Interrupts and communication

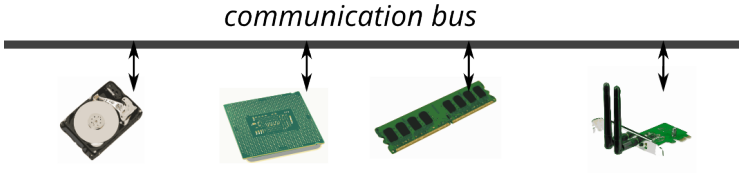
Gaël Thomas Mathieu Bacou

Contents

Communication buses

Communication buses

- Hardware components communicate via buses

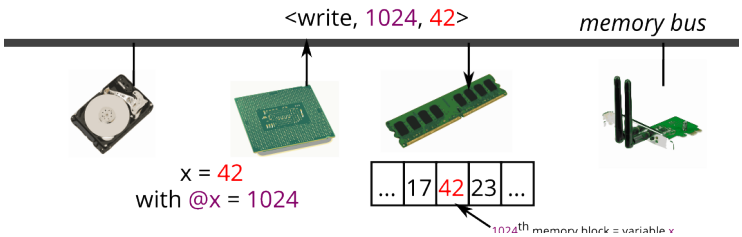


- From a software point of view, 3 main buses
  - Memory bus: mainly to access memory
  - Input / output bus: messages from CPUs to devices
  - Interrupt bus: messages from peripherals to CPUs
- From the hardware point of view: a set of hardware buses with different protocols that can multiplex the software buses

---

The memory bus

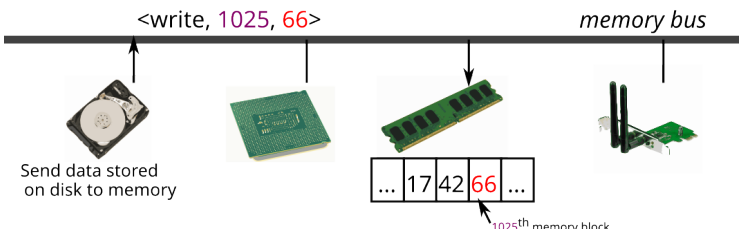
- Processors use the memory bus for reads / writes
  - Sender: the processor or a peripheral
  - Receiver: most often memory, but can also be a device (*memory-mapped IO*)



DMA: Direct Memory Access

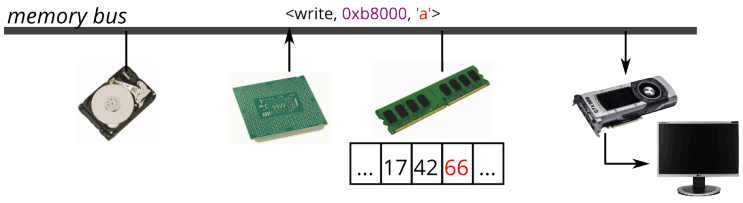
- Devices use the memory bus for reads/writes
  - Sender: a processor or a peripheral
  - Receiver: most often memory, but can also be a device (*memory-mapped IO*)
- The DMA controller manages the transfer between peripherals or memory
  - The processor configures the DMA controller
  - The DMA controller performs the transfer
  - When finished, the DMA controller generates an interrupt

  → The processor can execute instructions during an I/O



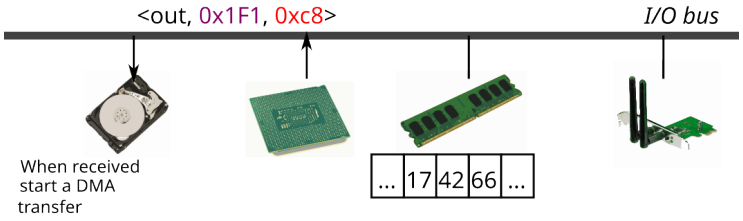
MMIO: Memory-Mapped IO

- Processors use memory bus to access devices
  - Sender: a processor or a peripheral
  - Receiver: most often memory, but can also be a device (*memory-mapped IO*)
- Device memory is *mapped* in memory
  - When the processor accesses this memory area, the data is transferred from / to the device



The input / output bus

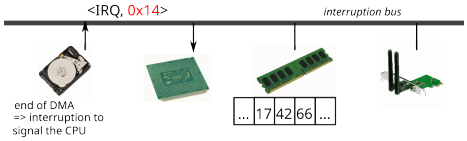
- Request / response protocol, special instructions `in`/`out`
  - Sender: a processor
  - Receiver: a peripheral
  - Examples: activate the caps-lock LED, start a DMA transfer, read the key pressed on a keyboard ...



