Computer Organization and Networks

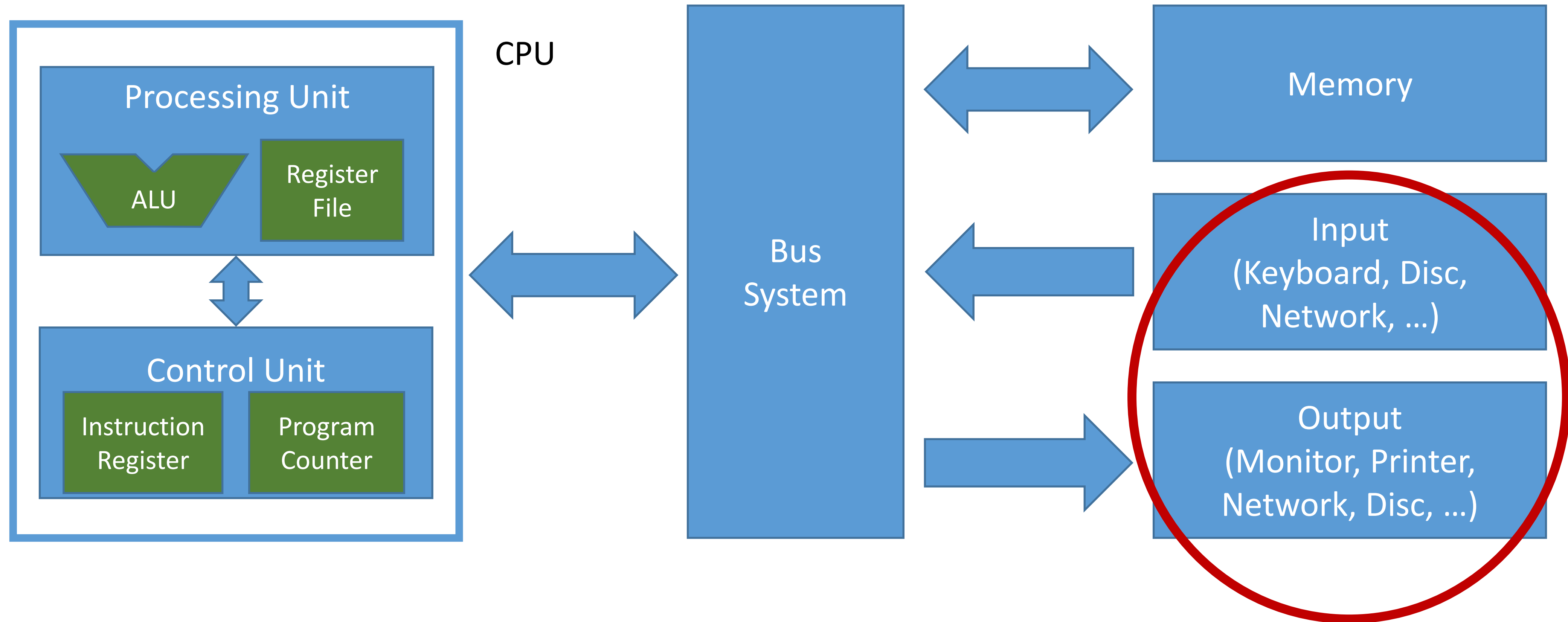
(INB.06000UF, INB.07001UF)

Chapter 11: Interrupts and Multitasking

Winter 2023/2024

Stefan Mangard, www.iaik.tugraz.at

Von Neumann Model



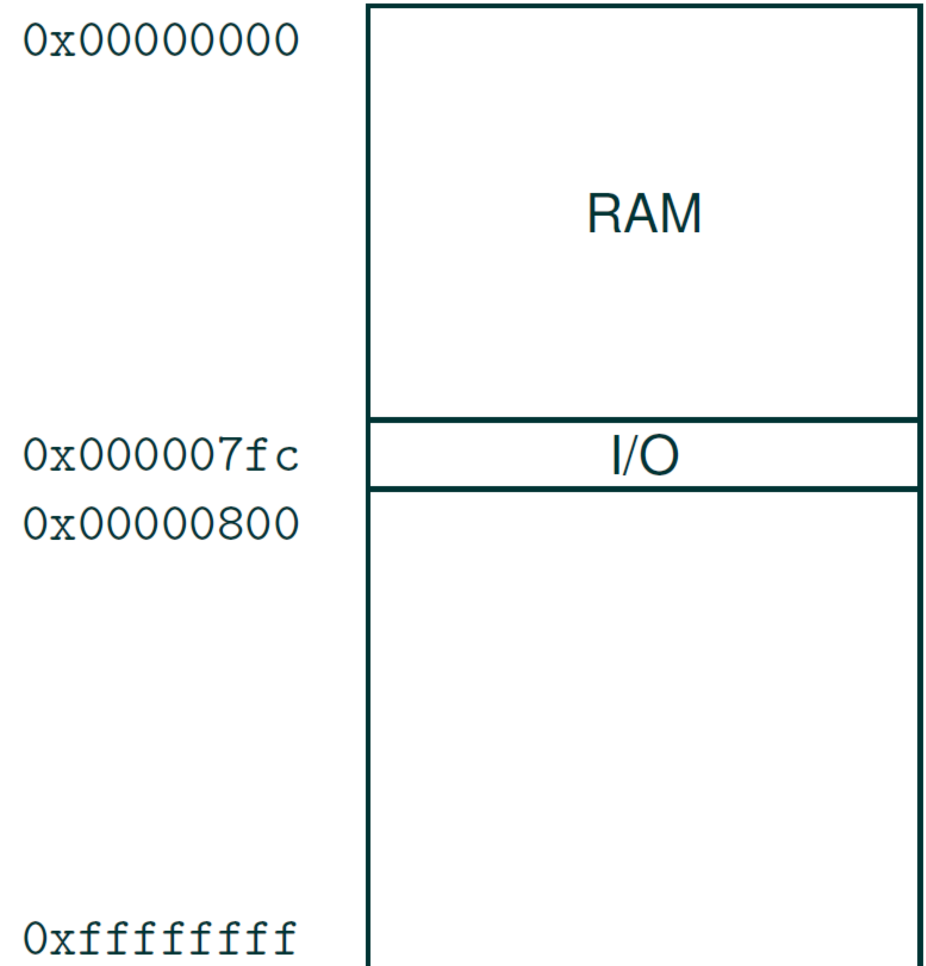
Example I/O

- The I/O interface that we discussed at the beginning of the lecture was idealized debug interface (data was 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

