

1.新建quard-star开发板 | TimerのBlog

👤 yanglianoo.github.io/2023/06/12/QEMU中自定义开发板

2023年6月12日

1. QEMU中新增虚拟开发板

参考链接：基于qemu-riscv从0开始构建嵌入式linux系统ch2. 添加qemu仿真板——Quard-Star板 — 主页 (quard-star-tutorial.readthedocs.io)

本文主要参考了上面的博文，复现一下在qemu中自定义板卡的过程，用于个人学习。

前言：qemu内置支持了一些开发板，我们可以基于这些内置的板子来做操作系统等软件的配置，但是实际市面上很多板子qemu中是没有提供支持的，要是直接在硬件中进行软件验证会十分麻烦，还好qemu中可以支持用户自定义开发板，这样就可以虚拟的对开发板进行验证了。

在向qemu中注册自定义的板子需要向qemu中添加源码，然后重新编译，qemu源码安装的编译过程我的这一篇博客：从源码构建Qemu | TimerのBlog (yanglianoo.github.io)

我们添加的板子cpu架构为riscv，进入qemu源码的hw/riscv目录下，可以看到如下图中的文件，其中qemu官方默认添加了几个riscv板子，比如：virt, sifive，其中virt，这块虚拟板子也是最常用的虚拟板子，常用于作为基于riscv操作系统的原型验证，比如xv6, rvos都是基于qemu-virt构建的。

```
timer@DESKTOP-JI9EVEH:~/qemu/qemu-8.0.2/hw/riscv$ ls
Kconfig  meson.build  numa.c      riscv_hart.c  sifive_e.c  spike.c      virt.c
boot.c   microchip_pfsoc.c  opentitan.c  shakti_c.c    sifive_u.c  virt-acpi-build.c
```

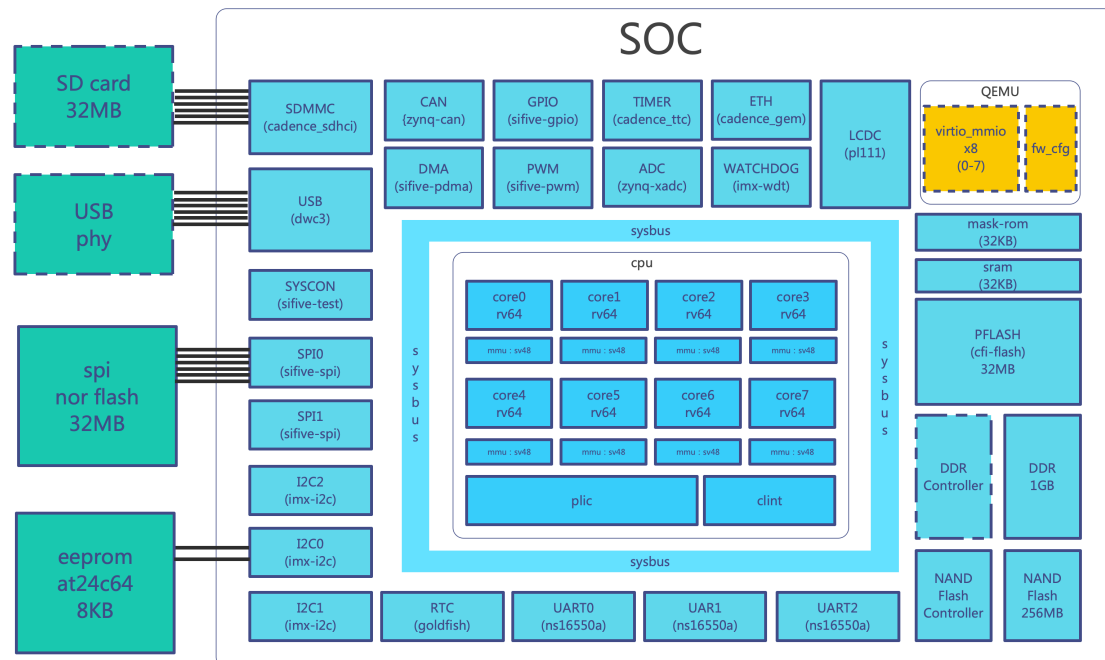
再进入include/hw/riscv目录，这里放着虚拟板卡的头文件：

```
timer@DESKTOP-JI9EVEH:~/qemu/qemu-8.0.2/include/hw/riscv$ ls
boot.h          microchip_pfsoc.h  opentitan.h  shakti_c.h    sifive_e.h  spike.h
boot_opensbi.h  numa.h             riscv_hart.h  sifive_cpu.h  sifive_u.h  virt.h
```

2.quard_star开发板的硬件资源

我们将自定义的开发板名字命名为quard_star，理论上这块板子的硬件资源你可以随便定义，开发板的资源如下：

Quard-Star-Board



- CPU: 8个64位的hart, **mmu**的虚拟地址翻译模式为**sv48**
- plic: 平台级中断控制器
- clint: 内部中断控制器
- mask-rom: 32KB
- PFLASH: 32MB
- DDR: 1GB
- NANO Flash: 256MB
- CAN:
- DMA:
- TIMER:
- ETH:
- USB:
- SPI:
- IIC:
- ADC:

