ISCAS 中国科学院软件研究所 Institute of Software Chinese Academy of Sciences | 智能软件研究中心 Intelligent Software Research Center

从零开始的RISC-V模拟器开发

第13讲 QEMU篇之总线虚拟化

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

本课内容

总线虚拟化

- QEMU总线是什么
- IIC总线介绍
- SPI总线介绍

总线虚拟化

进入monitor模式:

```
$qemu-system-riscv32  \
-bios none  \
-nographic -machine virt \
-nodefaults \
-monitor stdio
```

```
(qemu) info qom-tree
/machine (virt-machine)
  /unattached (container)
   /device[0] (riscv.sifive.clint)
     /riscv.sifive.clint[0] (memory-region)
   /device[11] (gpex-pcihost)
     /gpex_ioport[0] (memory-region)
     /gpex_ioport_window[0] (memory-region)
     /gpex_mmio[0] (memory-region)
     /gpex_mmio_window[0] (memory-region)
     /gpex_root (gpex-root)
       /bus master container[0] (memory-region)
       /bus master[0] (memory-region)
     /pcie-ecam[0] (memory-region)
     /pcie-mmcfg-mmio[0] (memory-region)
     /pcie-mmio-high[0] (memory-region)
     /pcie-mmio[0] (memory-region)
     /pcie.0 (PCIE)
  /device[1] (riscv.sifive.plic)
     /riscv.sifive.plic[0] (memory-region)
     /unnamed-gpio-in[0] (irq)
     /unnamed-gpio-in[100] (irq)
……
   /io[0] (memory-region)
   /riscv_virt_board.mrom[0] (memory-region)
   /riscv_virt_board.ram[0] (memory-region)
   /sysbus (System)
   /system[0] (memory-region)
```

```
(qemu) info qtree
bus: main-system-bus
  type System
  dev: cfi.pflash01, id ""
  dev: cfi.pflash01, id ""
  dev: gpex-pcihost, id ""
    gpio-out "sysbus-irq" 4
    allow-unmapped-accesses = true
    x-config-reg-migration-enabled = true
    bypass-iommu = false
    mmio ffffffffffffffff/0000000020000000
    mmio ffffffffffffffff/ffffffffffffffff
    mmio 0000000003000000/0000000000010000
    bus: pcie.0
      type PCIE
      dev: gpex-root, id ""
        addr = 00.0
        ……
        class Host bridge, addr 00:00.0, pci id 1b36:0008 (sub 1af4:1100)
  dev: virtio-mmio, id ""
    gpio-out "sysbus-irq" 1
    format_transport_address = true
    force-legacy = true
    ioeventfd = false
    mmio 0000000010008000/0000000000000200
    bus: virtio-mmio-bus.7
      type virtio-mmio-bus
……
```

总线虚拟化

进入monitor模式:

```
$qemu-system-riscv32  \
-bios none  \
-nographic -machine virt \
-nodefaults \
-monitor stdio
```

```
(qemu) info qtree
bus: main-system-bus
  type System
……
```

根bus:
main_system_bus

hw\core\sysbus.c

```
static BusState
          *main_system_bus;
```

```
(qemu) info qom-tree
/machine (virt-machine)
  /unattached (container)
   /device[0] (riscv.sifive.clint)
     /riscv.sifive.clint[0] (memory-region)
   /device[11] (gpex-pcihost)
     /gpex_ioport[0] (memory-region)
     /gpex_ioport_window[0] (memory-region)
     /gpex_mmio[0] (memory-region)
     /gpex_mmio_window[0] (memory-region)
     /gpex_root (gpex-root)
       /bus master container[0] (memory-region)
       /bus master[0] (memory-region)
     /pcie-ecam[0] (memory-region)
     /pcie-mmcfg-mmio[0] (memory-region)
     /pcie-mmio-high[0] (memory-region)
     /pcie-mmio[0] (memory-region)
     /pcie.0 (PCIE)
  /device[1] (riscv.sifive.plic)
     /riscv.sifive.plic[0] (memory-region)
     /unnamed-gpio-in[0] (irq)
     /unnamed-gpio-in[100] (irq)
……
   /io[0] (memory-region)
   /riscv_virt_board.mrom[0] (memory-region)
   /riscv_virt_board.ram[0] (memory-region)
   /sysbus (System)
   /system[0] (memory-region)
```

include\hw\pci-host\gpex.h

```
#define TYPE_GPEX_HOST "gpex-pcihost"
OBJECT_DECLARE_SIMPLE_TYPE(GPEXHost, GPEX_HOST)
struct GPEXHost {
    /*< private >*/
    PCIExpressHost parent_obj;
    /*< public >*/
    GPEXRootState gpex_root;
    ……
};
```