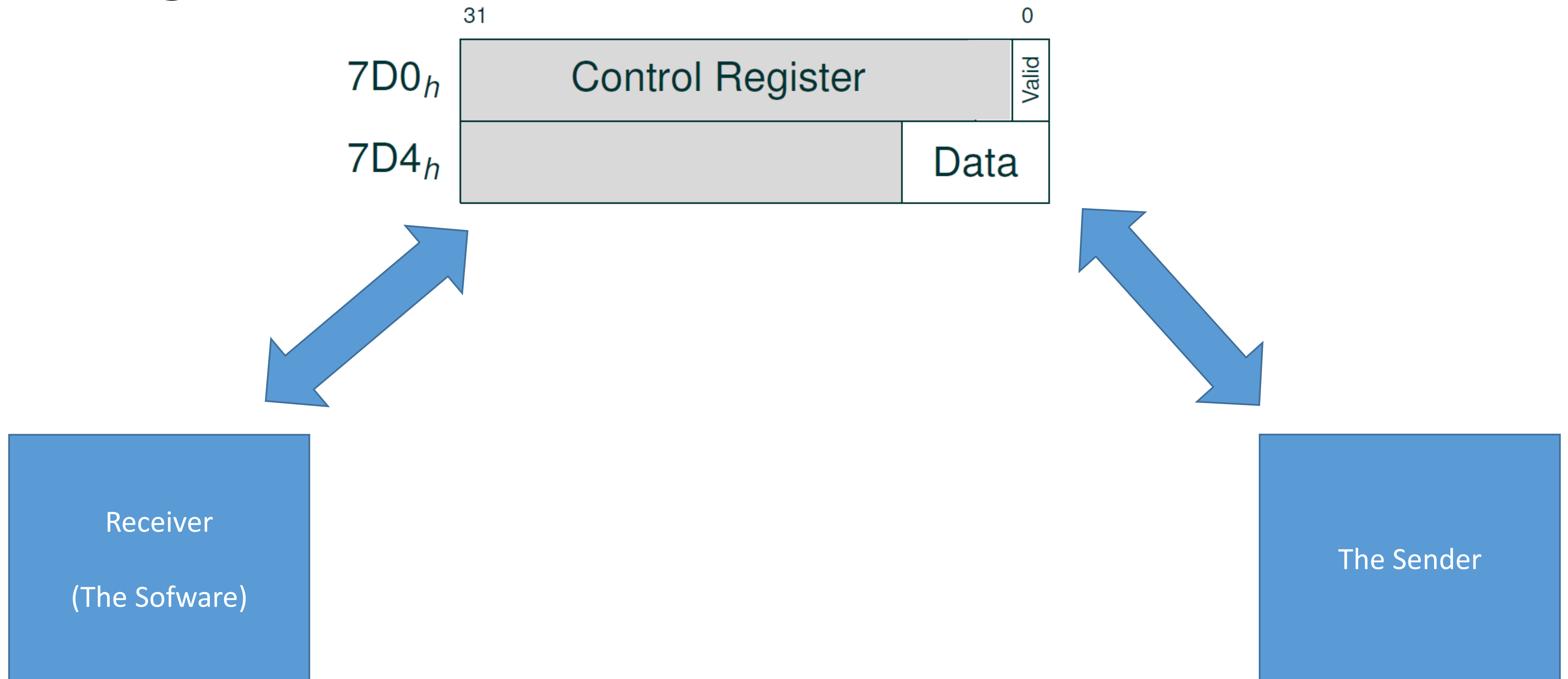
You've Got Mail

Synchronization with Control Signals

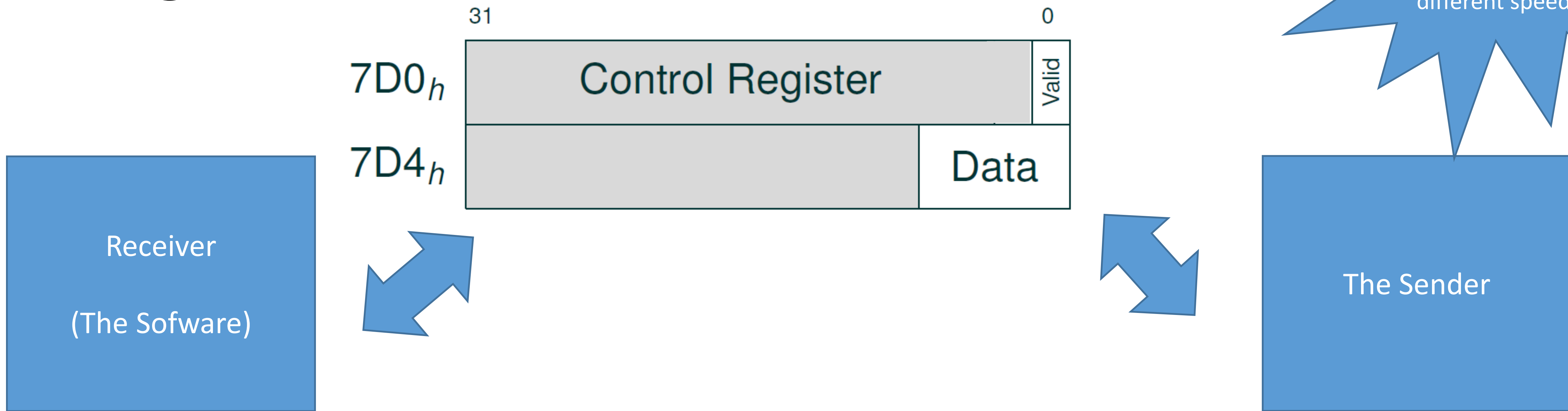
- On real communication channels, data is not always ready
- We need synchronization with control signals
- There exist different protocols and standards.
  - Serial protocols: RS232, SPI, USB, SATA, . . .
  - Parallel protocols: PATA/IDE, IEEE 1284 (Printer), . . .

