程，所以它相对于传输过程的**前两方**(**传输的发送者和接收者**)来说是**第三方**
    3. **标准DMA技术**主要有**两个缺点**
       - DMA的数据传输速度太慢，**不能与更高速的总线（如PCIe）配合使用**
       - 两个DMA一起只提供了8个DMA通道，这也成为了限制系统I/O吞吐率提升的瓶颈
    4. 鉴于上述两个原因，**PCI总线体系结构设计一种成为第一方DMA（first-party DMA）的DMA机制，也称为 Bus Mastering（总线主控）**。在这种情况下，进行传输的**PCI卡必须取得系统总线的主控权后才能进行数据传输**。实际的传输也不借助慢速的ISA DMA来进行，而是由**内嵌在PCI卡中的DMA电路**（比传统的ISA DMA要快）来完成。Bus Mastering方式的DMA可以让PCI外设得到它们想要的传输带宽，因此它比标准DMA功能满足现代高性能外设的要求
    5. ISA卡几乎不使用Bus Mastering模式的DMA
       - **而PCI只使用Bus Mastering模式的DMA，它从不使用标准DMA**
         - **这也是我一直没有找到PCIe系统下DMA相关代码的原因**，就是没有找对方向
    6. 之后在重点介绍**标准DMA**，不予介绍
  - https://www.cnblogs.com/chengqi521/p/8512510.html —— **PCIe DMA/XDMA**
    1. 在xilinx中生成IP核后，工程文件夹下会有这两个文件夹
       - `[Xilinx_PCIe_BMD]` xilinx FPGA 开发**pcie BMD DMA**的verilog HDL源码
         - **印证了PCIe设备上是有DMA接口电路的**
       - `[example_design]` xilinx pcie总线**pio模式**下的控制器代码。包含接收发送模块，存储模块，控制模块等
         - 处理器IO，就是利用处理器完成IO
    2. DMA读过程
       1. 驱动程序向操作系统申请一片**物理连续的内存**（PCIe初始化的时候做的）
       2. **主机向该地址写入数据**
       3. 通过**写寄存器**，**主机将这个内存的物理地址告诉FPGA DMA**，
       4. FPGA中的DMA开始从之前分配的物理内存中读数据，**直到完成**
       5. DMA完成IO后中断CPU
    3. DMA写过程如下
       1. 驱动程序向操作系统申请一片物理连续的内存
       2. 主机将这个内存的物理地址告诉FPGA的DMA
       3. FPGA的DMA向主机发起写TLP请求，并将数据放入TLP包中—连续发出多个写请求
          - **其实就是DMA写内存**
       4. FPGA DMA发送完数据后通过**中断**等形式通知主机DMA完成
       5. 主机从内存中获取数据

4. tips
    1. 内存一定要**物理连续**。因为DMA使用的是**物理地址**，没有页表参与
    2. PCIe的BAR空间中的寄存器有DMA的寄存器，如**源/目的**地址，数据位宽，长度等等

1.1.1 Third-Party —— slow and old

- The DMA controller, **built into the system chipset on modern PCs**, **manages standard DMA transfers**。**Standard DMA** is sometimes called **third party DMA**. This refers to the fact that the **system DMA controller is actually doing the transfer**
    - the first two parties are the **sender(host)** and **receiver(device)** of the transfer
- **Third-party DMA** utilizes a **system DMA engine resident on the main system board**, which has several DMA channels available for use by devices. The device relies on the system's DMA engine to perform the data transfers between the device and memory. The **driver** uses DMA engine routines to initialize and program the DMA engine. For each DMA data transfer, the **driver programs the DMA engine** and then gives the device a command to initiate the transfer in cooperation with that engine

```
# ls -1 /sys/class/dma/
dma0chan0  dma0chan1  dma1chan0  dma1chan1
# cat /proc/dma
4: cascade
```

- 一个**chipset**可以包含**多个DMA控制器（2个）**。**每个控制器**有**多个DMA通道（2个）**

1.1.2 First-Party DMA (Bus Mastering)

- There is also a type of DMA called **first party DMA**. In this situation, the peripheral doing the transfer actually takes control of the system bus to perform the transfer. This is also called **bus mastering**

2. DMA Test Guide

- ref
    - https://www.kernel.org/doc/html/v4.15/driver-api/dmaengine/dmatest.html

3. linux内核DMA控制器的发现与使能（Third-Party）—— 尽管已近被时代抛弃了

- ref
    - http://m.blog.chinaunix.net/uid-69947851-id-5828920.html —— **linux DMA框架**
        - 在Linux当中有一个**专门处理DMA的框架**，叫做**dmaengine**，它的代码实现在 `drivers/dma/dmaengine.c` 。这个文件主要是提供一套DMA使用的抽象层，但是封装的也比较简单。从使用上来讲，通常我们让DMA工作，大概都是5步，我叫做DMA 5步曲。是哪5步呢?
            1. **dma通道请求**，对应Linux API， `chan = dma_request_channel(....)`
                - 从**dma_device_list**上找到一个**合适的dma控制器**，并从控制器上获取**一个dma channel**
            2. dma通道配置，对应Linux API， `dmaengine_slave_config (chan....)`
                - 用户通过该函数，可以配置**指定通道的参数**，比如**目的**和**源地址**，**位宽**，**传输方向**等
                    - 其实这些**寄存器确定**之后就可以**发起DMA了**
            3. dma通道预处理，对应Linux的API， `dmaengine_prep_*`
                - 这个预处理函数会比较多，因为DMA支持很多**不同类型的处理**，比如**DEVTODEV**，**MEMTOMEM**，**MEMTODEV**，**DEVTOMEM**等。但是都会以dmaengine_prep开头，这个API返回做好预处理的DMA TX结构，用于第4步处理
            4. dma数据提交，对应Linux API， `dmaengine_submit()`