include\hw\pci\pcie_host.h

```
#define TYPE_PCIE_HOST_BRIDGE "pcie-host-bridge"
OBJECT_DECLARE_SIMPLE_TYPE(PCIExpressHost, PCIE_HOST_BRIDGE)
struct PCIExpressHost {
    PCIHostState pci;
};
```

include\hw\pci\pci_host.h

```
struct PCIHostState {
    SysBusDevice busdev;
    ……
    PCIBus *bus;

    QLIST_ENTRY(PCIHostState) next;
};
```

$$\begin{tikzcd}
\texttt{pci\_root\_bus\_new} \arrow[d] & & \\
\texttt{PCI\_BUS(qbus\_create(……))} \arrow[rr] & & \texttt{BUS(object\_new(typename))}
\end{tikzcd}$$

$$\begin{tikzcd}
\texttt{qdev\_new} \arrow[r] & \texttt{DEVICE(object\_new(name))}
\end{tikzcd}$$

总线虚拟化

进入monitor模式:

```
$qemu-system-riscv32  \
-bios none  \
-nographic -machine virt \
-nodefaults \
-monitor stdio
```

```
(qemu) info qtree
bus: main-system-bus
  type System
……
```

根bus:
main_system_bus

hw\core\sysbus.c

```
static BusState
        *main_system_bus;
```

```
(qemu) info qom-tree
/machine (virt-machine)
  /unattached (container)
   /device[0] (riscv.sifive.clint)
     /riscv.sifive.clint[0] (memory-region)
   /device[11] (gpex-pcihost)
     /gpex_ioport[0] (memory-region)
     /gpex_ioport_window[0] (memory-region)
     /gpex_mmio[0] (memory-region)
     /gpex_mmio_window[0] (memory-region)
     /gpex_root (gpex-root)
       /bus master container[0] (memory-region)
       /bus master[0] (memory-region)
     /pcie-ecam[0] (memory-region)
     /pcie-mmcfg-mmio[0] (memory-region)
     /pcie-mmio-high[0] (memory-region)
     /pcie-mmio[0] (memory-region)
     /pcie.0 (PCIE)
  /device[1] (riscv.sifive.plic)
     /riscv.sifive.plic[0] (memory-region)
     /unnamed-gpio-in[0] (irq)
     /unnamed-gpio-in[100] (irq)
……
   /io[0] (memory-region)
   /riscv_virt_board.mrom[0] (memory-region)
   /riscv_virt_board.ram[0] (memory-region)
   /sysbus (System)
   /system[0] (memory-region)
```

```
struct DeviceState {
    /*< private >*/
    Object parent_obj;
    /*< public >*/

    const char *id;
    char *canonical_path;
    bool realized;
    bool pending_deleted_event;
    QemuOpts *opts;
    int hotplugged;
    bool allow_unplug_during_migration;
    BusState *parent_bus;
    QLIST_HEAD(, NamedGPIOList) gpios;
    QLIST_HEAD(, NamedClockList) clocks;
    QLIST_HEAD(, BusState) child_bus;
    int num_child_bus;
    int instance_id_alias;
    int alias_required_for_version;
    ResettableState reset;
};
```

```
struct DeviceClass {
    /*< private >*/
    ObjectClass parent_class;
    /*< public >*/
    ……
    /* Private to qdev / bus.  */
    const char *bus_type;
};
```

总线虚拟化

根bus是什么，BusState类型
SysBusDevice 与 SysBus关系
类似于PCIBus *bus，I2C SPI使用什么bus

```
I2CBus *bus;
```

```
SSIBus *bus;
```

hw\ssi\ssi.c

```
struct SSIBus {
    BusState parent_obj;
};

#define TYPE_SSI_BUS "SSI"
OBJECT_DECLARE_SIMPLE_TYPE(SSIBus, SSI_BUS)
```

```
#define TYPE_BUS "bus"
DECLARE_OBJ_CHECKERS(BusState, BusClass,
                     BUS, TYPE_BUS)
```

include\hw\qdev-core.h

include\hw\sysbus.h

```
#define TYPE_SYSTEM_BUS "System"
DECLARE_INSTANCE_CHECKER(BusState, SYSTEM_BUS,
                         TYPE_SYSTEM_BUS)

#define TYPE_SYS_BUS_DEVICE "sys-bus-device"
OBJECT_DECLARE_TYPE(SysBusDevice, SysBusDeviceClass,
                    SYS_BUS_DEVICE)
```

include\hw\i2c\i2c.h

```
#define TYPE_I2C_BUS "i2c-bus"
OBJECT_DECLARE_SIMPLE_TYPE(I2CBus, I2C_BUS)
struct I2CBus {
    BusState qbus;
    QLIST_HEAD(, I2CNode) current_devs;
    uint8_t saved_address;
    bool broadcast;
};
```