- 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



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

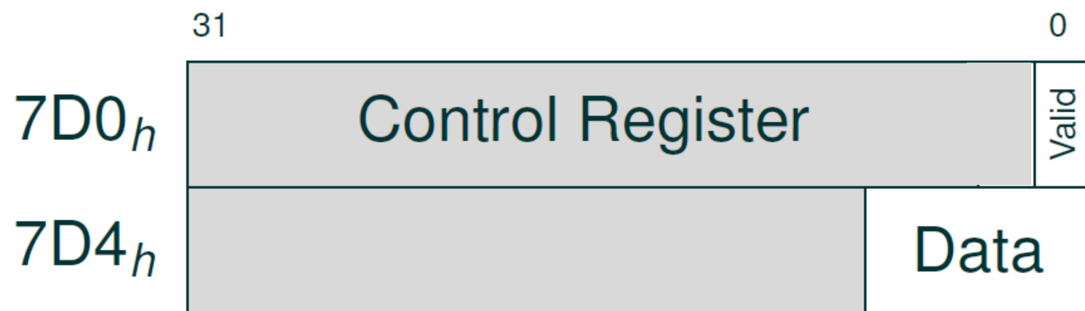
(6) Receiver clears the valid bit

(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




```
POLL_PARIN:
    LW x1, 0x7D0(x0)
    ANDI x1,x1, 1
    BEQZ x1, POLL_PARIN

    LW x2, 0x7D4(x0)
    SW zero, 0x7D0(x0)
```

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)

Polling Example in QtRVSim

- https://comparch.edu.cvut.cz/qtrvsim/app/

- See examples in directory con11.01_QtRVSim_IO:
  - Basic I/O example:
    - 01_playing_with_knobs.S
  - Polling example:
    - 02_polling.S

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 are like doing a function call that is not triggered by a caller, but by an external event

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
  - 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

Source 0
(e.g. keyboard)

Source 1
(e.g. timer)

Source 2
…

…
…

There are many options for connecting interrupt sources to interrupt service routines

