PCI Devices

Basics

This section instructs how to model a basic PCI device in QEMU. We will call out device `sample-pci` and will implement it in `hw/itc/sample-pci.c`. The specifications of `sample-pci` are as follows:

- 1 BAR that exists in memory space
- Reading register 0 returns 1
- Reading register 1 returns 0
- Writing to register 0 prints `Hello\n`
- Writing to register 1 prints `World!\n`
- The vendor is QEMU
- The device is a VGA device (even though it most certainly isn't)

First, let's add the device to `Makefile.objs`. Since not all architectures support PCI, `make` will crash if we use `common-obj-y` like the last samples, thus we have to use our own configuration variable, `CONFIG_SAMPLEPCI`.

Makefile.objs - Adding `sample-pci`

```
...

common-obj-$(CONFIG_SAMPLEPCI) += sample-pci.o
```

To define `CONFIG_SAMPLEPCI`, we must make a `Kconfig` file in `hw/itc`. This tells `make` to add `sample-pci` only if PCI is enabled.

Kconfig

```
config SAMPLEPCI
    bool
    default y if TEST_DEVICES
    depends on PCI
```

Now we can get to implementing `sample-pci`. We will start with a simple skeleton that is similar to the sample MMIO device.

sample-pci.c - Basic Skeleton

```
#include "qemu/osdep.h"
#include "hw/pci/pci.h"
#include "hw/hw.h"
#include "qemu/module.h"


#define TYPE_SAMPLE_PCI_DEVICE "sample-pci"
#define SAMPLE_PCI_DEVICE(obj) OBJECT_CHECK(SamplePCIDeviceState

typedef struct SamplePCIDeviceState {
    PCIDevice parent;

    MemoryRegion mmio;
} SamplePCIDeviceState;

static uint64_t sample_pci_device_read(void *opaque, hwaddr addr
printf("**READ**\n");
return 0;
}

static void sample_pci_device_write(void *opaque, hwaddr addr, u
printf("**WRITE**\n");
}

static const MemoryRegionOps sample_pci_device_ops = {
.read = sample_pci_device_read,
.write = sample_pci_device_write,
.endianness = DEVICE_LITTLE_ENDIAN
};

static void sample_pci_device_realize(PCIDevice *dev, Error **er
SamplePCIDeviceState *sample = SAMPLE_PCI_DEVICE(dev);

    memory_region_init_io(&sample->mmio, OBJECT(dev), &sample_pc
}

static void sample_pci_device_uninit(PCIDevice *dev) {

}

static void sample_pci_init(ObjectClass *klass, void *data) {
    DeviceClass *dc = DEVICE_CLASS(klass);
```

```
    PCIDeviceClass *k = PCI_DEVICE_CLASS(klass);

    k->realize = sample_pci_device_realize;
    k->exit = sample_pci_device_uninit;
    k->vendor_id = PCI_VENDOR_ID_QEMU;
    k->device_id = PCI_DEVICE_ID_QEMU_VGA;
    k->revision = 0x00;
    k->class_id = PCI_CLASS_DISPLAY_VGA;
    set_bit(DEVICE_CATEGORY_MISC, dc->categories);
}

static InterfaceInfo interfaces[] = {
        { INTERFACE_CONVENTIONAL_PCI_DEVICE },
        { },
    };
static const TypeInfo sample_pci_info = {
    .name          = TYPE_SAMPLE_PCI_DEVICE,
    .parent        = TYPE_PCI_DEVICE,
    .instance_size = sizeof(SamplePCIDeviceState),
    .class_init    = sample_pci_init,
    .interfaces = interfaces,
};

static void sample_pci_register_types(void) {
    type_register_static(&sample_pci_info);
}

type_init(sample_pci_register_types)
```

For the most part, this skeleton is the similar the MMIO sample. However, there are some important additions to make note of.

