Lab5

一、 实验目的

1. 理解中断的原理和机制，掌握CPU访问设备控制器的方法
2. 掌握Arm体系结构的中断机制和规范，实现时钟中断服务

二、 实验过程

1、理解Arm架构的中断系统

arm的中断分为两类IRQ(普通中断),与FIQ(快速中断)。
Arm采用的中断控制器叫做GIC，即general interrupt controller。gic包括多个版本，如GICv1（已弃用），GICv2，GICv3，GICv4。简单起见，我们实验将选用GICv2版本。

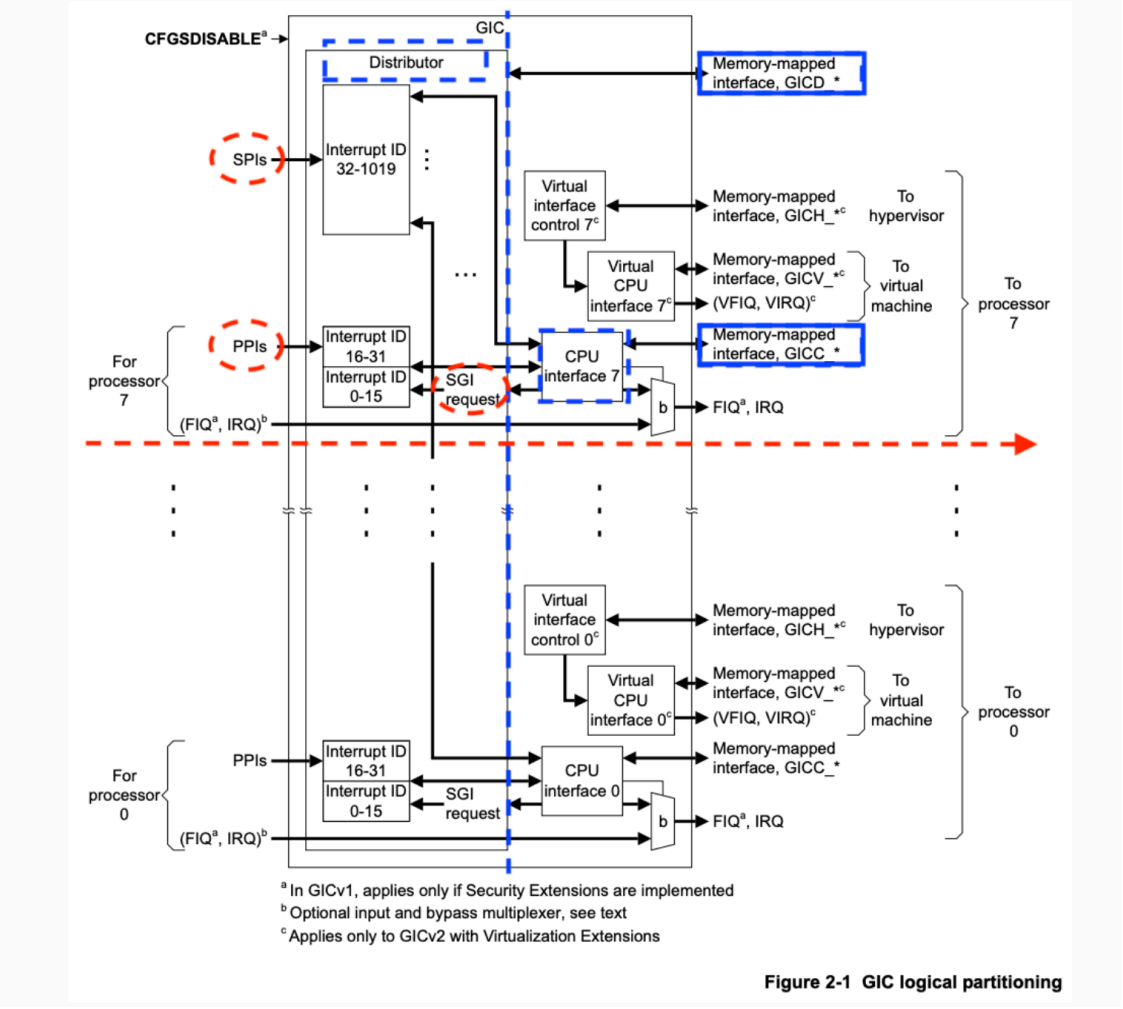

Figure 2-1 GIC logical partitioning

1. GICv2 最多支持8个核的中断管理。
2. GIC包括两大主要部分（由图中蓝色虚竖线分隔，Distributor和CPU Interface由蓝色虚矩形框标示），分别是:
3. Distributor，其通过GICD_开头的寄存器进行控制（蓝色实矩形框标示）
4. CPU Interface，其通过GICC_开头的寄存器进行控制（蓝色实矩形框标示）
5. 中断类型分为以下几类（由图中红色虚线椭圆标示）：

6. SPI：（shared peripheral interrupt），共享外设中断。该中断来源于外设，通过Distributor分发给特定的core，其中断编号为32-1019。从图中可以看到所有核共享SPI。
7. PPI：（private peripheral interrupt），私有外设中断。该中断来源于外设，但只对指定的core有效，中断信号只会发送给指定的core，其中断编号为16-31。从图中可以看到每个core都有自己的PPI。
8. SGI：（software-generated interrupt），软中断。软件产生的中断，用于给其他的core发送中断信号，其中断编号为0-15。
9. virtual interrupt，虚拟中断，用于支持虚拟机。图中也可以看到，因为我们暂时不关心，所以没有标注。

Distributor（中断分发器）

- 作用：
  - **全局中断管理**：负责接收所有外设和软件产生的中断（如硬件IRQ、软件生成的中断），并根据配置决定是否将中断转发给CPU Interface。
  - **优先级仲裁**：比较多个中断的优先级，确保高优先级中断优先被处理。
  - **中断路由**：支持多核场景，将中断分配到特定的CPU核心或一组核心（亲和性配置）。
