



DEPARTMENT OF
NETWORKED SYSTEMS
AND SERVICES

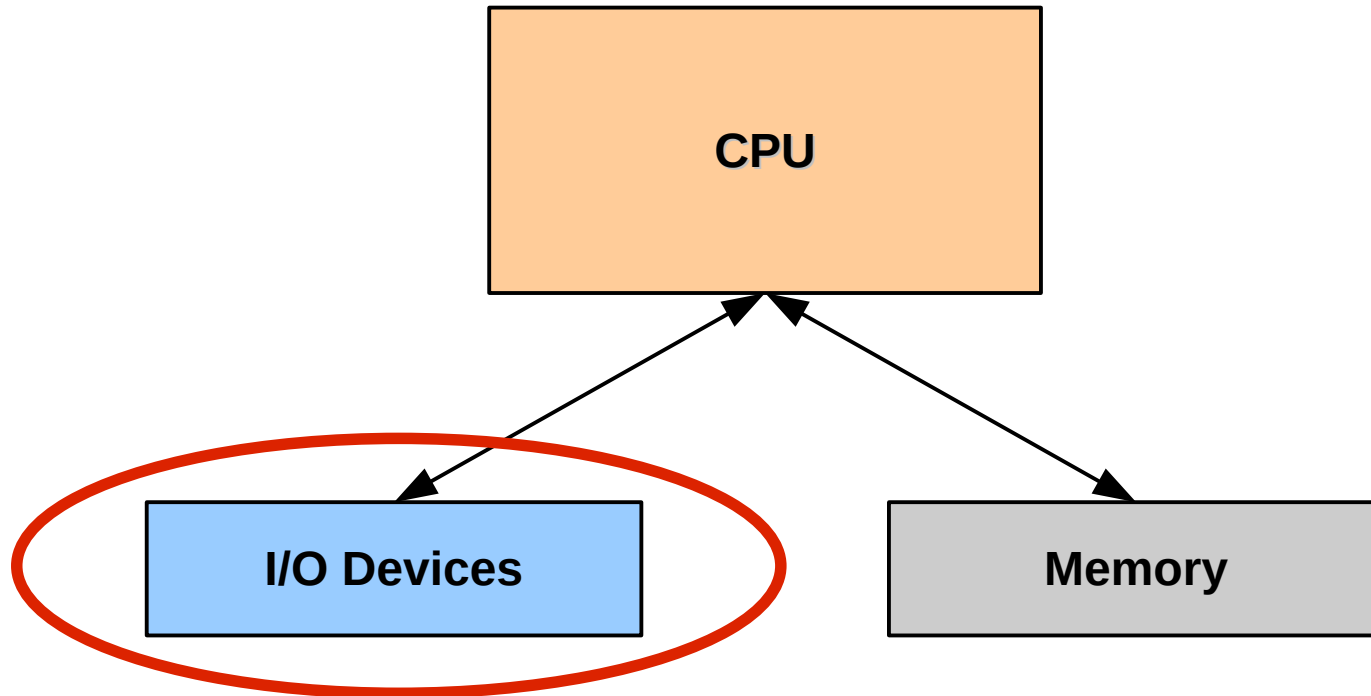
COMPUTER ARCHITECTURES

I/O Devices

Budapest,
2025. 02. 13.

Gábor Horváth, ghorvath@hit.bme.hu





- There are many of them:
 - Input / output
 - Delay sensitive / insensitive
 - Throughput sensitive / insensitive
 - Bit error tolerant / intolerant
 - Etc..

	Controlled by	Direction	Data traffic
Keyboard	Human	Input	ca. 100 byte/s
Mouse	Human	Input	ca. 200 byte/s
Sound device	Human	Output	ca. 96 kB/s
Printer	Human	Output	ca. 200 kB/s
Graphics unit	Human	Output	ca. 500 MB/s
Modem	Machine	In/Out	2-8 kB/s
Ethernet network interface	Machine	In/Out	ca. 12.5 MB/s
Disk (HDD)	Machine	In/Out	ca. 50 MB/s
GPS	Machine	Input	ca. 100 byte/s

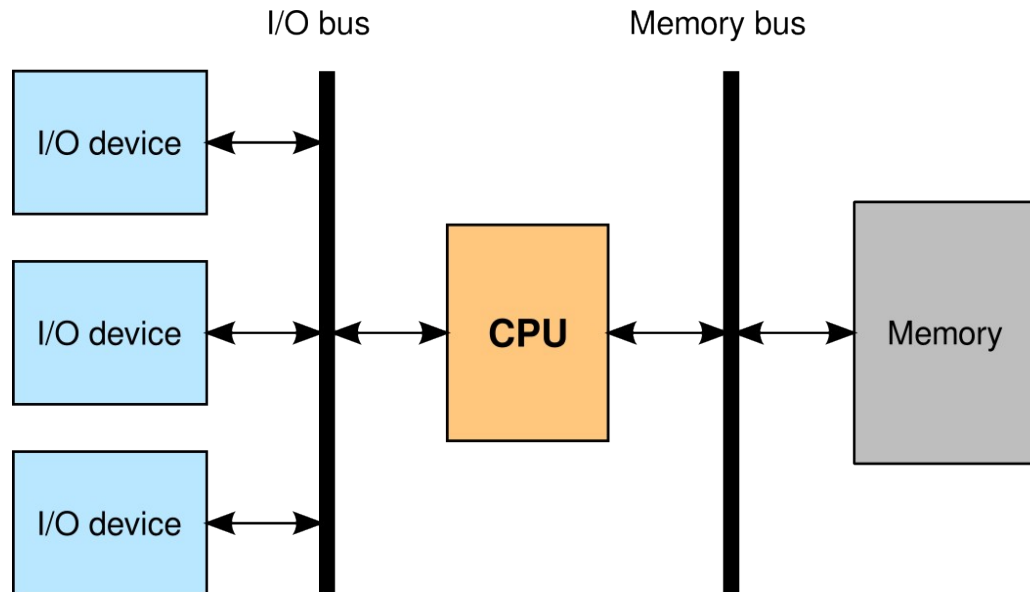
- Questions to investigate:
 - How does the CPU communicate with I/O devices?
 - How do I/O devices communicate with the CPU?
 - How to transmit data efficiently, without errors?
 - How to connect the I/O devices to the CPU?

- Questions to investigate:
 - **How does the CPU communicate with I/O devices?**
 - How do I/O devices communicate with the CPU?
 - How to transmit data efficiently, without errors?
 - How to connect the I/O devices to the CPU?

- Accessing memory content: by addressing
- Accessing I/O devices: by addressing
- Based on address space separation:
 - **Separate** memory and I/O address spaces
 - **Shared** memory and I/O address spaces

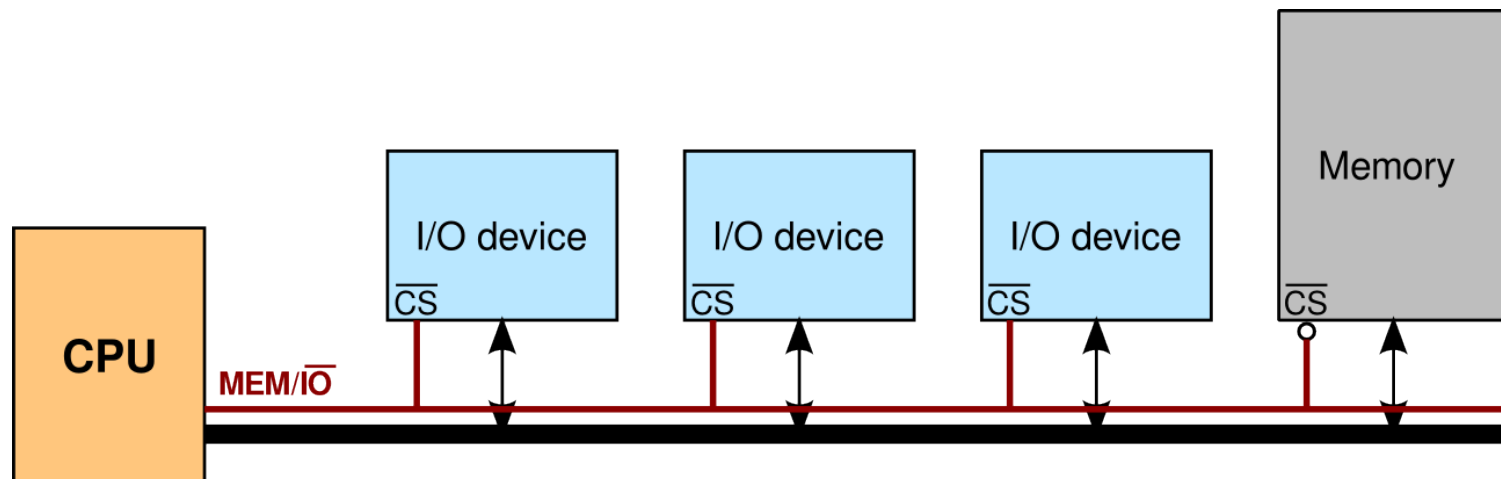
SEPARATE MEMORY AND I/O ADDRESSES

- Two separate address spaces:
 - x86: memory: 0 – 4GB, I/O: 0 – 64kB
- Separate instructions for I/O and memory operations
 - **R0** \leftarrow **MEM[0x60]**: data movement from memory
 - **R0** \leftarrow **IO[0x60]**: data movement from I/O peripheral
- Implementation:
 - Separate I/O and memory buses