Firstly, `PCIDevice` objects have additional exit functions. These functions allow for proper handling of freeing memory or finishing threads. As there are no extra things in our sample device, our exit function is empty.

```
static void sample_pci_device_uninit(PCIDevice *dev) {

}
```

The initialization function also has some changes, in particular for setting ID information.

```
static void sample_pci_init(ObjectClass *klass, void *data) {
    DeviceClass *dc = DEVICE_CLASS(klass);
```

```
    PCIDeviceClass *k = PCI_DEVICE_CLASS(klass);

    k->realize = sample_pci_device_realize;
    k->exit = sample_pci_device_uninit; 
    k->vendor_id = PCI_VENDOR_ID_QEMU; 
    k->device_id = PCI_DEVICE_ID_QEMU_VGA; 
    k->revision = 0x00;
    k->class_id = PCI_CLASS_DISPLAY_VGA; 
    set_bit(DEVICE_CATEGORY_MISC, dc->categories);
}
```

 This registers the exit function.
 This sets the vendor to be QEMU. This id is equivalent to 0x1234.
 This sets the device to be a QEMU VGA device. This id is equivalent to 0x1111.
 This sets the class to be a VGA device.

Constants for the vendor and device IDs can be found in `include/hw/pci/pci_ids.h`

Now, you can run `make` to make QEMU with `sample-pci` as a device. Then, in your home directory, you can run `./build/i386-softmmu/qemu-system-i386 -monitor stdio alpine.img -` This will add the device on bus `pci.0` which is the PCI bus for `i386`. Once this is up and running, you can run `info qtree` in the HMP terminal and you should get an output that contains:

```
dev: sample-pci, id "foo"
        addr = 04.0
        romfile = ""
        rombar = 1 (0x1)
        multifunction = false
        command_serr_enable = true
        x-pcie-lnksta-dllla = true
        x-pcie-extcap-init = true
        failover_pair_id = ""
        class VGA controller, addr 00:04.0, pci id 1234:1111 (su
```

And in Alpine's terminal, you can run `lspci` and get an output similar to:

```
...
00:04.0 VGA compatiblecontroller: Device 1234:1111
```

This is device is `sample-pci`.

Now that we have the working skeleton, we can now add the BAR for `sample-pci`. Adding this BAR will require some changes to `sample_pci_device_read`, `sample_pci_device_write`, and `sample_pci_device_realize`.

In `sample_pci_device_realize`, we want to register our BAR.

sample-pci.c - Registering a BAR

```
static const MemoryRegionOps sample_pci_device_ops = {
.read = sample_pci_device_read,
.write = sample_pci_device_write,
.endianness = DEVICE_LITTLE_ENDIAN
};

static void sample_pci_device_realize(PCIDevice *dev, Error **er
SamplePCIDeviceState *sample = SAMPLE_PCI_DEVICE(dev);

    memory_region_init_io(&sample->mmio, OBJECT(dev), &sample_pc
    pci_register_bar(dev, 0, PCI_BASE_ADDRESS_SPACE_MEMORY, &sam
}
```

 `pci_register_bar` registers `mmio` as a BAR.

Since `sample_pci_device_read` and `sample_pci_device_write` are the functions used for accessing our BAR, the changes to them will reflect the specifications for reading and writing to registers.

sample-pci.c - Read and Write

```
static uint64_t sample_pci_device_read(void *opaque, hwaddr addr
SamplePCIDeviceState *sample = *opaque;

  int val = -1;

  switch(addr) {
    case 0:
      val = 1;
      break;
    case 1:
      val = 0;
      break;
  }
return val;
}
```

```
static void sample_pci_device_write(void *opaque, hwaddr addr, u
SamplePCIDeviceState *sample = *opaque;

  switch(addr) { □
    case 0:
      printf("Hello\n");
      break;
    case 1:
      printf("World!\n");
      break;
  }
}
```

□ `addr` is the address being accessed, which we use to define which register is being accessed. So, here if `addr` is 0, we are accessing register 0 so we should return 1. Similarly, if `addr` is 1, we return 0.
□ Again, `addr` is the address being accessed. So, if `addr` is 0, we print `Hello\n`, and if `addr` is 1, we print `World!\n`.