The interrupt bus - principle

- Used to signal an event to a processor
  - Sender: a peripheral or a processor
  - Receiver: a processor

  - Examples: keyboard key pressed, end of a DMA transfer, millisecond elapsed ...
  - **IRQ** (*Interrupt ReQuest*): interrupt number. Identifies the sending device



Interrupts

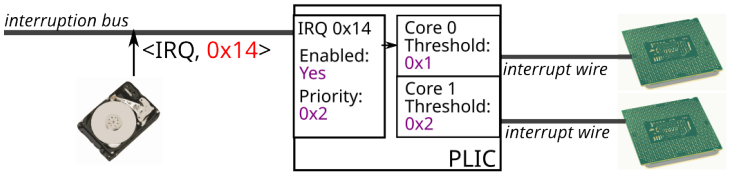
Receiving an interrupt: simple routing

- Devices are wired to a **Platform-Level Interrupt Controller** (PLIC)
  - **IRQ**: input wire number of a device on the PLIC
  - The configuration of the PLIC is done by MMIO
  - The PLIC is wired to every processor to actually interrupt them
- The OS configures interrupts for each processor and each privilege mode
  - IRQ routing is achieved by selecting which processors receive which interrupts
  - There are also **priorities** of interrupts
- The OS sets its **interrupt handler** by writing its address in register `stvec`
  - An interrupt handler (a function) usually checks the interrupt type (e.g., device, timer, etc.), and then delegates handling to other functions
  - Depending on the context (typically, user or system mode), the OS swaps handler

---

Receiving an interrupt: example

1. A block device on IRQ line `0X14` signals a data block is available
2. The PLIC reads the configured priority of IRQ `0x14`: `0x2`
3. The PLIC signals all processors with priority threshold $<$ `0x2`
4. All signaled processors compete to serve the interrupt
   - The first processor that gets to serve the interrupt (i.e., execute its interrupt handler) writes to the PLIC to indicate the interrupt is served
   - Other signaled processors check that the interrupt is not already served, and resume normal operation otherwise



Interrupt routing in NUMA architectures

In Non Uniform Memory Access architectures (NUMA), a device is linked to only one NUMA node. On `RISC-V` architectures, this means a device is linked to only one PLIC, as there is one PLIC per NUMA node. So only a processor from this NUMA node can serve interrupts from this device.

Interrupt routing on `x86` architecture

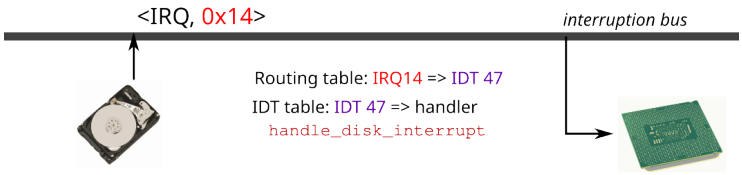
On `x86`, interrupt routing goes through two tables configured by the kernel:

1. **Routing table**: associates an **IRQ** with an **IDT** number
2. **IDT table** (*interrupt descriptor table*): associates an **IDT** number to a **interrupt handler**

Two tables allow more flexibility than a single table which associates an IRQ number directly with a manager. This is different from `RISC-V` architecture where there can only be one interrupt handler, that must check the kind of the interrupt to serve it (e.g., a device interrupt, a timer interrupt, etc.).

Example of routing:

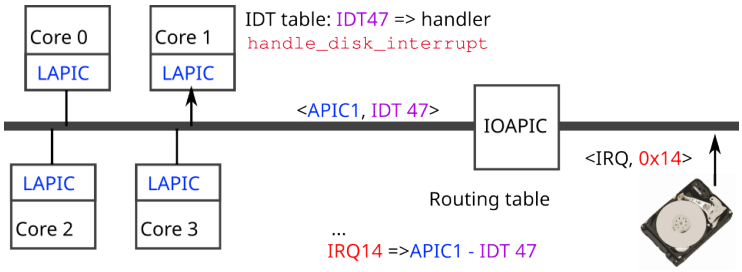
- A device sends an **IRQ** (for example `0x14`)
- The **routing table** associates `IRQ14` with `IDT47`
- The **IDT table** indicates that `IDT47` is managed by the function `handle_disk_interrupt`



This is with only one processor; on multicore `x86` systems:

- XAPIC protocol on pentium (x2APIC since Intel Core processors)
  - Each core has a number called APIC number (*Advanced Programmable Interrupt Controller*)
  - Each core handles interrupts via its LAPIC (*local APIC*)
  - An IOAPIC routes an interrupt to a given LAPIC

∗ Routing table configured by the system kernel

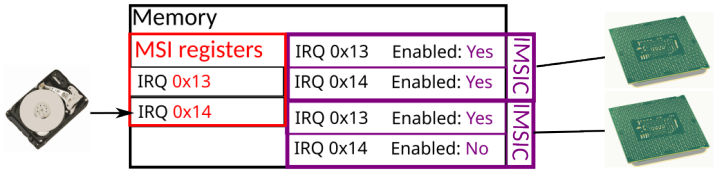


Receiving an interrupt: simple routing (continued)