SEPARATE MEMORY AND I/O ADDRESSES

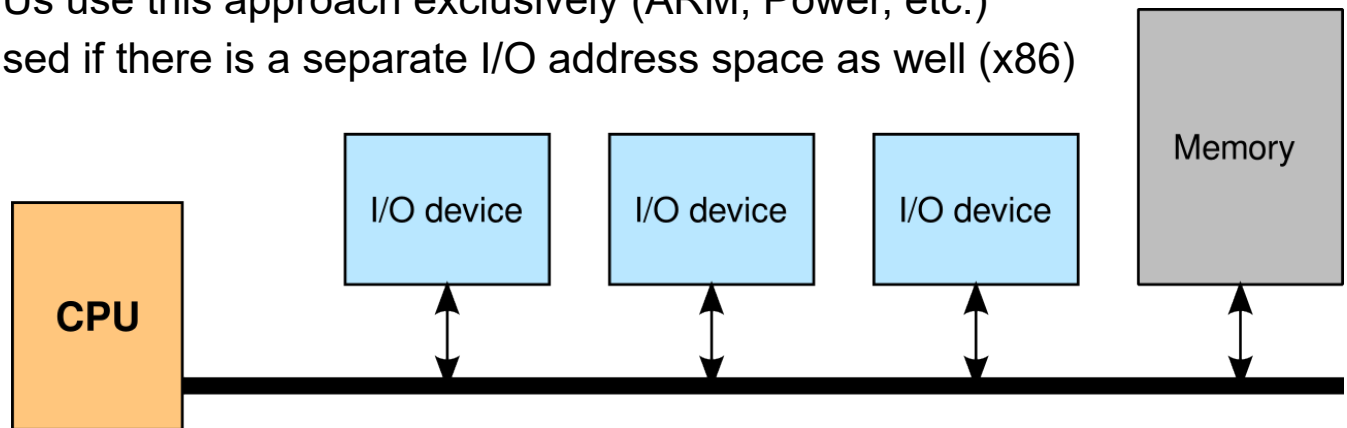
- Two separate address spaces:
 - x86: memory: 0 – 4GB, I/O: 0 – 64kB
- Separate instructions for I/O and memory operations
 - $R0 \leftarrow \text{MEM}[0x60]$: data movement from memory
 - $R0 \leftarrow \text{IO}[0x60]$: data movement from I/O peripheral
- Implementation:
 - Multiplexed I/O and memory buses



MEMORY MAPPED I/O DEVICE HANDLING

- The CPU has a single address space
- There are special memory addresses reserved for I/O communication
 - Memory read/write requests from/to these addresses are answered by I/O devices
 - Data exchange with I/O devices is very simple:


```
char* p = 0x60;
int x = *p;
```
- **Advantage:** there are many convenient data transfer instructions for the memory in every architecture
- **Drawback:** unreachable “holes” in the memory
- Examples:
 - RISC CPUs use this approach exclusively (ARM, Power, etc.)
 - Can be used if there is a separate I/O address space as well (x86)

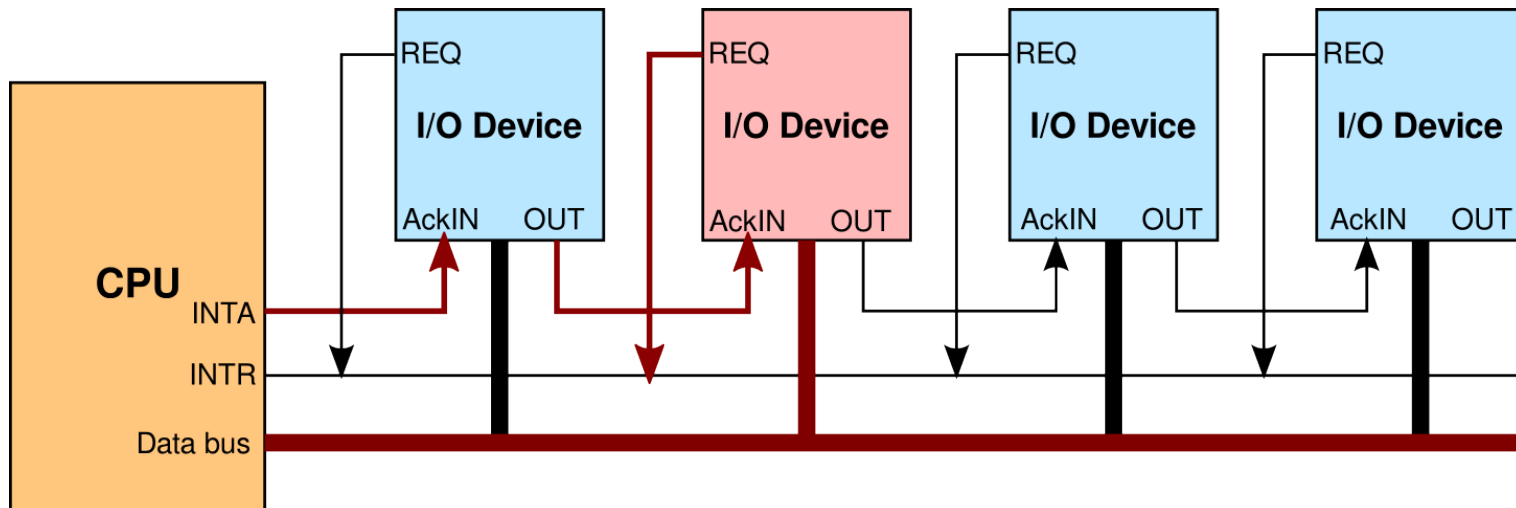


- Questions to investigate:
 - How does the CPU communicate with I/O devices? ✓
 - **How do I/O devices communicate with the CPU?**
 - How to transmit data efficiently, without errors?
 - How to connect the I/O devices to the CPU?