Now we can run `make` and `./build/i386-softmmu/qemu-system-i386 -monitor stdio alpine.img -` again to add `sample-pci` with its BAR. You can see this BAR if you run `info qtree` in the HMP terminal you should get a return that contains:

```
dev: sample-pci, id "foo"
        addr = 04.0
        romfile = ""
        rombar = 1 (0x1)
        multifunction = false
        command_serr_enable = true
        x-pcie-lnksta-dllla = true
        x-pcie-extcap-init = true
        failover_pair_id = ""
        class VGA controller, addr 00:04.0, pci id 1234:1111 (su
        bar 0: mem at 0xfebf1000 [0xfebf1fff]
```

`bar 0` is our newly added BAR.

Now we have implemented a basic PCI device that meets the full specifications as given.

Config Space

If you are wanting to model a PC compatible PCI device, QEMU already takes care of a lot of the config space for you. However, QEMU still leaves open the

possibility to make necessary changes to model a wider array of PCI devices.

If the device being modeled is a bridge, then the field `is_bridge` must be set to `true`. By default, it's value is `false`. `is_bridge` is used by QEMU to determine if the config space header is type 0 or type 1.

PCI Bridge

```
static void sample_pci_init(ObjectClass *klass, void *data) {
DeviceClass *dc = DEVICE_CLASS(klass);
PCIDeviceClass *k = PCI_DEVICE_CLASS(klass);
  ...
  k->is_bridge = true;
  ...
}
```

By default, QEMU assumes the device's config space is accessed using the `CONFIG_ADDRESS` and `CONFIG_DATA` registers. However, it is possible to write your own access methods and then use them to overwrite the default using `config_read` and `config_write`.

Config Read/Write

```
void sample_config_read(PCIDevice *pci_dev, uint32_t address, ui
...
}
uint32_t sample_config_write(PCIDevice *pci_dev. uint32_t addres
...
}
...
static void sample_pci_init(ObjectClass *klass, void *data) {
DeviceClass *dc = DEVICE_CLASS(klass);
PCIDeviceClass *k = PCI_DEVICE_CLASS(klass);
  ...
  k->config_read = sample_config_read;
  k->config_write = sample_config_write;
  ...
}
```

BARs

For the most part, BARs in QEMU are similar to doing MMIO. Whether the device being modeled uses I/O space or memory space, the steps are similar and just vary on whether the type is `PCI_BASE_ADDRESS_SPACE_IO` or `PCI_BASE_ADDRESS_SPACE_MEMORY`.

```
typedef struct SamplePCIDeviceState {
PCIDevice parent;

MemoryRegion mmio;
} SamplePCIDeviceState;

static uint64_t sample_pci_device_read(void *opaque, hwaddr addr
SamplePCIDeviceState *sample = *opaque;

  int val = -1;

  switch(addr) {
    case 0:
      val = 1;
      break;
    case 1:
      val = 0;
      break;
  }
return val;
}

static void sample_pci_device_write(void *opaque, hwaddr addr, u
SamplePCIDeviceState *sample = *opaque;

  switch(addr) {
    case 0:
      printf("Hello\n");
      break;
    case 1:
      printf("World!\n");
      break;
  }
}

static const MemoryRegionOps sample_pci_device_ops = {
.read = sample_pci_device_read,
.write = sample_pci_device_write,
.endianness = DEVICE_LITTLE_ENDIAN
};

static void sample_pci_device_realize(PCIDevice *dev, Error **er
SamplePCIDeviceState *sample = SAMPLE_PCI_DEVICE(dev);

memory_region_init_io(&sample->mmio, OBJECT(dev), &sample_pci_de
pci_register_bar(dev, 0, PCI_BASE_ADDRESS_SPACE_MEMORY, &sample-
```

```
}
```

 This function gives the read functionality for the memory region that will be used for our BAR.
 This switch statement lets us determine the proper actions to take based on what address is being accessed, represented in the value of `addr`. For our function, it returns 1 if address 0 is being accessed and 0 if address 1 is being accessed.
 This function gives the write functionality for the memory region that will be used for our BAR.

