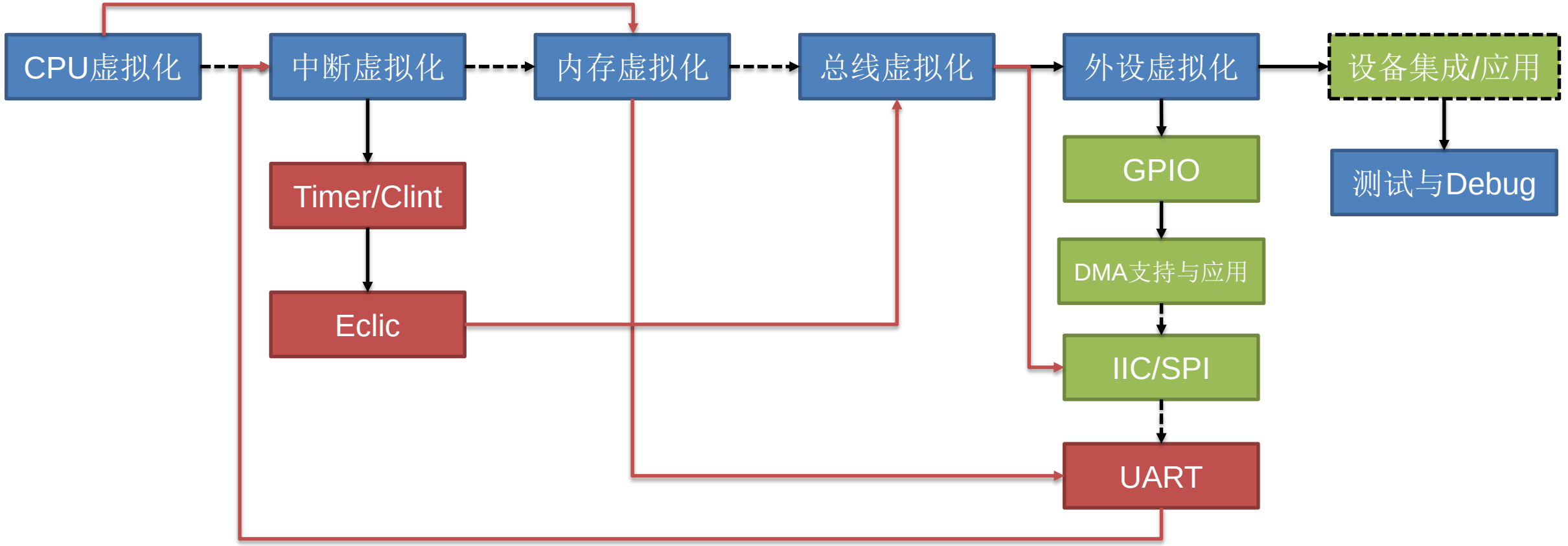
# 从零开始的RISC-V模拟器开发

## 第9讲 QEMU篇之内存虚拟化

中国科学院软件研究所
PLCT实验室

王俊强 wangjunqiang@iscas.ac.cn
李威威 liweiwei@iscas.ac.cn
吴伟 wuwei2016@iscas.ac.cn

# 课程内容调整


CPU虚拟化
中断虚拟化
内存虚拟化
总线虚拟化
外设虚拟化
设备集成/应用
Timer/Clint
Eclic
GPIO
DMA支持与应用
IIC/SPI
UART
测试与Debug

