Computer Organization and Networks

(INB.06000UF, INB.07001UF)

Chapter 10: Polling and Interrupts

Winter 2021/2022

Stefan Mangard, www.iaik.tugraz.at

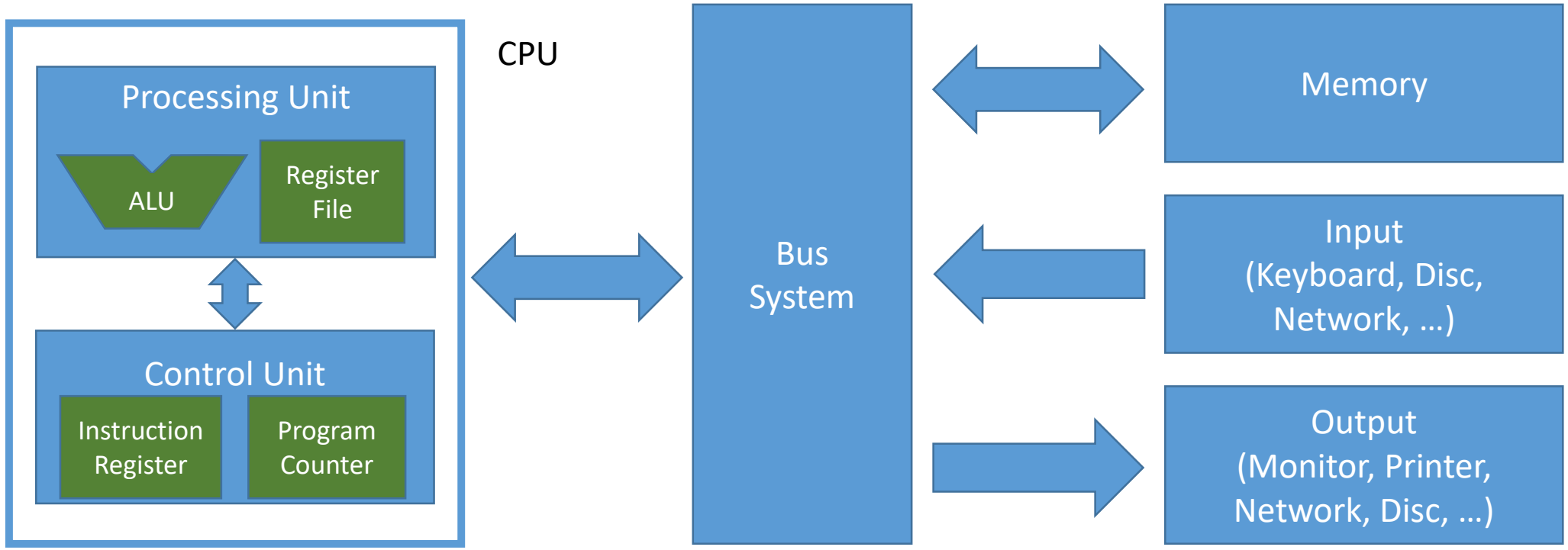
What We Covered So Far …

Logic Gates
Combinational Circuits
Finite State Machines
Instructions, Instructions Sets
CPUs
Assembler
Stack, Calling Conventions
C Programming
Link and Network Layer
Network and Transport Layer
Application Layer

- We skipped many details at the different abstraction layers
- We now focus again on CPUs