- In the processor, after executing **each instruction**
  1. Check if an interrupt has been received
  2. If so, switch to kernel mode and run the interrupt handler
  3. Then switch back to the previous mode and continue the execution
- Note: a handler can be run at **anytime**
  - Problem of concurrent access between handlers and the rest of the kernel code
  - The solution is to mask interrupts, two ways:
    1. raise priority threshold of IRQs accepted by the processor
    2. disable them by clearing bit `SIE` (Supervisor Interrupts Enable) in register `SSTATUS`

MSI: Message Signaling Interrupt for advanced interrupt management

- MSI: direct interrupts from devices to processors
  - Each processor has its own IMSIC (Incoming MSI Controller)
  - Different from the PLIC interrupting *all* processors that may serve an interrupt
- The OS configures an IMSIC via MMIO to enable or disable an interrupt
- The OS configures a device to direct its interrupts to MSI registers
- Used for performance or fine granularity in interrupt routing



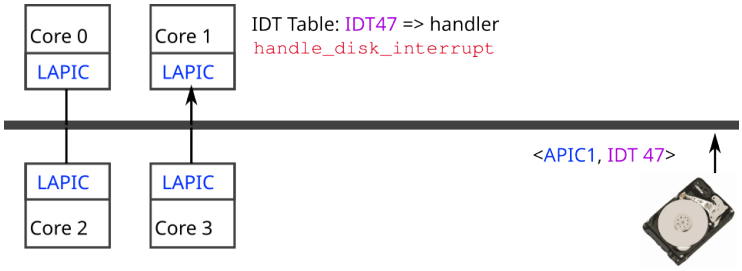
MSIs and PLIC in `RISC-V` architectures

The PLIC (usually an APLIC, *Advanced PLIC*) remains used by devices that do not (need to) support MSI. When the platform supports MSIs, the APLIC converts wired (i.e., non message signaled) interrupts into MSIs. This is configured by the OS, as if they were direct MSIs from external devices.

MSIs on `x86`

On `x86` systems, MSIs work roughly the same:

- MSI: direct interrupt from a device to a LAPIC without passing through the IOAPIC
  - The kernel must configure the device so that it knows which LAPIC / IDT pair should be generated
  - Used when the need for performance is important

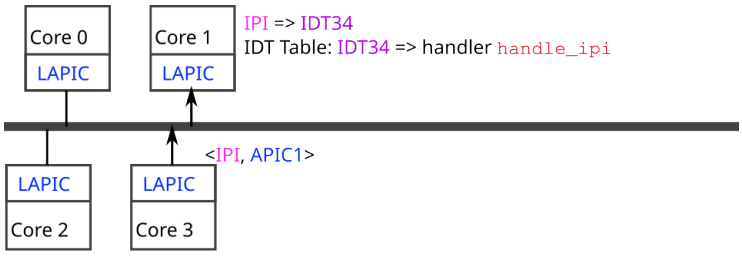


Inter-core communication

- One core can send an interrupt to another core: this is called Inter-Processor Interrupt (IPI)
- Without MSIs:
  - To send an IPI, a processor writes to another processor's Core-Local Interrupt Controller (CLINT) (in its MMIO registers)
  - The destination processor receives a *software interrupt*
- With MSIs and IMSICs:
  - A processor writes to another processor's IMSIC like a device

IPIs on `x86` architectures

- One core can send an interrupt to another core
  - LAPIC x sends an IPI to LAPIC y
  - In LAPIC y, receiving an IPI is associated with an IDT number



Other interruptions: system calls and exceptions

- The interrupt handler (the function addressed by register `stvec`) is also called when **system calls** and **exceptions** occur
  - system calls are called by the userspace by executing the instruction `ecall`, which triggers an interrupt of this type
  - exceptions are faults that occur when executing instructions
    - they trigger an interrupt that matches the fault type
    - for instance, trying to read from an illegal address triggers a software interrupt with exception code `0x5`
  - In other words, `stvec` points to the unique entrypoint into the kernel:
    - from the software, via system calls or IPIs
    - from the hardware, via external interruptions and exceptions

On `x86` systems, the IDT table is used for every possible interruption:

- Used by **interrupts** as seen previously
- But also for a **system call**: `int 0x64` simply generates the interrupt `IDT 0x64`
- But also to catch *faults* when executing instructions
  - a division by zero generates the interrupt `IDT 0x00`, an access illicit memory (`SIGSEGV`) the interrupt `IDT 0x0e` etc.

The IDT table is therefore the table that contains all of the entry points to the kernel:

- From the software via the system call
- From material for other IDTs

Time management: two sources

- **Jiffies**: global time source to update the date
  - A dedicated device or the CLINT regularly sends IRQ
  - Only a single core serves this IRQ to update the date

- **Tick**: core-local time source used for scheduling
  - CLINTs are also used to generate periodic interrupts to their cores
  - The system associates a handler with this timer interrupt
  - May be less precise than the **jiffies**