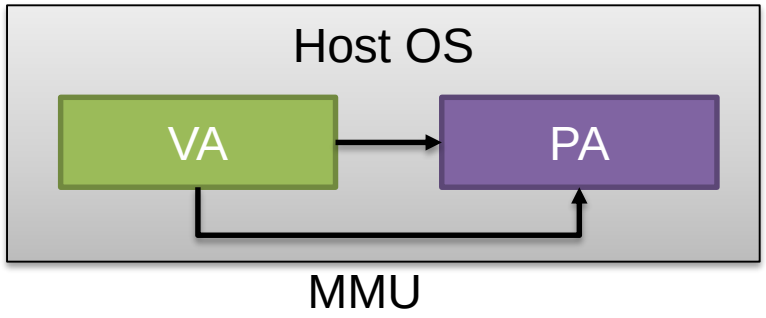
# 本课内容

## 内存虚拟化

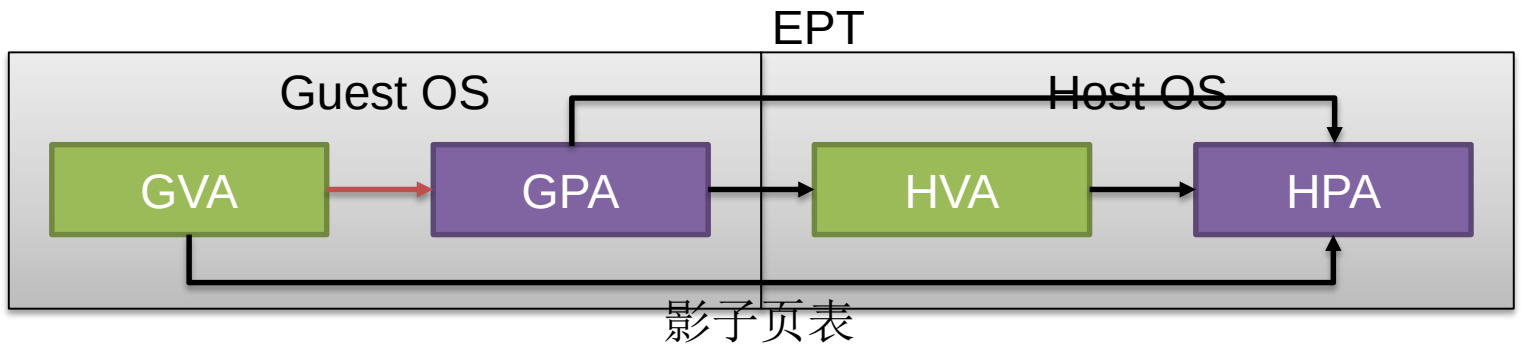
- 内存虚拟化介绍
- Nuclei内存分布实现

# 内存虚拟化

一般状态：




QEMU状态下：

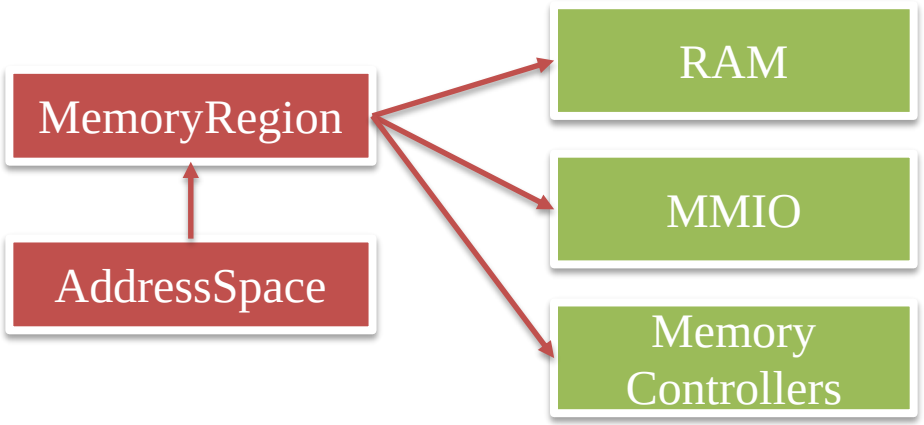



~~内存虚拟化~~ **内存模拟**

# 内存虚拟化

The memory API models the memory and I/O buses and controllers of a QEMU machine. It attempts to allow modelling of:
•ordinary RAM
•memory-mapped I/O (MMIO)
•memory controllers that can dynamically reroute physical memory regions to different destinations
……
Memory is modelled as an acyclic graph of MemoryRegion objects. Sinks (leaves) are RAM and MMIO regions, while other nodes represent buses, memory controllers, and memory regions that have been rerouted.

In addition to MemoryRegion objects, the memory API provides AddressSpace objects for every root and possibly for intermediate MemoryRegions too. These represent memory as seen from the CPU or a device's viewpoint.




参考: https://qemu.readthedocs.io/en/latest/devel/memory.html

# 内存虚拟化

There are multiple types of memory regions (all represented by a single C type MemoryRegion):

•**RAM**: a RAM region is simply a range of host memory that can be made available to the guest. You typically initialize these with memory_region_init_ram(). Some special purposes require the variants memory_region_init_resizeable_ram(), memory_region_init_ram_from_file(), or memory_region_init_ram_ptr().

•**MMIO**: a range of guest memory that is implemented by host callbacks; each read or write causes a callback to be called on the host. You initialize these with memory_region_init_io(), passing it a MemoryRegionOps structure describing the callbacks.

•**ROM**: a ROM memory region works like RAM for reads (directly accessing a region of host memory), and forbids writes. You initialize these with memory_region_init_rom().

•**ROM device**: a ROM device memory region works like RAM for reads (directly accessing a region of host memory), but like MMIO for writes (invoking a callback). You initialize these with memory_region_init_rom_device().

•**IOMMU region**: an IOMMU region translates addresses of accesses made to it and forwards them to some other target memory region. As the name suggests, these are only needed for modelling an IOMMU, not for simple devices. You initialize these with memory_region_init_iommu().

•**container**: a container simply includes other memory regions, each at a different offset. Containers are useful for grouping several regions into one unit. For example, a PCI BAR may be composed of a RAM region and an MMIO region.

A container's subregions are usually non-overlapping. In some cases it is useful to have overlapping regions; for example a memory controller that can overlay a subregion of RAM with MMIO or ROM, or a PCI controller that does not prevent card from claiming overlapping BARs.

You initialize a pure container with memory_region_init().

•**alias**: a subsection of another region. Aliases allow a region to be split apart into discontiguous regions. Examples of uses are memory banks used when the guest address space is smaller than the amount of RAM addressed, or a memory controller that splits main memory to expose a "PCI hole". Aliases may point to any type of region, including other aliases, but an alias may not point back to itself, directly or indirectly. You initialize these with memory_region_init_alias().

•**reservation region**: a reservation region is primarily for debugging. It claims I/O space that is not supposed to be handled by QEMU itself. The typical use is to track parts of the address space which will be handled by the host kernel when KVM is enabled. You initialize these by passing a NULL callback parameter to memory_region_init_io().

参考: https://qemu.readthedocs.io/en/latest/devel/memory.html

# 内存虚拟化

There are multiple types of memory regions (all represented by a single C type MemoryRegion):

•**RAM**: a RAM region is simply a range of host memory that can be made available to the guest. You typically initialize these with memory_region_init_ram(). Some special purposes require the variants memory_region_init_resizeable_ram(), memory_region_init_ram_from_file(), or memory_region_init_ram_ptr().

•**MMIO**: a range of guest memory that is implemented by host c

initialize these with memory_region_init_io(), passing it a Memo

•**ROM**: a ROM memory region works like RAM for reads (direc

with memory_region_init_rom().

•**ROM device**: a ROM device memory region works like RAM f

(invoking a callback). You initialize these with memory_region_

•**IOMMU region**: an IOMMU region translates addresses of acc

name suggests, these are only needed for modelling an IOMMU,

•**container**: a container simply includes other memory regions, e

one unit. For example, a PCI BAR may be composed of a RAM

A container's subregions are usually non-overlapping. In some c

that can overlay a subregion of RAM with MMIO or ROM, or a

You initialize a pure container with memory_region_init().

•**alias**: a subsection of another region. Aliases allow a region to b

when the guest address space is smaller than the amount of RAM

hole". Aliases may point to any type of region, including other al

these with memory_region_init_alias().

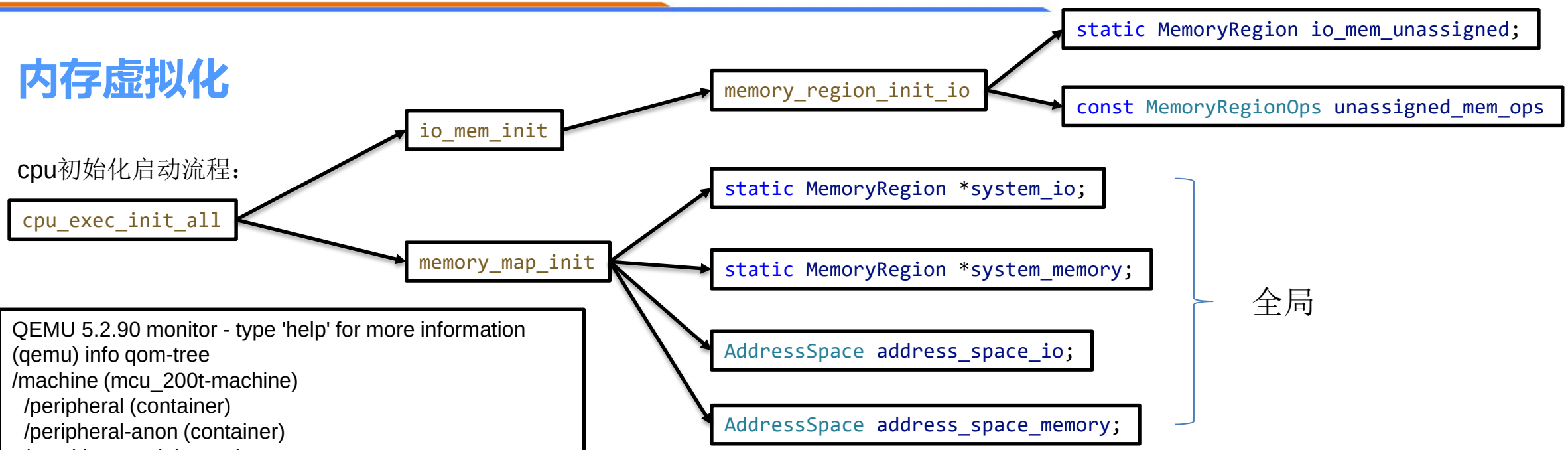
| Types of regions | initialize |
|---|---|
| RAM | memory_region_init_ram() |
| MMIO | memory_region_init_io() |
| ROM | memory_region_init_rom(). |
| ROM device | memory_region_init_rom_device() |
| IOMMU region | memory_region_init_iommu() |
| container | memory_region_init() |
| alias | memory_region_init_alias() |
| reservation region | memory_region_init_io() |