QOM TYPE:

$$\begin{tikzcd}
 & & \text{TYPE\_SYSTEM\_BUS} \\
 & & \text{TYPE\_SSI\_BUS} \\
\text{TYPE\_OBJECT} \arrow[r] & \text{TYPE\_BUS} \arrow[ur] \arrow[r] \arrow[uur] \arrow[dr] \arrow[ddr] \arrow[dddr] & \text{TYPE\_I2C\_BUS} \\
 & & \text{TYPE\_PCI\_BUS} \\
 & & \text{TYPE\_PCIE\_BUS}
\end{tikzcd}$$

类型定义：

```
typedef struct BusClass BusClass;
typedef struct BusState BusState;
typedef struct I2CBus I2CBus;
typedef struct SSIBus SSIBus;
……
typedef struct PCIBus PCIBus;
typedef struct ISABus ISABus;
```

include\hw\qdev-core.h

总线虚拟化之TYPE_BUS

```
struct BusState {
    Object obj;
    DeviceState *parent;
    char *name;
    HotplugHandler *hotplug_handler;
    int max_index;
    bool realized;
    int num_children;

    QTAILQ_HEAD(, BusChild) children;
    QLIST_ENTRY(BusState) sibling;
    ResettableState reset;
};
```

```
static const TypeInfo bus_info = {
    .name = TYPE_BUS,
    .parent = TYPE_OBJECT,
    .instance_size = sizeof(BusState),
    .abstract = true,
    .class_size = sizeof(BusClass),
    .instance_init = qbus_initfn,
    .instance_finalize = qbus_finalize,
    .class_init = bus_class_init,
    .interfaces = (InterfaceInfo[]) {
        { TYPE_RESETTABLE_INTERFACE },
        { }
    },
};
```

```
struct BusChild
 struct rcu_head rcu
DeviceState *child;
int index;
QTAILQ_ENTRY
(BusChild)sibling;
```

```
struct BusClass {
    ObjectClass parent_class;

    /* FIXME first arg should be BusState */
    void (*print_dev)(Monitor *mon, DeviceState *dev, int indent);
    char *(*get_dev_path)(DeviceState *dev);

    char *(*get_fw_dev_path)(DeviceState *dev);

    void (*reset)(BusState *bus);

    bool (*check_address)(BusState *bus, DeviceState *dev, Error **errp);

    BusRealize realize;
    BusUnrealize unrealize;

    /* maximum devices allowed on the bus, 0: no limit. */
    int max_dev;
    /* number of automatically allocated bus ids (e.g. ide.0) */
    int automatic_ids;
};
```

```
static void qbus_initfn(Object *obj)
{
    BusState *bus = BUS(obj);
    ……
    object_property_add_bool(obj, "realized",
                             bus_get_realized, bus_set_realized);
}
```

```
qbus_realize
qbus_unrealize
```

```
type_init(bus_register_types)
  type_register_static(&bus_info);
```

```
bc->get_fw_dev_path = default_bus_get_fw_dev_path;
bc->reset = bus_phases_reset;
```
bus_class_init

总线的创建

总线虚拟化之TYPE_BUS

hw\core\bus.c

```
static void bus_set_realized(Object *obj, bool value,
Error **errp)
{
    BusState *bus = BUS(obj);
    BusClass *bc = BUS_GET_CLASS(bus);
    BusChild *kid;

    if (value && !bus->realized) {
        if (bc->realize) {
            bc->realize(bus, errp);
        }
    }
    ……
}
```

```
BusState *qbus_create(const char *typename, DeviceState *parent, const char
 *name)
{
    BusState *bus;

    bus = BUS(object_new(typename));
    qbus_init(bus, parent, name);

    return bus;
}
```

```
void qbus_create_inplace(void *bus, size_t size, const char *typename,
                         DeviceState *parent, const char *name)
{
    object_initialize(bus, size, typename);
    qbus_init(bus, parent, name);
}
```

```
static void main_system_bus_create(void)
{
    /* assign main_system_bus before qbus_create_inplace()
     * in order to make "if (bus != sysbus_get_default())" work */
    main_system_bus = g_malloc0(system_bus_info.instance_size);
    qbus_create_inplace(main_system_bus, system_bus_info.instance_size,
                        TYPE_SYSTEM_BUS, NULL, "main-system-bus");
    OBJECT(main_system_bus)->free = g_free;
}
```