- GPIO:
- SDMMC:
- UART:
- RTC:
- WATHDOG:

我们的目标就是在qemu中创建这样一个开发板，需要依次在qemu中定义每一个硬件。接下来就来逐步添加每一个硬件

3.添加quard-star板子

声明：以下使用的qemu源码版本均为qemu-8.0.2，参考代码为qemu的virt这块板子的代码，我们需要先让qemu识别到quard-star这块板子，然后再逐渐丰富板子的外设。

注册quard-star板子需要修改如下几个文件：

plaintext

```
`qemu-8.0.2/configs/devices/riscv64-softmmu/default.mak`  
`qemu-8.0.2/configs/devices/riscv32-softmmu/default.mak`  
`qemu-8.0.2/hw/riscv/meson.build`  
`qemu-8.0.2/hw/riscv/Kconfig`
```

要在qemu中定义自己的虚拟板卡，需要在hw/riscv目录下增加一个和自己板子相关的.c文件，以及在include/hw/riscv目录下添加一个对应虚拟板子的.h文件，这里新增quard_star.c和quard_star.h文件，将其加入qemu编译体系内。

3.1 文件修改

qemu-8.0.2/hw/riscv/meson.build

```

hw > riscv > meson.build
1  riscv_ss = ss.source_set()
2  riscv_ss.add(files('boot.c'), fdt)
3  riscv_ss.add(when: 'CONFIG_RISCV_NUMA', if_true: files('numa.c'))
4  riscv_ss.add(files('riscv_hart.c'))
5  riscv_ss.add(when: 'CONFIG_OPENTITAN', if_true: files('opentitan.c'))
6  riscv_ss.add(when: 'CONFIG_RISCV_VIRT', if_true: files('virt.c'))
7  riscv_ss.add(when: 'CONFIG_SHAKTI_C', if_true: files('shakti_c.c'))
8  riscv_ss.add(when: 'CONFIG_SIFIVE_E', if_true: files('sifive_e.c'))
9  riscv_ss.add(when: 'CONFIG_SIFIVE_U', if_true: files('sifive_u.c'))
10 riscv_ss.add(when: 'CONFIG_SPIKE', if_true: files('spike.c'))
11 riscv_ss.add(when: 'CONFIG_MICROCHIP_PFSOC', if_true: files('microchip_pfsoc.c'))
12 riscv_ss.add(when: 'CONFIG_ACPI', if_true: files('virt-acpi-build.c'))
13 riscv_ss.add(when: 'CONFIG_QUARD_STAR', if_true: files('quard_star.c'))
14
15 hw_arch += {'riscv': riscv_ss}
16

```

qemu-8.0.2/hw/riscv/Kconfig: 这里只暂时只选中一个串口设备。

```

81  config SPIKE
82      bool
83      select RISCV_NUMA
84      select HTIF
85      select RISCV_ACLINT
86      select SIFIVE_PLIC
87
88  config QUARD_STAR
89      bool
90      select SERIAL

```

qemu-8.0.2/configs/devices/riscv32-softmmu/default.mak


```

configs > devices > riscv32-softmmu > M default.mak
1  # Default configuration for riscv32-softmmu
2
3  # Uncomment the following lines to disable these optional devices:
4  #
5  #CONFIG_PCI_DEVICES=n
6  CONFIG_SEMIHOSTING=y
7  CONFIG_ARM_COMPATIBLE_SEMIHOSTING=y
8
9  # Boards:
10 #
11 CONFIG_SPIKE=y
12 CONFIG_SIFIVE_E=y
13 CONFIG_SIFIVE_U=y
14 CONFIG_RISCV_VIRT=y
15 CONFIG_OPENTITAN=y
16 CONFIG_QUARD_STAR=y

```

qemu-8.0.2/configs/devices/riscv64-softmmu/default.mak

```
configs > devices > riscv64-sofmmu > M default.mak
1  # Default configuration for riscv64-sofmmu
2
3  # Uncomment the following lines to disable these optional devices:
4  #
5  #CONFIG_PCI_DEVICES=n
6  CONFIG_SEMIHOSTING=y
7  CONFIG_ARM_COMPATIBLE_SEMIHOSTING=y
8
9  # Boards:
10 #
11 CONFIG_SPIKE=y
12 CONFIG_SIFIVE_E=y
13 CONFIG_SIFIVE_U=y
14 CONFIG_RISCV_VIRT=y
15 CONFIG_MICROCHIP_PFSOC=y
16 CONFIG_SHAKTI_C=y
17 CONFIG_QUARD_STAR=y
18
```