•**reservation region**: a reservation region is primarily for debugging. It claims I/O space that is not supposed to be handled by QEMU itself. The typical use is to track parts of the address space which will be handled by the host kernel when KVM is enabled. You initialize these by passing a NULL callback parameter to memory_region_init_io().

参考: https://qemu.readthedocs.io/en/latest/devel/memory.html

# 内存虚拟化

cpu初始化启动流程：




```
QEMU 5.2.90 monitor - type 'help' for more information
(qemu) info qom-tree
/machine (mcu_200t-machine)
  /peripheral (container)
  /peripheral-anon (container)
  /soc (riscv.nuclei.n.soc)
    /cpus (riscv.hart_array)
      /harts[0] (nuclei-n600-riscv-cpu)
  /unattached (container)
    /io[0] (memory-region)
    /sysbus (System)
    /system[0] (memory-region)
(qemu) info mtree
address-space: memory
  0000000000000000-ffffffffffffffff (prio 0, i/o): system

address-space: I/O
  0000000000000000-000000000000ffff (prio 0, i/o): io

address-space: cpu-memory-0
  0000000000000000-ffffffffffffffff (prio 0, i/o): system
```

```
static void memory_map_init(void)
{
    system_memory = g_malloc(sizeof(*system_memory));

    memory_region_init(system_memory, NULL, "system", UINT64_MAX);
    address_space_init(&address_space_memory, system_memory, "memory");

    system_io = g_malloc(sizeof(*system_io));
    memory_region_init_io(system_io, NULL, &unassigned_io_ops, NULL, "io",
                          65536);
    address_space_init(&address_space_io, system_io, "I/O");
}
```

# 内存虚拟化

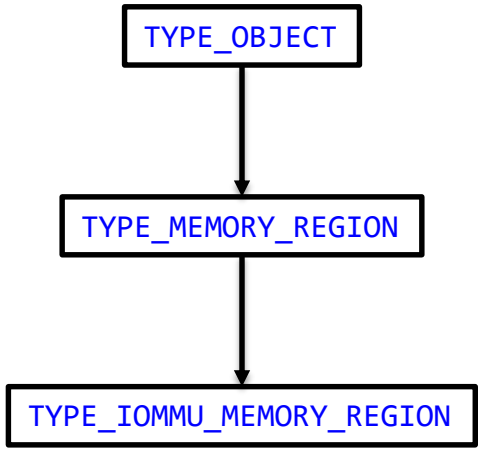
include/exec/memory.h

```
#define TYPE_MEMORY_REGION "memory-region"
DECLARE_INSTANCE_CHECKER(MemoryRegion, MEMORY_REGION,
                         TYPE_MEMORY_REGION)
```

```
static const TypeInfo memory_region_info = {
    .parent             = TYPE_OBJECT,
    .name               = TYPE_MEMORY_REGION,
    .class_size         = sizeof(MemoryRegionClass),
    .instance_size      = sizeof(MemoryRegion),
    .instance_init      = memory_region_initfn,
    .instance_finalize  = memory_region_finalize,
};
```

```
static void memory_register_types(void)
{
    type_register_static(&memory_region_info);
    type_register_static(&iommu_memory_region_info);
}

type_init(memory_register_types)
```




```
struct MemoryRegion {
    Object parent_obj;
    bool romd_mode;
    bool ram;
    bool subpage;
    bool readonly; /* For RAM regions */
    bool nonvolatile;
    bool rom_device;
    bool flush_coalesced_mmio;
    uint8_t dirty_log_mask;
    bool is_iommu;
    RAMBlock *ram_block;  //GPA
    Object *owner;
    const MemoryRegionOps *ops;  //TLB_MMIO
    void *opaque;
    MemoryRegion *container;
    Int128 size;
    hwaddr addr;
    void (*destructor)(MemoryRegion *mr);
    uint64_t align;
    bool terminates;
    bool ram_device;
    bool enabled;
    bool warning_printed; /* For reservations */
    uint8_t vga_logging_count;
    MemoryRegion *alias;
    hwaddr alias_offset;
    int32_t priority;
    QTAILQ_HEAD(, MemoryRegion) subregions;
    QTAILQ_ENTRY(MemoryRegion) subregions_link;
    QTAILQ_HEAD(, CoalescedMemoryRange) coalesced;
    const char *name;
    unsigned ioeventfd_nb;
    MemoryRegionIoeventfd *ioeventfds;
};
```

# 内存虚拟化

include/exec/memory.h

```
struct AddressSpace {
    /* private: */
    struct rcu_head rcu;
    char *name;
    MemoryRegion *root;

    /* Accessed via RCU.  */
    struct FlatView *current_map;

    int ioeventfd_nb;
    struct MemoryRegionIoeventfd *ioeventfds;
    QTAILQ_HEAD(, MemoryListener) listeners;
    QTAILQ_ENTRY(AddressSpace) address_spaces_link;
};
```

```
struct FlatView {
    struct rcu_head rcu;
    unsigned ref;
    FlatRange *ranges;
    unsigned nr;
    unsigned nr_allocated;
    struct AddressSpaceDispatch *dispatch;
    MemoryRegion *root;
};
```

```
struct AddressSpaceDispatch {
    MemoryRegionSection *mru_section;
    PhysPageEntry phys_map;
    PhysPageMap map;
};
```

```
/* Range of memory in the global
map.  Addresses are absolute. */
struct FlatRange {
    MemoryRegion *mr;
    hwaddr offset_in_region;
    AddrRange addr;
    uint8_t dirty_log_mask;
    bool romd_mode;
    bool readonly;
    bool nonvolatile;
};
```

```
struct AddrRange {
    Int128 start;
    Int128 size;
};
```