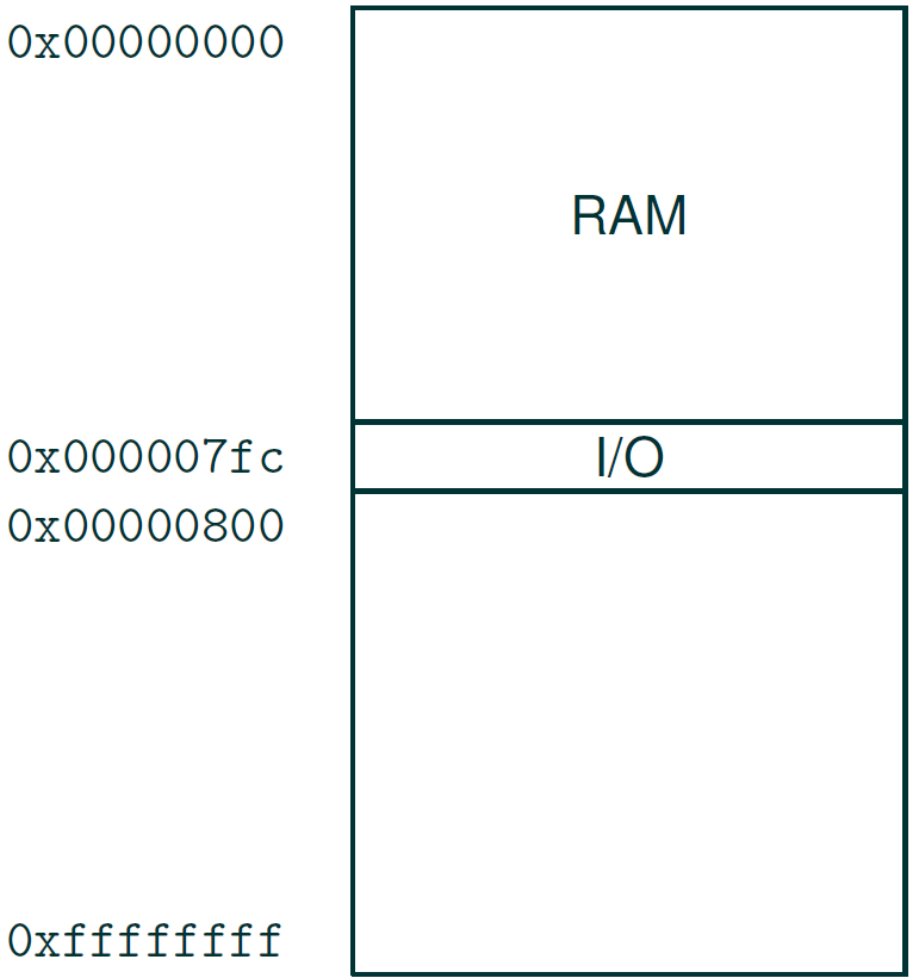
- First topic: How do to handle I/O efficiently at the CPU level?

Von Neumann Model



Our Example I/O

- The I/O interface that we discussed so far is idealized debug interface (data is always valid)
- In practice there is the following challenge:
  - The CPU executes one instruction after the other.
  - How should it know when the input is valid? Is it valid always (in every clock cycle)?



Example

- Assume an input port of a computer is set to a value 1 in one clock cycle
- It is still 1 in the next clock cycle
- Does this mean this is the “same” 1 or does this mean that there is a “second” 1?
- How should the computer know?

We Need to Add a Flag

No Mail for You

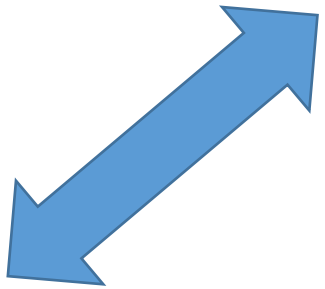
You've Got Mail

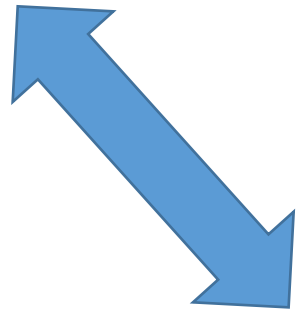
Synchronization with Control Signals

- On real communication channels, data is not always ready
- We need synchronization with control signals
- There exist different protocols and standards.
  - Serial protocols: RS232, SPI, USB, SATA, . . .
  - Parallel protocols: PATA/IDE, IEEE 1284 (Printer), . . .