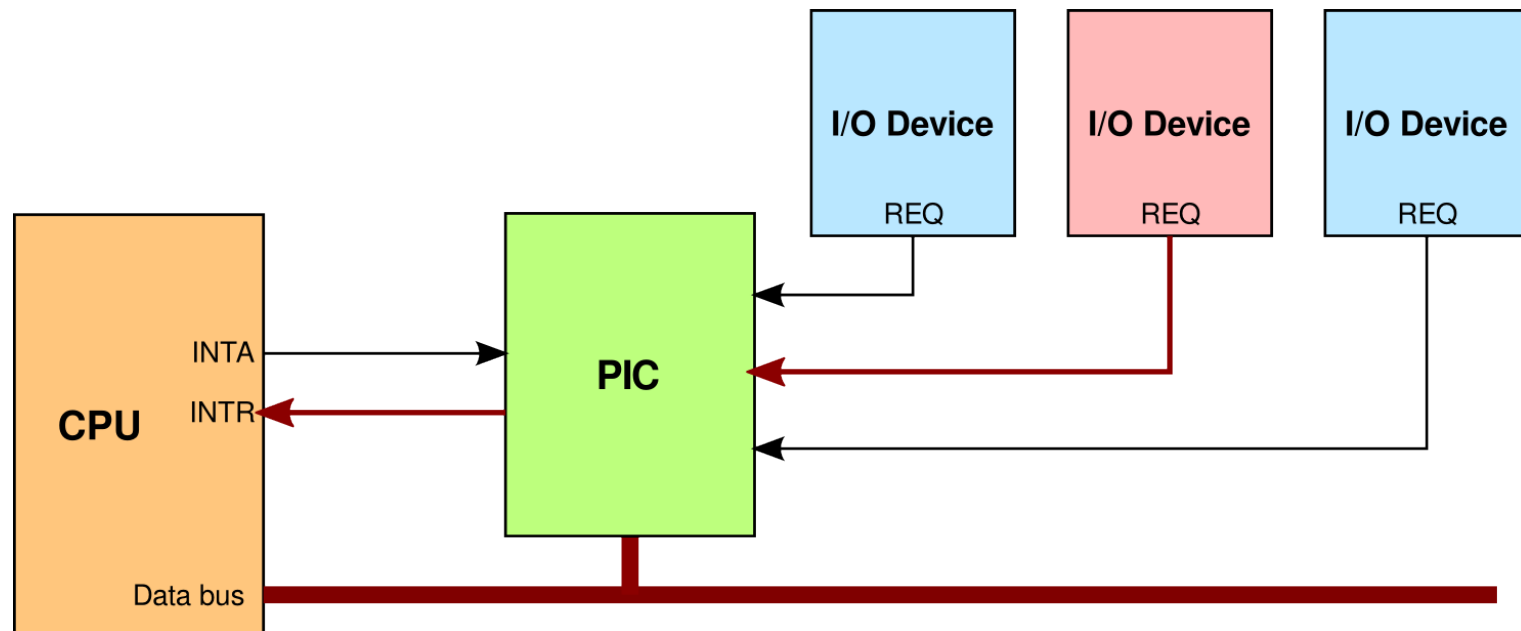
- An interrupt can be requested through the CPU „INT” input
- Problem: **number of interrupt sources > INT input pins of the CPU**
- Several I/O devices share the same interrupt lines
 - How does the CPU know which devices wanted the interrupt?
 - What happens, if several devices requested an interrupt at the same time?
- Solution 1: **Polling**
 - Every device uses a single interrupt line
 - On interrupt, the interrupt handler subroutine asks every device if it generated the interrupt → **polling**
 - What happens, if more devices request an interrupt at the same time?
 - Service order = polling order

- Solution 2: **interrupt vector table**
 - When the CPU accepts the interrupt, the device generating the interrupt puts its number to the data bus
 - This number determines which subroutine handles the interrupt
 - The CPU has a table: **interrupt vector table**
 - It contains pointers to interrupt handler subroutines
 - The interrupt is handled by the subroutine given by the number provided by the device
- What happens, if more devices request an interrupt at the same time?
 - Happens frequently
 - During the service of an interrupt several others may arrive
 - They have to be served one after the other
 - By daisy chain
 - Using an interrupt controller

- **Daisy chaining:**
 - The “interrupt acknowledge” signal is sent by the CPU
 - Devices not requesting an interrupt pass it on to their neighbor
 - If a device requests an interrupt, stops the signal
 - The order of devices define the priority
 - Devices at the end of the row starve



- **Programmable Interrupt Controller:**
 - It has more than one inputs
 - The PIC is an I/O peripheral itself
 - The CPU (op. system) configures it with I/O operations
 - What should happen, if more interrupts arise at the same time
 - Which devices are allowed to generate an interrupt



- Interrupts in **multi-processor systems**
 - Simple solution: every interrupt is handled by the default processor (the one that boots the op. system)
 - Alternative solution: advanced programmable interrupt controller (Intel: APIC, ARM: GIC, etc.)
 - Components:
 - Each processor has a local interrupt controller
 - There is a system-level interrupt controller that distributes interrupts
 - If an I/O device needs an interrupt, the system level interrupt distributor routes it to the appropriate processor → **interrupt routing**
 - The interrupt distributor and the interrupt routing is configured by the operating system as a part of the boot process
 - Local interrupt controllers can send interrupts to the other processors as well
 - This is a way of communication between the processors

- When there are too many interrupts...
 - There are devices that generate too many interrupts
 - E.g. gigabit speed network devices

Interrupt source	Typical rate	Maximal rate
10 Mbps Ethernet	812	14880
100 Mbps Ethernet	8127	148809
Gigabit Ethernet	81274	1488095

- The CPU has to handle interrupts continuously, it can not do anything else
- Rule of thumbs:
 - Interrupt handling routines should be as short as possible
 - Interrupt rates are maximized in critical systems
 - In the critical intervals of the programs interrupts should be disabled
 - **Interrupt moderation**
 - The device awaits several events and indicates it using a single interrupt
 - The CPU handles the multiple events in a single interrupt

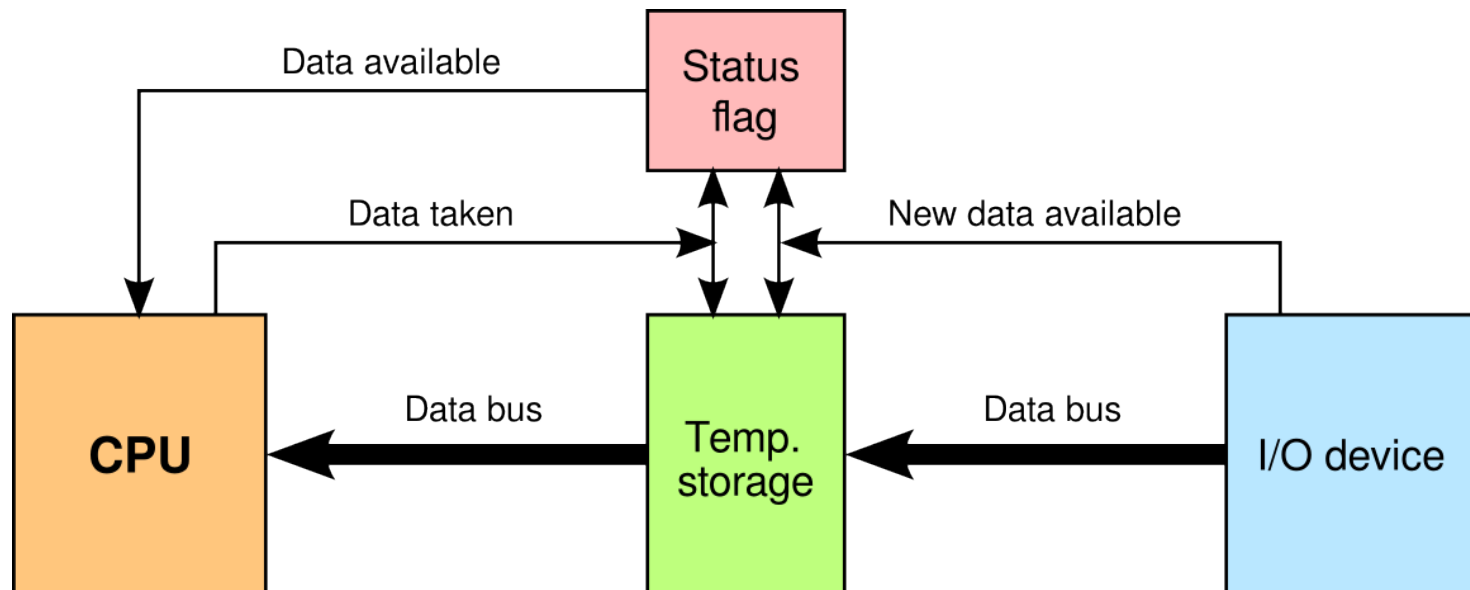
- Questions to investigate:
 - How does the CPU communicate with I/O devices? ✓
 - How do I/O devices communicate with the CPU? ✓
 - **How to transmit data efficiently, without errors?**
 - How to connect the I/O devices to the CPU?

- Ultimate goal:
 - Transferring data from the I/O device to the memory (or vice versa)
- Problems:
 - What to do, if the source and the target have different speed?
→ **Flow control**
 - Which path to choose to transfer the data to/from the memory?

- The principle problem is:
 - How do we know that both the CPU and the I/O device are ready to transmit data?
- 1) **Unconditional data transmission**
 - No flow control at all
 - Neither the CPU nor the I/O device can indicate its status to the other side
 - Two problems can occur:
 - Data over-run error (the sender is too fast, the receiver did not even process the previous message, when the new arrives)
 - Data deficiency (the sender is too slow, the receiver thought it got the next data, but it did not happen)
 - Usage: reading a button, control a LED , ...