“平坦化”接口（无环图->数组GPA）：

memory_region_transaction_begin/commit
>>flatviews_reset
  >>generate_memory_topology
    >>render_memory_region

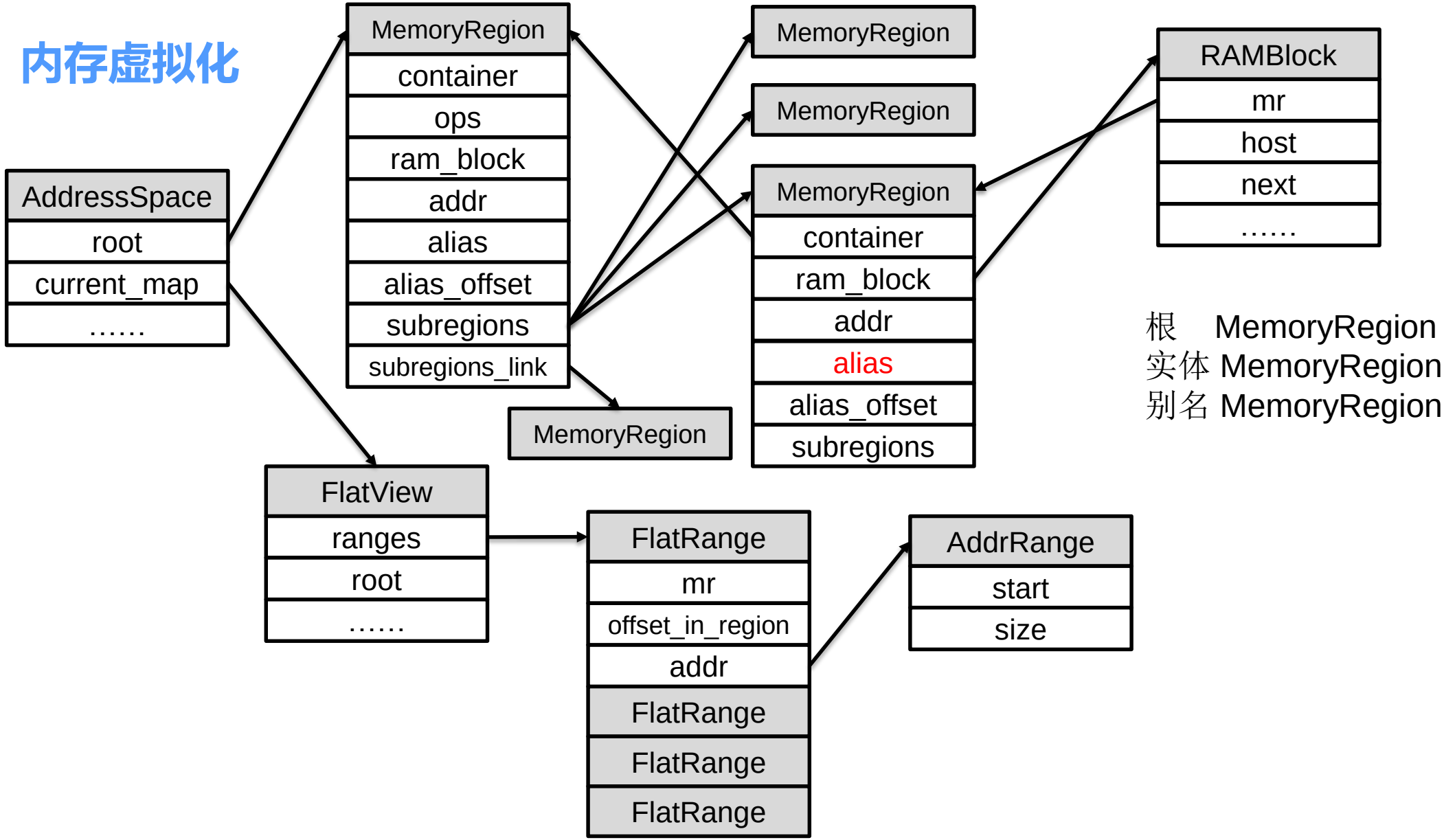
GVA->GPA

get_page_addr_code

get_page_addr_code_hostp

```
static struct TCGCPUOps riscv_tcg_ops = {
    ......
    .tlb_fill = riscv_cpu_tlb_fill,
};
```

# 内存虚拟化




根 MemoryRegion
实体 MemoryRegion
别名 MemoryRegion

# 内存模拟

hw/riscv/sifive_e.c

```
/* Mask ROM */
memory_region_init_rom(&s->mask_rom, OBJECT(dev), "riscv.sifive.e.mrom",
                       memmap[SIFIVE_E_DEV_MROM].size, &error_fatal);
memory_region_add_subregion(sys_mem,
        memmap[SIFIVE_E_DEV_MROM].base, &s->mask_rom);
```

ROM初始化

```
/* Data Tightly Integrated Memory */
memory_region_init_ram(main_mem, NULL, "riscv.sifive.e.ram",
    memmap[SIFIVE_E_DEV_DTIM].size, &error_fatal);
memory_region_add_subregion(sys_mem,
    memmap[SIFIVE_E_DEV_DTIM].base, main_mem);
```

RAM初始化

```
/* Map GPIO registers */
sysbus_mmio_map(SYS_BUS_DEVICE(&s->gpio), 0, memmap[SIFIVE_E_DEV_GPIO0].base);
```

MMIO初始化/MAP ?

```
rom_add_blob_fixed_as("mrom.reset", reset_vec, sizeof(reset_vec),
                      memmap[SIFIVE_U_DEV_MROM].base, &address_spa
ce_memory);
```

ROM操作

# 内存模拟

## MemoryRegion ROOT — system_memory:

```
MemoryRegion *get_system_memory(void)
```

## RAM初始化:

```
void memory_region_init_ram(MemoryRegion *mr,
                            struct Object *owner,
                            const char *name,
                            uint64_t size,
                            Error **errp)
```

## ROM初始:

```
void memory_region_init_rom(MemoryRegion *mr,
                            struct Object *owner,
                            const char *name,
                            uint64_t size,
                            Error **errp)
```

## 分配/挂载:

```
void memory_region_add_subregion(MemoryRegion *mr,
                                 hwaddr offset,
                                 MemoryRegion *subregion)
```

## MMIO:

```
void memory_region_init_io(MemoryRegion *mr,
                           Object *owner,
                           const MemoryRegionOps *ops,
                           void *opaque,
                           const char *name,
                           uint64_t size)
void sysbus_init_mmio(SysBusDevice *dev, MemoryRegion *memory)
```

