ARM64 IRQ handling

Hello There!

Let's continue on our journey of understanding a mammoth that is the ARM64 architecuture.

In the previous article, we went through the flow involved in switching from EL0 to EL1. In this one, we will cover the handling of interrupts by an ARM-A Class core involving the interactions with GIC (Generic Interrupt Controller) and how does the flow look in case of an SMP system.

Bird's eye view of the entire flow would be (we will expand on each of the item in this post):

- The CPU detects the IRQ and switches to EL1 (kernel mode, assuming a typical OS setup).
- Control jumps to the IRQ entry in the vector table.
- The hardware saves some state automatically, but we need to save the rest.
- We interact with the Generic Interrupt Controller (GIC) to figure out which interrupt fired.
- We handle the interrupt, acknowledge it, and signal its completion.
- Finally, we restore state and return to EL0.

When an IRQ Strikes at EL0:

Imagine our ARMv8-A core is diligently crunching numbers at EL0, the lowest exception level, where user applications typically run (may be calcuting the value of Pi to a trillionth decimal). Suddenly, an external interrupt (IRQ) signal asserts. What happens?

1. Vector Table

When an interrupt occurs, the ARM processor performs a semi hardware-assisted exception handling (What do I mean by semi-assisted, we will find answer real soon).

The first step is the transfer of control to the appropriate exception vector (recall we had talked about vector tables in our previous posts as well).

For synchronous exceptions (like data aborts) or asynchronous exceptions (like IRQs and FIQs) taken from EL0, the processor jumps to an entry point specified in the Vector Base Address Register (VBAR_EL1).

Each exception level (EL1, EL2, EL3) has its own VBAR. When an exception is taken, the processor determines the target exception level (which is EL1 if we're coming from EL0) and uses that EL's VBAR to locate the vector table. The vector table contains addresses of the exception handlers.

For an IRQ taken from EL0, the processor will jump to the "Synchronous/SError/IRQ/-FIQ taken from EL0" offset within the EL1's vector table.

NOTE: While we are executing at EL0, any interrupt will be taken to EL1. This is a fundamental concept in ARMv8-A exception handling. We can't directly handle interrupts at EL0; the architecture mandates a transition to a higher exception level (typically EL1) to handle them. IRQs/FIQs and other exceptions are deemed crucial enough to not be handled at EL0 user space.

Code snippet which covers everything we have talked so far.

```
// EL1 Vector Table
.align 11 // Vector table must be 2KB aligned (2^11)

el1_vector_table: // This address will be loaded in VBAR_EL1 during boot time
    // ... other exception entries ...

    // EL1_SYNC: Synchronous exception from EL0 (At offset 0x400)
    b   el1_sync_handler

    // EL1_IRQ: IRQ from EL0 (At offset 0x480)
    b   el1_irq_handler     // This is where our IRQ handling journey begins!

    // EL1_FIQ: FIQ from EL0 (At offset 0x500)
    b   el1_fiq_handler

    // EL1_SERROR: SError from EL0 (At offset 0x580)
    b   el1_serror_handler

    // ... other exception entries ...

el1_irq_handler:
    // This is our main IRQ handler entry point
    // We'll dive into the details of this handler shortly
    // We will cover the full vector table structure in some other posts
    b   .
```

In our above snippet, when an IRQ occurs at EL0, control is transferred to the `el1_irq_handler` which is at an offset of `0x480` of the main vector table.

NOTE: The design aspect of ARM64 vector table is also fascinating. At first glance, it might seem like its riddled with a lot of redundant entries. But, each of them serve a purpose. We will cover each of them in detail in one of the next articles.

2. Are Interrupts Masked by default in an interrupt context?

To answer it shortly, **Yes, when an exception is taken, the hardware automatically masks asynchronous exceptions (IRQs and FIQs) at the target exception level.**

Specifically, upon entering an exception handler at EL1 (from EL0), the PSTATE.I (IRQ mask bit) and PSTATE.F (FIQ mask bit) bits are set to 1, effectively disabling further IRQs and FIQs. This prevents immediate re-entry into an interrupt handler before the current one has had a chance to save its context or establish its own interrupt handling policy.

Think of it like this: The hardware gives us a brief "breather" to get our act together before potentially dealing with another interrupt. This initial masking allows us to save the context of the interrupted program and set up our exception handling environment without being immediately interrupted again.

We can re-enable interrupts within our handler if we intend to support nested interrupts, but let's revist that topic a bit later. For now, we will keep our IRQs and FIQs masked.