> **Note**: `opaque` in both the read and write function are a `SamplePCIDeviceState` because we pass our instance `sample` to `memory_region_init_io` in the `opaque` argument.

 This switch statement serves the same purpose as the one in . For our function, it prints “Hello” if address 0 is accessed and “World!” if address 1 is accessed.
 This is where our memory region is actually registered as a BAR for our device. We pass the `PCIDevice` `dev`, the region number 0 since this is the first BAR, the type `PCI_BASE_ADDRESS_SPACE_MEMORY` since our BAR uses memory space, and finally a reference to the `MemoryRegion` `sample->mmio`. Since there can only be 6 BARs, the region number must be between 0 and 5. Additionally, to set up this BAR as I/O space, `PCI_BASE_ADDRESS_SPACE_IO` would be used in place of `PCI_BASE_ADDRESS_SPACE_MEMORY`.

Interrupts

For PCI interrupts, QEMU uses the 4 interrupt lines, `INTA#`, `INTB#`, `INTC#`, and `INTD#`, which will be referred to as A, B, C, and D for the rest of this document. Setting up and using these interrupts in recent QEMU releases is mostly simple, as there are many PCI specific interrupt functions.

edu.c - Interrupts

```
typedef struct {
    PCIDevice pdev;
    MemoryRegion mmio;

    QemuThread thread;
    QemuMutex thr_mutex;
    QemuCond thr_cond;
    bool stopping;

    uint32_t addr4;
    uint32_t fact;
```

```
#define EDU_STATUS_COMPUTING    0x01
#define EDU_STATUS_IRQFACT      0x80
    uint32_t status;

    uint32_t irq_status;

#define EDU_DMA_RUN             0x1
#define EDU_DMA_DIR(cmd)        (((cmd) & 0x2) >> 1)
# define EDU_DMA_FROM_PCI       0
# define EDU_DMA_TO_PCI         1
#define EDU_DMA_IRQ             0x4
    struct dma_state {
        dma_addr_t src;
        dma_addr_t dst;
        dma_addr_t cnt;
        dma_addr_t cmd;
    } dma;
    QEMUTimer dma_timer;
    char dma_buf[DMA_SIZE];
    uint64_t dma_mask;
} EduState;

...

static void edu_raise_irq(EduState *edu, uint32_t val)
{
    edu->irq_status |= val;
    if (edu->irq_status) {
        if (edu_msi_enabled(edu)) {
            msi_notify(&edu->pdev, 0);
        } else {
            pci_set_irq(&edu->pdev, 1); □
        }
    }
}

static void edu_lower_irq(EduState *edu, uint32_t val)
{
    edu->irq_status &= ~val;

    if (!edu->irq_status && !edu_msi_enabled(edu)) {
        pci_set_irq(&edu->pdev, 0); □
    }
}

...
```

```
static void pci_edu_realize(PCIDevice *pdev, Error **errp)
{
    EduState *edu = EDU(pdev);
    uint8_t *pci_conf = pdev->config;

    pci_config_set_interrupt_pin(pci_conf, 1); □

    if (msi_init(pdev, 0, 1, true, false, errp)) {
        return;
    }

    timer_init_ms(&edu->dma_timer, QEMU_CLOCK_VIRTUAL, edu_dma_t

    qemu_mutex_init(&edu->thr_mutex);
    qemu_cond_init(&edu->thr_cond);
    qemu_thread_create(&edu->thread, "edu", edu_fact_thread,
                       edu, QEMU_THREAD_JOINABLE);

    memory_region_init_io(&edu->mmio, OBJECT(edu), &edu_mmio_ops
                    "edu-mmio", 1 * MiB);
    pci_register_bar(pdev, 0, PCI_BASE_ADDRESS_SPACE_MEMORY, &ed
}
```

□ `pci_set_irq` either raises or lowers the interrupt based on the second argument, `level`. `level` is 1 here so the interrupt is being raised.
□ `level` is 0 here so the interrupt is being lowered.