3.2 添加源码

这里先把源码添加上去，后续慢慢分析，源码来自于文章开头参考的项目以及qemu中virt的源码。这里只定义了和初始化了MROM、SRAM、DRAM三种硬件。

3.2.1 quard_star.h

C

```
#ifndef HW_RISCV_QUARD_STAR__H
#define HW_RISCV_QUARD_STAR__H

#include "hw/riscv/riscv_hart.h"
#include "hw/sysbus.h"
#include "qom/object.h"
#include "hw/block/flash.h"

#define QUARD_STAR_CPUS_MAX 8
#define QUARD_STAR_SOCKETS_MAX 8

#define TYPE_RISCV_QUARD_STAR_MACHINE MACHINE_TYPE_NAME("quard-
star")
typedef struct QuardStarState QuardStarState;
DECLARE_INSTANCE_CHECKER(QuardStarState, RISCV_VIRT_MACHINE,
                        TYPE_RISCV_QUARD_STAR_MACHINE)

struct QuardStarState {
    /*< private >*/
    MachineState parent;

    /*< public >*/
    RISCVHartArrayState soc[QUARD_STAR_SOCKETS_MAX];
};

enum {
    QUARD_STAR_MROM,
    QUARD_STAR_SRAM,
    QUARD_STAR_UART0,
    QUARD_STAR_DRAM,
};

enum {
    QUARD_STAR_UART0_IRQ = 10,    //定义了串口中断号为10
};
#endif
```

c

```

#include "qemu/osdep.h"
#include "qemu/units.h"
#include "qemu/error-report.h"
#include "qemu/guest-random.h"
#include "qapi/error.h"
#include "hw/boards.h"
#include "hw/loader.h"
#include "hw/sysbus.h"
#include "hw/qdev-properties.h"
#include "hw/char/serial.h"
#include "target/riscv/cpu.h"

#include "hw/riscv/riscv_hart.h"
#include "hw/riscv/quard_star.h"
#include "hw/riscv/boot.h"
#include "hw/riscv/numa.h"
#include "hw/intc/riscv_aclint.h"
#include "hw/intc/riscv_aplic.h"

#include "chardev/char.h"
#include "sysemu/device_tree.h"
#include "sysemu/sysemu.h"
#include "sysemu/kvm.h"
#include "sysemu/tpm.h"

static const MemMapEntry quard_star_memmap[] = {
    [QUARD_STAR_MROM] = { 0x0, 0x8000 },
    [QUARD_STAR_SRAM] = { 0x8000, 0x8000 },
    [QUARD_STAR_UART0] = { 0x10000000, 0x100 },
    [QUARD_STAR_DRAM] = { 0x80000000, 0x80 },
};
/* 创建CPU */
static void quard_star_cpu_create(MachineState *machine)
{
    int i, base_hartid, hart_count;
    char *soc_name;
    QuardStarState *s = RISC_VIRT_MACHINE(machine);

    if (QUARD_STAR_SOCKETS_MAX < riscv_socket_count(machine)) {
        error_report("number of sockets/nodes should be less than %d",
            QUARD_STAR_SOCKETS_MAX);
        exit(1);
    }

    for (i = 0; i < riscv_socket_count(machine); i++) {
        if (!riscv_socket_check_hartids(machine, i)) {
            error_report("discontinuous hartids in socket%d", i);
            exit(1);
        }

        base_hartid = riscv_socket_first_hartid(machine, i);
        if (base_hartid < 0) {
            error_report("can't find hartid base for socket%d", i);
            exit(1);
        }

        hart_count = riscv_socket_hart_count(machine, i);
        if (hart_count < 0) {

```



```

        error_report("can't find hart count for socket%d", i);
        exit(1);
    }

    soc_name = g_strdup_printf("soc%d", i);
    object_initialize_child(OBJECT(machine), soc_name, &s->soc[i],
                           TYPE_RISCV_HART_ARRAY);
    g_free(soc_name);
    object_property_set_str(OBJECT(&s->soc[i]), "cpu-type",
                            machine->cpu_type, &error_abort);
    object_property_set_int(OBJECT(&s->soc[i]), "hartid-base",
                            base_hartid, &error_abort);
    object_property_set_int(OBJECT(&s->soc[i]), "num-harts",
                            hart_count, &error_abort);
    sysbus_realize(SYS_BUS_DEVICE(&s->soc[i]), &error_abort);
}
}

/* 创建内存 */
static void quard_star_memory_create(MachineState *machine)
{
    QuardStarState *s = RISCV_VIRT_MACHINE(machine);
    MemoryRegion *system_memory = get_system_memory();
    //分配三片存储空间 dram sram mrom
    MemoryRegion *dram_mem = g_new(MemoryRegion, 1); //DRAM
    MemoryRegion *sram_mem = g_new(MemoryRegion, 1); //SRAM
    MemoryRegion *mask_rom = g_new(MemoryRegion, 1); //MROM

    memory_region_init_ram(dram_mem, NULL, "riscv_quard_star_board.dram",
                           quard_star_memmap[QUARD_STAR_DRAM].size,
&error_fatal);
    memory_region_add_subregion(system_memory,
                               quard_star_memmap[QUARD_STAR_DRAM].base,
dram_mem);

    memory_region_init_ram(sram_mem, NULL, "riscv_quard_star_board.sram",
                           quard_star_memmap[QUARD_STAR_SRAM].size,
&error_fatal);
    memory_region_add_subregion(system_memory,
                               quard_star_memmap[QUARD_STAR_SRAM].base,
sram_mem);

    memory_region_init_rom(mask_rom, NULL, "riscv_quard_star_board.mrom",
                           quard_star_memmap[QUARD_STAR_MROM].size,
&error_fatal);
    memory_region_add_subregion(system_memory,
                               quard_star_memmap[QUARD_STAR_MROM].base,
mask_rom);

    riscv_setup_rom_reset_vec(machine, &s->soc[0],
                              quard_star_memmap[QUARD_STAR_MROM].base,
                              quard_star_memmap[QUARD_STAR_MROM].base,
                              quard_star_memmap[QUARD_STAR_MROM].size,
                              0x0, 0x0);
}

/* quard-star 初始化各种硬件 */

static void quard_star_machine_init(MachineState *machine)
{