        - **主要是将DMA处理事务提交到dma通道处理链上**，这个submit用的是第四步得到的DMA TX结构
    5. dma数据处理，用于启动一次事务处理，对应Linux API， `dma_async_issue_pending()`
        - 这个函数和其他几个函数一样，就是调用dma_device的对应回调，这里给出原型。直接回调dma控制器的devie_issue_pending函数
    - 以上这些所谓的dmaengine框架提供的DMA函数，**其实都是间接调用了封装在struct dma_device里面的回调函数**
        - **即具体的功能是由具体的dma引擎决定的**
- http://www.wowotech.net/linux_kenrel/dma_engine_overview.html —— **Linux DMA Engine framework**
    - **DMA channels**
        - DMA channels是DMA controller为了方便，**抽象**出来的概念，让consumer以为独占了一个channel，**实际上所有channel的DMA传输请求都会在DMA controller中进行仲裁，进而串行传输**
    - **DMA request lines**
        - DMA channel是Provider（**提供传输服务**），DMA request line是Consumer（**消费传输服务**）。在一个系统中**DMA request line的数量通常比DMA channel的数量多**
            - 因为并不是每个request line在**每一时刻**都需要传输
    - **传输参数**
        - **transfer size:传输大小**
        - **transfer width:每一次传输的大小，一次一个太慢了**
    - **scatter-gather** —— 简称**SG**
        - 一般情况下，DMA传输一般只能处理在物理上连续的buffer。但在有些场景下，我们需要将一些非连续的buffer拷贝到一个连续buffer中（这样的操作称作**scatter gather**）
    - **master-slave**
        - **DMA中的slave指的是参与DMA传输的设备**
        - **而对应的master就是指DMA controller自身**
    - kernel的**dmaengine框架**就**薄薄的一层**
- 描述DMA的数据结构是**struct dma_device**，所有的DMA设备位于系统**全局list中**

```
# 先说下founder机器上的dma位于哪里
/sys/devices/pci0000:00/0000:00:15.0/dma/dma0chan0
/sys/devices/pci0000:00/0000:00:15.0/dma/dma0chan1
/sys/devices/pci0000:00/0000:00:15.1/dma/dma1chan0
/sys/devices/pci0000:00/0000:00:15.1/dma/dma1chan1
# lspci 可以发现对应的PCIe是两个串口
00:15.0 Serial bus controller [0c80]: Intel Corporation Device 43e8 (rev 11)
00:15.1 Serial bus controller [0c80]: Intel Corporation Device 43e9 (rev 11)
```

- 先学习两个名词概念
    - **MFD:Multifunction device**
    - **LPSS:Low Power Subsystem**

```
# ACPI 模式暂不考虑
config MFD_INTEL_LPSS_ACPI
        tristate "Intel Low Power Subsystem support in ACPI mode"
        select MFD_INTEL_LPSS
        depends on X86 && ACPI
        help
          This driver supports Intel Low Power Subsystem (LPSS) devices such as
          I2C, SPI and HS-UART starting from Intel Sunrisepoint (Intel Skylake
          PCH) in ACPI mode.

config MFD_INTEL_LPSS_PCI
        tristate "Intel Low Power Subsystem support in PCI mode"
        select MFD_INTEL_LPSS
        depends on X86 && PCI
        help
          This driver supports Intel Low Power Subsystem (LPSS) devices such as
          I2C, SPI and HS-UART starting from Intel Sunrisepoint (Intel Skylake
          PCH) in PCI mode.
# SPI means Serial Peripheral Interface
```

```
// 这些物理地址怎么看起来是固定的
#define LPSS_IDMA64_OFFSET      0x800   //2K，DMA在LPSS中的偏移
#define LPSS_IDMA64_SIZE        0x800   //2K
static const struct resource intel_lpss_idma64_resources[] = {
        DEFINE_RES_MEM(LPSS_IDMA64_OFFSET, LPSS_IDMA64_SIZE),
        DEFINE_RES_IRQ(0),
};
// ...
static const struct mfd_cell intel_lpss_idma64_cell = { // intel DMA
        .name = LPSS_IDMA64_DRIVER_NAME,
        .num_resources = ARRAY_SIZE(intel_lpss_idma64_resources),
        .resources = intel_lpss_idma64_resources,
};
// ...
static const struct mfd_cell intel_lpss_spi_cell = {  // 串口
        .name = "pxa2xx-spi",
        .num_resources = ARRAY_SIZE(intel_lpss_dev_resources),
        .resources = intel_lpss_dev_resources,
};
```