- We use a simple interface with few control signals to illustrate this
  - 8-bit data port
  - Simple valid/ready flow-control
  - Registers (memory mapped)
    - 0x7D0 (control register)
    - 0x7D4 (data register)

Implementing an Interface With a Control Register



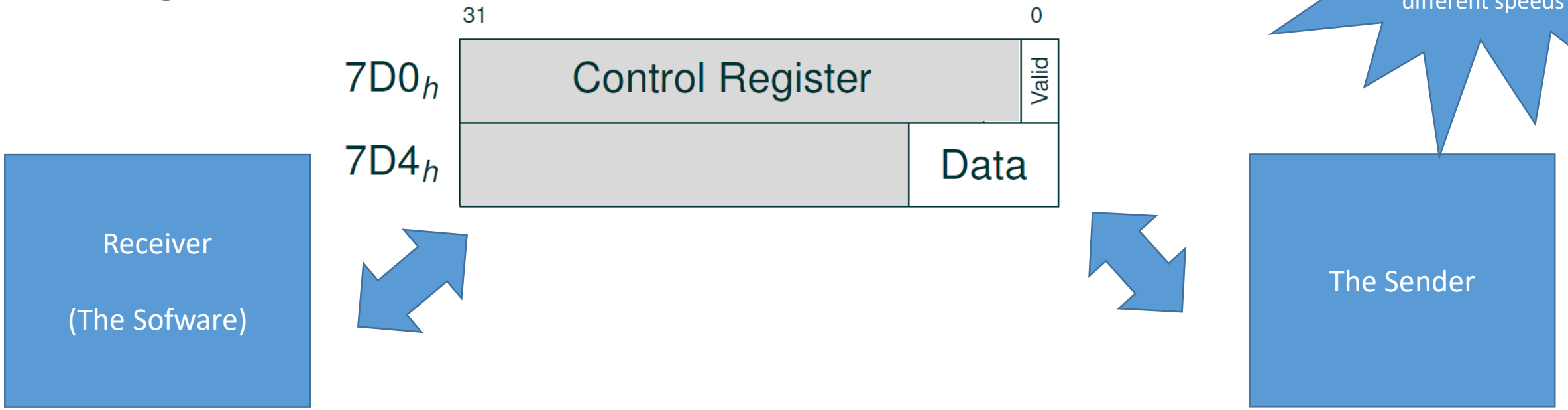


Receiver

(The Sofware)

The Sender

Implementing an Interface With a Control Register



Example of a basic protocol:

(4) Receiver (the software) waits until valid bit is set

(5) Receiver reads the data

(6) Receiver clears the valid bit

(1) Sender waits until valid bit is cleared (set to 0)

(2) Sender sets the data value

(3) Sender sets the valid bit

Polling Using a Control Register by the sender

Pseudoinstruction for

beq t1,zero, POLL_PARIN


```
POLL_PARIN:
  LW t1, 0x7D0(x0)     # Read PARIN CTRL REG
  ANDI t1, t1, 1
  BEQZ t1, POLL_PARIN

  LW t2,   0x7D4(x0)   # Read available data
  SW zero, 0x7D0(x0)   # Acknowledge Data
```

Control Signals

- There is a wide range of options for implementing communication between entities (FSMs, software, humans, …) of with different speeds

- However, in all cases, there needs to be signals to ensure that
  - The sender knows that the resource (bus, register, …) is available
  - The receiver knows that there is valid input
  - The sender knows that the receiver has received the signal (acknowledge)

Communication via a Slow Communication Interface

- Polling is highly inefficient: the CPU is stuck in a loop until e.g.
  - an I/O peripheral sets a ready signal
  - a timer has reached a certain value
  - the user has pressed a key
  - ….
- Alternative
  - CPU keeps executing some useful code in the first place
  - We use concept of interrupts to react to “unexpected” events
  - Basic idea: Instead of waiting for an event, we execute useful code and then let an event trigger a redirection of the instruction stream

How to handle unexpected external events?