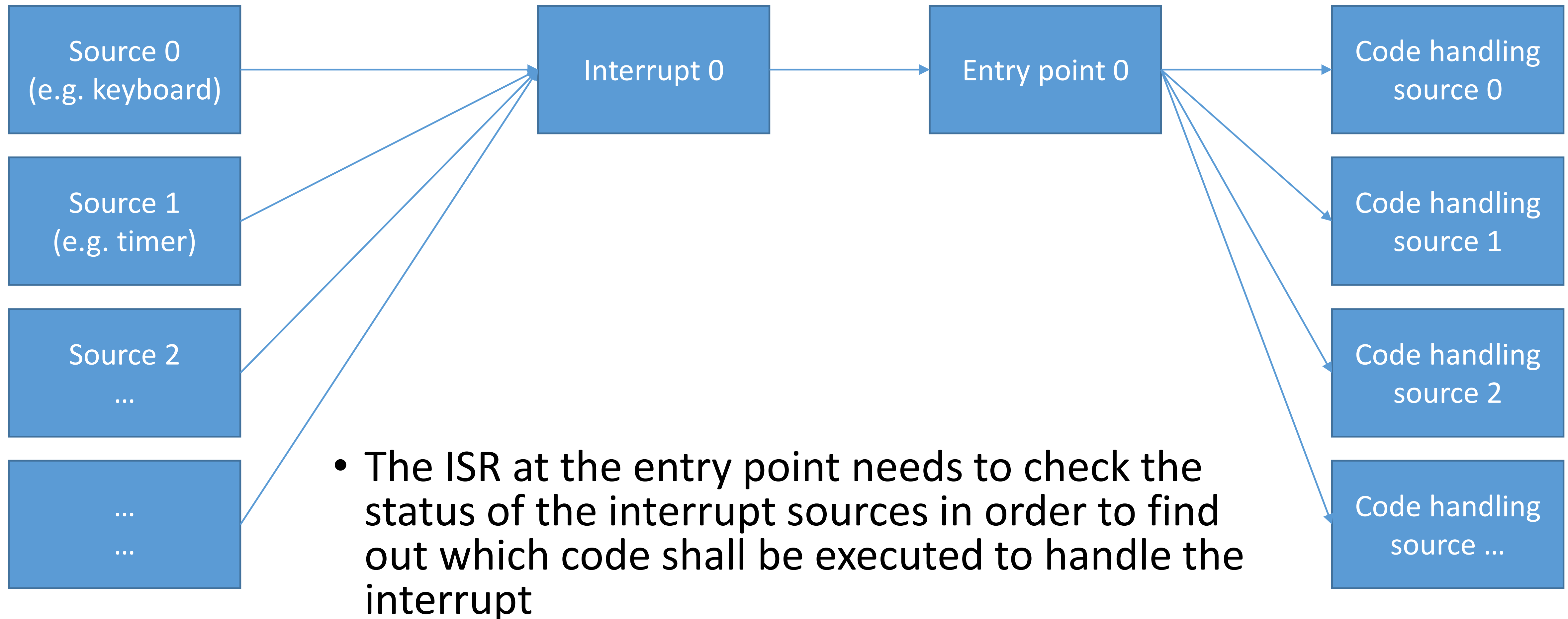
Code handling source 0

Code handling source 1

Code handling source 2

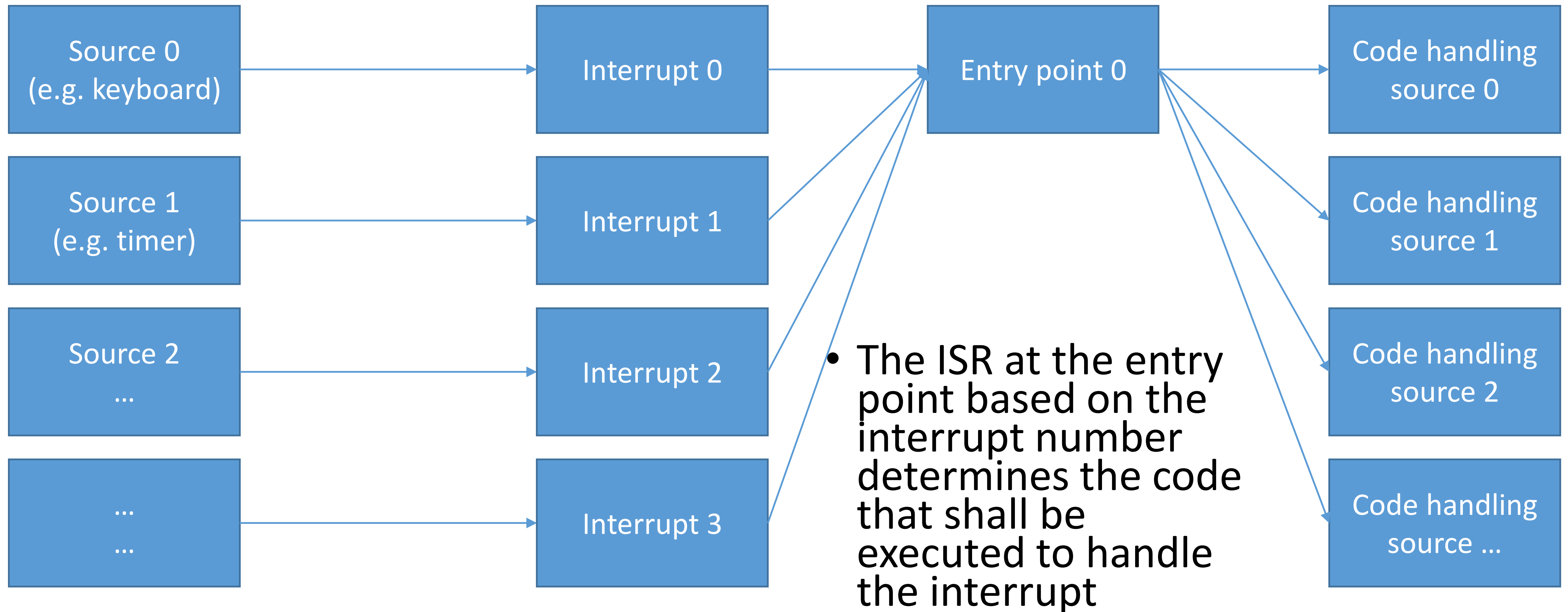
Code handling source …

Connecting Interrupt Sources to Interrupt Service Routines (one Interrupt)



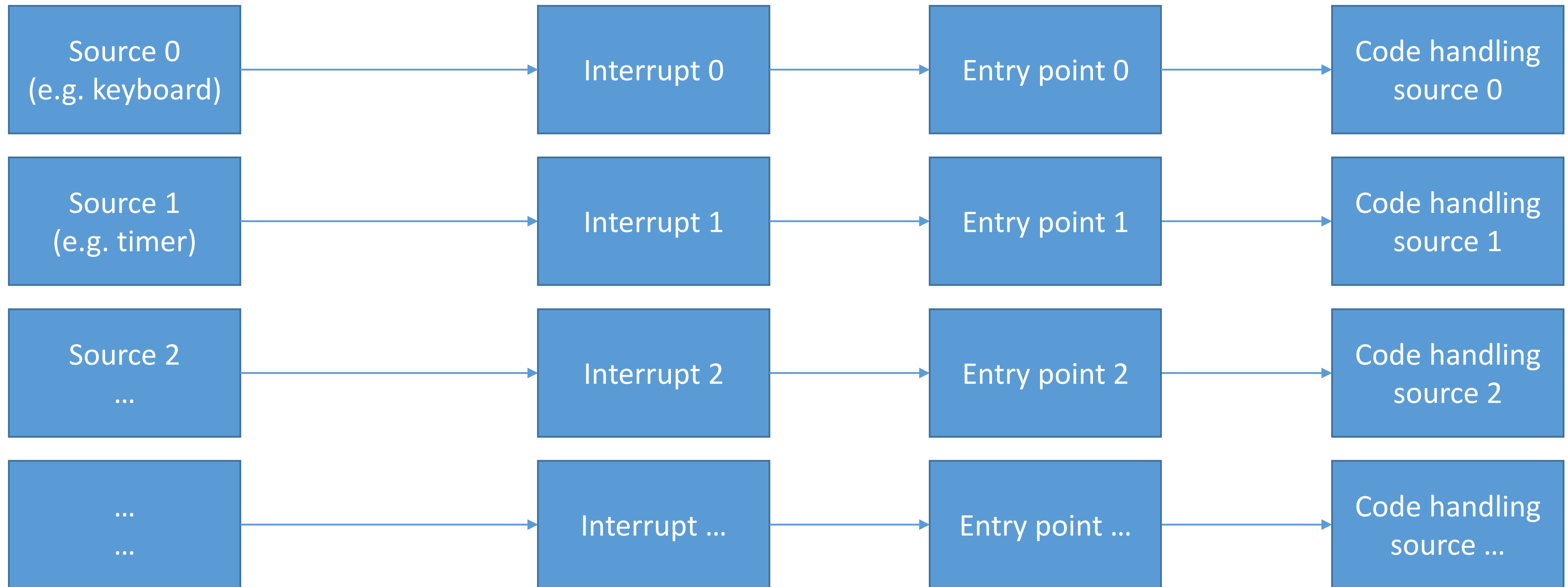
- The ISR at the entry point needs to check the status of the interrupt sources in order to find out which code shall be executed to handle the interrupt

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, …

Direct Memory Access

Direct Memory Access (DMA)

- Observe:
  - I/O typically runs at lower speeds than the CPU
  - With interrupts we a mechanisms that prevents the need for polling
  - The CPU is directly notified when data is available at a peripheral

- Interrupts are useful for communication with devices like
  - Network interfaces
  - Harddrives
  - USB
  - …

This is nice, but let’s look at the job of the CPU upon receiving an interrupt.