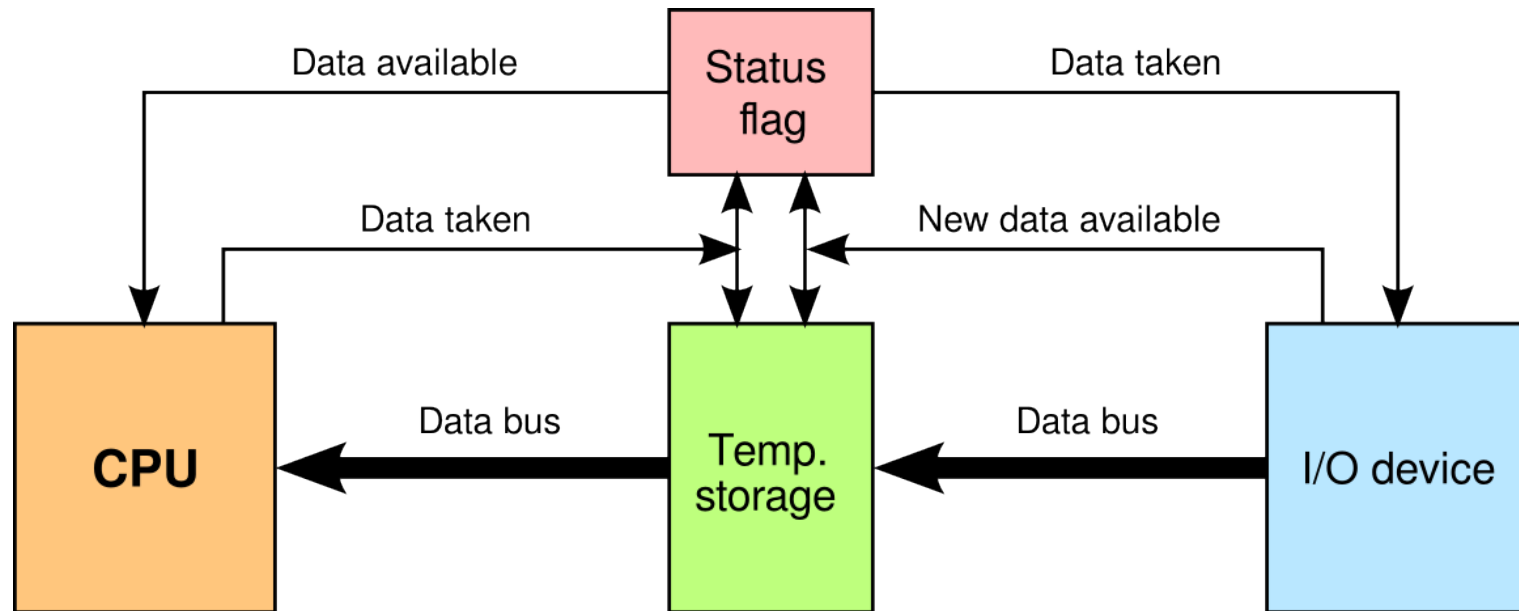
2) Conditional transmission with one-sided handshake

- The speed of either the CPU or the I/O device can not be influenced
- A status flag is used: is there valid data available?
- Example: voice input, network card
- Example: an input device
→ with the status register the “data deficiency” errors can be avoided



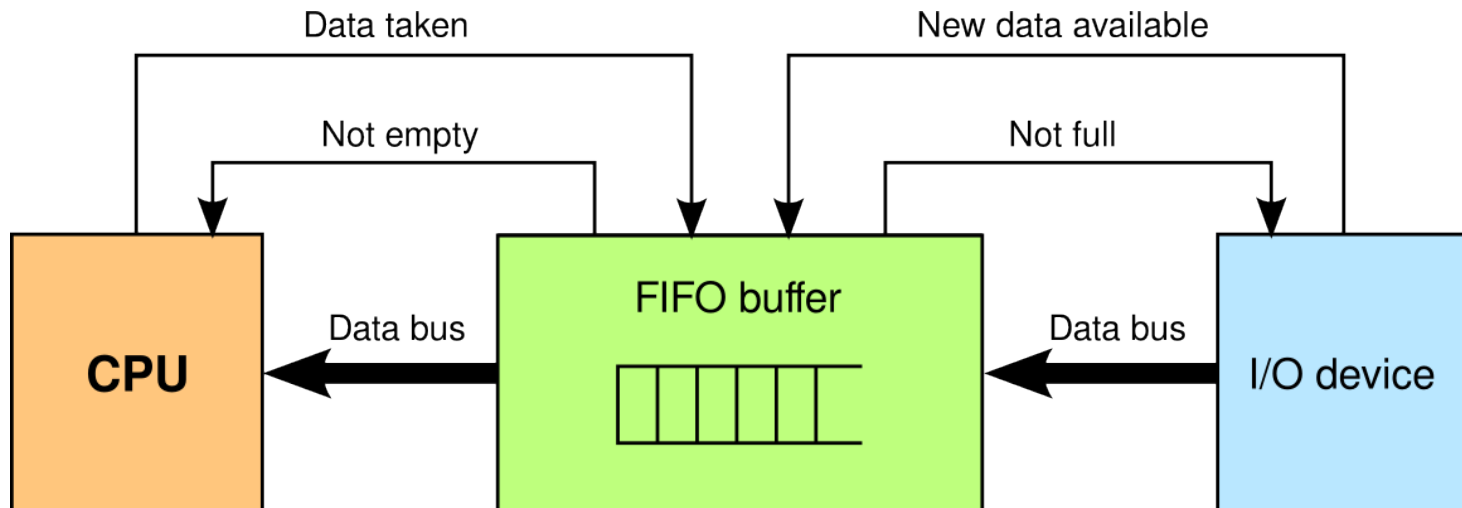
3) Conditional transmission with two-sided handshake

- The speed of both sides can be influenced
- Status flag: is there valid data available?
 - Both sides check the status flag
- Example: an input device
 - this way both the “data over-run” and the “data deficiency” problems can be avoided



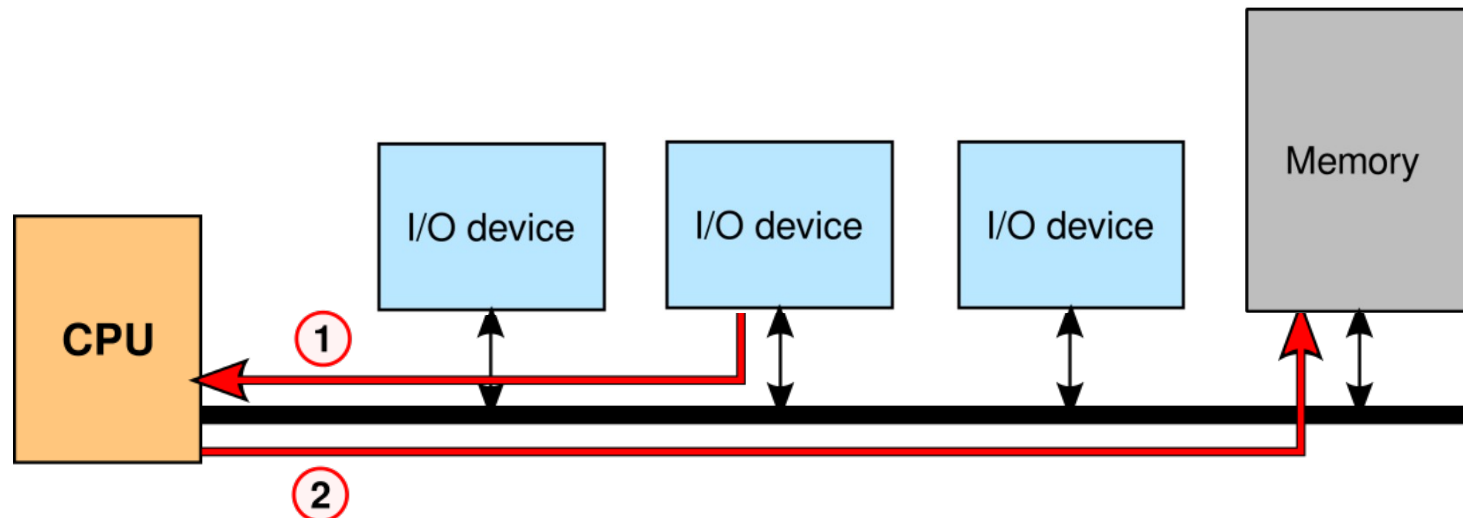
4) Conditional transmission with a FIFO buffer