- We add an input signal to the CPU called “interrupt”.
- An external source can activate this input signal “interrupt”.
- After executing an instruction, the CPU checks for the value of this input signal “interrupt” before it fetches the next instruction.
- If the signal “interrupt” is active, the next instruction to be executed is the first instruction of the “interrupt-service routine”.
- After “handling” the interrupt by executing the interrupt-service routine, the CPU returns to the interrupted program.

Interrupts in RISC-V

- **Hardware Aspects**
  - External interrupt is an input signal to the processor core
  - Control & Status registers (CSRs) for interrupt configuration (e.g. mie, mtvec, mip, …)
  - Additional instructions for interrupt handling (mret)
  - Dedicated interrupt controllers on bigger processors

- **Software Aspects**
  - When an interrupt occurs, the program execution is interrupted
  - Special functions have to be provided to handle interrupts → Interrupt Service Routines (ISR)
  - Software needs to configure and enable interrupts
  - Software has to preserve the interrupted context
    → Interrupt entry points are typically written in assembly

Control & Status Registers (CSRs) in RISC-V

- We so far only considered memory-mapped peripherals whose registers can be accessed via standard load and store instructions

- RISC-V also features dedicated so called “Control & Status Registers”
  - The ISA allows addressing 4096 registers (32 bit each)

  - Dedicated instructions allow to read and write these registers: CSRRW, CSRRS, CSRRC, CSRRWI, CSRRSI, CSRRCI

The Interrupt Service Routine (ISR)

- Entering the ISR
  - Upon an interrupt, the processor
    - jumps to a location in memory specified by the **mtvec** CSR.
    - automatically stores the previous location into **mepc** CSR.
- Executing the ISR
  - The ISR can execute arbitrary code; However, the processor context (program counter, register) needs to have exactly the same values when returning to the interrupted code → “From the view of the interrupted program, the execution after the interrupt continues as if nothing had happened”
- Leaving the ISR
  - Upon the execution of the **mret** instruction, the processor
    - returns to the original location stored in the mepc CSR

