# Lab 3: Interrupt And Exception

18307130024 Jimmy Tan



# 1. `AArch64` Exception Handling

I have briefly introduced the exception handling of `AArch64` in my report of `lab1`. This section will further discuss upon this topic, based on ARMv8-A Programmer Guide.

## 1.1 Where to jump

When an exception occurs, the system will jump to the exception handler.

There are multiple exception handlers. The entry of the handlers are determined by both the vector base and the offset, i.e., $Base + Offset$.

The vector base is stored in the `vector based address registers`, or `VBAR`. The `VBAR` is replicated for exception level 1, 2 and 3.

The offset to be selected is determined by:

- The type of the exception
- If the exception is being taken at the same Exception level, the Stack Pointer to be used (SP0 or SPx).
- If the exception is being taken at a lower Exception level, the execution state of the next lower level (AArch64 or AArch32).

## 1.2 The Generic Interrupt Controller

ARM provides a standard interrupt controller, the programming interface to which is defined in the `GIC` architecture. This is a brief introduction of it.

`GIC` can be divided into two major parts:

- **Distributor**. This is where all interrupt sources are connected. The distributor manages the status of each individual interrupt and selects one to send to the CPU.
- **CPU interface**. One CPU interface for each core. It's where a CPU core receives an interrupt.

Interrupts are identified by an `interrupt ID` and are grouped as:

- **Software Generated Interrupt (SGI)**. The ID range is 0 to 15.

- **Private Peripheral Interrupt (PPI)**. The ID range is 16 to 31.
- **Shared Peripheral Interrupt (SPI)**. The ID range is 32 to 1020.

Interrupts can have several states:

- Inactive
- Pending
- Active
- Active and pending

# 2. Homework

## 2.1 What happens when an interrupt occurs

Interrupts, including `IRQs` (normal priority interrupt) and `FRQs` (fast interrupt), are asynchronous exceptions.

When an interrupt occurs, the system checks whether the interrupt mask bit for this kind of interrupt is 0 or 1. If this kind of interrupt is disabled, the interrupt would not be responded; otherwise, the system will handle the interrupt by executing the following steps:

- Save the `PC` to be returned to `ELR` register and the `PSTATE` to `SPSR` register.
- Disable the `DAIF` bits.
- Jump to the handler, which is usually written in assembly code. Switch the stack pointer to `SP_ELn` where n is the exception level of the handler.
- Save corruptible registers, including all common registers, `ELR`, `SPSR` of the exception level taken to and the `SP` of the exception level where the interrupt occurs.
- Call a C subroutine to handle the interrupt, and then return to the interrupt handler written by `asm`.
- Restore the registers which are saved previously.
- Use `eret` instruction to continue executing the code where the interrupt occurs. Switch the stack pointer to `SP_ELm` where m is the exception level of where the interrupt occurs.

## 2.2 Trap Frame

The trap frame structure contains the registers to be saved when an interrupt occurs.

The trap frame is designed as:

```
struct trapframe {
    uint64_t elr_el1, spsr_el1, sp_el0;
    uint64_t r0, r1, /* ... */ r29, r30;
};
```

The design of the trap frame should consider:

- Interrupts are not function calls, so both caller-saved registers and the callee-saved registers should be involved in the trap frame.
- The `push` and `pop` sequence in the interrupt handler should be uniform to the placement of the `trapframe` structure. The trap frame structure pointer would be used.

## 2.3 Store and Reload Registers

Building the trap frame involves a set of `push` operations:

```
stp x27, x28, [sp, #-16]!
```

System registers can not communicate directly with the memory. Store the value of system registers to common registers, and then `push` it:

```
mrs x2, spsr_el1
mrs x3, elr_el1
stp x3, x2, [sp, #0x100]
```

After building the trap frame, jump to the `trap` function via `bl trap` instruction. The `trap` function has an argument `struct *trapframe tf` which is transfered by `r0` register.

Similarly, reloading the registers involves a set of `pop` operations. When it comes to system registers, the value is transfered in some common registers and then sent to the system registers. After restoring all the registers, execute the `eret` instruction.

## 2.4 Test Snapshot