设备实现

```
void sysbus_mmio_map(SysBusDevice *dev, int n, hwaddr addr)
```

```
/*
 * Memory region callbacks
 */
struct MemoryRegionOps {
    /* Read from the memory region. @addr is relative to
@mr; @size is
     * in bytes. */
    uint64_t (*read)(void *opaque,
                     hwaddr addr,
                     unsigned size);
    /* Write to the memory region. @addr is relative to @
mr; @size is
     * in bytes. */
    void (*write)(void *opaque,
                  hwaddr addr,
                  uint64_t data,
                  unsigned size);
    ……
}
```

# 内存模拟

hw/riscv/sifive_e.c

```
/* Mask ROM */
memory_region_init_rom(&s->mask_rom, OBJECT(dev), "riscv.sifive.e.mrom",
                       memmap[SIFIVE_E_DEV_MROM].size, &error_fatal);
memory_region_add_subregion(sys_mem,
        memmap[SIFIVE_E_DEV_MROM].base, &s->mask_rom);
```

```
/* Data Tightly Integrated Memory */
memory_region_init_ram(main_mem, NULL, "riscv.sifive.e.ram",
    memmap[SIFIVE_E_DEV_DTIM].size, &error_fatal);
memory_region_add_subregion(sys_mem,
    memmap[SIFIVE_E_DEV_DTIM].base, main_mem);
```

```
memory_region_init_rom
 >>memory_region_init_rom_nomigrate
   >>memory_region_init_ram_shared_nomigrate
     >>memory_region_init
       mr->ram_block = qemu_ram_alloc(size, share, mr, &err);
```

初始化建立 MR

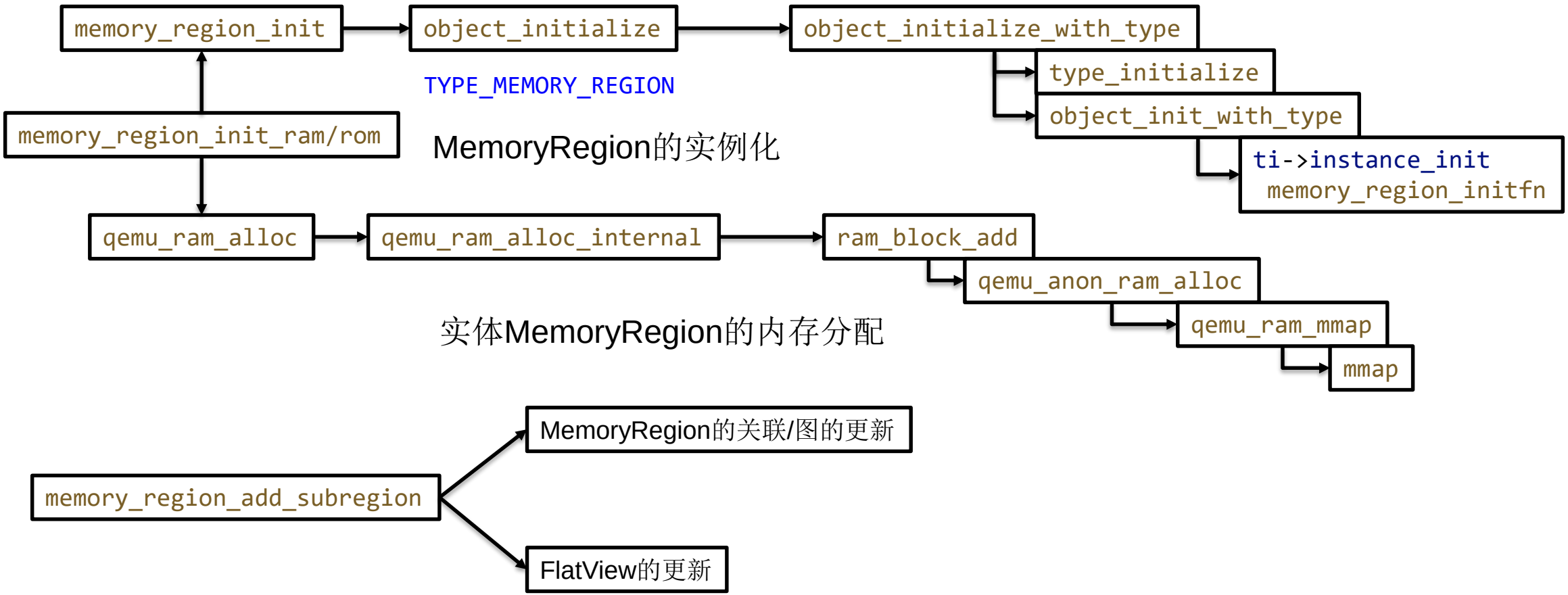
```
memory_region_add_subregion
 >>memory_region_add_subregion_common
   >>memory_region_update_container_subregions
     >>memory_region_transaction_begin/commit
       >>flatviews_reset
```

```
generate_memory_topology
```

```
render_memory_region
```

分派/更新FlatView

# 内存模拟


memory_region_init
object_initialize
object_initialize_with_type
type_initialize
object_init_with_type
ti->instance_init
memory_region_initfn
TYPE_MEMORY_REGION
memory_region_init_ram/rom
MemoryRegion的实例化
qemu_ram_alloc
qemu_ram_alloc_internal
ram_block_add
qemu_anon_ram_alloc
qemu_ram_mmap
mmap
实体MemoryRegion的内存分配
memory_region_add_subregion
MemoryRegion的关联/图的更新
FlatView的更新

# 内存模拟之创建

hw/riscv/sifive_e.c

```
/* GPIO */
if (!sysbus_realize(SYS_BUS_DEVICE(&s->gpio), errp)) {
        return;
    }

/* Map GPIO registers */
sysbus_mmio_map(SYS_BUS_DEVICE(&s->gpio),
    0, memmap[SIFIVE_E_DEV_GPIO0].base);
```

```
bool sysbus_realize(SysBusDevice *dev, Error **errp)
{
    return qdev_realize(DEVICE(dev), sysbus_get_default(), errp);
}
```

```
struct SIFIVEGPIOState {
    SysBusDevice parent_obj;
    MemoryRegion mmio;
    qemu_irq irq[SIFIVE_GPIO_PINS];
    qemu_irq output[SIFIVE_GPIO_PINS];

    uint32_t value;
    uint32_t input_en;
    uint32_t output_en;
    ……
};
```

```
struct SysBusDevice {
    /*< private >*/
    DeviceState parent_obj;
    /*< public >*/
    int num_mmio;
    struct {
        hwaddr addr;
        MemoryRegion *memory;
    } mmio[QDEV_MAX_MMIO];
    int num_pio;
    uint32_t pio[QDEV_MAX_PIO];
};
```

```
static void sifive_gpio_realize(DeviceState *dev, Error **errp)
{
    SIFIVEGPIOState *s = SIFIVE_GPIO(dev);

    memory_region_init_io(&s->mmio, OBJECT(dev), &gpio_ops, s,
            TYPE_SIFIVE_GPIO, SIFIVE_GPIO_SIZE);

    sysbus_init_mmio(SYS_BUS_DEVICE(dev), &s->mmio);

    ……
}
```

```
static const MemoryRegionOps gpio_ops = {
    .read =  sifive_gpio_read,
    .write = sifive_gpio_write,
    .endianness = DEVICE_LITTLE_ENDIAN,
    .impl.min_access_size = 4,
    .impl.max_access_size = 4,
};
```