What Happens Upon an Interrupt?

- When a CPU receives data from a peripheral typically needs to do two things:

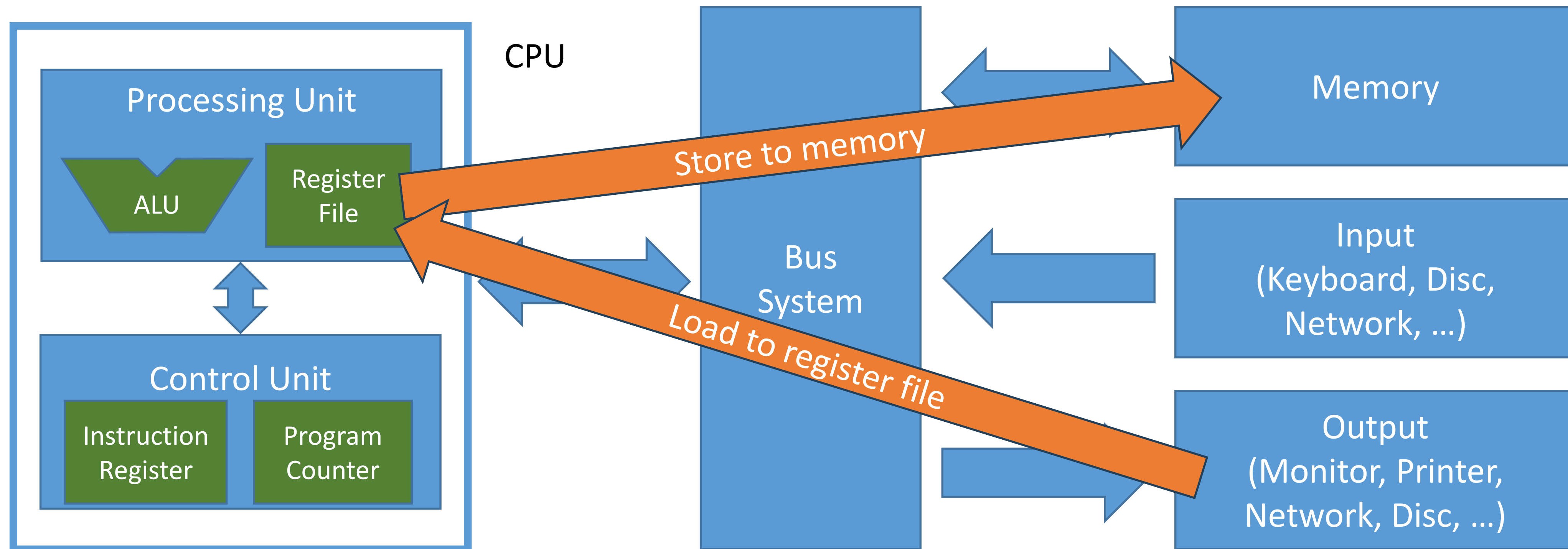
    1. The CPU first copies the data from the peripheral to the memory

    2. The CPU then works with the data in the memory to perform specific tasks

Note:

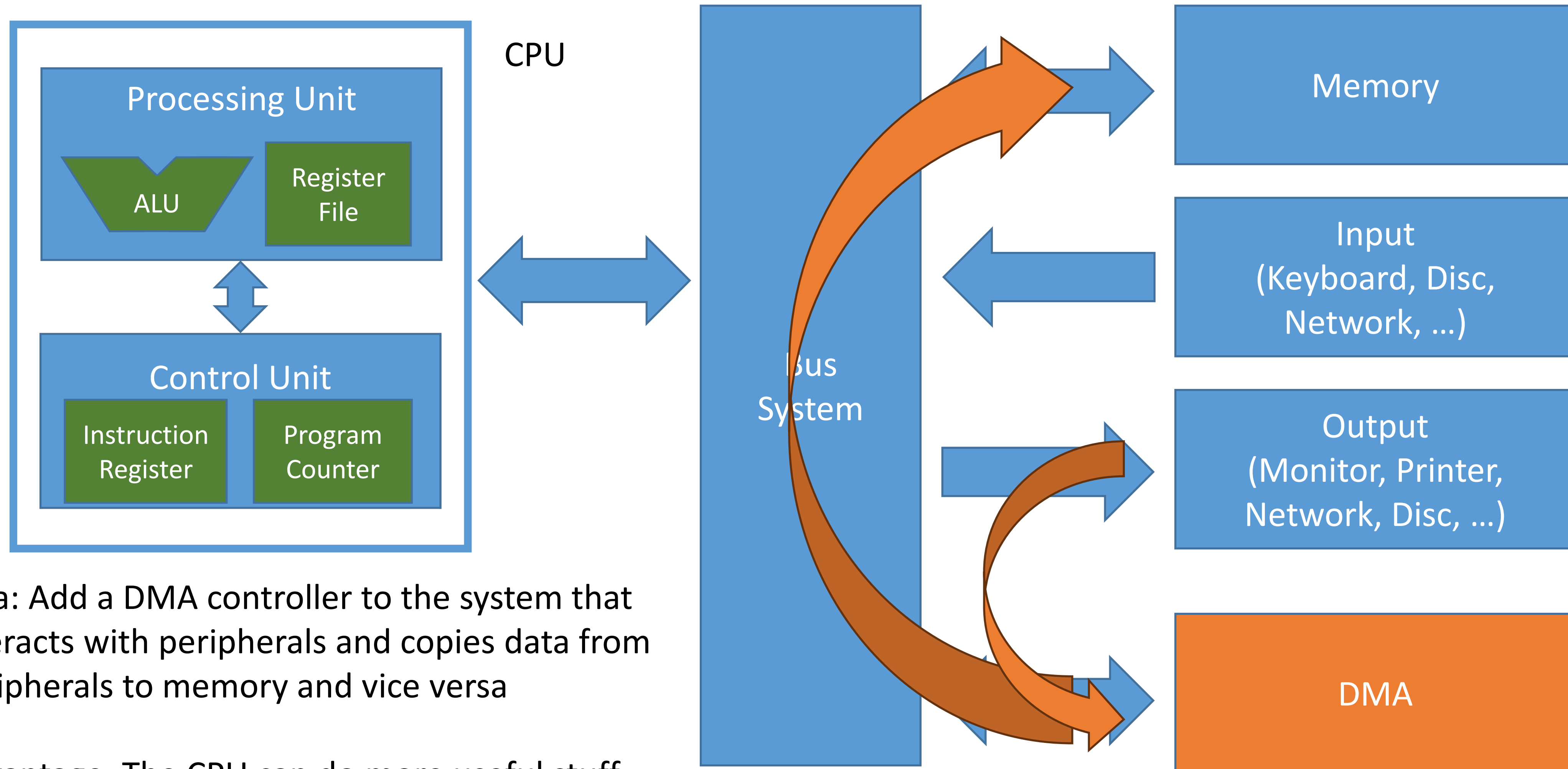
- The first task is a very simple task that is just a sequence of load and store operations;
- This might can a long sequence in case of larger blocks of data that need to be moved → It's a loop that continuously loads data from a peripheral and that then stores to memory