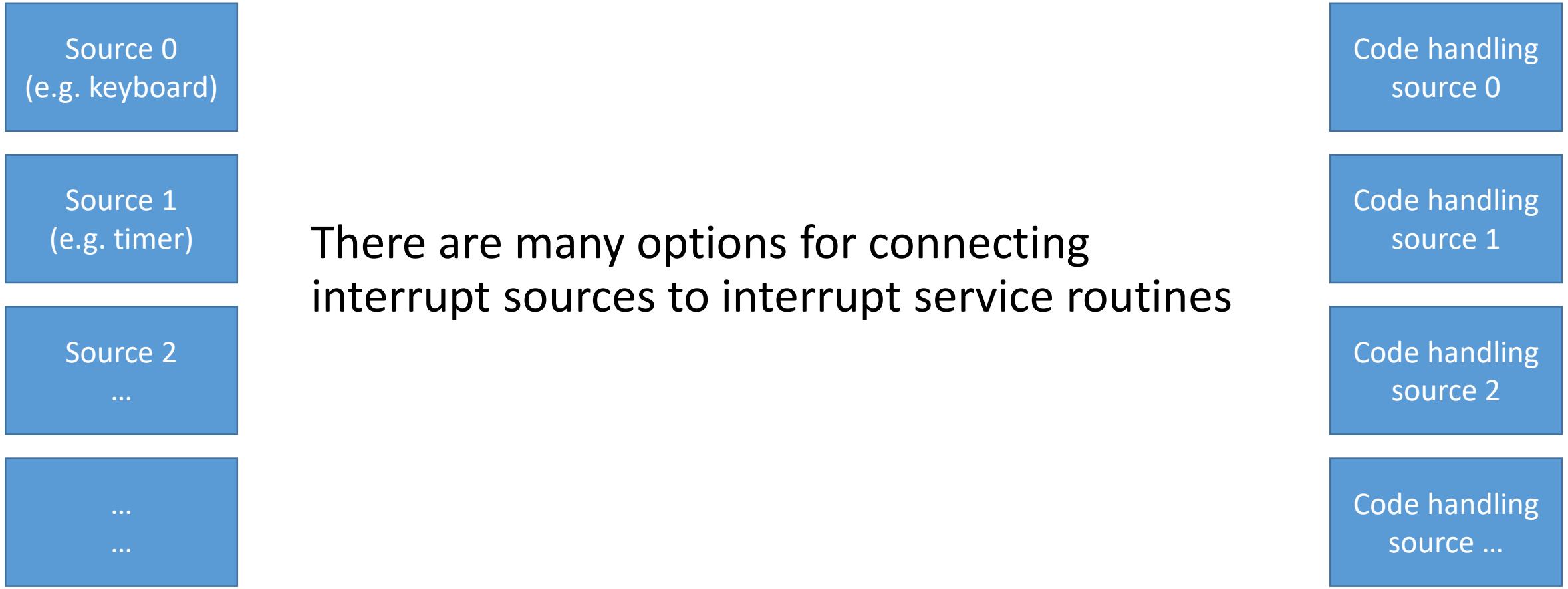
Finding the Interrupt Service Routine

- Two approaches are common:
  - Single entrypoint for all interrupts.
    - the ISR has to determine what caused the interrupt and then handles the corresponding interrupt
  - Multiple entrypoints for different interrupts organized in a table (**vectored interrupts**)
    - A table defines the entry point for different causes of interrupts
    - E.g. each interrupt vector table entry has 4 bytes
      - Interrupt cause 0 leads to a jump to mtvec
      - Interrupt cause 1 leads to a jump to mtvec+4
      - Interrupt cause 2 leads to a jump to mtvec+8
      - …

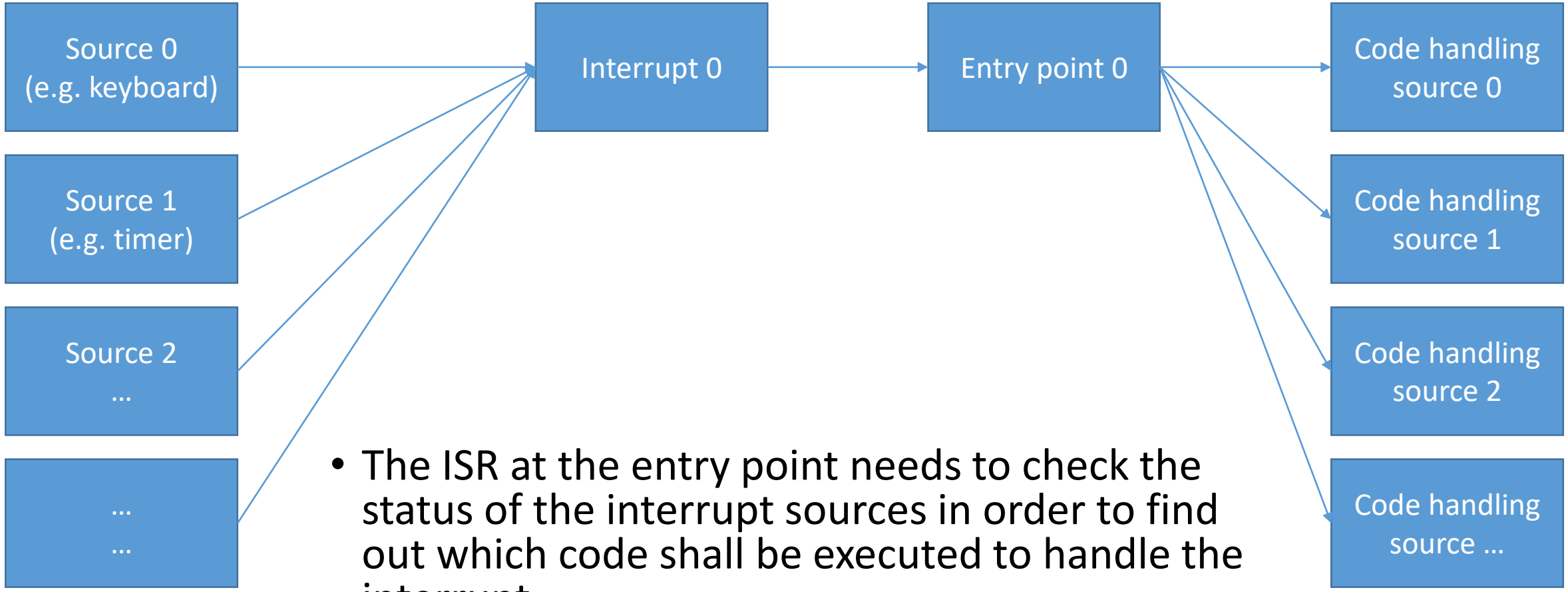
      →just enough space to place a single **jal** instruction to the actual ISR handler code at each entry location