```

```

//创建CPU
quard_star_cpu_create(machine);
// 创建主存
quard_star_memory_create(machine);
}

static void quard_star_machine_instance_init(Object *obj)
{

}

/* 创建machine */
static void quard_star_machine_class_init(ObjectClass *oc, void *data)
{
    MachineClass *mc = MACHINE_CLASS(oc);

    mc->desc = "RISC-V Quard Star board";
    mc->init = quard_star_machine_init;
    mc->max_cpus = QUARD_STAR_CPUS_MAX;
    mc->default_cpu_type = TYPE_RISCV_CPU_BASE;
    mc->pci_allow_0_address = true;
    mc->possible_cpu_arch_ids = riscv_numa_possible_cpu_arch_ids;
    mc->cpu_index_to_instance_props = riscv_numa_cpu_index_to_props;
    mc->get_default_cpu_node_id = riscv_numa_get_default_cpu_node_id;
    mc->numa_mem_supported = true;
}
/* 注册 quard-star */
static const TypeInfo quard_star_machine_typeinfo = {
    .name = MACHINE_TYPE_NAME("quard-star"),
    .parent = TYPE_MACHINE,
    .class_init = quard_star_machine_class_init,
    .instance_init = quard_star_machine_instance_init,
    .instance_size = sizeof(QuardStarState),
    .interfaces = (InterfaceInfo[]) {
        { TYPE_HOTPLUG_HANDLER },
        { }
    },
};

static void quard_star_machine_init_register_types(void)
{
    type_register_static(&quard_star_machine_typeinfo);
}
type_init(quard_star_machine_init_register_types)

```

3.3.3 源码分析

可以看见创建新的板子的流程为：

1.在`quard-star.h`中的`QuardStarState`结构体中为板子新建硬件，这些硬件表现在软件中为一个一个的结构体，各种硬件结构体定义在`hw/`目录下，比如我现在只创建了CPU。

plaintext

```
struct QuardStarState {
    /*< private >*/
    MachineState parent;

    /*< public >*/
    RISCVHartArrayState
    soc[QUARD_STAR_SOCKETS_MAX];
};
```

2.往static const MemMapEntry quard_star_memmap[]结构体数组中添加硬件的地址和映射的地址长度，注意这里的长度不能为0，不然会报错，其实这里DRAM的大小应该是qemu启动时需要用户输入的，例如-m 1G，这里暂不知道如何实现，所以指定了一个长度，不然会assert报错。MemMapEntry 结构体定义如下：

C

```
typedef struct
MemMapEntry {
    hwaddr base; //基
    址
    hwaddr size; //长
    度
} MemMapEntry;
```

3.创建硬件，比如我这里创建并初始化了ram和rom，新建了quard_star_memory_create函数。在函数的最后这里调用了一个很重要的函数 riscv_setup_rom_reset_vec，这个函数定义在boot.c中，函数主体如下：

参考链接：[notes/多核启动基本逻辑 at master · wangzhou/notes · GitHub](#)