3. GIC and CPU Interface's Interaction

The Generic Interrupt Controller (GIC) is the unsung hero of interrupt management in ARM-based systems. It's a separate hardware block responsible for aggregating interrupt sources, prioritizing them, and presenting them to the ARM core. The CPU interface is the part of the GIC that directly communicates with the ARM processor.

Macro level view of its working:

1. **Interrupt Source:** An external peripheral (e.g., a timer, a network controller, a GPIO pin) asserts its interrupt line.
2. **GIC Distributor:** The GIC Distributor collects these interrupt requests, manages their priority, and determines which core (in an SMP system) should receive the interrupt.
3. **GIC CPU Interface:** The GIC CPU Interface then signals the ARM core that an interrupt is pending. This is what causes the ARM processor to take the exception and jump to our `el1_irq_handler`.

To identify which interrupt has occurred and to acknowledge it, our CPU will interact with the GIC CPU Interface registers. These registers are memory-mapped, meaning we access them by reading from and writing to specific memory addresses.

4. Interrupt handling flow

Let's looks at a typical generic interrupt handler at EL1. I would also encourage folks to take a look at other kernel implementation and compare how there handling logic is written.

1. **Save Context:** Preserve the state of the interrupted EL0 program. This includes general-purpose registers (X0-X30), floating point registers, and possibly some other peripheral registers whose state might be cobbled by EL1. * NOTE: The stack pointer (SP_EL0), and the Exception Link Register (ELR_EL1) and Saved Program Status Register (SPSR_EL1) are automatically saved by hardware on exception entry.
2. **Identify Interrupt Source:** Read a GIC CPU Interface register to determine the Interrupt ID (INTID) of the pending interrupt. (Detailed below)
3. **Acknowledge Interrupt:** Write to a GIC CPU Interface register to acknowledge that the interrupt has been received by the CPU. This prevents the GIC from re-signaling the same interrupt. (Detauled below)
4. **Enable Interrupts (Optional, only required for Interrupt nesting):** If nested interrupts are desired, clear the PSTATE.I bit (and potentially PSTATE.F for FIQs) to allow higher-priority interrupts to be preempt the current context handling.
5. **Call Specific Handler:** Based on the INTID, dispatch to the appropriate device-specific interrupt handler (e.g., a timer interrupt handler, a UART interrupt handler).
6. **Disable Interrupts (for nesting):** Before returning from the generic handler, re-mask interrupts if they were enabled for nesting. This ensures we return to a controlled state. This is required as we are going to restore the context back and signal the GIC to mark the interrupt state correctly.
7. **Signal End of Interrupt (EOI):** Write to a GIC CPU Interface register to inform the GIC that the interrupt has been fully processed. This allows the GIC to clear the pending state of the interrupt and potentially forward other pending interrupts of lower priority.
8. **Restore Context:** Restore the saved state of the interrupted EL0 program.
9. **Return from Exception:** Use the `eret` instruction to return to EL0 and resume the interrupted program.

5. GIC - Interrupt #ID finding and acknowledgement

As per `Step-2` and `Step-3` we would want to identify the interrupt and acknowledge it using GIC CPU Interface registers. These are:

- **ICC_IAR1_EL1 (Interrupt Acknowledge Register):** Reading this register provides the Interrupt ID (INTID) of the highest priority pending interrupt. It also implicitly acknowledges the interrupt by causing the GIC to de-assert the interrupt signal to the CPU.

Code snippet:

```
// Assuming we are in el1_irq_handler

get_irq_id_and_ack:
    mrs x0, icc_iar1_el1    // Read Interrupt Acknowledge Register to get Interrupt ID (INTID)
                            // This also implicitly acknowledges the interrupt to the GIC.
                            // x0 now holds the INTID.

    // At this point, x0 contains the Interrupt ID.
    // We can now use this ID to dispatch to the appropriate handler.
    // For example, a jump table or a series of comparisons.

    // Example dispatch (simplified)
    // Most systems would typically have a better way of dispatching
    // most likely an array of function pointers.
    mov x1, #0x30           // Example: Assuming 0x30 is a timer interrupt
    cmp x0, x1
    b.eq handle_timer_irq

handle_timer_irq:
    // ... perform device-specific interrupt handling based on x0 (INTID) ...

    // After the device-specific handler completes, we need to signal End Of Interrupt.
    b   signal_eoi_and_return
```

NOTE: I'd want to remind ourselves that reading `ICC_IAR1_EL1` both retrieves the INTID *and* implicitly acknowledges the interrupt. We don't need a separate write to acknowledge. That's a clever design if you ask me.