- RISC-V permits both approaches

Connecting Interrupt Sources to Interrupt Service Routines

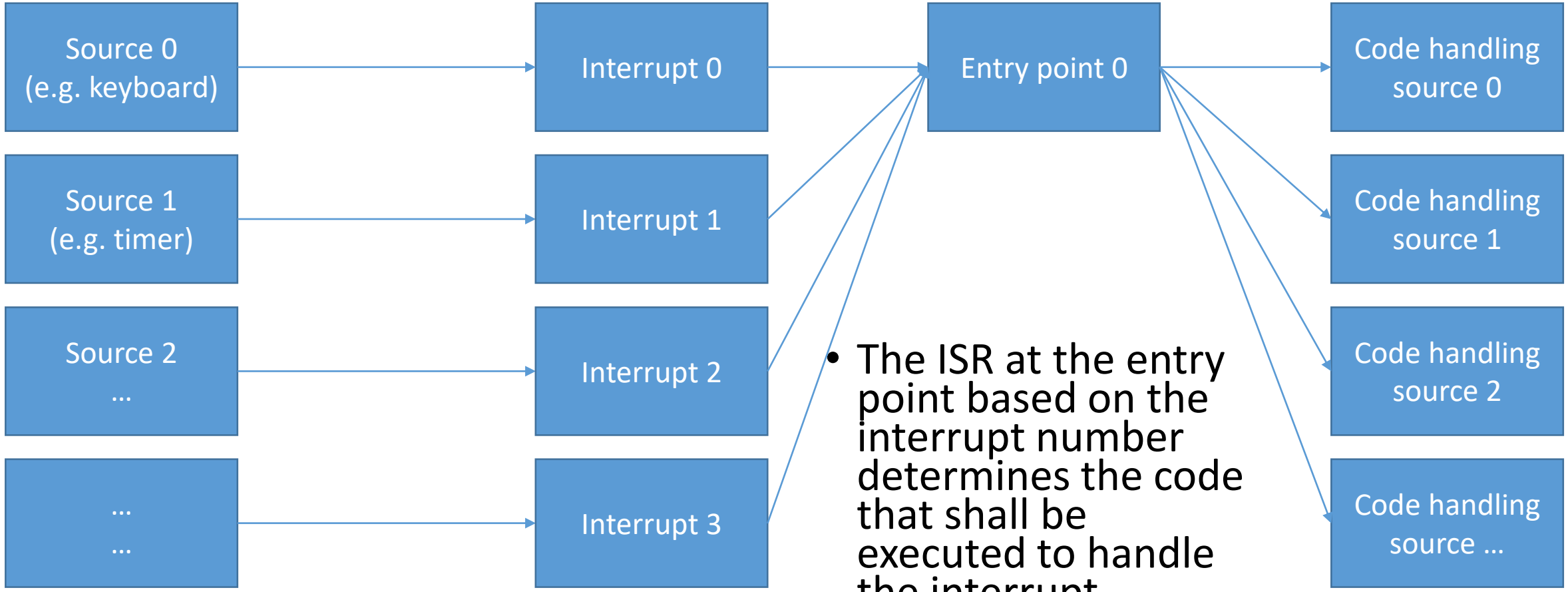


Connecting Interrupt Sources to Interrupt Service Routines (one Interrupt)