- 明确一点，**intel的DMA位于lpss子系统下，lpss可以按照PCIe模式来初始化**

```
// intel_lpss_pci_driver 相当于就是一个串口的驱动
static struct pci_driver intel_lpss_pci_driver = {
        .name = "intel-lpss",
        .id_table = intel_lpss_pci_ids,
        .probe = intel_lpss_pci_probe,
        .remove = intel_lpss_pci_remove,
        .driver = {
                .pm = &intel_lpss_pci_pm_ops,
        },
};
```

- `intel_lpss_pci_probe`
  - struct intel_lpss_platform_info *info;
  - `ret = pcim_enable_device(pdev);`
  - `info = devm_kmemdup(&pdev->dev, (void *)id->driver_data, sizeof(*info), GFP_KERNEL);`
    - `info->mem = &pdev->resource[0];`
      - 后续用到的**mem_base**其实就是lpss的**bar0的基址**
  - `ret = intel_lpss_probe(&pdev->dev, info);`

- `if (intel_lpss_has_idma(lpss))`
  - `ret = mfd_add_devices(dev, lpss->devid, &intel_lpss_idma64_cell, 1, info->mem, info->irq, NULL);` —— **register child devices**
    - `mfd_add_device` —— **简单理解，DMA控制器添加到了platform总线上，在自己的例子中DMA控制器的数量为2**
      - `pdev = platform_device_alloc(cell->name, platform_id);`
        - **这个pdev是DMA控制器，因为这个cell是intel_lpss_idma64_cell**
      - `pdev->dev.parent = parent;`
        - **父设备是lpss**
      - `pdev->dev.type = &mfd_dev_type;`
      - `pdev->dev.dma_mask = parent->dma_mask;` —— 学习一下**dma_mask**
        - DMA能够寻址的物理内存是**有限制的**，**并非64位物理内存均可寻址**，但是PCIe规范下基本认为就是**64位全部可寻址的**
          - 如果设备有DMA寻址的限制，那么驱动需要将这个限制通知到内核。如果驱动不通知内核，那么内核缺省情况下认为外设的DMA可以访问所有的系统总线的32 bit地址线。对于64 bit平台，情况类似，不再赘述
          - 是否有DMA寻址限制是和硬件设计相关，有时候标准总线协议也会规定这一点。例如
            - **PCI-X规范规定，所有的PCI-X设备必须要支持64 bit的寻址**
        - In order to communicate the maximum addressing width, every generic device has a parameter, called the DMA mask, that contains a map of set bits corresponding to the accessible address lines that **must be set up by the device driver**
          - `u64 *dma_mask; /* dma mask (if dma'able device) */`
            - **struct device 中的一个字段**
          - **能够DMA的物理内存有限制**
        - The **dma_mask** represents a bit mask of the **addressable region for the device**. I.e., if the physical address of the memory anded with the dma_mask is still equal to the physical address, then the device can perform DMA to the memory
          - On some ARM systems, memory does not start at a physical address of zero; the physical address of the first byte can be as high as 3GB (**0xc0000000**). If a system configured in this way has a device with a **26-bit address limitation** (with the upper bits being filled in by the bus hardware), then its DMA mask should be set to **0xc3ffffff**. Any physical address within the device's range will be **unchanged** by a logical AND operation with this mask, while any address outside of that range will not
            - dma_mask在一定范围内均为1意味着任何在这个范围内的地址做**按位与**都不会变，但是dma_mask=0的位置就会改变，意味着**不能DMA**
      - `pdev->dev.dma_parms = parent->dma_parms;`
      - `pdev->dev.coherent_dma_mask = parent->coherent_dma_mask;`
        - **基本同上**
      - `for (r = 0; r < cell->num_resources; r++) {`
        - `res[r].name = cell->resources[r].name;`
        - `res[r].flags = cell->resources[r].flags;`
        - `if ((cell->resources[r].flags & IORESOURCE_MEM) && mem_base) {`
          - `res[r].start = mem_base->start + cell->resources[r].start;`
            - **DMA寄存器在PCIe域中原来是这个意思**
            - **mem_base->start**是lpss（**PCIe设备**）的**bar0的基址**
          - `res[r].end = mem_base->start + cell->resources[r].end;`
        - `}`
      - `}`
      - `ret = platform_device_add_resources(pdev, res, cell->num_resources);`