sysbus_mmio_map → sysbus_mmio_map_common → memory_region_add_subregion

# 内存模拟之加载

include/hw/loader.h

```
#define rom_add_blob_fixed_as(_f, _b, _l, _a, _as)      \
    rom_add_blob(_f, _b, _l, _l, _a, NULL, NULL, NULL, _as, true)
```

```
/* Mask ROM reset vector */
    uint32_t reset_vec[4];

    if (s->revb) {
        reset_vec[1] = 0x200102b7;  /* 0x1004: lui     t0,0x20010 */
    } else {
        reset_vec[1] = 0x204002b7;  /* 0x1004: lui     t0,0x20400 */
    }
    reset_vec[2] = 0x00028067;      /* 0x1008: jr      t0 */

    reset_vec[0] = reset_vec[3] = 0;

    /* copy in the reset vector in little_endian byte order */
    for (i = 0; i < sizeof(reset_vec) >> 2; i++) {
        reset_vec[i] = cpu_to_le32(reset_vec[i]);
    }
    rom_add_blob_fixed_as("mrom.reset", reset_vec, sizeof(reset_vec),
                          memmap[SIFIVE_E_DEV_MROM].base, &address_space_memory);

    if (machine->kernel_filename) {
        riscv_load_kernel(machine->kernel_filename,
                          memmap[SIFIVE_E_DEV_DTIM].base, NULL);
    }
```

```
/* Default Reset Vector adress */
#define DEFAULT_RSTVEC      0x1000

static void rv32_nuclei_n_cpu_init(Object *obj)
{
    ......
    set_resetvec(env, DEFAULT_RSTVEC);
    ......
}
```

freedom-e-sdk/bsp/qemu-sifive-e31/metal.default.lds

```
MEMORY
{
  ram (airwx) : ORIGIN = 0x80000000, LENGTH = 0x400000
  rom (irx!wa) : ORIGIN = 0x20400000, LENGTH = 0x1fc00000
}
```

# 内存模拟之加载

Linux(MMU)系统模拟式：

```
firmware_end_addr = riscv_find_and_load_firmware(machine,
                        "opensbi-riscv64-generic-fw_dynamic.bin",
                         start_addr, NULL);
```

OpenSBI

```
kernel_start_addr = riscv_calc_kernel_start_addr(&s->soc.u_cpus,firmware_end_addr);

kernel_entry = riscv_load_kernel(machine->kernel_filename,kernel_start_addr, NULL);

if (machine->initrd_filename) {
      hwaddr end = riscv_load_initrd(machine->initrd_filename,machine->ram_size,
                        kernel_entry,&start);
……
```

Uboot

Linux Kernel

Initrd

```
fdt_load_addr = riscv_load_fdt(memmap[SIFIVE_U_DEV_DRAM].base,machine->ram_size, s->fdt);
```

dtb

```
rom_add_blob_fixed_as("mrom.reset", reset_vec, sizeof(reset_vec),
                        memmap[SIFIVE_U_DEV_MROM].base, &address_space_memory);
riscv_rom_copy_firmware_info(machine, memmap[SIFIVE_U_DEV_MROM].base,
                        memmap[SIFIVE_U_DEV_MROM].size,
                        sizeof(reset_vec), kernel_entry);
```

ROM

OpenSBI INFO

# Nuclei 内存模拟之Memmap

**Table 5-1 Address Allocation of SoC**

| | Component | Address Spaces | Description |
|---|---|---|---|
| Core Private Peripherals | TIMER | 0x0200_0000 ~ 0x0200_0FFF | TIMER Unit address space. |
| | ECLIC | 0x0C00_0000 ~ 0x0C00_FFFF | ECLIC Unit address space. |
| | DEBUG | 0x0000_0000 ~ 0x0000_0FFF | DEBUG Unit address space. |
| Memory Resource | ILM | 0x8000_0000 ~ | ILM address space. |
| | DLM | 0x9000_0000 ~ | DLM address space. |
| | ROM | 0x0000_1000 ~ 0x0000_1FFF | Internal ROM. |
| | Off-Chip QSPI0 Flash Read | 0x2000_0000 ~ 0x3FFF_FFFF | QSPI0 with XiP mode read-only address space. |
| Peripherals | GPIO | 0x1001_2000 ~ 0x1001_2FFF | GPIO Unit address space. |
| | UART0 | 0x1001_3000 ~ 0x1001_3FFF | First UART address space. |
| | QSPI0 | 0x1001_4000 ~ 0x1001_4FFF | First QSPI address space. |
| | PWM0 | 0x1001_5000 ~ 0x1001_5FFF | First PWM address space. |
| | UART1 | 0x1002_3000 ~ 0x1002_3FFF | Second UART address space. |
| | QSPI1 | 0x1002_4000 ~ 0x1002_4FFF | Second QSPI address space. |
| | PWM1 | 0x1002_5000 ~ 0x1002_5FFF | Second PWM address space. |
| | QSPI2 | 0x1003_4000 ~ 0x1003_4FFF | Third QSPI address space. |
| | PWM2 | 0x1003_5000 ~ 0x1003_5FFF | Third PWM address space. |
| | I2C Master | 0x1004_2000 ~ 0x1004_2FFF | I2C Master address space. |
| Default slave | The other space is write-ignored and read-as zero. | | |

```
typedef struct MemMapEntry {
    hwaddr base;
    hwaddr size;
} MemMapEntry;
```

hw/riscv/nuclei_n.c

```
static MemMapEntry nuclei_n_memmap[] = {
    [NUCLEI_N_DEBUG] = {0x0, 0x1000},
    [NUCLEI_N_ROM] = {0x1000, 0x1000},
    [NUCLEI_N_TIMER] = {0x2000000, 0x1000},
    [NUCLEI_N_ECLIC] = {0xc000000, 0x10000},
    [NUCLEI_N_GPIO] = {0x10012000, 0x1000},
    [NUCLEI_N_UART0] = {0x10013000, 0x1000},
    [NUCLEI_N_QSPI0] = {0x10014000, 0x1000},
    [NUCLEI_N_PWM0] = {0x10015000, 0x1000},
    [NUCLEI_N_UART1] = {0x10023000, 0x1000},
    [NUCLEI_N_QSPI1] = {0x10024000, 0x1000},
    [NUCLEI_N_PWM1] = {0x10025000, 0x1000},
    [NUCLEI_N_QSPI2] = {0x10034000, 0x1000},
    [NUCLEI_N_PWM2] = {0x10035000, 0x1000},
    [NUCLEI_N_XIP] = {0x20000000, 0x10000000},
    [NUCLEI_N_DRAM] = {0xa0000000, 0x0},
    [NUCLEI_N_ILM] = {0x80000000, 0x20000},
    [NUCLEI_N_DLM] = {0x90000000, 0x20000},
};
```

```
static MemMapEntry nuclei_u_memmap[] = {
    ......
    [NUCLEI_U_CLINT] = {0x2001000, 0x10000},
    [NUCLEI_U_PLIC] = {0x8000000, 0x4000000},
    ......
    [NUCLEI_U_DRAM] = {0xa0000000, 0x0},
};
```