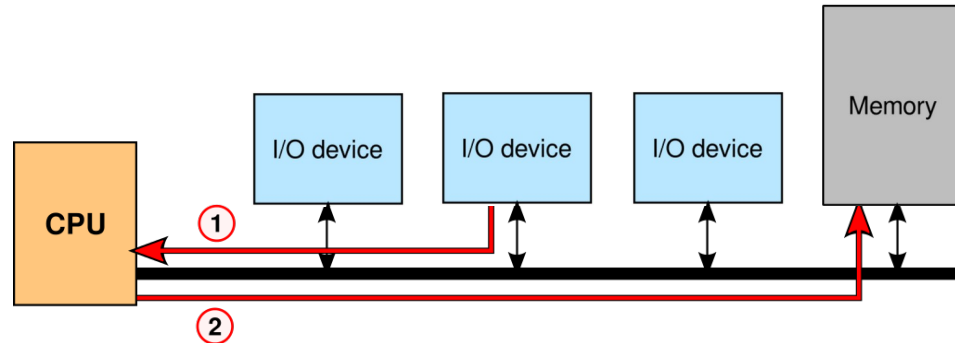
- The speed of both sides can be influenced
- The partners don't have to wait for the other each times a single data is transmitted
- It is beneficial if
 - the data rate is varying
 - the availability of the CPU and/or the I/O device is varying



- Ultimate goal:
 - Transferring data from the I/O device to the memory (or vice versa)
- Problems:
 - What to do if the source and the target have different speed?
→ Flow control ✓
 - **Which path to choose to transfer the data to/from the memory?**

- Ultimate goal:
 - Transferring data from the I/O device to the memory (or vice versa)
- Which path to choose to transfer the data to/from the memory?
- Solution 1: **through the CPU**





- **Polling based data transfer:**

- The status of the I/O device is monitored continuously:

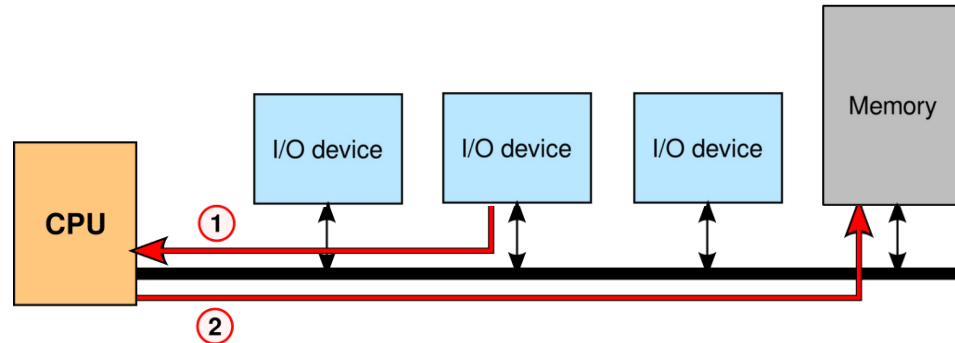
```
loop: R0 ← IO[0x64]
```

```
      JUMP loop IF R0==0
```

```
      R0 ← IO[0x60]
```

```
      MEM[0x142] ← R0
```

- Critical problem: choosing the right polling period
 - Too frequent → high CPU load
 - Too rare → loss of information



- Interrupt based data transfer:**

- The status of the I/O device is monitored by interrupts
- Interrupt handling routine:

keyhandler: PUSH R0

R0 ← IO[0x60]

MEM[0x142] ← R0

POP R0

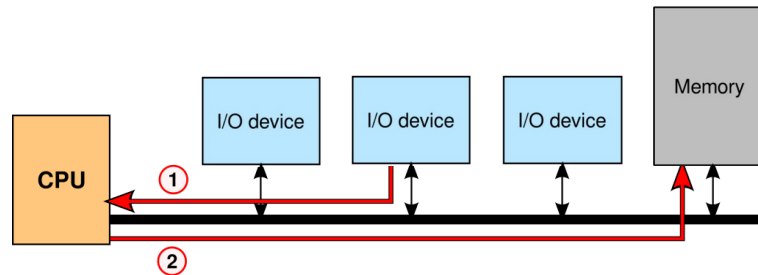
RET

- Only as high load to the CPU as necessary

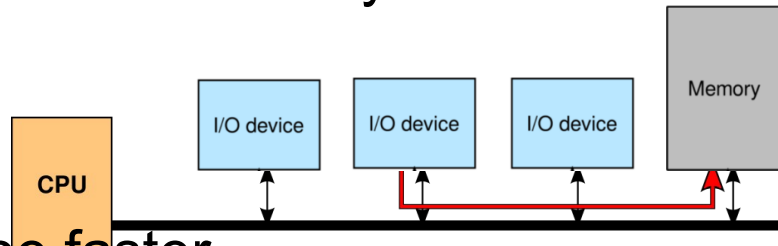
- Example for polling (CPU: 1 GHz, 600 cycles/poll)
 - Mouse:
 - Enough to have 30 polls/s
 - $30 \text{ poll/s} * 600 \text{ cycles/poll} = 18\,000 \text{ cycles/s}$
 - CPU: $10^9 \text{ cycles/s} \rightarrow 18\,000 / 10^9 = 0.0018\%$, OK.
 - Disk:
 - Interface: $100 \cdot 10^6 \text{ byte/s}$, 500 byte/block
 - Polling period: $100 \cdot 10^6 \text{ byte/s} / 500 \text{ byte/block} = 200\,000 \text{ poll/s}$
 - $200\,000 \text{ poll/s} * 600 \text{ cycles/poll} = 120\,000\,000$
 - CPU: $10^9 \text{ clocks/s} \rightarrow 120 \cdot 10^6 / 10^9 = 12\%$
 - Too much load for a single I/O device!

- Example for interrupt-based data transfer
 - Disk
 - Assumptions:
 - The disk is active in 10% of the time
 - Interrupt processing time: 600 cycles
 - Data transfer time: 100 cycles
 - CPU speed: 1 GHz
 - Time devoted to interrupt processing:
 - $0.1 \cdot (100 \cdot 10^6 \text{ byte/s} / 500 \text{ byte/block} \cdot 600 \text{ cycles/interrupt})$
 $= 12\,000\,000 \text{ cycles/s}$
 - CPU: $10^9 \text{ cycles/s} \rightarrow 12 \cdot 10^6 / 10^9 = 1.2\%$
 - Data transfer time:
 - $0.1 \cdot (100 \cdot 10^6 \text{ byte/s} / 500 \text{ byte/block} \cdot 100 \text{ clocks/transfer})$
 $= 2\,000\,000 \text{ clocks/s}$
 - CPU: $10^9 \text{ clocks/s} \rightarrow 2 \cdot 10^6 / 10^9 = 0.2\%$
 - In total: $1.2\% + 0.2\% = 1.4\%$

- I/O device → memory data transfer, as seen so far:

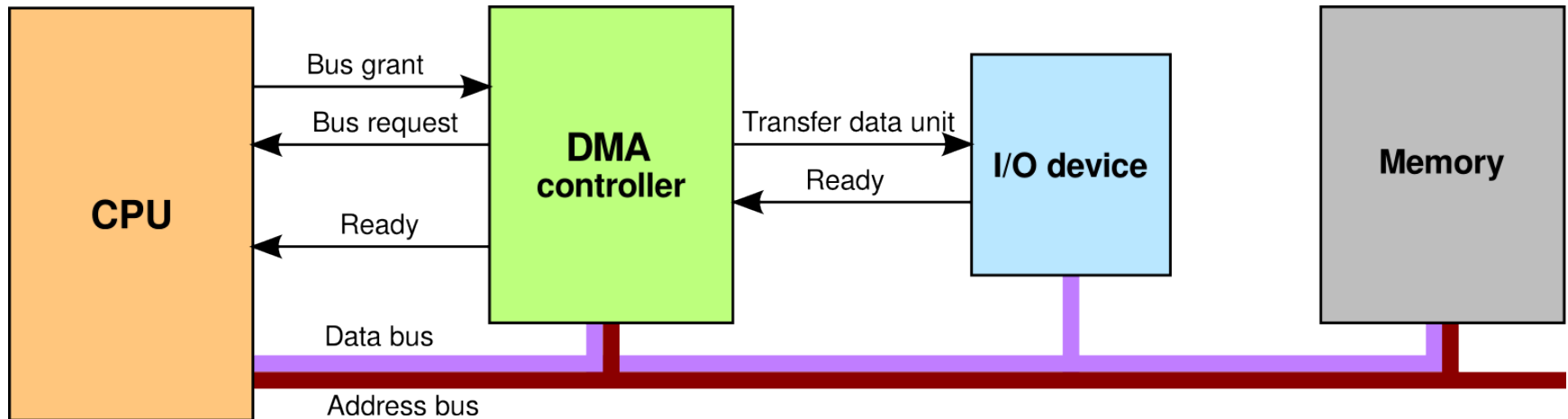


- Can we do it in a direct way?