        - pdev->resource = r;
        - pdev->num_resources = num;
    - ret = platform_device_add(pdev);
        - **Add a platform device to device hierarchy**

```
static struct platform_driver idma64_platform_driver = {
        .probe          = idma64_platform_probe,
        .remove         = idma64_platform_remove,
        .driver = {
                .name   = LPSS_IDMA64_DRIVER_NAME,
                .pm     = &idma64_dev_pm_ops,
        },
};
// ...
module_platform_driver(idma64_platform_driver);
// ...
struct idma64_chip {
        struct device   *dev;
        struct device   *sysdev;
        int             irq;
        void __iomem    *regs;  // DMA寄存器基址，iomem space
        struct idma64   *idma64;
};
//
struct idma64 {
        struct dma_device dma;
        void __iomem *regs;
        /* channels */
        unsigned short all_chan_mask;
        struct idma64_chan *chan;
};
// DMA一个channel的寄存器 memmap IO space
/* Channel registers */
#define IDMA64_CH_SAR           0x00    /* Source Address Register */
#define IDMA64_CH_DAR           0x08    /* Destination Address Register */
#define IDMA64_CH_LLP           0x10    /* Linked List Pointer */
#define IDMA64_CH_CTL_LO        0x18    /* Control Register Low */
#define IDMA64_CH_CTL_HI        0x1c    /* Control Register High */
#define IDMA64_CH_SSTAT         0x20
#define IDMA64_CH_DSTAT         0x28
#define IDMA64_CH_SSTATAR       0x30
#define IDMA64_CH_DSTATAR       0x38
#define IDMA64_CH_CFG_LO        0x40    /* Configuration Register Low */
#define IDMA64_CH_CFG_HI        0x44    /* Configuration Register High */
#define IDMA64_CH_SGR           0x48
#define IDMA64_CH_DSR           0x50
#define IDMA64_CH_LENGTH        0x58    // 一个channel寄存器的长度
```

- static int idma64_platform_probe(struct platform_device *pdev)
    - struct idma64_chip *chip;
    - chip = devm_kzalloc(dev, sizeof(*chip), GFP_KERNEL);
    - mem = platform_get_resource(pdev, IORESOURCE_MEM, 0);
        - **mem还是物理地址，这个是DMA的bar0基址**
    - chip->regs = devm_ioremap_resource(dev, mem);
        - **将DMA的寄存器物理地址map到VM中**，regs是DMA寄存器的**基址**
    - ret = idma64_probe(chip);
        - struct idma64 *idma64;
        - unsigned short nr_chan = IDMA64_NR_CHAN; —— **#define IDMA64_NR_CHAN 2**
        - idma64 = devm_kzalloc(chip->dev, sizeof(*idma64), GFP_KERNEL);

    - DMA控制器对象，包含**关键数据结构 struct dma_device**，这个是**DMA的关键核心数据结构**
- `idma64->regs = chip->regs;`
    - **寄存器还是它**
- `chip->idma64 = idma64;`
- `idma64->chan = devm_kcalloc(chip->dev, nr_chan, sizeof(*idma64->chan), GFP_KERNEL);`
    - 为DMA分配通道，**nr_chan代表通道个数**
- `idma64->all_chan_mask = (1 << nr_chan) - 1;`
- `idma64_off(idma64);`
- `for (i = 0; i < nr_chan; i++) {`
    - `struct idma64_chan *idma64c = &idma64->chan[i];`
    - `idma64c->vchan.desc_free = idma64_vdesc_free;`
    - `vchan_init(&idma64c->vchan, &idma64->dma);`
    - `idma64c->regs = idma64->regs + i * IDMA64_CH_LENGTH;`
        - 一个DMA控制器的**多个channel的寄存器**是**并排的**
    - `idma64c->mask = BIT(i);`
- `}`
- `idma64->dma.device_alloc_chan_resources = idma64_alloc_chan_resources;`
- **...**
- **DMA设备的一顿初始化**
- **...**
- `ret = dma_async_device_register(&idma64->dma);` —— **registers DMA devices found，该代码位于drivers/dma/dmaengine.c这是一个dma框架（因为DMA控制器有很多，DMA驱动也有很多），这里是将DMA注册到系统中，全局有一个list，描述了系统中的所有dma_device**
    - `rc = get_dma_id(device);`
    - `ida_init(&device->chan_ida);`
    - `list_for_each_entry(chan, &device->channels, device_node) {` —— **represent channels in sysfs. Probably want devs too**
        - `rc = __dma_async_device_channel_register(device, chan);`
            - `chan->chan_id = ida_alloc(&device->chan_ida, GFP_KERNEL);`
            - `chan->dev->device.class = &dma_devclass;`
            - `chan->dev->device.parent = device->dev;`
            - `chan->dev->chan = chan;`
            - `chan->dev->dev_id = device->dev_id;`
            - `dev_set_name(&chan->dev->device, "dma%dchan%d", device->dev_id, chan->chan_id);`
                - **sys文件系统下有所表现**
            - `rc = device_register(&chan->dev->device);`
            - `chan->client_count = 0;`
            - `device->chancnt++;`
    - `}`
    - `list_add_tail_rcu(&device->global_node, &dma_device_list);`
    - `dma_channel_rebalance();`
- `dev_info(chip->dev, "Found Intel integrated DMA 64-bit\n");`

4. PCIe DMA（first-Party）

```
[ddhao@zc-r750 iommu_groups]$ lspci -s 00:11.5 -vvv
00:11.5 SATA controller: Intel Corporation C620 Series Chipset Family SSATA Controller [AHCI mode] (rev 0a) (prog-if 01 [AHC
        Subsystem: Dell Device 090e
        Control: I/O+ Mem+ BusMaster+ SpecCycle- MemWINV- VGASnoop- ParErr+ Stepping- SERR+ FastB2B- DisINTx+
        Status: Cap+ 66MHz+ UDF- FastB2B+ ParErr- DEVSEL=medium >TAbort- <TAbort- <MAbort- >SERR- <PERR- INTx-
        Latency: 0
        Interrupt: pin A routed to IRQ 256
        NUMA node: 0
        Region 0: Memory at 92dfa000 (32-bit, non-prefetchable) [size=8K]
        Region 1: Memory at 92e05000 (32-bit, non-prefetchable) [size=256]
        Region 2: I/O ports at 2068 [size=8]
        Region 3: I/O ports at 2074 [size=4]
        Region 4: I/O ports at 2040 [size=32]
        Region 5: Memory at 92b80000 (32-bit, non-prefetchable) [size=512K]
        Capabilities: <access denied>
        Kernel driver in use: ahci
        Kernel modules: ahci
```