  - **中断屏蔽与使能**：通过 `GICD_*` 寄存器控制每个中断的全局使能/禁用状态。
  - **中断触发方式配置**：设置中断为电平触发或边沿触发。

CPU Interface（CPU接口）**

- 作用：
  - **核本地中断管理**：每个CPU核心有独立的CPU Interface，用于处理该核心接收到的中断。
  - **中断应答与结束**：当CPU核心处理中断时，通过 `GICC_*` 寄存器完成中断的应答（ACK）和结束（EOI）通知。
  - **优先级掩码控制**：动态调整当前CPU核心能处理的最低中断优先级（通过 `GICC_PMR`），避免低优先级中断抢占关键任务。
  - **中断抢占配置**：支持嵌套中断（高优先级中断可抢占正在处理的低优先级中断）。

2.分析virt.dts中intc和timer的部分

```
intc@8000000 {
        phandle = <0x8001>;
        reg = <0x00 0x8000000 0x00 0x10000 0x00 0x8010000 0x00
0x10000>;
        compatible = "arm,cortex-a15-gic";
        ranges;
        #size-cells = <0x02>;
        #address-cells = <0x02>;
        interrupt-controller;
        #interrupt-cells = <0x03>;

        v2m@8020000 {
                phandle = <0x8002>;
                reg = <0x00 0x8020000 0x00 0x1000>;
                msi-controller;
                compatible = "arm,gic-v2m-frame";
        };
};

timer {
        interrupts = <0x01 0x0d 0x104 0x01 0x0e 0x104 0x01 0x0b
0x104 0x01 0x0a 0x104>;
        always-on;
        compatible = "arm,armv8-timer\0arm,armv7-timer";
};
```