总线虚拟化之TYPE_SYSTEM_BUS

```
#define TYPE_SYSTEM_BUS "System"
DECLARE_INSTANCE_CHECKER(BusState, SYSTEM_BUS,
                         TYPE_SYSTEM_BUS)
```

```
static const TypeInfo system_bus_info = {
    .name = TYPE_SYSTEM_BUS,
    .parent = TYPE_BUS,
    .instance_size = sizeof(BusState),
    .class_init = system_bus_class_init,
};
```

```
#define TYPE_SYS_BUS_DEVICE "sys-bus-device"
OBJECT_DECLARE_TYPE(SysBusDevice, SysBusDeviceClass,
                    SYS_BUS_DEVICE)
```

```
static const TypeInfo sysbus_device_type_info = {
    .name = TYPE_SYS_BUS_DEVICE,
    .parent = TYPE_DEVICE,
    .instance_size = sizeof(SysBusDevice),
    .abstract = true,
    .class_size = sizeof(SysBusDeviceClass),
    .class_init = sysbus_device_class_init,
};
```

```
type_register_static(&system_bus_info);
type_register_static(&sysbus_device_type_info);
```

```
static void qemu_create_machine
{
    ……
    object_property_add_child(o
                              O
    object_property_add_child(container_get(OBJECT(current_machine),
                                            "/unattached"),
                              "sysbus", OBJECT(sysbus_get_default()));
}
```

sysbus创建

```
BusState *sysbus_get_default(void)
{
    if (!main_system_bus) {
        main_system_bus_create();
    }
    return main_system_bus;
}
```

```
static void system_bus_class_init(ObjectClass *klass, void *data)
{
    BusClass *k = BUS_CLASS(klass);

    k->print_dev = sysbus_dev_print;
    k->get_fw_dev_path = sysbus_get_fw_dev_path;
}
```

```
static void sysbus_device_class_init(ObjectClass *klass, void *data)
{
    DeviceClass *k = DEVICE_CLASS(klass);
    k->realize = sysbus_device_realize;
    k->bus_type = TYPE_SYSTEM_BUS;
}
```

接口示例

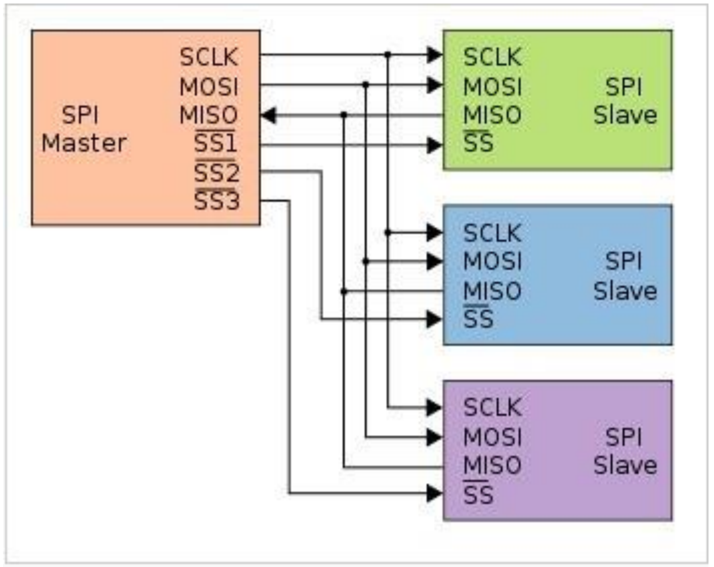
```
void sysbus_connect_irq(SysBusDevice *dev, int n, qemu_irq irq);
```

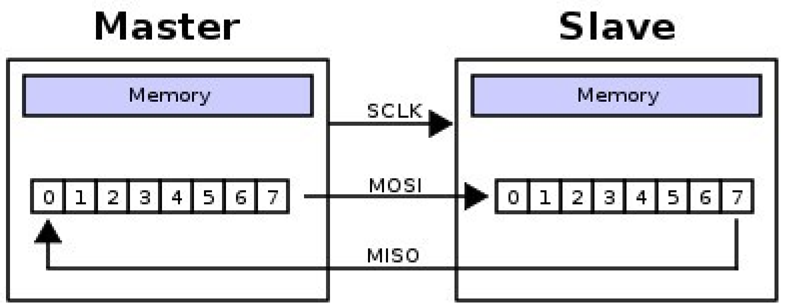
```
void sysbus_mmio_map(SysBusDevice *dev, int n, hwaddr addr);
```