- **详见io.md**
- lspci看到的SATA控制器，**其中BusMaster+就表明它的DMA是first-part的**

5. Linux kernel scatterlist API介绍

- ref
  - http://www.wowotech.net/memory_management/scatterlist.html
    - **写的真牛逼易懂，下面就是该ref的关键提取**
- 假设在一个系统中有**三个模块可以访问memory。CPU、DMA控制器和某个外设**
  - CPU通过MMU以虚拟地址（VA）的形式访问memory
  - DMA直接以物理地址（PA）的形式访问memory
  - Device通过自己的IOMMU以设备地址（DA）的形式访问memory
- 然后，某个**软件实体**分配并使用了一片存储空间。该存储空间在CPU视角上（虚拟空间）是连续的，起始地址是va1（实际上，它映射到了3块不连续的物理内存上，我们以pa1,pa2,pa3表示）
  - 那么，如果该软件单纯的以CPU视角访问这块空间（操作va1），则完全没有问题，因为MMU实现了连续VA到非连续PA的映射
  - 不过，如果软件经过一系列操作后，**要把该存储空间交给DMA控制器，最终由DMA控制器将其中的数据搬移给某个外设的时候，由于DMA控制器只能访问物理地址，只能以 不连续的物理内存块 为单位递交**
    - 而不是我们所熟悉的虚拟地址
- **此时，scatterlist就诞生了**
  - 为了方便，我们需要使用一个数据结构来描述这一个个**不连续的物理内存块**（起始地址、长度等信息），这个数据结构就是scatterlis
    - 而多个scatterlist组合在一起形成一个表（可以是一个struct scatterlist类型的数组，也可以是kernel帮忙抽象出来的struct sg_table），**就可以完整的描述这个虚拟地址了**

6. DMA映射的3种方式（其实说的都是kernel提供的API，本质都是分配一片内存，区别如下）

- ref
  - https://blog.csdn.net/tangtang_yue/article/details/50635451
    - 一致DMA映射

        - 使用 `dma_alloc_coherent()`，如果不需要大块的内存（**小于一个page**），**则使用poll来分配**
    - 流映射需要自己flush缓存
    - sg，详见ahci部分