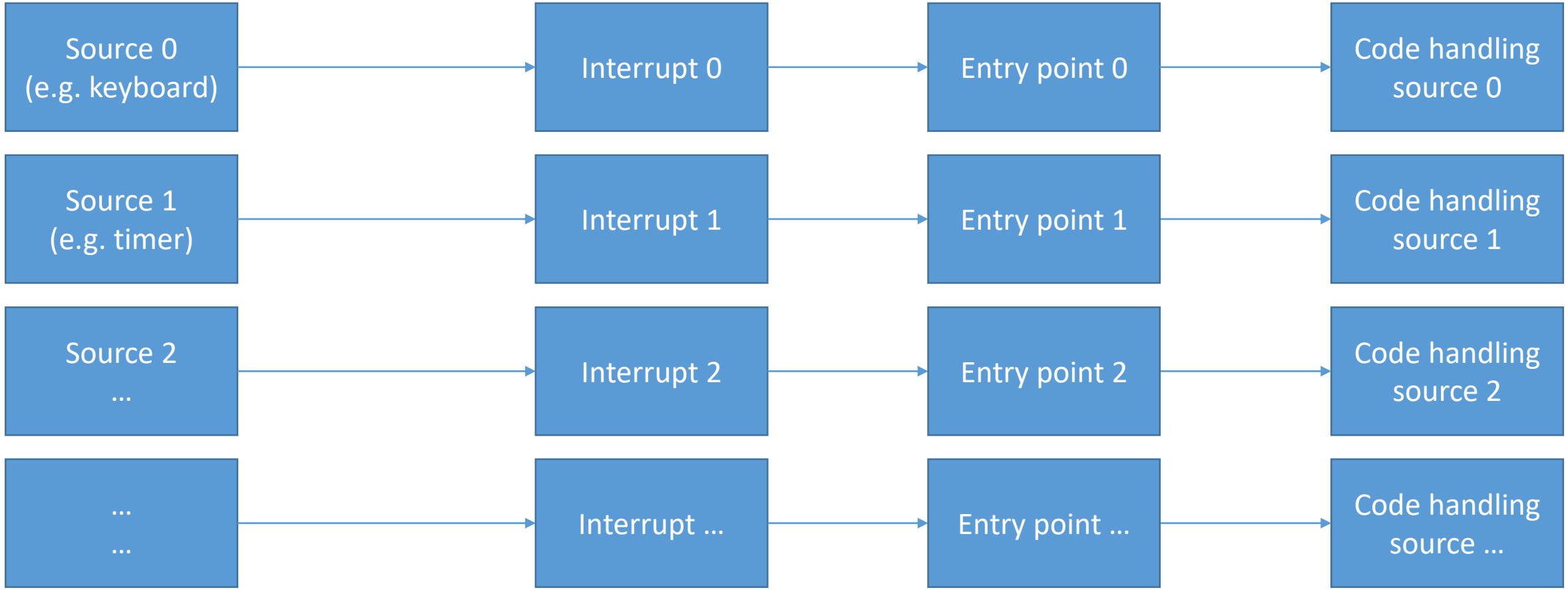
- The ISR at the entry point needs to check the status of the interrupt sources in order to find out which code shall be executed to handle the interrupt

Connecting Interrupt Sources to Interrupt Service Routines (one entry point)



- The ISR at the entry point based on the interrupt number determines the code that shall be executed to handle the interrupt

Connecting Interrupt Sources to Interrupt Service Routines (vectored approach)



- Vectored handling with different entry points for different interrupts

Connecting Interrupt Sources to Interrupt Service Routines

- In practice all kinds of combinations are possible for interrupt handling
- There is also the option for having interrupts with different priorities
- Dedicated interrupt controllers are available on larger systems to handle priorities, entry points, nested interrupts, …