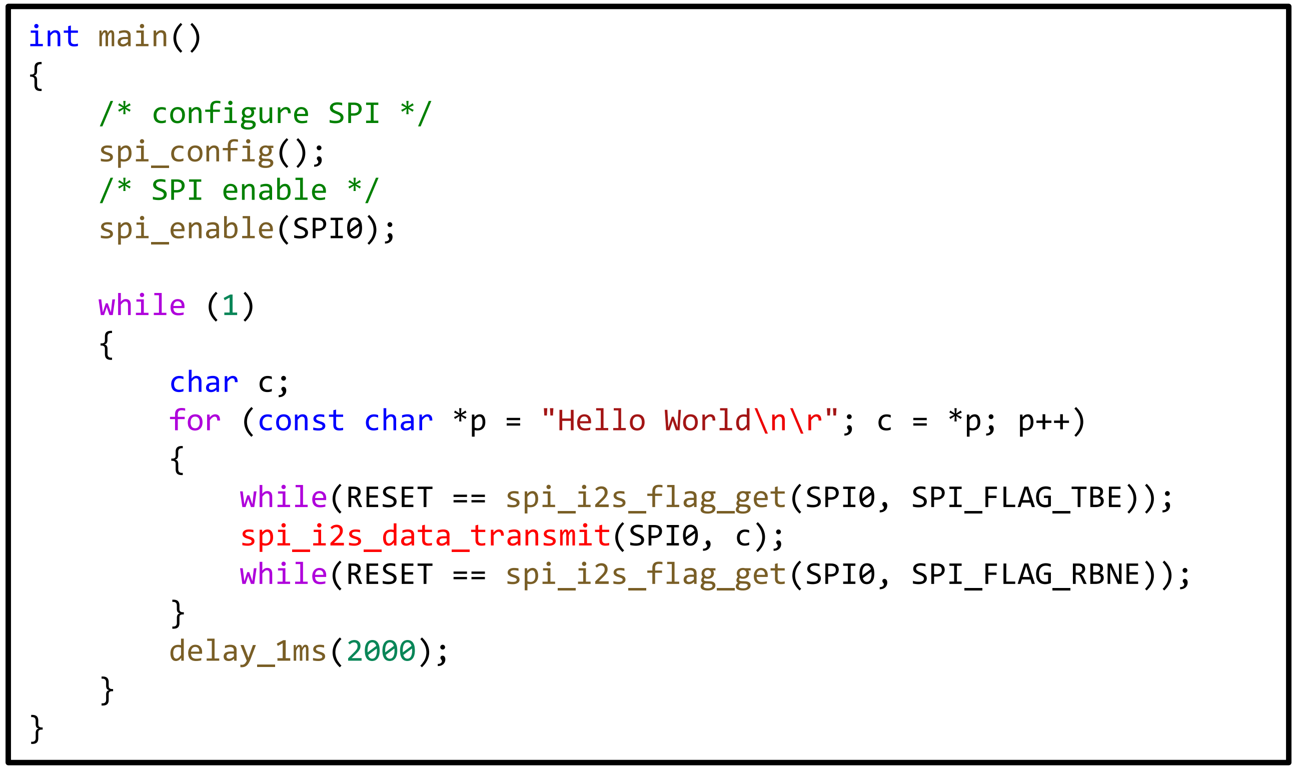
```
bool sysbus_realize(SysBusDevice *dev, Error **errp);
```

总线虚拟化之SSIBus

SPI是一种同步、高速、全双工的通信总线，全称为Serial Peripheral Interface（串行外设接口），由Motorola公司提出。在嵌入式系统设计时，常使用SPI接口连接一些传感器、外接存储器或通信模组。







```c
int main()
{
    /* configure SPI */
    spi_config();
    /* SPI enable */
    spi_enable(SPI0);

    while (1)
    {
        char c;
        for (const char *p = "Hello World\n\r"; c = *p; p++)
        {
            while(RESET == spi_i2s_flag_get(SPI0, SPI_FLAG_TBE));
            spi_i2s_data_transmit(SPI0, c);
            while(RESET == spi_i2s_flag_get(SPI0, SPI_FLAG_RBNE));
        }
        delay_1ms(2000);
    }
}
```