7. DMA tips

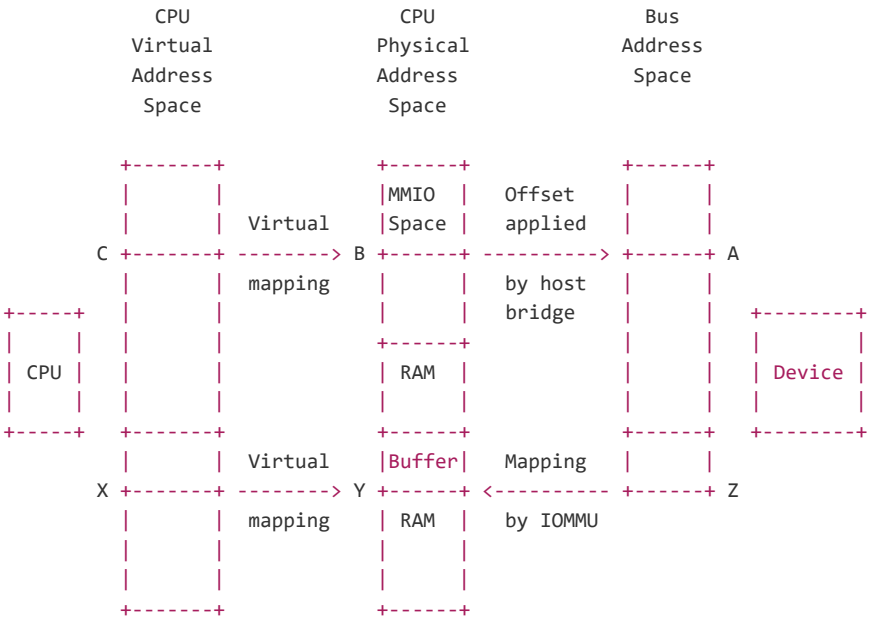
- ref
    - https://breezetemple.github.io/2018/12/19/linux-kernel-dma-cache-coherence/
        1. 存在**DMA ZONE**的原因是某些硬件的DMA引擎**不能访问** 到所有的内存区域，因此，加上一个 **DMA ZONE**，当使用 **GFP_DMA** 方式申请内存时，获得的内存限制在**DMA ZONE**的范围内，这些特定的硬件需要使用**GFP_DMA**方式获得可以做DMA的内存
            - 如果系统中所有的设备都可选址所有的内存，那么**DMA ZONE**覆盖所有内存。**现在几乎都可以**
        2. **DMA ZONE**的内存**做什么都可以**
        3. **dma_mask**与**coherent_dma_mask**这两个参数表示**它能寻址的物理地址的范围**，内核通过这两个参数分配**合适的物理内存**给**device**
            - **dma_mask是设备DMA能访问的内存范围**
                - `dma_set_mask(&pdev->dev, DMA_BIT_MASK(64))`
            - **coherent_dma_mask则作用于申请一致性DMA缓冲区**
                - `dma_set_coherent_mask(&pdev->dev, DMA_BIT_MASK(64))`
    - https://blog.csdn.net/21cnbao/article/details/79133658
        1. **dma_alloc_coherent()申请的内存来自DMA ZONE**
            - **mask覆盖了整个4GB，64位同理**
        2. **dma_alloc_coherent()申请的内存是非cache**的吗
            - 大多数是uncache的，**但是现在技术有带cache的**
        3. **dma_alloc_coherent()申请的内存一定是物理连续**的吗
            - 没有IOMMU，物理地址必须是物理连续的
            - 有IOMMU之后就可以不连续了
    - http://www.wowotech.net/memory_management/DMA-Mapping-api.html
        - **DMA寻址限制以及mask等概念的清晰解析** —— wowo科技
    - https://elinux.org/images/3/32/Pinchart--mastering_the_dma_and_iommu_apis.pdf
        - DMA接口函数的一些使用，**主要就看了dma_alloc_coherent**
            - **实现详见IO部分**
        - `void *dma_alloc_coherent(struct device *dev, size_t size, dma_addr_t *dma_handle, gfp_t flag);`
            - This routine allocates **a region of @size bytes of coherent memory**. It also returns a @dma_handle which may be cast to an unsigned integer the same width as the bus and **used as the device address base of the region**
            - Returns: a pointer to the allocated region (in the processor's virtual address space) or NULL if the allocation failed
            - Note: coherent memory can be **expensive** on some platforms, and the minimum allocation length may be **as big as a page**, so you should consolidate your requests for consistent memory as much as possible. The simplest way to do that is to use the **dma_pool** calls
    - **dma的实现是需要一个临时的mapping的**
        - **host将需要dma的内存地址写到硬件的寄存器上实际就意味着做一次mapping**
    - https://zhuanlan.zhihu.com/p/336616452
        - CPU访问内存用virtual_address，然后通过MMU转换成physical_address，外设访问内存用的bus_address，由bus把bus_address定位到物理内存上，不同的bus可能处理方法还不一样，但在pci下，目前bus_address就是phiscal_address。DMA内存由CPU分配，CPU需要把virtual_address转换成bus_address，然后DMA engine进行DMA操作。DMA_ZONE表示DMA可使用的内存范围，现在x86_64下一般设备所有内存都可用，**所以说DMA_ZONE和bus_address为了兼容而保留其实不用特殊考虑**

8. PCIe bar的uncached属性以及DMA的缓存一致性问题 —— 硕哥说了一个PCIe bar uncached以及dma是否需要考虑缓存一致性的问题，我也忽然发现我需要额外仔细理解一下这个

- ref
  - https://docs.kernel.org/driver-api/device-io.html
    - __iomem pointer tokens
      - The data type for an MMIO address is an `__iomem` qualified pointer, such as void `__iomem *reg` . On most architectures it is a regular pointer that points to a virtual memory address
      - While **on most architectures**, **ioremap() creates a page table entry for an uncached virtual address pointing to the physical MMIO address**, some architectures require special instructions for MMIO, and the __iomem pointer just encodes the physical address or an offsettable cookie that is interpreted by `readl()/writel()`
    - Device memory mapping modes
      - `ioremap()` 是最常见的映射类型，适用于典型的设备内存（例如 I/O 寄存器）。如果架构支持，其他模式可以提供更弱或更强的保证。从最常见到最不常见，如下所示，基本上就是优化性的操作都没有
        - **Uncached - CPU-side caches are bypassed**
        - 没有推测性操作
        - ...
      - `ioremap_wc()`
      - `ioremap_wt()`
      - `ioremap_np()`
      - `ioremap_uc()`
      - `ioremap_cache()`
    - 更高级的接口，实际上leapiosas是照着XDMA来写的，使用的接口是 `pci_iomap`
      - 鼓励驱动程序使用**更高级别的API**，而不是使用**上述原始ioremap()模式**。这些 API 可以实现特定于平台的逻辑，以在任何给定总线上自动选择适当的**ioremap**模式，从而允许与平台无关的驱动程序在没有任何特殊情况的情况下在这些平台上工作
        - 其中 `pci_iomap` 就是去调用了 `ioremap`
  - 所以这里首先需要得出一个结论，PCIe的bar空间是需要map到虚拟地址的，这部分内存应该如何对待，其实是提供了不同的接口的
    - 我们最常用也就是uncached的情况，也是默认的情况
- DMA的缓存一致性问题
  - DMA能不能直接访问cache？有类似的技术ddio(data direct IO)
  - DMA如果只访问物理内存会不会读到老的数据等等？
  - 实际上到了驱动这一层，缓存不一致什么的这些行为都被处理或者说屏蔽掉了
    - `dma_alloc_coherent`
    - `struct dma_pool *dma_pool_create(const char *name, struct device *dev, size_t size, size_t align, size_t alloc);`
      - `dma_pool_create()` initializes a pool of **DMA-coherent** buffers for use with a given device. It must be called in a context which can sleep
  - 描述符是缓存一致（一致性dma映射）的了，那数据呢？
    - 在真正开启DMA之前，会调用 `dma_map_sg` ，而该函数**负责缓存一致性**，**这个函数到底都干了啥呢，后期还是有一些疑惑且这里本身有一些误解**
      - 后期理解了，区别于描述符的一致性dma映射，这里使用的是流式DMA映射。通常来说
        - 如果是写操作
          - 在 `dma_map_sg` 的过程中，**会刷新cache到内存**
        - 如果是读操作
          - 在 `dma_unmap_sg` 的过程中，**会使cache无效**
    - 在这个操作过程期间，数据的一致性是无法得到保证的，但是正常情况下，这个阶段也没人需要数据，但是如果需要数据的话，则需要**主动的flush**