Von Neumann Model



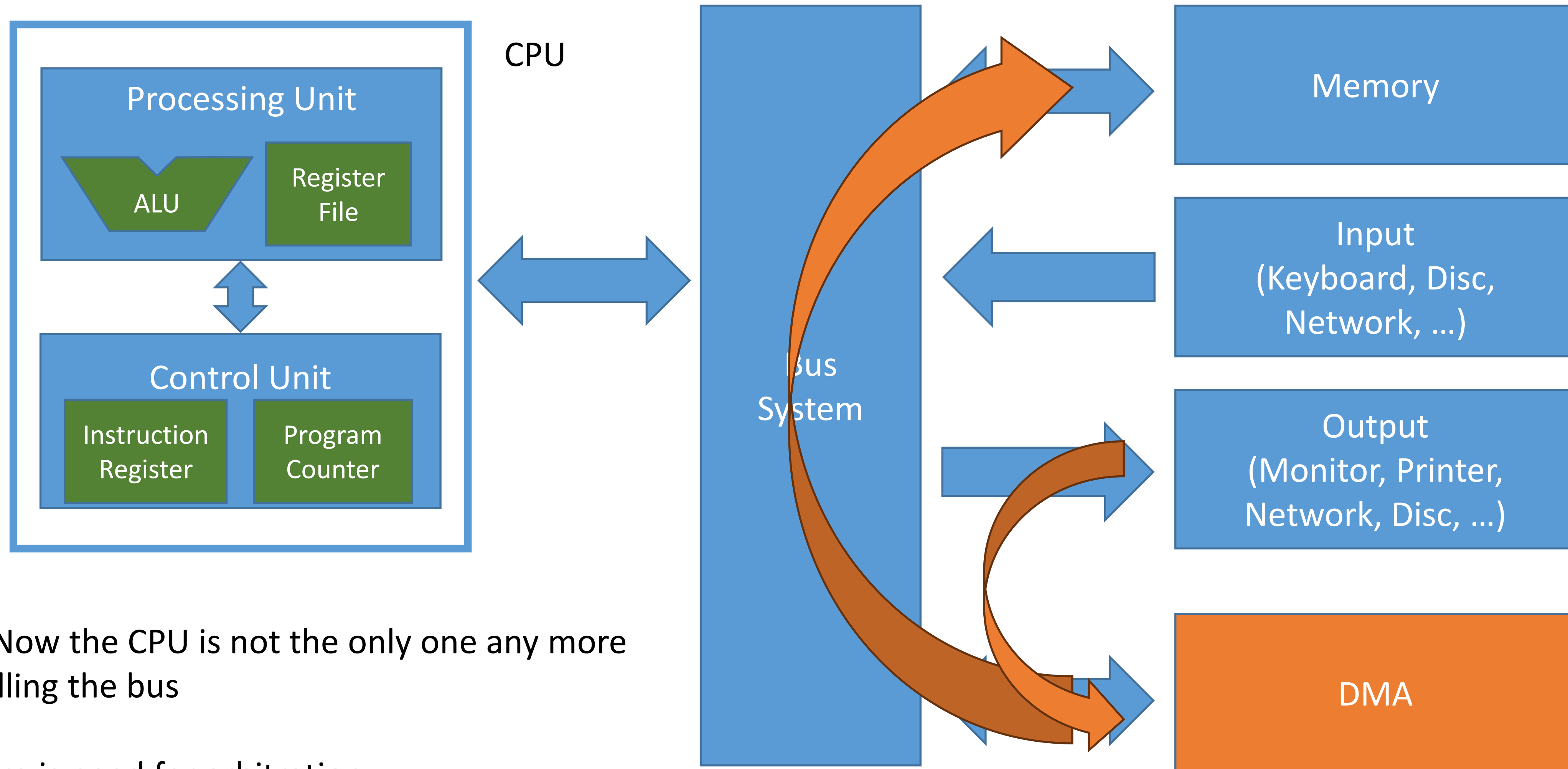
This is is very simple – should the CPU do this or can't something else do this?

Von Neumann Model



- Idea: Add a DMA controller to the system that interacts with peripherals and copies data from peripherals to memory and vice versa
- Advantage: The CPU can do more useful stuff while the data is copied

Von Neumann Model



Note: Now the CPU is not the only one any more controlling the bus

→ There is need for arbitration

Basic Idea of DMA

- Goal: Offload memory transfer tasks from the CPU to DMA controller

- Implementation
  - CPU configures DMA control with the following basic paramters:
    - Where to read from (source address)
    - Where to write to (destination address)
    - Amount of data to be transferred

  - After completion of the transfer, the DMA triggers an interrupt

  - Implementation variants:
    - There are many variants of how to implement DMA (big differences between small microcontrollers and server CPUs)
    - DMA can be a dedicated peripheral or integrated directly into relevant peripherals
    - There are different variants on how to avoid bus contentions: dedicated additional buses, different types of arbitration