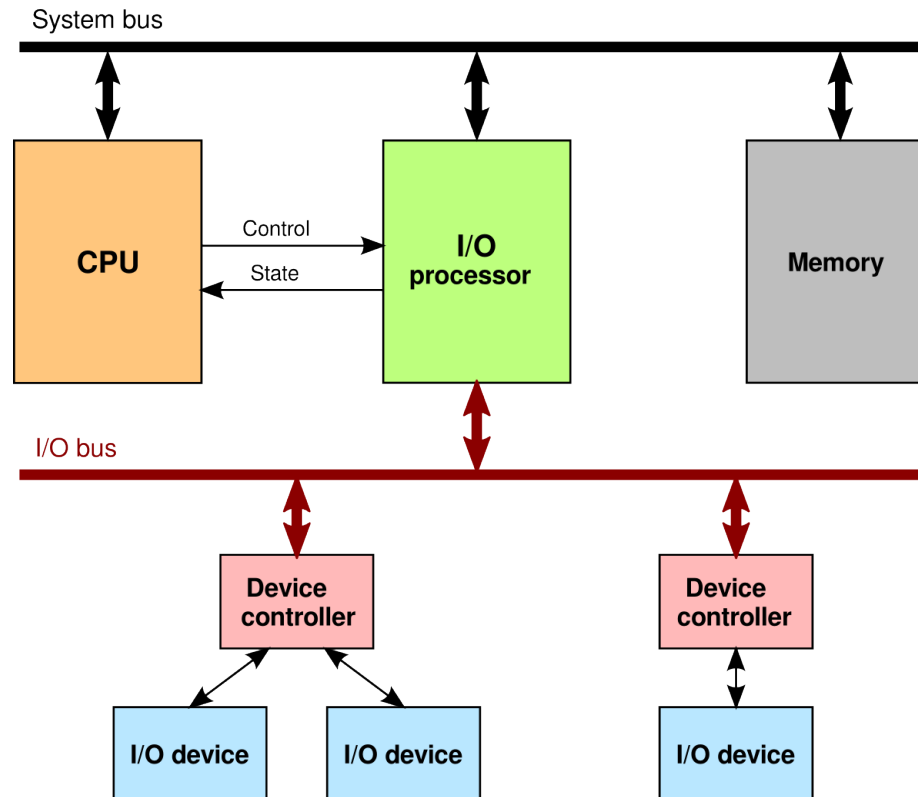
- It would be faster
- CPU could do something else during the data transfer
- Solutions:
 - DMA
 - I/O processor

- “I/O device → memory” data transfer without the CPU
- Steps:
 1. Setting up the DMA controller
 - Which peripheral
 - Which memory address
 - Direction of the data transfer (reading or writing)
 - Number of data units to transfer
 2. The DMA controller controls the data transfer
 - It obtains the right to use the bus (from the CPU)
 - It does the data transfer (possibly with flow control)
 - Plays the role of the CPU
 3. The DMA controller sends an interrupt whenever the data transfer is ready



- **CPU has to work only when setting up the controller and when the data transfer is accomplished**
- This was the system level (third party) DMA controller
- Instead of this, nowadays they are using first party DMA controllers
 - Each I/O device has its own DMA controller
 - They are all competing for the bus
 - ... with the CPU and with each other
 - The winner can transfer the data of its I/O device to the memory

- Evolution of the DMA concept
- The I/O processor has an **own instruction set**
- **I/O program:**
 - A series of transfer requests
 - Simple data processing tasks
 - CRC, checking parity
 - Compression/decompression
 - Byte order conversion
 - Etc.
- Execution:
 1. CPU gives the pointer of the I/O program to the I/O processor
 2. The I/O processor executes them one by one
 3. The I/O processor generates an interrupt when the I/O program is accomplished



- The CPU can access the I/O devices only through the I/O processor
→ **device independence!**
- Task of the device controller:
- Device specific interface ↔ I/O bus protocol translation

- Questions to investigate:
 - How does the CPU communicate with I/O devices? ✓
 - How do I/O devices communicate with the CPU? ✓
 - How to transmit data efficiently, without errors? ✓
 - **How to connect the I/O devices to the CPU?**

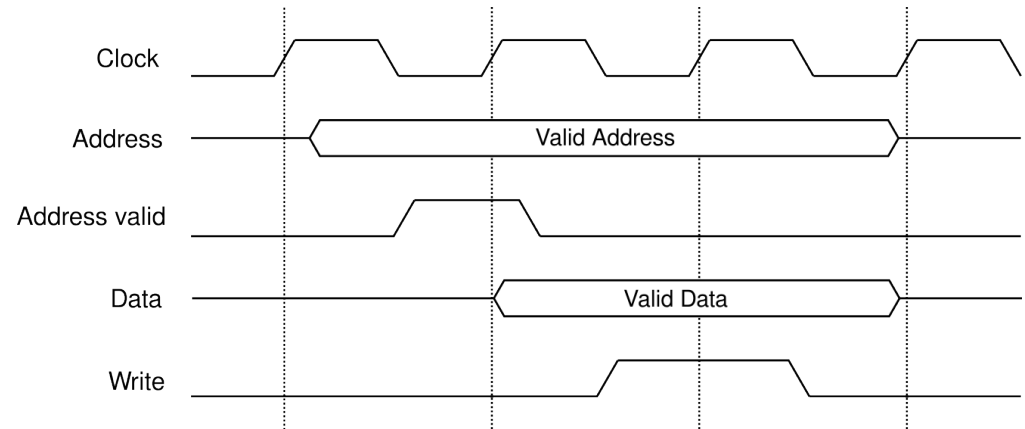
- How to interconnect
 - the CPU,
 - the memory,
 - I/O devices?
- So far: memory and I/O devices were sitting on the CPU bus
- It can be done more efficiently

- **Point-to-point** interconnects
 - Dedicated channel
 - No contention, no waiting → faster
 - The more I/O devices we have, the more point-to-point connections are needed → expensive
- **Bus based** interconnects
 - Shared channel
 - Shared resource → can be a bottleneck (contention, waiting for each other, etc.)
 - Everybody shares the same bus → cheaper
 - We need algorithms to control access to shared channel

- Width of the bus = number of wires used for transmitting data
- Wide bus:
 - More bits can be transmitted at the same time → can be faster
 - More expensive
- **Contradiction: the fastest buses are serial!**
 - Wide bus → more wires
 - The length and the electrical behavior of the wires is not the same
 - signals transmitted at the same time are shifted at arrival!
 - It is only a problem if the amount of shift is not much thinner than the clock tick
- Trend: serial transmission to everywhere → no shift, nothing can stop up to send fast