hw/riscv/nuclei_u.c

# Nuclei 内存模拟之初始化

hw/riscv/nuclei_n.c

ROM初始化 nuclei_n_soc_realize

```
MemoryRegion *sys_mem = get_system_memory();
/* Internal ROM */
memory_region_init_rom(&s->internal_rom, OBJECT(dev), "riscv.nuclei.n.irom",
                       memmap[NUCLEI_N_ROM].size, &error_fatal);
memory_region_add_subregion(sys_mem,
                       memmap[NUCLEI_N_ROM].base, &s->internal_rom);
```

创建unimplemented设备

```
/* SysTimer */
create_unimplemented_device("riscv.nuclei.n.timer",
        memmap[NUCLEI_N_TIMER].base, memmap[NUCLEI_N_TIMER].size);

/* Eclic */
create_unimplemented_device("riscv.nuclei.n.eclic",
        memmap[NUCLEI_N_ECLIC].base, memmap[NUCLEI_N_ECLIC].size);

/* GPIO */
create_unimplemented_device("riscv.nuclei.n.gpio",
        memmap[NUCLEI_N_GPIO].base, memmap[NUCLEI_N_GPIO].size);
……
```

```
static inline void create_unimplemented_device(const char *name,
                                               hwaddr base,
                                               hwaddr size)
{
    DeviceState *dev = qdev_new(TYPE_UNIMPLEMENTED_DEVICE);

    qdev_prop_set_string(dev, "name", name);
    qdev_prop_set_uint64(dev, "size", size);
    sysbus_realize_and_unref(SYS_BUS_DEVICE(dev), &error_fatal);

    sysbus_mmio_map_overlap(SYS_BUS_DEVICE(dev), 0, base, -1000);
}
```

# Nuclei 内存模拟之初始化

hw/riscv/nuclei_n.c

RAM初始化 nuclei_mcu_machine_init

```
MemoryRegion *sys_mem = get_system_memory();
/* Initialize ilm dlm */
memory_region_init_ram(&s->soc.ilm, NULL, "riscv.nuclei.n.ilm",
                       memmap[NUCLEI_N_ILM].size, &error_fatal);
memory_region_add_subregion(sys_mem,
                            memmap[NUCLEI_N_ILM].base, &s->soc.ilm);
memory_region_init_ram(&s->soc.dlm, NULL, "riscv.nuclei.n.dlm",
                       memmap[NUCLEI_N_DLM].size, &error_fatal);

……
```

```
switch (s->msel)
{
  case MSEL_ILM:
       start_addr = memmap[NUCLEI_N_ILM].base;
       break;
  ……
  case MSEL_DDR:
       start_addr = memmap[NUCLEI_N_DRAM].base;
       break;
  default:
       start_addr = memmap[NUCLEI_N_ILM].base;
       break;
}
```

ROM设置和kernel加载：

```
/* reset vector */
uint32_t reset_vec[8] = {
    0x00000297, /* 1:  auipc  t0, %pcrel_hi(dtb) */
    0x02028593, /*     addi   a1, t0, %pcrel_lo(1b) */
    0xf1402573, /*     csrr   a0, mhartid  */
#if defined(TARGET_RISCV32)
    0x0182a283, /*     lw     t0, 24(t0) */
#elif defined(TARGET_RISCV64)
    0x0182b283, /*     ld     t0, 24(t0) */
#endif
    0x00028067, /*     jr     t0 */
    0x00000000,
    start_addr, /* start: .dword DRAM_BASE */
    0x00000000,
};
/* copy in the reset vector in little_endian byte order */
for (i = 0; i < sizeof(reset_vec) >> 2; i++)
{
    reset_vec[i] = cpu_to_le32(reset_vec[i]);
}
rom_add_blob_fixed_as("mrom.reset", reset_vec, sizeof(reset
_vec),memmap[NUCLEI_N_ROM].base, &address_space_memory);

/* boot rom */
if (machine->kernel_filename)
{
    riscv_load_kernel(machine>kernel_filename, start_addr,
                      NULL);
}
```