6. GIC - Signalling End of Interrupt

After we've handled the specific interrupt, we must inform the GIC that we're done. This is achieved by writing the same INTID we received from `ICC_IAR1_EL1` back to `ICC_EOIR1_EL1`.

Code snippet

```
signal_eoi_to_gic:
    // Assuming x0 still holds the INTID that was read from icc_iar1_el1 in earlier snippet
    msr icc_eoir1_el1, x0   // Write INTID to End Of Interrupt Register
                            // This tells the GIC we're done with this interrupt.
```

Thinking time: Why do we need ICC_EOIR1_EL1 when ICC_IAR1_EL1 already acknowledges the interrupt? The ICC_IAR1_EL1 acknowledges the *CPU's reception* of the interrupt, stopping the GIC from sending it again to the CPU interface. ICC_EOIR1_EL1, on the other hand, tells the GIC to clear the *pending state* of the interrupt within the GIC Distributor. This allows other interrupts of lower priority (that might have been masked by the current interrupt's priority) to now be forwarded. Without signaling EOI, the GIC might think the interrupt is still active and prevent other interrupts from reaching the CPU.

7. Putting everything together

Let's put all these pieces together into a more complete generic IRQ handler.

Code snippet

```
.align 11 // Vector table must be 2KB aligned (2^11)

el1_vector_table:
    // ... other exception entries ...

    // EL1_IRQ: IRQ from EL0 (At offset 0x480)
    b   el1_irq_handler

    // ... other exception entries ...



// Our main IRQ handler entry point from EL0
el1_irq_handler:
    // --- Step 1: Save Context ---
    // Save general-purpose registers that might be clobbered.
    // x0, x1 are used in this example, so we'll save them only.
    // In a real system, you'd save all caller-saved registers or push them to a stack.
    // Recall our save and restore snippets from the previous articles

    stp x0, x1, [sp, #-16]!      // Push x0, x1 onto stack
    // REMEBER: The hardware automatically saves ELR_EL1 (Exception Link Register) and
    // SPSR_EL1 (Saved Program Status Register) upon exception entry.
    // They will be restored back when we return back to EL0 using eret.

    // --- Step 2: Identify Interrupt Source and Acknowledge ---
    mrs x0, icc_iar1_el1         // Read Interrupt Acknowledge Register
                                 // x0 now holds the Interrupt ID (INTID).
                                 // This also implicitly acknowledges the interrupt.
    // --- Step 3: Handle Nested Interrupts (Optional) ---
    // If we want to allow higher-priority interrupts to preempt us, we can re-enable IRQs here.
    // This involves clearing the I bit in PSTATE.
    // For now, let's keep them masked for simplicity in this initial flow.
    // For nested interrupts, you would typically have a more complex
    // interrupt prioritization and re-enabling strategy.

    // --- Step 4: Dispatch to Specific Handler ---
    // In a real system, you'd have a dispatch table or a series of comparisons
    // to call the appropriate device driver handler based on the INTID in x0.
    // For demonstration, let's just simulate a simple handler.

    // Example: If INTID is 0x30 (as an example)
    cmp x0, #0x30
    b.eq handle_timer_irq

    // If it's not our specific handler, perhaps a default or error handler
    b handle_unknown_irq

handle_timer_irq:
    // This is our device-specific timer interrupt handler.
    // Here, you would interact with the timer hardware, clear its pending bit, etc.
    // For example, read a timer register, increment a counter.
    mov x1, #0xDEADBEEF        // Simulate some work
    // ... actual timer handling code ...

    b irq_handler_done

handle_unknown_irq:
    // Log an error or take appropriate action for an unhandled interrupt
    mov x1, #0xBADCODE         // Simulate error handling
    // ... error handling code ...

irq_handler_done:
    // --- Step 5: Signal End of Interrupt (EOI) ---
    // Assuming x0 still holds the original INTID
    msr icc_eoir1_el1, x0       // Write INTID to End Of Interrupt Register

    // --- Step 6: Restore Context ---
    ldp x0, x1, [sp], #16       // Pop x0, x1 from stack
    // --- Step 7: Return from Exception ---
    eret                        // Return from Exception to EL0
```

I tried to keep the final snippet simplified enough yet covering all the key details which are necessary while handling an interrupt and returing back to the user space context. In the next one, we will look at FIQ handling in detail.