- intc中的 **`reg`** 指明GICD寄存器映射到内存的位置为0x8000000，长度为0x10000， GICC寄存器映射到内存的位置为0x8010000，长度为0x10000
- intc中的 **`#interrupt-cells`** 指明 interrupts 包括3个cells。第一个文档 指明：第一个cell为中断类型，0表示SPI，1表示PPI；第二个cell为中断号，SPI范围为[0-987]，PPI为[0-15]；第三个cell为flags，其中[3:0]位表示触发类型，4表示高电平触发，[15:8]为PPI的cpu中断掩码，每1位对应一个cpu，为1表示该中断会连接到对应的cpu。
- 以timer设备为例，其中包括4个中断。以第二个中断的参数 **`0x01 0x0e 0x104`** 为例，其指明该中断为PPI类型的中断，中断号14， 路由到第一个cpu，且高电平触发。但注意到PPI的起始中断号为16，所以实际上该中断在GICv2中的中断号应为16 + 14 = 30。

什么是中断源：

中断源（Interrupt Source）是指能够**触发 CPU 中断的硬件或软件事件**。它可以是外设（如 UART、GPIO、定时器）、异常（如除零错误）或软件生成的中断信号。中断源向中断控制器（如 ARM GIC）发送请求，由中断控制器决定是否通知 CPU 进行处理。

什么是`interrupt-cell`：

在设备树（Device Tree）中，`interrupt-cells` 是一个关键属性，用于定义**如何编码和描述中断源**。它指定了设备节点中 `interrupts` 属性所需的 **数据单元（cell）的数量和格式**，以便中断控制器（如 ARM GIC）能正确解析中断配置。

`interrupt-cells` 的常见值：

```
#interrupt-cells = <3>
```

- **用途**：ARM GIC 常见格式，三个 cell 分别描述：
  1. **中断类型**：`0` 为 SPI（共享外设中断），`1` 为 PPI（私有外设中断）。
  2. **中断号**：GIC 中的硬件中断号（如 25）。
  3. **触发类型和标志**：同前述编码（如 `4` 为高电平触发）。
- **示例**：

```
interrupts = <0 25 4>; // SPI中断，中断号25，高电平触发
```

3.在bsp目录下新建文件hwi_init.c，初始化GIC

4.在include目录下创建文件prt_config.h

设置中断时间间隔，tick处理时间不能超过1/OS_TICK_PER_SECOND(s)

原因：防止中断堆积

Tick频率 = 1000Hz（1ms/次），若处理时间超过1ms，下次Tick到来时，前一次还未完成，导致中断丢失或系统崩溃。

5.在include目录下创建os_cpu_armv8.h文件

这个文件定义了异常处理等级，同时定义了Debug、Abort、IRQ和FIQ四个异常的屏蔽位，分别代表调试异常、终止异常、IRQ中断和FIQ中断，以及一些ARMv8架构中的内存屏障汇编指令，用来保证特定的内存访问顺序。

6.在bsp目录下创建time.c文件对定时器和相应的中断进行配置

7.构建时钟中断处理

1. prt_vector.S 中的 EXC_HANDLE 5 OsExcDispatch 改为 EXC_HANDLE 5 OsHwiDispatcher，设置IRQ类型的异常处理函数
2. 在 prt_vector.S 中加入 OsHwiDispatcher 处理代码，整体上下文的恢复与保存的结构与系统调用完全一致，C语言处理函数的入口变为OsExcDispatch

3. 在prt_exc.c中引用头文件 os_attr_armv8_external.h，os_cpu_armv8.h，同时定义C语言处理函数OsHwiDispatch，用于IRQ 类型的中断。
4. 在src/kernel/tick目录下新建文件prt_tick.c文件，实现 OsTickDispatcher 时钟中断处理函数
5. 将新创建的文件纳入构建系统
6. 在prt_exc.c文件中实现OsIntLock 和 OsIntRestore 函数
7. 在src/bsp目录下创建头文件os_cpu_armv8_external.h，用于相关宏的定义
8. 在include目录下新建 prt_tick.h，声明 Tick 相关的接口函数