# Nuclei 内存模拟之创建

测试命令：

```
$qemu-system-riscv32 \
-nographic -M mcu_200t \
-kernel helloworld.elf
```

```
(qemu) info qom-tree
/machine (mcu_200t-machine)
  /peripheral (container)
  /peripheral-anon (container)
  /soc (riscv.nuclei.n.soc)
    /cpus (riscv.hart_array)
      /harts[0] (nuclei-n600-riscv-cpu)
    /riscv.nuclei.n.irom[0] (memory-region)
  /unattached (container)
    /device[0] (unimplemented-device)
      /riscv.nuclei.n.timer[0] (memory-region)
    /device[10] (unimplemented-device)
      /riscv.nuclei.n.pwm2[0] (memory-region)
    /device[1] (unimplemented-device)
      /riscv.nuclei.n.eclic[0] (memory-region)
    /device[2] (unimplemented-device)
      /riscv.nuclei.n.gpio[0] (memory-region)
    /device[3] (unimplemented-device)
      /riscv.nuclei.n.uart0[0] (memory-region)
    /device[4] (unimplemented-device)
      /riscv.nuclei.n.uart1[0] (memory-region)
    /device[5] (unimplemented-device)
      /riscv.nuclei.n.qspi0[0] (memory-region)
    /device[6] (unimplemented-device)
      /riscv.nuclei.n.qspi1[0] (memory-region)
    /device[7] (unimplemented-device)
      /riscv.nuclei.n.qspi2[0] (memory-region)
    /device[8] (unimplemented-device)
      /riscv.nuclei.n.pwm0[0] (memory-region)
    /device[9] (unimplemented-device)
      /riscv.nuclei.n.pwm1[0] (memory-region)
    /io[0] (memory-region)
    /riscv.nuclei.n.dlm[0] (memory-region)
    /riscv.nuclei.n.dram[0] (memory-region)
    /riscv.nuclei.n.ilm[0] (memory-region)
    /riscv.nuclei.n.xip[0] (memory-region)
    /sysbus (System)
    /system[0] (memory-region)
```

```
QEMU 5.2.90 monitor - type 'help' for more information
(qemu) info mtree
address-space: memory
  0000000000000000-ffffffffffffffff (prio 0, i/o): system
    0000000000001000-0000000000001fff (prio 0, rom): riscv.nuclei.n.irom
    0000000002000000-0000000002000fff (prio -1000, i/o): riscv.nuclei.n.timer
    000000000c000000-000000000c00ffff (prio -1000, i/o): riscv.nuclei.n.eclic
    0000000010012000-0000000010012fff (prio -1000, i/o): riscv.nuclei.n.gpio
    0000000010013000-0000000010013fff (prio -1000, i/o): riscv.nuclei.n.uart0
    0000000010014000-0000000010014fff (prio -1000, i/o): riscv.nuclei.n.qspi0
    0000000010015000-0000000010015fff (prio -1000, i/o): riscv.nuclei.n.pwm0
    0000000010023000-0000000010023fff (prio -1000, i/o): riscv.nuclei.n.uart1
    0000000010024000-0000000010024fff (prio -1000, i/o): riscv.nuclei.n.qspi1
    0000000010025000-0000000010025fff (prio -1000, i/o): riscv.nuclei.n.pwm1
    0000000010034000-0000000010034fff (prio -1000, i/o): riscv.nuclei.n.qspi2
    0000000010035000-0000000010035fff (prio -1000, i/o): riscv.nuclei.n.pwm2
    0000000020000000-000000002fffffff (prio 0, ram): riscv.nuclei.n.xip
    0000000080000000-000000008001ffff (prio 0, ram): riscv.nuclei.n.ilm
    0000000090000000-000000009001ffff (prio 0, ram): riscv.nuclei.n.dlm
    00000000a0000000-00000000a7ffffff (prio 0, ram): riscv.nuclei.n.dram

address-space: I/O
  0000000000000000-000000000000ffff (prio 0, i/o): io

address-space: cpu-memory-0
  0000000000000000-ffffffffffffffff (prio 0, i/o): system
    0000000000001000-0000000000001fff (prio 0, rom): riscv.nuclei.n.irom
    0000000002000000-0000000002000fff (prio -1000, i/o): riscv.nuclei.n.timer
    000000000c000000-000000000c00ffff (prio -1000, i/o): riscv.nuclei.n.eclic
    0000000010012000-0000000010012fff (prio -1000, i/o): riscv.nuclei.n.gpio
    0000000010013000-0000000010013fff (prio -1000, i/o): riscv.nuclei.n.uart0
    0000000010014000-0000000010014fff (prio -1000, i/o): riscv.nuclei.n.qspi0
    0000000010015000-0000000010015fff (prio -1000, i/o): riscv.nuclei.n.pwm0
    0000000010023000-0000000010023fff (prio -1000, i/o): riscv.nuclei.n.uart1
    0000000010024000-0000000010024fff (prio -1000, i/o): riscv.nuclei.n.qspi1
    0000000010025000-0000000010025fff (prio -1000, i/o): riscv.nuclei.n.pwm1
    0000000010034000-0000000010034fff (prio -1000, i/o): riscv.nuclei.n.qspi2
    0000000010035000-0000000010035fff (prio -1000, i/o): riscv.nuclei.n.pwm2
    0000000020000000-000000002fffffff (prio 0, ram): riscv.nuclei.n.xip
    0000000080000000-000000008001ffff (prio 0, ram): riscv.nuclei.n.ilm
    0000000090000000-000000009001ffff (prio 0, ram): riscv.nuclei.n.dlm
    00000000a0000000-00000000a7ffffff (prio 0, ram): riscv.nuclei.n.dram
```

# Nuclei 内存模拟之加载运行

测试命令：

```
qemu-system-riscv32  \
-nographic -M mcu_200t,msel=1  \
-d in_asm \
 -kernel ../hbird/ilm/helloworld.elf
```

```
QEMU 5.2.90 monitor - type 'help' for more information
(qemu) ---------------
IN:
Priv: 3; Virt: 0
0x00001000:  00000297          auipc           t0,0            # 0x1000
0x00001004:  02028593          addi            a1,t0,32
0x00001008:  f1402573          csrrs           a0,mhartid,zero

---------------
IN:
Priv: 3; Virt: 0
0x0000100c:  0182a283          lw              t0,24(t0)
0x00001010:  00028067          jr              t0

---------------
IN:
Priv: 3; Virt: 0
0x80000000:  0cc0006f          j               204             # 0x800000cc

---------------
IN: _start
Priv: 3; Virt: 0
0x800000cc:  30047073          csrrci          zero,mstatus,8

---------------
IN: _start
Priv: 3; Virt: 0
0x800000d0:  10000197          auipc           gp,268435456    # 0x900000d0
0x800000d4:  79018193          addi            gp,gp,1936
0x800000d8:  10010117          auipc           sp,268500992    # 0x900100d8
0x800000dc:  f2810113          addi            sp,sp,-216
0x800000e0:  20000293          addi            t0,zero,512
0x800000e4:  7d02a073          csrrs           zero,0x7d0,t0
```

# Nuclei 内存模拟之加载运行

hw/riscv/nuclei_u.c： 类似的ROM RAM DDR创建，不一样的加载

```
firmware_end_addr = riscv_find_and_load_firmware(machine, BIOS_FILENAME,
                                                 start_addr, NULL);
```

OpenSBI

```
kernel_start_addr = riscv_calc_kernel_start_addr(&s->soc.cpus,
                                                 firmware_end_addr);
kernel_entry = riscv_load_kernel(machine->kernel_filename,
                                 kernel_start_addr, NULL);
hwaddr end = riscv_load_initrd(machine->initrd_filename,
                               machine->ram_size, kernel_entry,
                                    &start);
……
```

Uboot

Linux Kernel

Initrd

```
fdt_load_addr = riscv_load_fdt(memmap[NUCLEI_U_DRAM].base,machine->ram_size, s->fdt);
```

dtb ← create_fdt

```
/* load the reset vector */
riscv_setup_rom_reset_vec(machine, &s->soc.cpus, start_addr,
                          memmap[NUCLEI_U_MROM].base,
                          memmap[NUCLEI_U_MROM].size, kernel_entry,
                          fdt_load_addr, s->fdt);
```

ROM

OpenSBI INFO

# 下节课内容：

## 中断虚拟化

- QEMU RISCV IRQ中断介绍
- Timer中断介绍
- Eclic中断介绍

## 外设虚拟化

- Nuclei Uart设备实现
- Nuclei Timer设备实现
- Nuclei Eclic设备实现

# 谢谢

wangjunqiang@iscas.ac.cn