

HW4-1

Entry.S modified to call IRQ_Handler() and FIQ_Handler()

From the lab slides, Handler.c already defines these two C handlers:

```
Lab04_interrupt
#include "stdbool.h"
#include "stdint.h"
#include "HallInterrupt.h"
```

```
__attribute__ ((interrupt ("IRQ"))) void IRQ_Handler(void)
{
    Hal_interrupt_run_handler();
}
```

```
__attribute__ ((interrupt ("FIQ"))) void FIQ_Handler(void)
{
    while (true);
}
```

So in Entry.S we only need to make the **IRQ and FIQ vectors jump to these symbols** instead of just looping forever.

```
.section .text
.global _start
.global reset_handler

.extern main      @ C main()
.extern IRQ_Handler @ C IRQ handler in Handler.c
.extern FIQ_Handler @ C FIQ handler in Handler.c

/* -----
 * Exception vector table (address 0x00000000)
 * Each entry uses "LDR PC, [addr]" to jump to the real handler.
 * -----
_start:
vector_start:
    LDR  PC, reset_handler_addr    @ 0x00 Reset
    LDR  PC, undef_handler_addr   @ 0x04 Undefined instruction
    LDR  PC, svc_handler_addr     @ 0x08 Supervisor call (SVC)
    LDR  PC, pftch_abt_handler_addr @ 0x0C Prefetch abort
    LDR  PC, data_abt_handler_addr @ 0x10 Data abort
    B   .             @ 0x14 Reserved
    LDR  PC, irq_handler_addr    @ 0x18 IRQ → IRQ_Handler()
    LDR  PC, fiq_handler_addr    @ 0x1C FIQ → FIQ_Handler()

/* -----
 * Addresses of each handler (used by the LDR PC instructions above)
 * -----
reset_handler_addr:  .word reset_handler
undef_handler_addr:  .word undef_handler
svc_handler_addr:   .word svc_handler
pftch_abt_handler_addr: .word pftch_abt_handler
data_abt_handler_addr: .word data_abt_handler
irq_handler_addr:   .word IRQ_Handler  @ NEW: C IRQ handler
fiq_handler_addr:   .word FIQ_Handler  @ NEW: C FIQ handler

/* -----
 * Reset handler: set up stack and call main()
 * -----
reset_handler:
    LDR  SP,=_stack_top    @ set stack pointer
    BL   main            @ jump to C code
    1: B   1b            @ if main returns, loop forever

/* Simple default handlers that just spin (unused in this lab) */
undef_handler:
    B   undef_handler
```

```

svc_handler:
B svc_handler

pftch_abt_handler:
B pftch_abt_handler

data_abt_handler:
B data_abt_handler

```

– The IRQ vector (0x18) now uses LDR PC, irq_handler_addr and irq_handler_addr holds the address of Irq_Handler.
– The FIQ vector (0x1C) now uses LDR PC, fiq_handler_addr and fiq_handler_addr holds the address of Fiq_Handler.
So when an IRQ or FIQ occurs, the CPU branches into the C handlers in Handler.c.

Build and run:

make

make run

HW4-2

Explain how keyboard input is handled

After the binary is running, **what happens when you press a key in QEMU?**

Here is the step-by-step flow, referencing the lab code.

1. UART hardware receives the key and raises an interrupt

- QEMU connects the terminal keyboard to **UART0** (PL011).
- When you press a key:
The UART hardware stores the byte in its RX FIFO,
Sets the RX interrupt status bit, and
Drives its interrupt line into the **GIC** (Generic Interrupt Controller).

In Hal_uart_init() we enabled RX interrupt and registered a handler:

```

void Hal_uart_init(void)
{
    // Enable UART (TX and RX)
    Uart->uartcr.bits.UARTEN = 0;
    Uart->uartcr.bits.TXE = 1;
    Uart->uartcr.bits.RXE = 1;
    Uart->uartcr.bits.UARTEN = 1;

    // Enable UART RX interrupt in the UART
    Uart->uartimsc.bits.RXIM = 1;

    // Enable this interrupt in the GIC and register handler
    Hal_interrupt_enable(UART_INTERRUPT0);
    Hal_interrupt_register_handler(interrupt_handler, UART_INTERRUPT0);
}

```

So the UART input interrupt (number 44) is connected to the GIC and has a C handler called `interrupt_handler`.

2. GIC sends IRQ to the CPU, CPU jumps to Irq_Handler()

- The GIC asserts **nIRQ** to the Cortex-A8 when the UART interrupt is pending.

Lab04_interrupt

- The CPU switches to IRQ mode and **fetches the instruction at the IRQ vector** (0x18).
- Thanks to our modified Entry.S, that entry loads PC with Irq_Handler's address:

LDR PC, irq_handler_addr @ from vector table

...

irq_handler_addr: .word Irq_Handler

So `Irq_Handler()` in Handler.c is executed.

3. Irq_Handler() asks the HAL to run the correct device handler

In Handler.c:

```

Lab04_interrupt
__attribute__((interrupt ("IRQ"))) void Irq_Handler(void)
{
    Hal_interrupt_run_handler();
}
```

- }
- The interrupt("IRQ") attribute makes the compiler generate prologue/epilogue to save/restore registers and return with the proper **exception return**.
 - Inside, we call `Hal_interrupt_run_handler()` to dispatch to the correct peripheral handler.

4. `Hal_interrupt_run_handler()` talks to the GIC

Implementation from Interrupt.c:

Lab04_interrupt

```
void Hal_interrupt_run_handler(void)
{
    uint32_t interrupt_num = GicCpu->interruptack.bits.InterruptID;

    if (sHandlers[interrupt_num] != NULL)
    {
        sHandlers[interrupt_num](); // call registered handler
    }

    GicCpu->endofinterrupt.bits.InterruptID = interrupt_num;
}
```

- It reads the **Interrupt Acknowledge Register** to get the interrupt ID (for UART, `interrupt_num = UART_INTERRUPT0 = 44`).
- It looks up the function pointer in `sHandlers[44]` and calls it.
- Finally it writes the ID to **End Of Interrupt** register to tell the GIC the interrupt has been serviced.

We registered the UART's handler earlier in `Hal_uart_init()` with `Hal_interrupt_register_handler(interrupt_handler, UART_INTERRUPT0);`.

5. The UART interrupt handler reads and echoes the key

The registered handler in Uart.c:

```
static void interrupt_handler(void)
{
    uint8_t ch = Hal_uart_get_char(); // 1) read received byte
    Hal_uart_put_char(ch); // 2) echo it back to console
}



- Hal_uart_get_char() waits until the UART RX FIFO is not empty and then reads Uart->uartdr.bits.DATA.
- Hal_uart_put_char() waits until TX FIFO has space and writes the same byte into the transmit register.

```