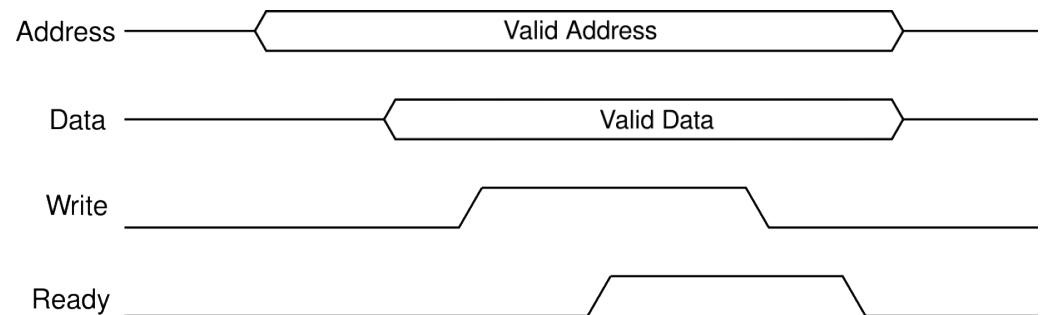
- Synchronous:**

- Shared clock signal
- Data validity is tied to the clock signal



- Asynchronous:**

- No clock signal
- Validity of the data is given by strobe signals

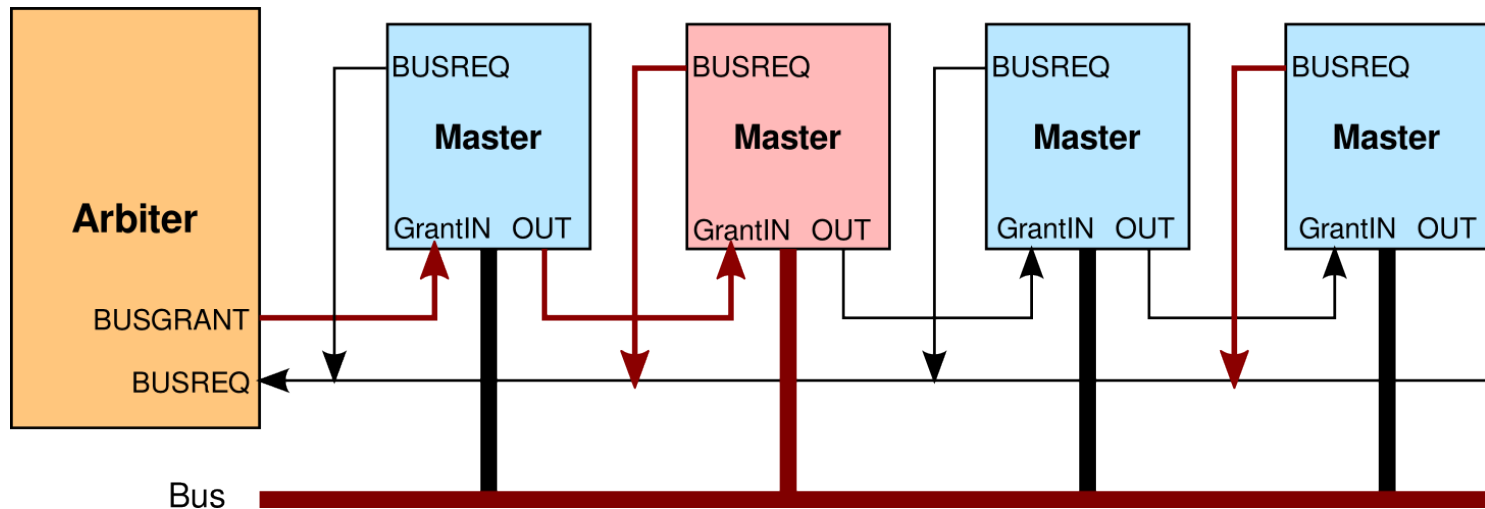


- Synchronous – asynchronous: **which is better?**

- Synchronous – asynchronous: which is better?
 - None of them!
 - **There can be a signal shift between the data and the clock/strobe lines!**
- Solution:
 - Remove clock/strobe
 - Just a single wire remains, for the data transfer itself
- How does the target know where the bit boundaries are?
 - It detects it automatically from the 0-1 transitions
 - **self-clocked timing**
 - It is feasible only if there are enough 0-1 transitions
- **8b/10b encoding:**
 - It creates 0 – 1 transitions
 - Gigabit Ethernet, PCI Express 1.0/2.0, USB 3.0, SATA, HDMI, DVI, etc.
 - Drawback: overhead
 - Enhancement
 - 64b/66b encoding: 10 GB Ethernet, 100 GB Ethernet
 - 128b/130b encoding: PCI Express 3.0/4.0, USB 3.1, SATA 3.2, DisplayPort 2.0
 - 256b/257b: Fibre Channel

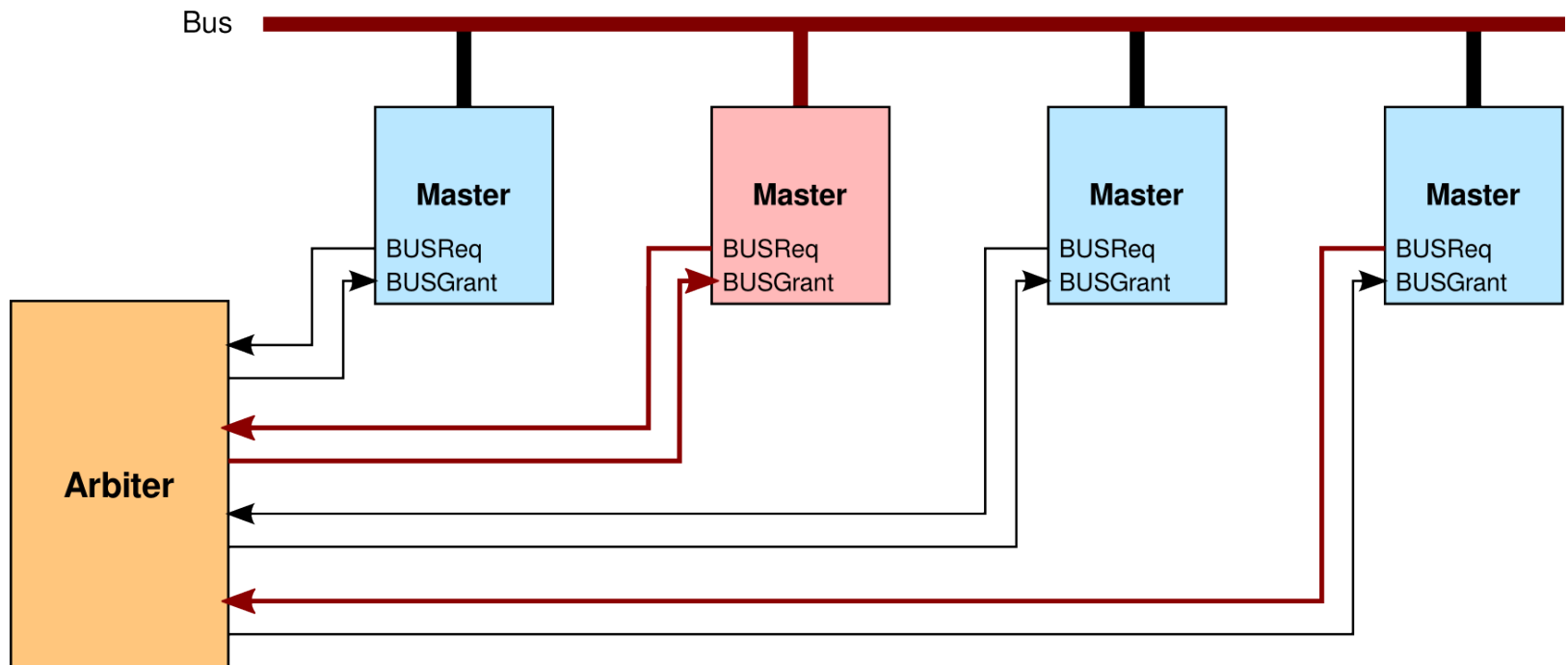
- Devices on the bus
 - **Bus Master**: a device that can grab the right to use the bus
 - **Bus Slave**: can not grab the bus, not able to manage data transfer by its own
- Bus is a shared resource
 - More masters can request to use it at the same time
 - Only one can grab the right to use the bus
- **Arbitration**
 - The decision of the contention to capture the bus

- A special unit: **Arbiter**
 - Decides who can use the bus next
- Serial Arbitration: **Daisy Chain**