参考链接：[QEMU 启动方式分析 \(3\) : QEMU 代码与 RISCV virt 平台 ZSBL 分析 - 泰晓科技 \(tinylab.org\)](#)


```

void riscv_setup_rom_reset_vec(MachineState *machine, RISCVHartArrayState
*harts,

                                hwaddr start_addr,
                                hwaddr rom_base, hwaddr rom_size,
                                uint64_t kernel_entry,
                                uint64_t fdt_load_addr)
{
    int i;
    uint32_t start_addr_hi32 = 0x00000000;
    uint32_t fdt_load_addr_hi32 = 0x00000000;

    if (!riscv_is_32bit(harts)) {
        start_addr_hi32 = start_addr >> 32;
        fdt_load_addr_hi32 = fdt_load_addr >> 32;
    }
    /* reset vector */
    uint32_t reset_vec[10] = {
        0x00000297,          /* 1: auipc  t0, %pcrel_hi(fw_dyn) */
        0x02828613,          /*      addi  a2, t0, %pcrel_lo(1b) */
        0xf1402573,          /*      csrr  a0, mhartid */
        0,
        0,
        0x00028067,          /*      jr    t0 */
        start_addr,          /* start: .dword */
        start_addr_hi32,
        fdt_load_addr,        /* fdt_laddr: .dword */
        fdt_load_addr_hi32,
                                /* fw_dyn: */
    };
    if (riscv_is_32bit(harts)) {
        reset_vec[3] = 0x0202a583; /*      lw     a1, 32(t0) */
        reset_vec[4] = 0x0182a283; /*      lw     t0, 24(t0) */
    } else {
        reset_vec[3] = 0x0202b583; /*      ld     a1, 32(t0) */
        reset_vec[4] = 0x0182b283; /*      ld     t0, 24(t0) */
    }
}

if (!harts->harts[0].cfg.ext_icsr) {
    /*
     * The Zicsr extension has been disabled, so let's ensure we don't
     * run the CSR instruction. Let's fill the address with a non
     * compressed nop.
     */
    reset_vec[2] = 0x00000013; /*      addi    x0, x0, 0 */
}

/* copy in the reset vector in little_endian byte order */
for (i = 0; i < ARRAY_SIZE(reset_vec); i++) {
    reset_vec[i] = cpu_to_le32(reset_vec[i]);
}
rom_add_blob_fixed_as("mrom.reset", reset_vec, sizeof(reset_vec),
                      rom_base, &address_space_memory);
riscv_rom_copy_firmware_info(machine, rom_base, rom_size,
sizeof(reset_vec),
                                kernel_entry);
}

```

这段代码执行以下操作：

1. 根据传入的参数，计算 `start_addr` 和 `fdt_load_addr` 的高 32 位（如果处理器不是 32 位的话）。
2. 定义一个长度为 10 的 `reset_vec` 数组，用于存储复位向量的指令序列。
3. 根据处理器是否为 32 位来设置不同的指令序列：
 - 如果是 32 位处理器，使用 `lw` 指令来加载 `a1` 和 `t0` 的值。
 - 如果是 64 位处理器，使用 `ld` 指令来加载 `a1` 和 `t0` 的值。
4. 如果处理器的 `ext_icsr` 属性为假（即禁用了 Zicsr 扩展），则将复位向量的第 2 个指令替换为一个不压缩的 `nop` 指令（`addi x0, x0, 0`）。
5. 将复位向量的指令按小端字节序进行拷贝。
6. 使用 `rom_add_blob_fixed_as` 函数将复位向量的指令添加到固定地址的 ROM 中。
7. 调用 `riscv_rom_copy_firmware_info`

我们先来看最后先调用了 `rom_add_blob_fixed_as` 函数将 `reset_vec` 中的代码拷贝到 rom 的起始位置，板子上电后最先执行的指令就是 ROM 起始位置处的这些指令。

C

```
rom_add_blob_fixed_as("mrom.reset", reset_vec,
sizeof(reset_vec),
rom_base, &address_space_memory);
```