8.最后修改main函数，完成时钟中断的构建

三、作业

实现 hwi_init.c 中缺失的 OsGicEnableInt 和 OsGicClearInt 函数

OsGicDisableInt 函数

```c
OS_SEC_L4_TEXT void OsGicDisableInt(U32 intId)
{
    // 计算中断对应的GICD_ICENABLERn寄存器的地址
    volatile U32* addr = (volatile U32*)(GICD_ICENABLERn + (intId / GICD_ICENABLER_SIZE) * sizeof(U32));
    // 生成需要清除的位掩码
    U32 mask = 1 << (intId % GICD_ICENABLER_SIZE);
    // 写入寄存器以禁用中断
    GIC_REG_WRITE(addr, mask);
}
```

1. 函数作用

`OsGicDisableInt` 用于 **禁用** GICv2 中断控制器中的某个中断（`intId`）。

- 通过写入 `GICD_ICENABLERn`（Interrupt Clear-Enable Registers）来 **清除** 中断使能位，从而禁用该中断。

---

2. 关键部分详解

(1) 计算 `GICD_ICENABLERn` 寄存器地址

```
volatile U32* addr = (volatile U32*)(GICD_ICENABLERn + (intId / GICD_ICENABLER_SIZE) * sizeof(U32));
```

- 为什么在此处必须用 `volatile`?
  - **GICD_ICENABLERn 是内存映射的硬件寄存器**，其值可能随时被硬件（如中断控制器）修改。
  - 若未用 `volatile`，编译器可能：
    - **优化掉"看似无用"的写入**（例如认为连续写入相同值无意义）。
    - **缓存旧值**（用寄存器变量代替内存访问）。
- `GICD_ICENABLERn`:
  - 基地址 `0x08000180`（`GICD_ICENABLERn` 的起始地址）。
  - 每个 `GICD_ICENABLERn` 寄存器管理 **32 个中断**（`GICD_ICENABLER_SIZE = 32`）。
- `intId / 32`:
  - 计算该中断属于哪个 `GICD_ICENABLERn` 寄存器。
  - 例如，`intId = 30` → `30 / 32 = 0`，表示使用第一个 `GICD_ICENABLERn` 寄存器。
- `* sizeof(U32)`:
  - 每个寄存器占 4 字节（`U32`），计算偏移量。
- **最终地址**:
  - `GICD_ICENABLERn + (intId / 32) * 4` → `0x08000180 + 0 = 0x08000180`（`intId=30` 时）。

---

(2) 生成位掩码

```
U32 mask = 1 << (intId % GICD_ICENABLER_SIZE);
```

- `intId % 32`:
  - 计算该中断在寄存器中的 **位偏移**。
  - 例如，`intId = 30` → `30 % 32 = 30`，表示第 30 位。
- `1 << 30`:

- 生成掩码 `0x40000000`（二进制 `0100_0000_0000_0000_0000_0000_0000_0000`）。
- 写入 `GICD_ICENABLERn` 的第 30 位，可以 **清除** 该中断的使能位。

---

(3) 写入寄存器

```
GIC_REG_WRITE(addr, mask);
```

- `GIC_REG_WRITE`：
  - 宏定义：`*(volatile U32 *)(addr) = mask`。
  - 向 `GICD_ICENABLERn` 写入 `mask`，**清除** 对应中断的使能位。
- **效果**：
  - 写入 `1` 到某位会 **禁用** 该中断（`GICD_ICENABLERn` 是 **写 1 清除** 寄存器）。

OsGicEnableInt函数

```c
OS_SEC_L4_TEXT void OsGicEnableInt(U32 intId)
{
    // 计算中断对应的GICD_ISENABLERn寄存器的地址
    volatile U32* addr = (volatile U32*)(GICD_ISENABLERn + (intId
/ GICD_ISENABLER_SIZE) * sizeof(U32));
    // 生成需要设置的位掩码
    U32 mask = 1 << (intId % GICD_ISENABLER_SIZE);
    // 写入寄存器以使能中断
    GIC_REG_WRITE(addr, mask);
}
```

四、测试及分析

运行内核程序，发现正常引发时钟中断

五、心得体会

1. 深刻理解了中断的原理和机制，深刻理解CPU访问设备控制器的方法。
2. 掌握Arm体系结构的中断机制和规范，实现时钟中断服务。
3. 实现了时钟中断，并完成了一个定时器进行时钟中断处理。