> **Note**: There are additional wrappers `pci_irq_assert` and `pci_irq_deassert` which only take a `PCIDevice` as an argument, and do `pci_set_irq` with `level` as 1 and 0, respectively.

□ This function call sets up interrupts on a desired pin. The first argument, `pci_conf` is the config space of the device. The second argument is which interrupt to use. 1, 2, 3, and 4 correspond to A, B, C, and D, respectively.

> **Note**: All the work of checking the interrupt pin value is handled by QEMU through `pci_set_irq`. There are still some devices that do the old way of using `qemu_set_irq`, but this should be avoided when making PCI devices.

DMA

There is no standard method of doing DMA in QEMU, however all there are many similarities between the current implementations. For this section, we will again use `edu.c` as our example code.

edu.c - DMA

```
#define FACT_IRQ        0x00000001
#define DMA_IRQ         0x00000100

#define DMA_START       0x40000
#define DMA_SIZE        4096

typedef struct {
    PCIDevice pdev;
    MemoryRegion mmio;

    QemuThread thread;
    QemuMutex thr_mutex;
    QemuCond thr_cond;
    bool stopping;

    uint32_t addr4;
    uint32_t fact;
#define EDU_STATUS_COMPUTING    0x01
#define EDU_STATUS_IRQFACT      0x80
    uint32_t status;

    uint32_t irq_status;

#define EDU_DMA_RUN             0x1
#define EDU_DMA_DIR(cmd)        (((cmd) & 0x2) >> 1)
# define EDU_DMA_FROM_PCI       0
# define EDU_DMA_TO_PCI         1
#define EDU_DMA_IRQ             0x4
    struct dma_state {
        dma_addr_t src;
        dma_addr_t dst;
        dma_addr_t cnt;
        dma_addr_t cmd;
    } dma;
    QEMUTimer dma_timer;
    char dma_buf[DMA_SIZE];
    uint64_t dma_mask;
} EduState;

...

static dma_addr_t edu_clamp_addr(const EduState *edu, dma_addr_t
{
    dma_addr_t res = addr & edu->dma_mask;

    if (addr != res) {
```

```
        printf("EDU: clamping DMA %#.16"PRIx64" to %#.16"PRIx64"
    }

    return res;
}

static void edu_dma_timer(void *opaque)
{
    EduState *edu = opaque;
    bool raise_irq = false;

    if (!(edu->dma.cmd & EDU_DMA_RUN)) {
        return;
    }

    if (EDU_DMA_DIR(edu->dma.cmd) == EDU_DMA_FROM_PCI) {
        uint64_t dst = edu->dma.dst;
        edu_check_range(dst, edu->dma.cnt, DMA_START, DMA_SIZE);
        dst -= DMA_START;
        pci_dma_read(&edu->pdev, edu_clamp_addr(edu, edu->dma.sr
                edu->dma_buf + dst, edu->dma.cnt);
    } else {
        uint64_t src = edu->dma.src;
        edu_check_range(src, edu->dma.cnt, DMA_START, DMA_SIZE);
        src -= DMA_START;
        pci_dma_write(&edu->pdev, edu_clamp_addr(edu, edu->dma.d
                edu->dma_buf + src, edu->dma.cnt);
    }

    edu->dma.cmd &= ~EDU_DMA_RUN;
    if (edu->dma.cmd & EDU_DMA_IRQ) {
        raise_irq = true;
    }

    if (raise_irq) {
        edu_raise_irq(edu, DMA_IRQ);
    }
}

static void dma_rw(EduState *edu, bool write, dma_addr_t *val, d
                bool timer)
{
    if (write && (edu->dma.cmd & EDU_DMA_RUN)) {
        return;
    }

    if (write) {
```

```
        *dma = *val;
    } else {
        *val = *dma;
    }

    if (timer) {
        timer_mod(&edu->dma_timer, qemu_clock_get_ms(QEMU_CLOCK_
    }
}

static uint64_t edu_mmio_read(void *opaque, hwaddr addr, unsigne
{
    EduState *edu = opaque;
    uint64_t val = ~0ULL;

    if (addr < 0x80 && size != 4) {
        return val;
    }

    if (addr >= 0x80 && size != 4 && size != 8) {
        return val;
    }

    switch (addr) {

..

    case 0x80:
        dma_rw(edu, false, &val, &edu->dma.src, false);
        break;
    case 0x88:
        dma_rw(edu, false, &val, &edu->dma.dst, false);
        break;
    case 0x90:
        dma_rw(edu, false, &val, &edu->dma.cnt, false);
        break;
    case 0x98:
        dma_rw(edu, false, &val, &edu->dma.cmd, false);
        break;
    }

    return val;
}

static void edu_mmio_write(void *opaque, hwaddr addr, uint64_t v
                unsigned size)
{
```

```
    EduState *edu = opaque;

    if (addr < 0x80 && size != 4) {
        return;
    }

    if (addr >= 0x80 && size != 4 && size != 8) {
        return;
    }

    switch (addr) {

..

    case 0x80:
        dma_rw(edu, true, &val, &edu->dma.src, false);
        break;
    case 0x88:
        dma_rw(edu, true, &val, &edu->dma.dst, false);
        break;
    case 0x90:
        dma_rw(edu, true, &val, &edu->dma.cnt, false);
        break;
    case 0x98:
        if (!(val & EDU_DMA_RUN)) {
            break;
        }
        dma_rw(edu, true, &val, &edu->dma.cmd, true);
        break;
    }
}

...
```