然后调用 `riscv_rom_copy_firmware_info`，我们来看看这个函数，定义在 `boot.c` 中：

```

void riscv_rom_copy_firmware_info(MachineState *machine, hwaddr
rom_base,
                                hwaddr rom_size, uint32_t
reset_vec_size,
                                uint64_t kernel_entry)
{
    struct fw_dynamic_info dinfo;
    size_t dinfo_len;

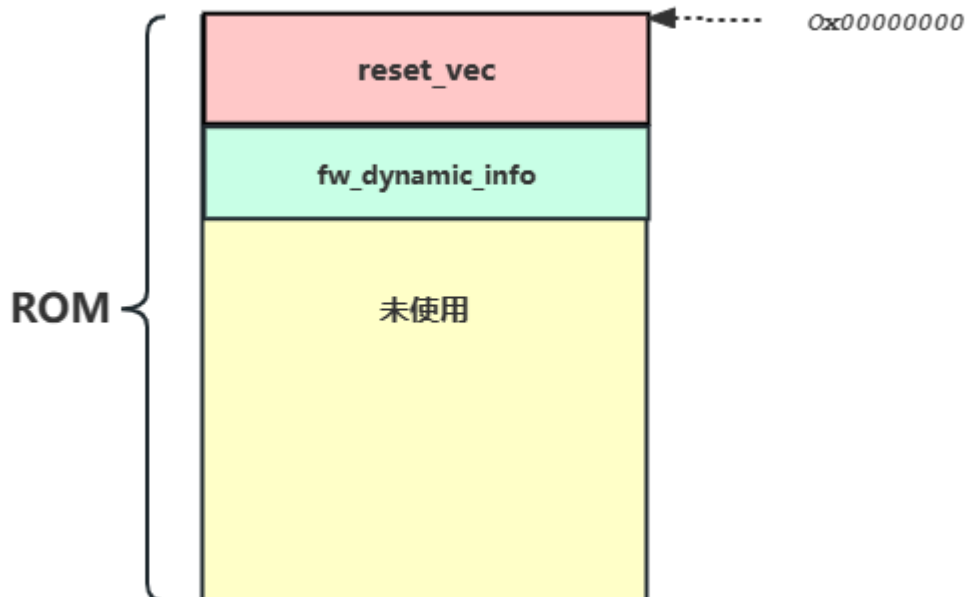
    if (sizeof(dinfo.magic) == 4) {
        dinfo.magic = cpu_to_le32(FW_DYNAMIC_INFO_MAGIC_VALUE);
        dinfo.version = cpu_to_le32(FW_DYNAMIC_INFO_VERSION);
        dinfo.next_mode = cpu_to_le32(FW_DYNAMIC_INFO_NEXT_MODE_S);
        dinfo.next_addr = cpu_to_le32(kernel_entry);
    } else {
        dinfo.magic = cpu_to_le64(FW_DYNAMIC_INFO_MAGIC_VALUE);
        dinfo.version = cpu_to_le64(FW_DYNAMIC_INFO_VERSION);
        dinfo.next_mode = cpu_to_le64(FW_DYNAMIC_INFO_NEXT_MODE_S);
        dinfo.next_addr = cpu_to_le64(kernel_entry);
    }
    dinfo.options = 0;
    dinfo.boot_hart = 0;
    dinfo_len = sizeof(dinfo);

    /**
     * copy the dynamic firmware info. This information is specific
to
     * OpenSBI but doesn't break any other firmware as long as they
don't
     * expect any certain value in "a2" register.
     */
    if (dinfo_len > (rom_size - reset_vec_size)) {
        error_report("not enough space to store dynamic firmware
info");
        exit(1);
    }

    rom_add_blob_fixed_as("mrom.finfo", &dinfo, dinfo_len,
                        rom_base + reset_vec_size,
                        &address_space_memory);
}

```

可以看到这个函数初始化了一个`fw_dynamic_info`类型的结构体，这个结构体包含了下一阶段程序启动的地址、魔数、下一阶段CPU位于S态，初始化完毕后又调用`rom_add_blob_fixed_as`函数将`fw_dynamic_info`拷贝到rom的`reset_vec`之后，用于下一阶段的启动。在这里其实可以不用使用这个函数来传递设备树，后续需要我们自己来编写设备树然后编译，在下一阶段将固件中设备树的地址传给启动的下一阶段。



所以现在再来看上面`reset_vec`的代码，将上面的代码翻译一下如下，以32位的cpu为例：

C

```
reset_vec[0] = 0x0000297; // auipc t0, %pcrel_hi(fw_dyn)
reset_vec[1] = 0x02828613; // addi a2, t0, %pcrel_lo(1b)
reset_vec[2] = 0xf1402573; // csrr a0, mhartid
reset_vec[3] = 0x0202a583; // lw a1, 32(t0)
reset_vec[4] = 0x0182a283; // lw t0, 24(t0)
reset_vec[5] = 0x00028067; // jr t0
reset_vec[6] = start_addr; // start: .dword (32-bit
address)
reset_vec[7] = 0; // unused
reset_vec[8] = fdt_load_addr; // fdt_laddr: .dword (32-bit
address)
reset_vec[9] = 0; // unused
```

具体来说，这段汇编代码完成了以下操作：

1. `auipc t0, %pcrel_hi(fw_dyn)`：使用当前 PC（程序计数器）的高 20 位（相对于 `fw_dyn` 标签的偏移量）来设置 `t0` 寄存器的值。这里的 `fw_dyn` 就是储存在 rom 的 `fw_dynamic_info` 的地址了，此时 `PC=0x00000000`。从汇编语意上看，这句的意思是，`%pcrel_hi(fw_dyn)` 表示计算 `fw_dyn` 这个符号相对于当前 PC 的偏移的高 20 bit，而 `auipc t0, imm` 表示把 `imm` 和当前 PC 相加，结果保存到 `t0`。所以，这条指令整体上的结果是将 `fw_dyn` 相对于 `pc` 的高 20 位地址取出然后拓展为 32 位与 `pc` 相加，得到的结果保存到 `t0`。这里执行完毕后 `t0=0x00000000`。
2. `addi a2, t0, %pcrel_lo(1b)`：使用当前 PC（相对于标签 `1b` 的偏移量）的低 12 位来设置 `a2` 寄存器的值。这个 `1b` 符号是啥我一直没搞懂，有没有大神告诉我，呜呜。看起来这两条指令的意思是将 `fw_dynamic_info` 的地址存到了 `a2` 中用于下一阶段启动的参数。
3. `csrr a0, mhartid`：将处理器的硬件线程 ID（`mhartid`）存储到 `a0` 寄存器中。