Multitasking

Typical Software Stack

- We have powerful CPUs and typically not just have one program that we would like to execute → we would like multiple programs

- We need a resource manager that takes care of
  - Which program executed when using what resources
  - Context switching (efficiently switching from one program to the other)
  - Managing the sharing all kinds of resources (CPU, memory, hard drive, I/O, …) between the program
  - …

→ All this done by the operating system (from small embedded OS to Linux/Windows)

Privilege Levels Provided by the Hardware

- Clearly not every program should be able to do everything. The operating system should have different rights
- Computers have typically at least two modes:
  - **User Mode:** A mode for user applications providing only limited access rights to hardware resources
  - **Supervisor Mode:** A mode for the operating system providing full access
- RISC-V defines three modes: User mode (U), Supervisor Mode (S), and Machine Mode (M)
- Switching between different privilege levels is done via “software exceptions”

Interrupts, Exceptions, Traps

- There are different wordings on different system for events that cause a disruption of the execution flow of a CPU

- There are essentially the following reasons for a disruption:

  - **Hardware Exception** (asynchronous – can occur any time)
    - A device signals that it requires attention by the CPU (see beginning of slideset)

  - **Software Exception** (synchronous – occurs based on an instruction)
    - Caused by an error: An instruction causes some error (e.g. division by zero, page fault, memory access violation, …)

    - Caused by system call: There are instructions to trigger exceptions on purpose by the software (e.g. ECALL on RISC V)

Typical Exception Handling

Independent of the source and cause, the essential procedure for handling exceptions is typically as follows:

- Exceptions are picked up by an exception handler (we called it interrupt service routine (ISR) at the beginning of the slide set) typically running at a higher privilege level (e.g. an exception handler by the operating system)
- The exception handler stacks registers, determines what caused the exception, reacts to/handles the exception, and as quickly as possible returns to the program that was interrupted
- Note: There is not always a return to the program the caused the execption. E.g. when there is an access violation, a division by 0, or another error, the program is terminated by the OS.

Operating Systems

- There are many things to learn about operating systems and there is tight interaction between the hardware and the operating systems

→ See course “Operating Systems” on how the operating system manages the resources provided by the hardware

Options for Executing Multiple Programs

- The hardware provides essentially three options to execute multiple programs on a computer

1. Execution of multiple programs on a single CPU
2. Execution on Multiple CPUs
3. Simultaneous Multithreading

Multiple Programs on a Single CPU

Two basic approaches

- **Cooperative multitasking**: The programs that execute on a CPU yield control of the CPU to each other
- **Preemptive multitasking** (the common approach): The operating system decides which tasks runs on the CPU. This approach requires a timer interrupt

Timer Interrupt

- Essentially all systems implement a peripheral called “Timer”, which allows to receive an interrupt after a given time

- Example Design:
  - The timer is clocked with a fixed clock frequency
  - It contains a counter that counts from a starting value down to zero and triggers an interrupt when reaching zero
  - A memory-mapped interface allows the software to set the starting value and to enable the timer interrupt

- Applications:
  - Multitasking
  - Measuring time

High-Level View on Preemptive Multitasking

- The operating system maintains a list of tasks to be executed
- It schedules the first task and sets a timer interrupt
- The first task runs for a given time and is then interrupted
- In the exception handler, the OS implements a scheduler that essentially saves the state of the current tasks and yields control to the next task
- The second task runs for a given time and is then interrupted
- The scheduler switches to the third task
- The scheduler switches to the fourth task
- …

→ This technique is also referred to as time-slicing (each task gets a slice of execution time on the CPU before there is switch to the next task)

Multiprocessor Systems

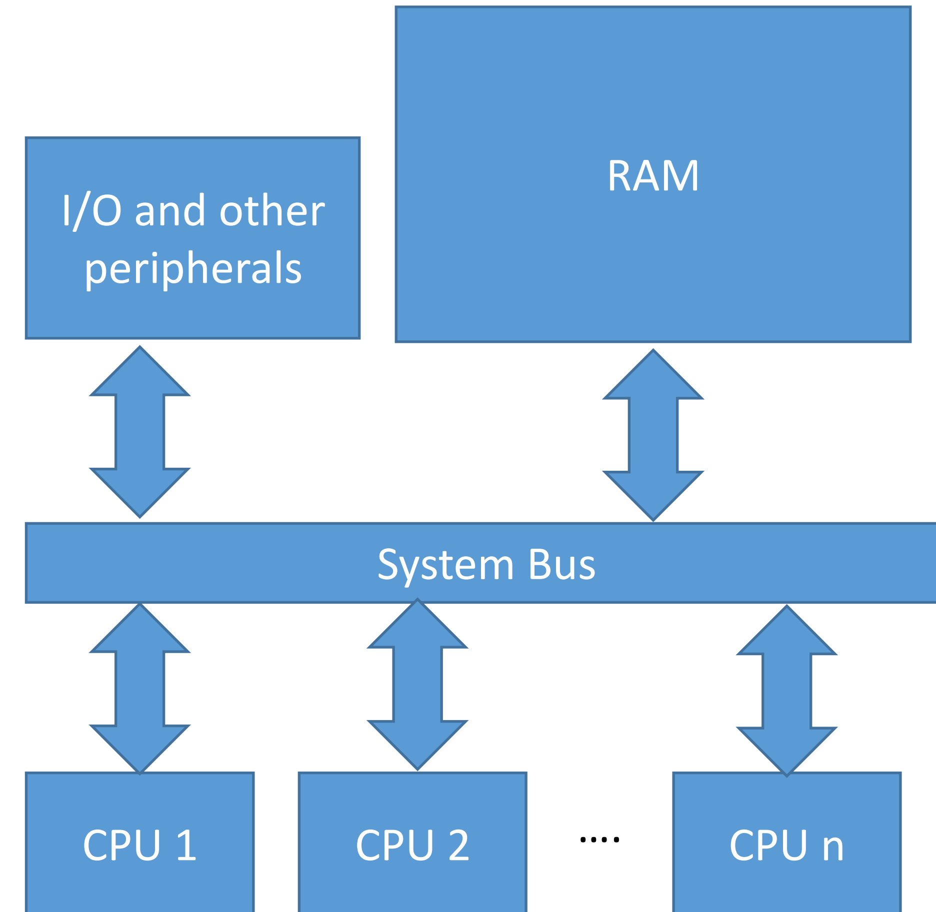
- During the last decades, the number of instructions per time have
  - first increased by increasing the clock frequency
  - then increased by using more and more transistors to build a single processor