关注一下DMA的安全性问题

linux/Documentation/core-api/dma-api-howto.rst —— 该论文是DMA的殿堂级论文 —— 内核里的这几句英文描述真准确



- 首先先描述**不同的地址类型**
  - 虚拟地址（Virtual Address Space），`void*` 类型
  - CPU物理地址空间，`phys_addr_t` 类型
    - The kernel manages device resources like registers as physical addresses
  - IO设备使用第三种地址 —— **bus address**
    - If a device has registers at an MMIO address, or if it performs DMA to read or write system memory, **the addresses used by the device are bus addresses**
    - In some systems, **bus addresses are identical to CPU physical addresses**, but in general they are not

        - **IOMMUs and host bridges can produce arbitrary mappings between physical and bus addresses**
- 总线地址特别说一下
    - It can then use, e.g., `ioread32(C)`, to access the **device registers at bus address A**
- In some simple systems, the device can do DMA directly to physical address Y. But in many others, there is **IOMMU hardware that translates DMA addresses to physical addresses**, e.g., it translates Z to Y
    - 是否虚拟化都不影响IOMMU是否起作用，是独立的2个机制
- dma的map是动态的，例如在调用函数 `dma_map_single` 时，传参类型包括 `void*` ，返回值类型是 `dma_addr_t` ，在得到 `void*` 对应的 `dma_addr_t` 后，就可以用这个 `bus address` 来编程DMA寄存器了

What memory is DMA'able? —— 要想的内容真多，又学习到了

- vmalloc这类函数分配出来的物理page不一定是连续的，很难直接用于dma，强行使用是有可能的，但是完全没有必要
- 在驱动中定义的全局变量可以用于DMA吗
    - 如果编译到内核，那么全局变量位于内核的数据段或者bss段。在内核初始化的时候，会建立kernel image mapping，因此全局变量所占据的内存都是连续的，并且VA和PA是有固定偏移的线性关系，因此可以用于DMA操作
        - 不过，在定义这些全局变量的DMA buffer的时候，我们要小心的进行cacheline的对齐，并且要处理CPU和DMA controller之间的操作同步，以避免cache coherence问题
    - 如果驱动编译成模块会怎么样呢?
        - 这时候，驱动中的全局定义的DMA buffer不在内核的线性映射区域，其虚拟地址是在模块加载的时候，通过**vmalloc**分配，因此这时候如果DMA buffer如果**大于一个page frame**，那么实际上我们也是无法保证其底层物理地址的连续性，也无法保证VA和PA的线性关系，**这一点和编译到内核是不同的**

DMA addressing capabilities —— 寻址能力

- PCI-X specification requires PCI-X devices to support 64-bit addressing (DAC) for all transactions

```
static int cetc52_set_dma_mask(struct pci_dev *pdev)
{
        if (!dma_set_mask(&pdev->dev, DMA_BIT_MASK(64))) {
                dma_set_coherent_mask(&pdev->dev, DMA_BIT_MASK(64));
                dev_info(&pdev->dev, "Using a 64-bit DMA mask.\n");
        } else if (!dma_set_mask(&pdev->dev, DMA_BIT_MASK(32))) {
                dma_set_coherent_mask(&pdev->dev, DMA_BIT_MASK(32));
                dev_info(&pdev->dev, "Using a 32-bit DMA mask.\n");
        } else {
                dev_err(&pdev->dev, "No suitable DMA possible.\n");
                return -EINVAL;
        }
        return 0;
}
/* 下面这个是内核推荐的标准写法 */
static inline int dma_set_mask_and_coherent(struct device *dev, u64 mask)
{
        int rc = dma_set_mask(dev, mask); /* 这一步是通知设备，我想要把你set成mask，你如果返回成功那就是说明你可以的 */
        if (rc == 0)
                dma_set_coherent_mask(dev, mask); /* 假设设备被通知到了一个mask，并且dma_set_mask成功，那么这个时候需要通知kernel
        return rc;
}
```