参考: https://www.rvmcu.com/column-topic-id-595.html

总线虚拟化之SSIBus

include\hw\ssi\ssi.h

```
#define TYPE_SSI_PERIPHERAL "ssi-peripheral"
OBJECT_DECLARE_TYPE(SSIPeripheral, SSIPeripheralClass,
                    SSI_PERIPHERAL)
```

```
struct SSIPeripheral {
    DeviceState parent_obj;

    /* Chip select state */
    bool cs;
};
```

```
struct SSIPeripheralClass {
    DeviceClass parent_class;

    void (*realize)(SSIPeripheral *dev, Error **errp);

    uint32_t (*transfer)(SSIPeripheral *dev, uint32_t val);

    int (*set_cs)(SSIPeripheral *dev, bool select);
    SSICSMode cs_polarity;

    uint32_t (*transfer_raw)(SSIPeripheral *dev, uint32_t v
al);
};
```

hw\ssi\ssi.c

```
struct SSIBus {
    BusState parent_obj;
};

#define TYPE_SSI_BUS "SSI"
OBJECT_DECLARE_SIMPLE_TYPE(SSIBus, SSI_BUS)

static const TypeInfo ssi_bus_info = {
    .name = TYPE_SSI_BUS,
    .parent = TYPE_BUS,
    .instance_size = sizeof(SSIBus),
};
```

```
type_init(ssi_peripheral_register_types)
    type_register_static(&ssi_bus_info);
    type_register_static(&ssi_peripheral_info);
```

```
static const TypeInfo ssi_peripheral_info = {
    .name = TYPE_SSI_PERIPHERAL,
    .parent = TYPE_DEVICE,
    .class_init = ssi_peripheral_class_init,
    .class_size = sizeof(SSIPeripheralClass),
    .abstract = true,
};
```

设备接口

```
DeviceState *ssi_create_peripheral(SSIBus *bus, const char *name);
bool ssi_realize_and_unref(DeviceState *dev, SSIBus *bus, Error **errp);
```

qdev_new

BUS接口

```
SSIBus *ssi_create_bus(DeviceState *parent, const char *name);
uint32_t ssi_transfer(SSIBus *bus, uint32_t val);
```

qbus_create

总线虚拟化之SSIBus

include\hw\ssi\sifive_spi.h

```c
typedef struct SiFiveSPIState {
    SysBusDevice parent_obj;
    MemoryRegion mmio;
    qemu_irq irq;
    uint32_t num_cs;
    qemu_irq *cs_lines;
    SSIBus *spi;
    Fifo8 tx_fifo;
    Fifo8 rx_fifo;
    uint32_t regs[SIFIVE_SPI_REG_NUM];
} SiFiveSPIState;
```

```c
static void sifive_spi_realize(DeviceState *dev, Error **errp)
{
    SysBusDevice *sbd = SYS_BUS_DEVICE(dev);
    SiFiveSPIState *s = SIFIVE_SPI(dev);
    int i;

    s->spi = ssi_create_bus(dev, "spi");
    ……
}
```

hw\ssi\sifive_spi.c

```c
static void sifive_spi_flush_txfifo(SiFiveSPIState *s)
{
    uint8_t tx;
    uint8_t rx;

    while (!fifo8_is_empty(&s->tx_fifo)) {
        tx = fifo8_pop(&s->tx_fifo);
        rx = ssi_transfer(s->spi, tx);

        if (!fifo8_is_full(&s->rx_fifo)) {
            if (!(s->regs[R_FMT] & FMT_DIR)) {
                fifo8_push(&s->rx_fifo, rx);
            }
        }
    }
}
```

```c
ssi_transfer
```

SSIPeripheralClass

```c
ssc->transfer_raw(peripheral, val);
```

```c
ssc->transfer_raw = ssi_transfer_raw_default;
```

```c
ssc->transfer(dev, val);
```

hw\riscv\sifive_u.c

```c
/* Connect an SD card to SPI2 */
sd_dev = ssi_create_peripheral(s->soc.spi2.spi, "ssi-sd");
```

```
TYPE_SSI_SD
```

TypeInfo ssi_sd_info	.parent = TYPE_SSI_PERIPHERAL
ssi_sd_class_init	k->transfer = ssi_sd_transfer;