4. `lw a1, 32(t0)`: 从 `t0` 寄存器指向的地址偏移 32 处加载一个字 (32 位) 的数据到 `a1` 寄存器中。`32(t0)`的地址刚好是`reset_vec`, 所以`fdt`的地址被送到了`a1`寄存器中, `fdt`为设备树的地址, 这里还没定义。
5. `lw t0, 24(t0)`: 从 `t0` 寄存器指向的地址偏移 24 处加载一个字 (32 位) 的数据到 `t0` 寄存器中。`24(t0)`的地址刚好是`reset_vec[6]`, 存储的是`start_addr`, 这了传入的参数为`flash`的地址。
6. `jr t0`: 跳转到 `t0` 寄存器中保存的地址, 即跳转到了`flash`处开始执行下一阶段的引导程序。

4.将所用创建硬件的函数用`static void quard_star_machine_init`包含起来,这里创建了CPU和主存。

C

```
/* quard-star 初始化各种硬件 */

static void quard_star_machine_init(MachineState
*machine)
{
    //创建CPU
    quard_star_cpu_create(machine);
    // 创建主存
    quard_star_memory_create(machine);
    // 其他硬件
}
```

5.注需要去初始化`machine`: 创建`static void quard_star_machine_class_init`函数, 并将`machine`结构体各个字段更新。

C

```
/* 创建machine */
static void quard_star_machine_class_init(ObjectClass *oc, void
*data)
{
    MachineClass *mc = MACHINE_CLASS(oc);

    mc->desc = "RISC-V Quard Star board";
    mc->init = quard_star_machine_init;
    mc->max_cpus = QUARD_STAR_CPUS_MAX;
    mc->default_cpu_type = TYPE_RISCV_CPU_BASE;
    mc->pci_allow_0_address = true;
    mc->possible_cpu_arch_ids =
riscv_numa_possible_cpu_arch_ids;
    mc->cpu_index_to_instance_props =
riscv_numa_cpu_index_to_props;
    mc->get_default_cpu_node_id =
riscv_numa_get_default_cpu_node_id;
    mc->numa_mem_supported = true;
}
```

6.注册quard-star

C

```
/* 注册 quard-star */
static const TypeInfo quard_star_machine_typeinfo
= {
    .name          = MACHINE_TYPE_NAME("quard-star"),
    .parent        = TYPE_MACHINE,
    .class_init    = quard_star_machine_class_init,
    .instance_init =
quard_star_machine_instance_init,
    .instance_size = sizeof(QuardStarState),
    .interfaces = (InterfaceInfo[]) {
        { TYPE_HOTPLUG_HANDLER },
        { }
    },
};

static void
quard_star_machine_init_register_types(void)
{
    type_register_static(&quard_star_machine_typeinfo)
;
}

type_init(quard_star_machine_init_register_types)
```

3.3 文件夹目录变更

我原本qemu源码的目录为：

sh

```
timer@DESKTOP-JI9EVEH:~/qemu/qemu-8.0.2/
```

修改文件夹名变为如下：将最上层的qemu目录重命名为了quard_star

sh

```
timer@DESKTOP-JI9EVEH:~/quard_star/qemu-8.0.2/
```

然后在此目录下新建一个脚本文件用于编译qemu：

sh

```
touch  
build.sh
```

build.sh的内容如下：

shell

```
# 获取当前脚本文件所在的目录  
SHELL_FOLDER=$(cd "$(dirname "$0")";pwd)  
  
cd qemu-8.0.2  
if [ ! -d "$SHELL_FOLDER/output/qemu" ]; then  
./configure --prefix=$SHELL_FOLDER/output/qemu --target-list=riscv64-softmmu --enable-  
gtk --enable-virtfs --disable-gio  
fi  
make -j16  
sudo make install  
cd ..
```

1. cd qemu-8.0.2：切换到 qemu-8.0.2 目录下。
2. if [! -d "\$SHELL_FOLDER/output/qemu"]; then：如果目录 \$SHELL_FOLDER/output/qemu 不存在，则执行下面的命令。

3. `./configure --prefix=$SHELL_FOLDER/output/qemu --target-list=riscv64-sofmmu --enable-gtk --enable-virtfs --disable-gio`: 运行 `configure` 脚本，用于配置编译参数。这里指定了安装路径为 `$SHELL_FOLDER/output/qemu`，目标平台为 `riscv64-sofmmu`，开启了 GTK 支持和 VirtFS 支持，禁用了 GIO 支持。
4. `fi`: 结束条件语句的块。
5. `make -j16`: 使用并发编译，编译生成目标文件。
6. `make install`: 将编译得到的目标文件安装到系统中。
7. `cd ..`: 切换回上一级目录。

执行完build脚本后，编译完成后的qemu位于output文件夹下，再创建一个脚本文件：

sh

```
timer@DESKTOP-JI9EVEH:~/quard_star$ touch
run.sh
timer@DESKTOP-JI9EVEH:~/quard_star$ chmod +x
run.sh
```