Types of DMA mappings

一致性DMA映射

- 一致性的DMA映射依然需要考虑**内存屏障**，因为CPU可能会重排序（在重排序方面，CPU依然认为它们是**normal memory**）
  - 对于设备侧，如果有数据依赖的更新关系，则必须显示的加 `wmb()`
    - `desc->word0 = address;`
    - `wmb();`
    - `desc->word1 = DESC_VALID;`

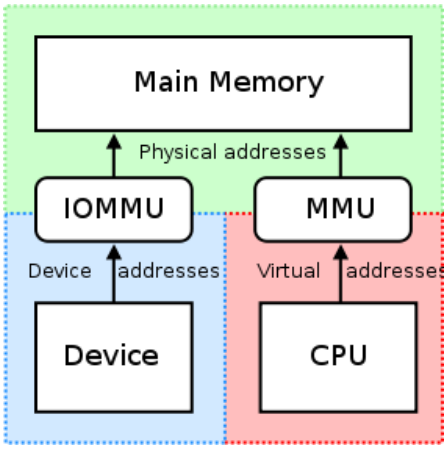
流式DMA映射

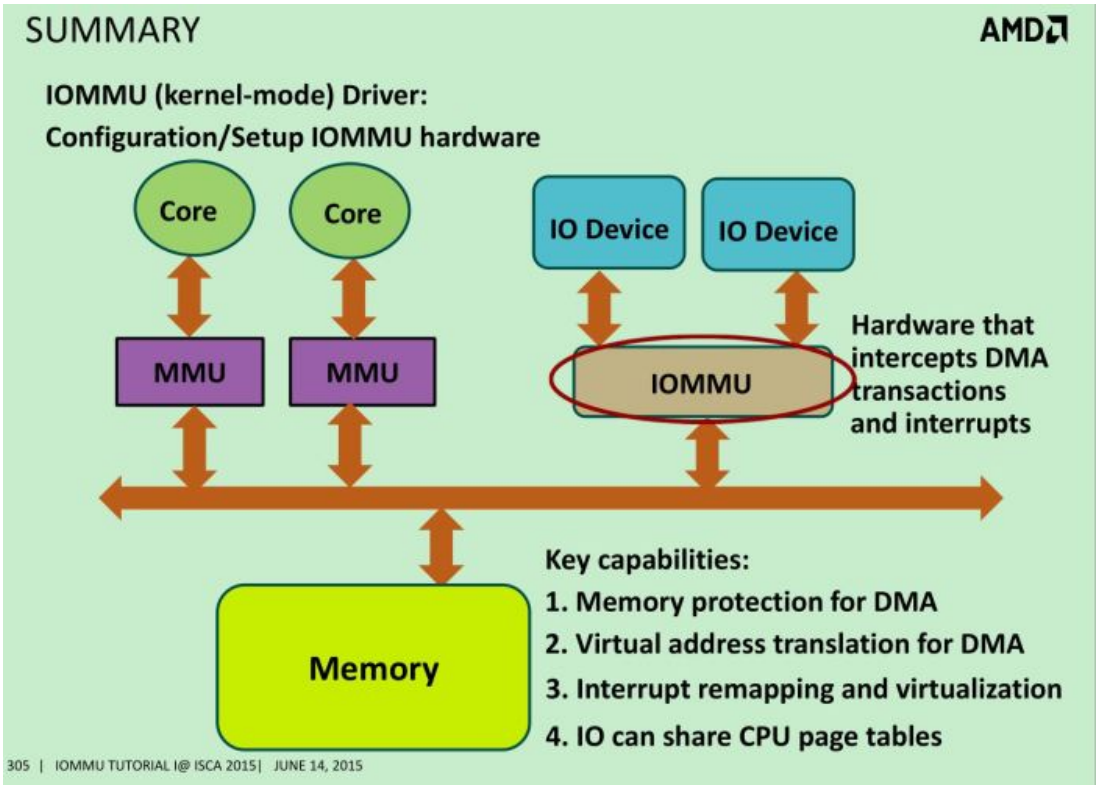
- data本身就是流式DMA映射，描述符通常是一致性DMA映射

DMA操作方向

- 在linux kernel中DMA有4个方向
  - **DMA_BIDIRECTIONAL** —— 不确定传输方向可以用这个
  - **DMA_TO_DEVICE**
  - **DMA_FROM_DEVICE**
  - **DMA_NONE** —— 主要用作**调试**

IOMMU





- https://nanxiao.me/iommu-introduction/
- IOMMU（Input/Output Memory Management Unit）是一个内存管理单元（Memory Management Unit），它的作用是连接DMA-capable I/O总线（Direct Memory Access-capable I/O Bus）和主存（main memory）
  - **DMA可以直接将设备数据move到主存储，这里需要的是物理地址**
    - 所以需要将设备访问的虚拟地址转换成物理地址
    - 就是DMA引擎也是拿着虚拟地址去访问物理内存的，IOMMU主要是用于隔离，因为寄存器都到了用户态，**不隔离就要GG**
  - IOMMU通常位于北桥，但是北桥基本已经被集成到SOC中了，**所以IOMMU现在就是在北桥**
- **传统的内存管理单元会把CPU访问的虚拟地址转化成实际的物理地址（内存）。而IOMMU则是把对分配给设备（device）的地址的访问转化成物理地址（内存）**