总线虚拟化之SSIBus

m25p80为例

spi flash

hw\sd\ssi-sd.c

```
static uint32_t ssi_sd_transfer(SSIPeripheral *dev, uint32_t val)
{
    ssi_sd_state *s = SSI_SD(dev);
    SDRequest request;
    uint8_t longresp[16];
    ……
    switch (s->mode) {
    case SSI_SD_CMD:
        switch (val) {
        case SSI_DUMMY:
            DPRINTF("NULL command\n");
            return SSI_DUMMY;
            break;
        case SSI_TOKEN_SINGLE:
        case SSI_TOKEN_MULTI_WRITE:
            DPRINTF("Start write block\n");
            s->mode = SSI_SD_DATA_WRITE;
            return SSI_DUMMY;
        case SSI_TOKEN_STOP_TRAN:
```

```
static const TypeInfo m25p80_info = {
    .name           = TYPE_M25P80,
    .parent         = TYPE_SSI_PERIPHERAL,
    ……
}
```

```
.parent     = TYPE_M25P80,
.class_init = m25p80_class_init,
```

hw\block\m25p80.c

```
static void m25p80_class_init(ObjectClass *klass, void *data)
{
    DeviceClass *dc = DEVICE_CLASS(klass);
    SSIPeripheralClass *k = SSI_PERIPHERAL_CLASS(klass);
    ……
    k->transfer = m25p80_transfer8;
    ……
}
```

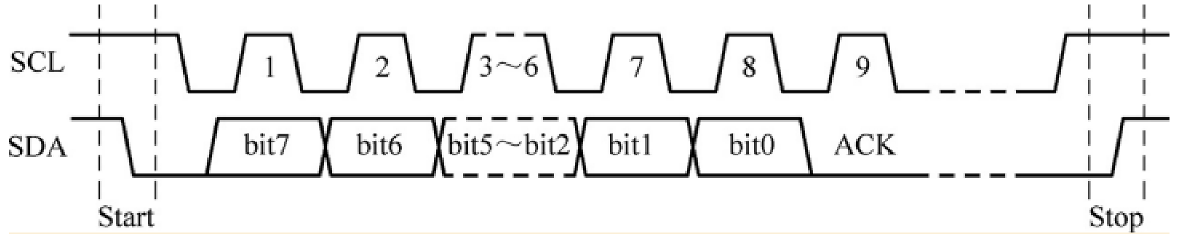
```
static uint32_t m25p80_transfer8(SSIPeripheral *ss, uint32_t tx)
{
    Flash *s = M25P80(ss);
    uint32_t r = 0;
    ……
    switch (s->state) {

    case STATE_PAGE_PROGRAM:
        trace_m25p80_page_program(s, s->cur_addr, (uint8_t)tx);
        flash_write8(s, s->cur_addr, (uint8_t)tx);
……
}
```

总线虚拟化之I2C

I²C是由NXP（原PHILIPS）公司开发的两线式串行总线，用于连接微控制器及其外围芯片，目前已成为一种行业标准，在微控制器设计中被大量采用，GD32VF103微控制器上也集成了I²C接口。

I²C总线的主要特点是接线简单，硬件上只需两条线，一根SCL时钟线用于收发双方的时钟节拍，一根SDA数据线负责传输数据，因此I²C是一种同步通信。



```c
void I2C_WriteByte(uint8_t addr, uint8_t data)
{
    /* wait until I2C bus is idle */
    while(i2c_flag_get(I2C1, I2C_FLAG_I2CBSY));
    /* send a start condition to I2C bus */
    i2c_start_on_bus(I2C1);
    /* wait until SBSEND bit is set */
    while(!i2c_flag_get(I2C1, I2C_FLAG_SBSEND));
    /* send slave address to I2C bus*/
    i2c_master_addressing(I2C1, 0x78, I2C_TRANSMITTER);
    /* wait until ADDSEND bit is set*/
    while(!i2c_flag_get(I2C1, I2C_FLAG_ADDSEND));
    /* clear ADDSEND bit */
    i2c_flag_clear(I2C1, I2C_FLAG_ADDSEND);

    /* send a addr byte */
    i2c_data_transmit(I2C1, addr);
    /* wait until the transmission data register is empty*/
    while(!i2c_flag_get(I2C1, I2C_FLAG_TBE));

    /* send a data byte */
    i2c_data_transmit(I2C1, data);
    /* wait until the transmission data register is empty*/
    while(!i2c_flag_get(I2C1, I2C_FLAG_TBE));

    /* send a stop condition to I2C bus*/
    i2c_stop_on_bus(I2C1);
    /* wait until stop condition generate */
    while(I2C_CTL0(I2C1)&0x0200);
}
```

来自: https://www.rvmcu.com/column-topic-id-563.html

中国科学院软件研究所 Institute of Software Chinese Academy of Sciences　智能软件研究中心 Intelligent Software Research Center