The values for the DMA registers are held in the `dma` struct. `src` is the address to move data from, `dst` is the address to move data to, `cnt` is the number of bytes to transfer, and `cmd` initiates DMA and acts partially as a status register for the DMA process. `dma_buf` is the local buffer that stores data read from memory and writes to memory. `dma_mask` is used to mask off `src` and `dst` to make sure they have the proper number of bits.
Another method of implementing the DMA registers is to use an array. This form can be seen in `hw/dma/sparc32_dma.c`.

edu.c - `dma` *struct*

```
...
```

```
struct dma_state {
    dma_addr_t src;
    dma_addr_t dst;
    dma_addr_t cnt;
    dma_addr_t cmd;
} dma;
QEMUTimer dma_timer;
char dma_buf[DMA_SIZE];
uint64_t dma_mask;

...
```

The DMA registers can be accessed through BARs, as is done in `edu.c`. The `dma_rw` function does the setting or returning of values, and even triggering the timer which will then trigger the reading or writing to memory.
In `edu.c`, the DMA registers are accessed through the BAR at addresses 0x80, 0x88, 0x90, 0x98, which correspond to `src`, `dst`, `cnt`, and `cmd`, respectively. Writing to `cmd` is the only access that will trigger a DMA read or write, accessing any of the other registers will just read or change the state of the DMA controller.
It is possbile to use other ways of accessing the DMA registers, such as their own MMIO region.

edu.c - Accessing DMA Registers

```
...

static void dma_rw(EduState *edu, bool write, dma_addr_t *val, d
                   bool timer)
{
    if (write && (edu->dma.cmd & EDU_DMA_RUN)) {
        return;
    }

    if (write) {
        *dma = *val;
    } else {
        *val = *dma;
    }

    if (timer) {
        timer_mod(&edu->dma_timer, qemu_clock_get_ms(QEMU_CLOCK_
    }
}
```

```
static uint64_t edu_mmio_read(void *opaque, hwaddr addr, unsigne
{
    EduState *edu = opaque;
    uint64_t val = ~0ULL;

    if (addr < 0x80 && size != 4) {
        return val;
    }

    if (addr >= 0x80 && size != 4 && size != 8) {
        return val;
    }

    switch (addr) {

..

    case 0x80:
        dma_rw(edu, false, &val, &edu->dma.src, false);
        break;
    case 0x88:
        dma_rw(edu, false, &val, &edu->dma.dst, false);
        break;
    case 0x90:
        dma_rw(edu, false, &val, &edu->dma.cnt, false);
        break;
    case 0x98:
        dma_rw(edu, false, &val, &edu->dma.cmd, false);
        break;
    }

    return val;
}

static void edu_mmio_write(void *opaque, hwaddr addr, uint64_t v
                unsigned size)
{
    EduState *edu = opaque;

    if (addr < 0x80 && size != 4) {
        return;
    }

    if (addr >= 0x80 && size != 4 && size != 8) {
        return;
    }
```

```
    switch (addr) {

..

    case 0x80:
        dma_rw(edu, true, &val, &edu->dma.src, false);
        break;
    case 0x88:
        dma_rw(edu, true, &val, &edu->dma.dst, false);
        break;
    case 0x90:
        dma_rw(edu, true, &val, &edu->dma.cnt, false);
        break;
    case 0x98:
        if (!(val & EDU_DMA_RUN)) {
            break;
        }
        dma_rw(edu, true, &val, &edu->dma.cmd, true);
        break;
    }
}

...
```