- Since about 2005, there are so many transistors available on the silicon that there has been a major shift to implementing multiprocessor systems

- Two types:
  - Symmetric Multiprocessors:
    The same core is simply instantiated multiple times
  - Heterogeneous Multiprocessors:
    The system has cores with different properties (e.g. energy efficiency, performance, …)

Multiprocessor Systems

- **Basic Designs**
  - Multiple CPU cores
  - All are connected to a shared memory
- **Warehouse-Scale Computers**
  - Datacenters like at AWS, Microsoft, etc. have dedicated designs
  - Clustering
- **Important Topics**
  - Scheduling of processes on the different CPUs
  - Handling of shared resources
  - Security



Hardware Multithreading

- Observation:
  - In a superscalar CPU design, it is likely that a single thread will not need all hardware resources on the different pipeline stages
- Idea:
  - Instead of just executing one threads in a CPU, execute multiple threads in parallel → there are no dependencies between the instructions of the different threads and this should increase throughput
- Implementation aspects
  - It is necessary to implement a CPU state (in particular register file) for every thread

Simultaneous Multithreading

There are essentially three options to switching between threads

- Coarse Grained: Switch between threats upon major stalls
- Fine Grained: Switch between threats in every clock cycle
- Simultaneous Multithreading: Permanently Fetch, decode, and execute instructions from different threads

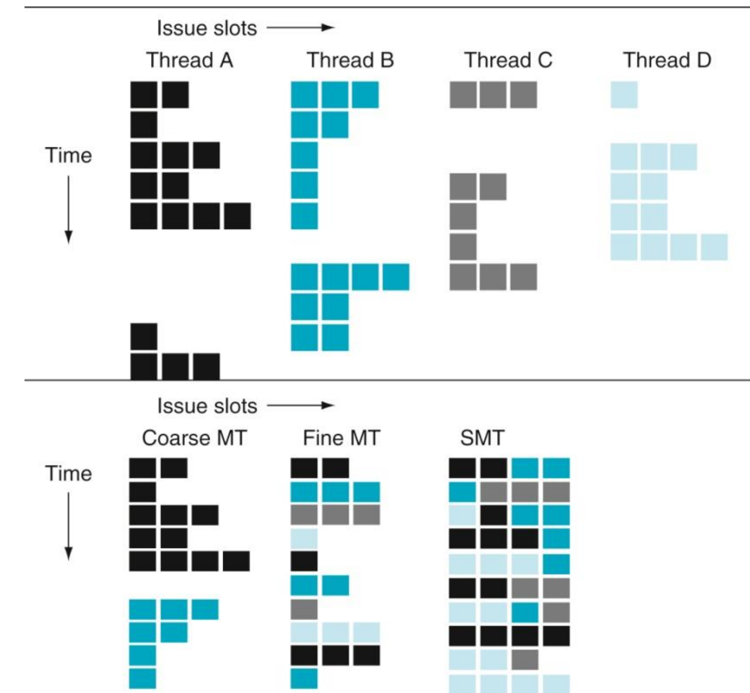


Figure Copyright © 2019, Elsevier Inc. All rights Reserved

Security Examples

Cybersecurity Challenges Related to Processors and Low-Level Software are Manifold

Birds Eye Perspective on Top-Level Challenges:

- There is a program running on the system

  Can the user through interacting the program make this program do things it is not supposed to do?

- We have different security contexts

  E.g. Operating System vs. User Application, User Application vs. User Application, Virtual Machine vs. Host, Sandbox vs. Host, Enclave vs Operating System, …

  Can the attacker learn anything form outside his security context?

  Can the attacker change anything outside his security context?

Buffer Overflow

More on this?
Take the course "Secure Software Development"

- A computer performs one instruction after the other
- If return addresses on the stack are overwritten by user input, the computer will jump to a target defined by the user input
- Simple buffer overflows are detected on today's computer systems. However, there are many more options of how a user can attack a computer system.
- See example **07_stack_buffer_overflow.asm**

Learn Things From Outside Your Security Context

Isolation of program components is crucial

- You don’t want your favorite game to read your bank account
- You don’t want that browsing to a website with javascript gives the website full access to your phone or labtop
- You don’t want that another tenant in the cloud can read your data from a cloud server
- You might also not want that the cloud provider can access all the data you process in the cloud
- …

Note: The security contexts that need to be isolated share the same hardware and also share software → Attacker and Victim execute code on the same computer

Protecting and Overcoming Isolation Bounds

Two classes for attacks:

- Software Attacks: The attacker exploits a vulnerability through a software interface
- Side-Channel Attacks: The attacker learns information indirectly (not through a classical software interface, but e.g. through timing behavior, power consumption

A Glimpse on Attacks

- Power Analysis
- Fault Analysis
- RowHammer
- Meltdown/Spectre
- …
- Interested in more?
  - TU Graz: https://graz.elsevierpure.com/de/organisations/institute-of-applied-information-processing-and-communications-70/publications/
  - Conferences: USENIX Security, IEEE Security & Privacy, NDSS, ACM CCS, …

Exam and Outlook

Exam

- 90 Minutes
- Next Dates
  - Feb 1, 2024
  - March 19, 20024

Content Continues in the Following Courses

- Operating Systems (Bachelor)
  Learn how to manage all the hardware resources that we have covered in this course
- Information Security (Bachelor)
  Learn about cryptography and the security challenges in processors, software, and networks
- Digital System Design (Bachelor / Master)
  Design your own chip in SystemVerilog and learn more about hardware design
- Digital System Integration and Programming (Master)
  Build your own hardware, write a Linux driver, put it on an FPGA → boot your own system
- Secure Software Development (Master)
  Learn how attacks are conducted and how to secure your applications