总线虚拟化之I2C

include\hw\i2c\i2c.h

```
#define TYPE_I2C_BUS "i2c-bus"
OBJECT_DECLARE_SIMPLE_TYPE(I2CBus, I2C_BUS)
typedef struct I2CNode I2CNode;
struct I2CNode {
    I2CSlave *elt;
    QLIST_ENTRY(I2CNode) next;
};

struct I2CBus {
    BusState qbus;
    QLIST_HEAD(, I2CNode) current_devs;
    uint8_t saved_address;
    bool broadcast;
};
```

```
type_init(i2c_slave_register_types)
```

```
type_register_static(&i2c_bus_info);
type_register_static(&i2c_slave_type_info);
```

```
static const TypeInfo i2c_slave_type_info = {
    .name = TYPE_I2C_SLAVE,
    .parent = TYPE_DEVICE,
    .instance_size = sizeof(I2CSlave),
    .abstract = true,
    .class_size = sizeof(I2CSlaveClass),
    .class_init = i2c_slave_class_init,
};
```

```
k->bus_type = TYPE_I2C_BUS;
```

hw\i2c\core.c

```
#define TYPE_I2C_SLAVE "i2c-slave"
OBJECT_DECLARE_TYPE(I2CSlave, I2CSlaveClass,
                    I2C_SLAVE)

struct I2CSlaveClass {
    DeviceClass parent_class;

    /* Master to slave. Returns non-zero for a NAK, 0 for success. */
    int (*send)(I2CSlave *s, uint8_t data);

    uint8_t (*recv)(I2CSlave *s);

    int (*event)(I2CSlave *s, enum i2c_event event);
};
```

```
struct I2CSlave {
    DeviceState qdev;

    uint8_t address;
};
```

```
I2CBus *i2c_init_bus(DeviceState *parent, const char *name);
void i2c_set_slave_address(I2CSlave *dev, uint8_t address);
int i2c_bus_busy(I2CBus *bus);
int i2c_start_transfer(I2CBus *bus, uint8_t address, int recv);
void i2c_end_transfer(I2CBus *bus);
void i2c_nack(I2CBus *bus);
int i2c_send_recv(I2CBus *bus, uint8_t *data, bool send);
int i2c_send(I2CBus *bus, uint8_t data);
uint8_t i2c_recv(I2CBus *bus);
```

I2C操作接口

设备创建接口

```
I2CSlave *i2c_slave_new(const char *name, uint8_t addr);
```

```
I2CSlave *i2c_slave_create_simple(I2CBus *bus, const char *name, uint8_t addr);
```

```
bool i2c_slave_realize_and_unref(I2CSlave *dev, I2CBus *bus, Error **errp);
```

总线虚拟化之I2C使用

```
typedef struct AspeedI2CBus {
    struct AspeedI2CState *controller;

    MemoryRegion mr;

    I2CBus *bus;
    uint8_t id;
    qemu_irq irq;

    ……
} AspeedI2CBus;
```

hw\i2c\aspeed_i2c.c

```
struct AspeedI2CState {
    SysBusDevice parent_obj;
    MemoryRegion iomem;
    qemu_irq irq;
    ……
    MemoryRegion pool_iomem;
    uint8_t pool[ASPEED_I2C_MAX_POOL_SIZE];

    AspeedI2CBus busses[ASPEED_I2C_NR_BUSSES];
    MemoryRegion *dram_mr;
    AddressSpace dram_as;
};
```

```
aspeed_i2c_realize → s->busses[i].bus = i2c_init_bus(dev, name);
```

hw\arm\aspeed.c

```
dev = DEVICE(i2c_slave_new(TYPE_PCA9552, 0x60));
qdev_prop_set_string(dev, "description", "pca1");
i2c_slave_realize_and_unref(I2C_SLAVE(dev),
                            aspeed_i2c_get_bus(&soc->i2c, 3),
                            &error_fatal);
```

```
static void pca955x_class_init(ObjectClass *klass, void *data)
{
    DeviceClass *dc = DEVICE_CLASS(klass);
    I2CSlaveClass *k = I2C_SLAVE_CLASS(klass);

    k->event = pca955x_event;
    k->recv = pca955x_recv;
    k->send = pca955x_send;
    dc->realize = pca955x_realize;
 }
```

hw\misc\pca9552.c

```
static int pca955x_send(I2CSlave *i2c, uint8_t data)
{
    PCA955xState *s = PCA955X(i2c);
    if (s->len == 0) {
        s->pointer = data;
        s->len++;
    } else {
        pca955x_write(s, s->pointer & 0xf, data);
        pca955x_autoinc(s);
    }

    return 0;
}
```

下节课内容：

外设虚拟化

- Nuclei GPIO设备实现
- Nuclei IIC设备实现
- Nuclei SPI设备实现
- Nuclei DMA设备实现与应用

谢谢

wangjunqiang@iscas.ac.cn