Doing data transfers to and from memory is possible using `pci_dma_read` and `pci_dma_write`. `pci_dma_read` takes a memory address as the second argument, a buffer as the third argument, and a count as the fourth argument. "count" many bytes are then transfered from the memory region starting at the given address to the buffer. `pci_dma_write` does the reverse, tranferring "count" many bytes from the buffer to the memory region starting at the memory address. Both of these functions work synchronously, and do all proper blocking on their own.
In `edu.c`, these are used to transfer data between `dma buf` and the 4096 bytes of memory starting at `DMA_START`. Once they are finished, the device then raises an interrupt if commanded to.

> TODO: See why `edu.c` has DMA because it doesn't seem a driver can access the data that is read/written.

If you take a look at other DMA controllers, such as `hw/dma/sparc32_dma.c`, you may see them using `dma_memory_read` and `dma_memory_write`. These are both wrappers for the same functions `pci_dma_read` and `pci_dma_write` are wrappers for.

edu.c - DMA Reading and Writing

```
static dma_addr_t edu_clamp_addr(const EduState *edu, dma_addr_t
```

```
{
    dma_addr_t res = addr & edu->dma_mask;

    if (addr != res) {
        printf("EDU: clamping DMA %#.16"PRIx64" to %#.16"PRIx64"
    }

    return res;
}

static void edu_dma_timer(void *opaque)
{
    EduState *edu = opaque;
    bool raise_irq = false;

    if (!(edu->dma.cmd & EDU_DMA_RUN)) {
        return;
    }

    if (EDU_DMA_DIR(edu->dma.cmd) == EDU_DMA_FROM_PCI) {
        uint64_t dst = edu->dma.dst;
        edu_check_range(dst, edu->dma.cnt, DMA_START, DMA_SIZE);
        dst -= DMA_START;
        pci_dma_read(&edu->pdev, edu_clamp_addr(edu, edu->dma.sr
                edu->dma_buf + dst, edu->dma.cnt);
    } else {
        uint64_t src = edu->dma.src;
        edu_check_range(src, edu->dma.cnt, DMA_START, DMA_SIZE);
        src -= DMA_START;
        pci_dma_write(&edu->pdev, edu_clamp_addr(edu, edu->dma.d
                edu->dma_buf + src, edu->dma.cnt);
    }

    edu->dma.cmd &= ~EDU_DMA_RUN;
    if (edu->dma.cmd & EDU_DMA_IRQ) {
        raise_irq = true;
    }

    if (raise_irq) {
        edu_raise_irq(edu, DMA_IRQ);
    }
}

...
```

The main take away from this section should be there are many different ways to

implement DMA. This section outlined a method that works for implementing DMA for a PCI device without an IOMMU, but to investigate other implementations it would be best to analyze `hw/dma/sparc32_dma.c` some.

References:

- Maydell, Peter. Ask about how DMA is implemented in qemu-arm-system. April 27, 2018.
- Polard, Tic Le. Emulate a PCI device with Qemu. Jan 10, 2015.
- Terenceli. QEMU interrupt emulation. Sept, 6, 2018.
- OSDev Wiki