`run.sh`的内容如下：

sh

```
SHELL_FOLDER=$(cd "$(dirname "$0")";pwd)
$SHELL_FOLDER/output/qemu/bin/qemu-system-
riscv64 \
-M quard-star \
-m 1G \
-smp 8 \
```

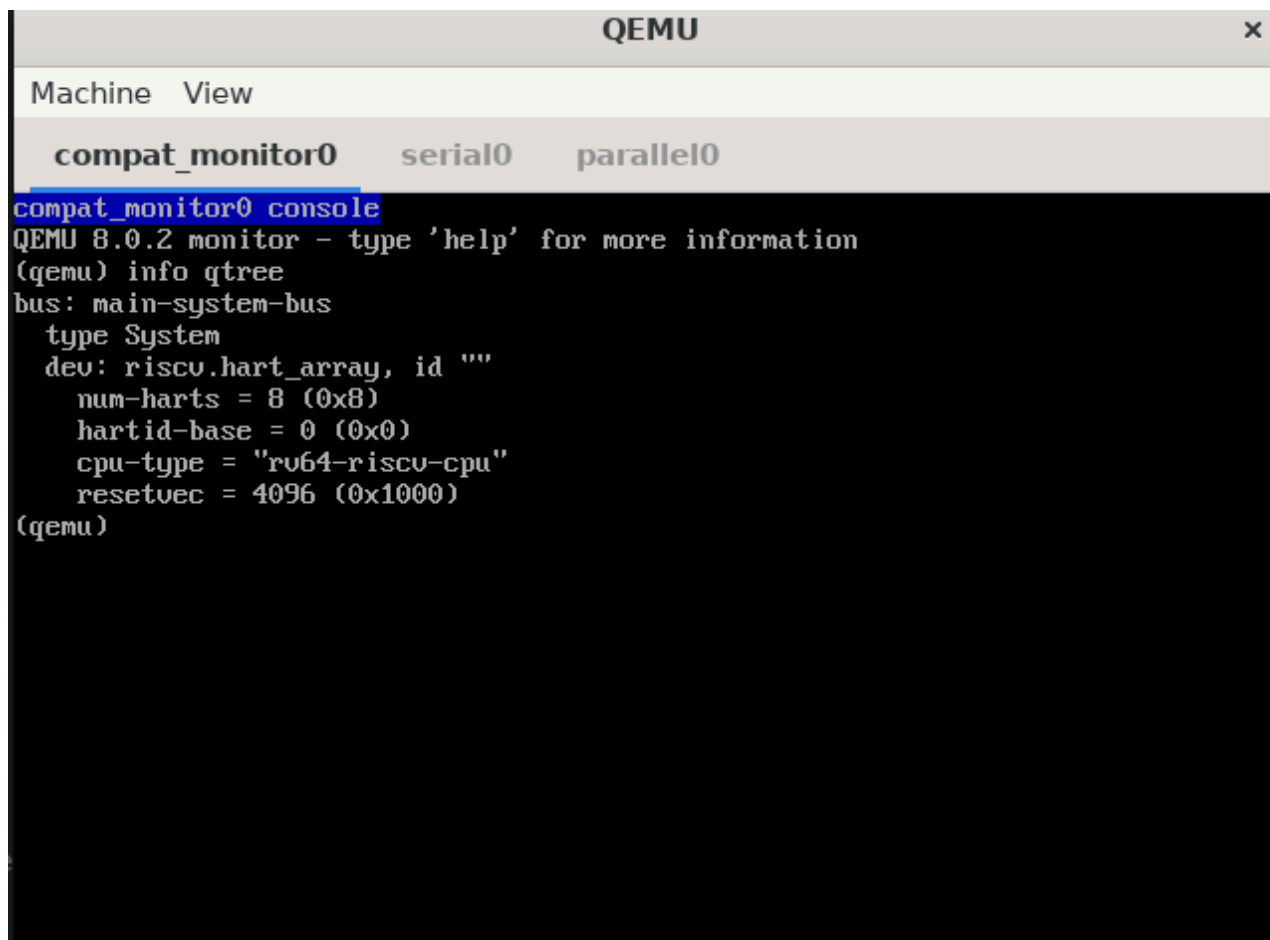
3.4 测试

运行脚本：

sh

```
timer@DESKTOP-JI9EVEH:~/quard_star$
./build.sh
timer@DESKTOP-JI9EVEH:~/quard_star$
./run.sh
```

板子启动后在qemu的monitor界面输入`info qtree`就可看见`quard-star`的硬件信息，如下：



这里我们只为quard-star板子创建了主存。

源码地址: [yangliano0/quard-star](https://github.com/yangliano0/quard-star): 从零基于qemu创建riscv嵌入式开发板, 并移植操作系统 (github.com)

有问题请与我联系: wechat: 13699648817

文章作者: Timer

文章链接: <https://yangliano0.github.io/2023/06/12/QEMU中自定义开发板/>

版权声明: 本博客所有文章除特别声明外, 均采用 CC BY-NC-SA 4.0 许可协议。转载请注明来自 TimerのBlog!

相关推荐