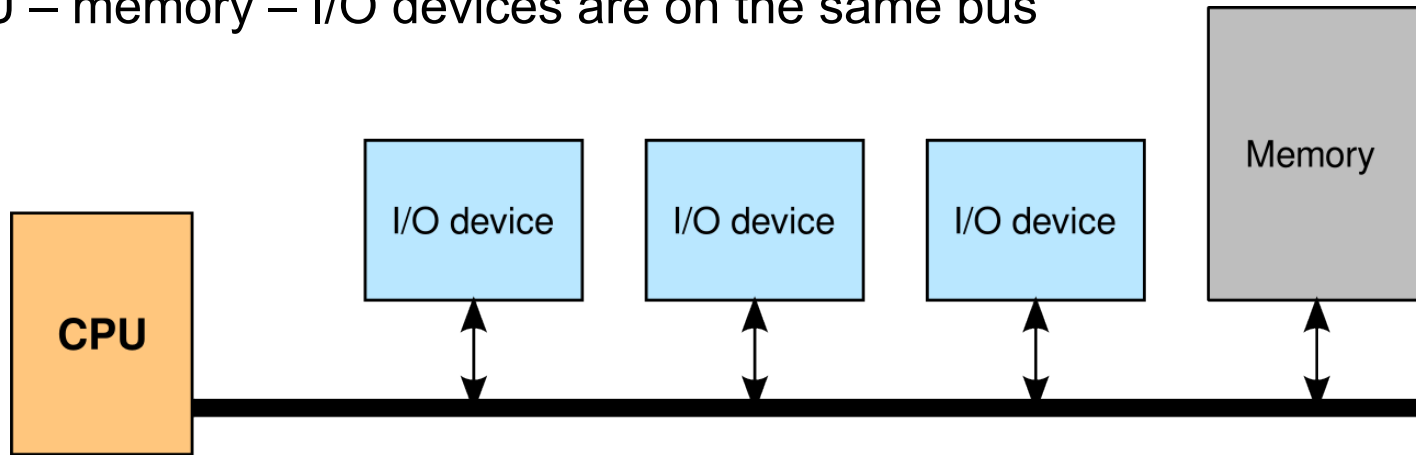
- Advantage:
 - Easy to extend
- Drawback:
 - Not fair (starvation at the end of the row)

- **Parallel** centralized arbitration
- More flexible in assigning priorities
 - Round-robin
 - Delay sensitive devices can have priority
 - Etc.



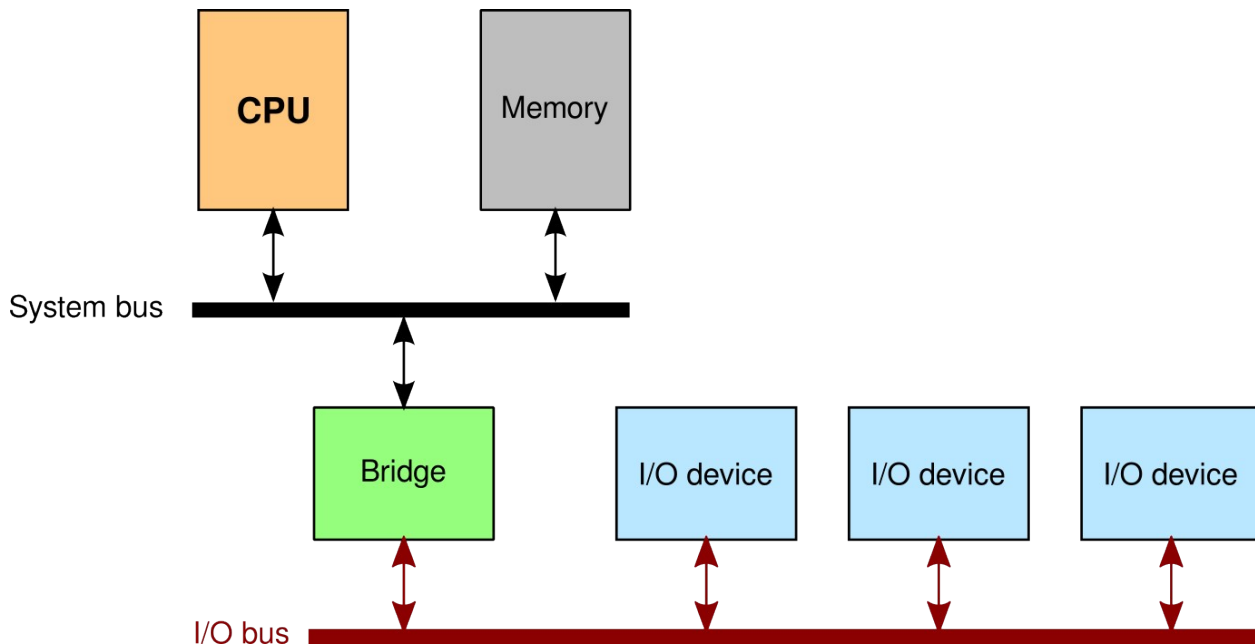
- No arbiter
- **Distributed arbitration** (e.g. SCSI):
 - Everybody sees all the requests
 - Everybody knows its own and the others' priority
 - The one having the highest priority gets the bus, the others have to wait
- **Collision detection based arbitration**
 - No arbitration at all
 - If somebody wants to use the bus, it can start data transmission immediately
 - During data transmission it listens to the bus as well
 - If it can hear its own transmission clearly → OK
 - If it cant → there was a collision. It waits a bit and tries again later.
- **Advantages of distributed arbitration:**
 - **No arbiter, that can break down. No critical component.**

- CPU – memory – I/O devices are on the same bus



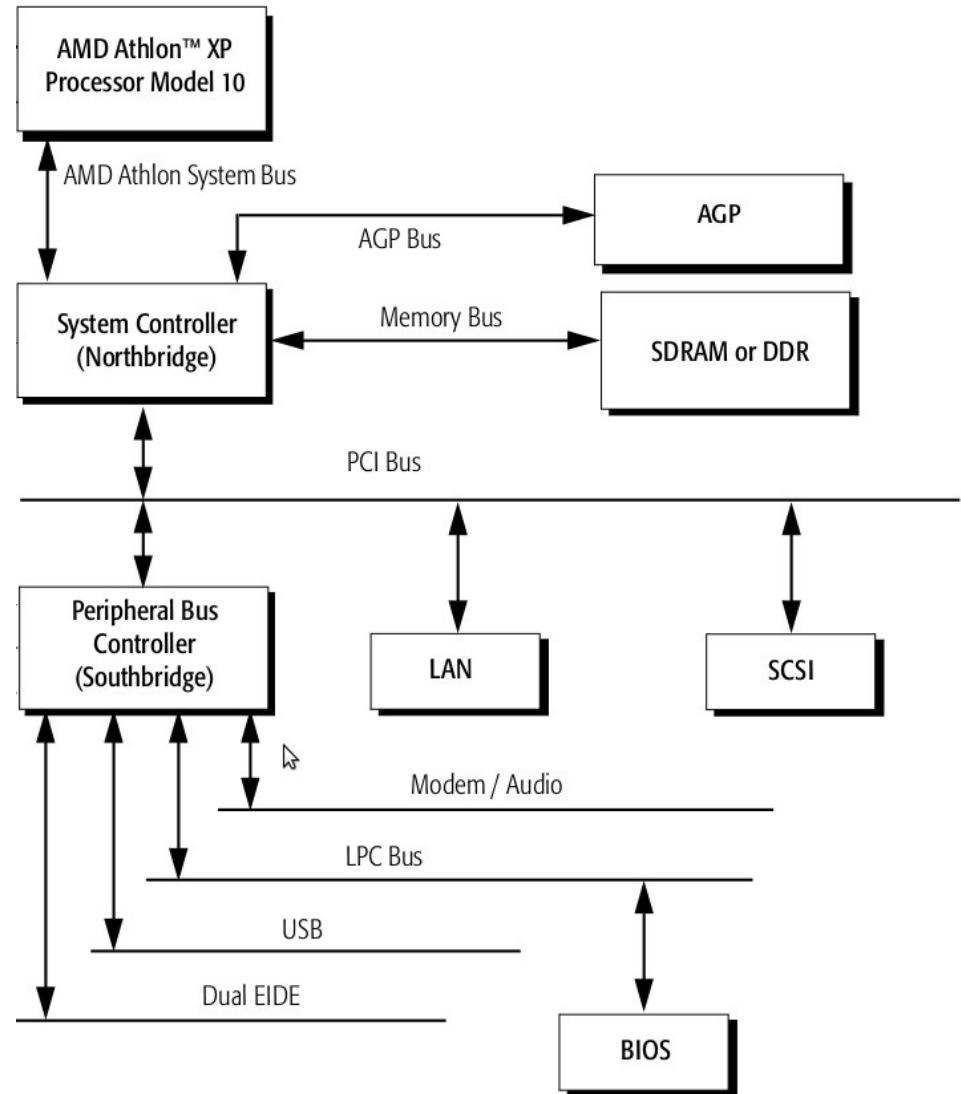
- Easy to implement
- Drawback:
 - If we replace the CPU → it uses a higher clocked bus / different bus protocol → old I/O devices may not support it
 - The clock freq. and the bus protocol of CPU-s change from model to model
 - We don't want to throw out our I/O devices, when buying a new CPU → I/O devices need a constant, standard interface

- I/O devices are connected to an I/O bus, memory is connected to the system bus
- I/O devices are reached through a **bridge**

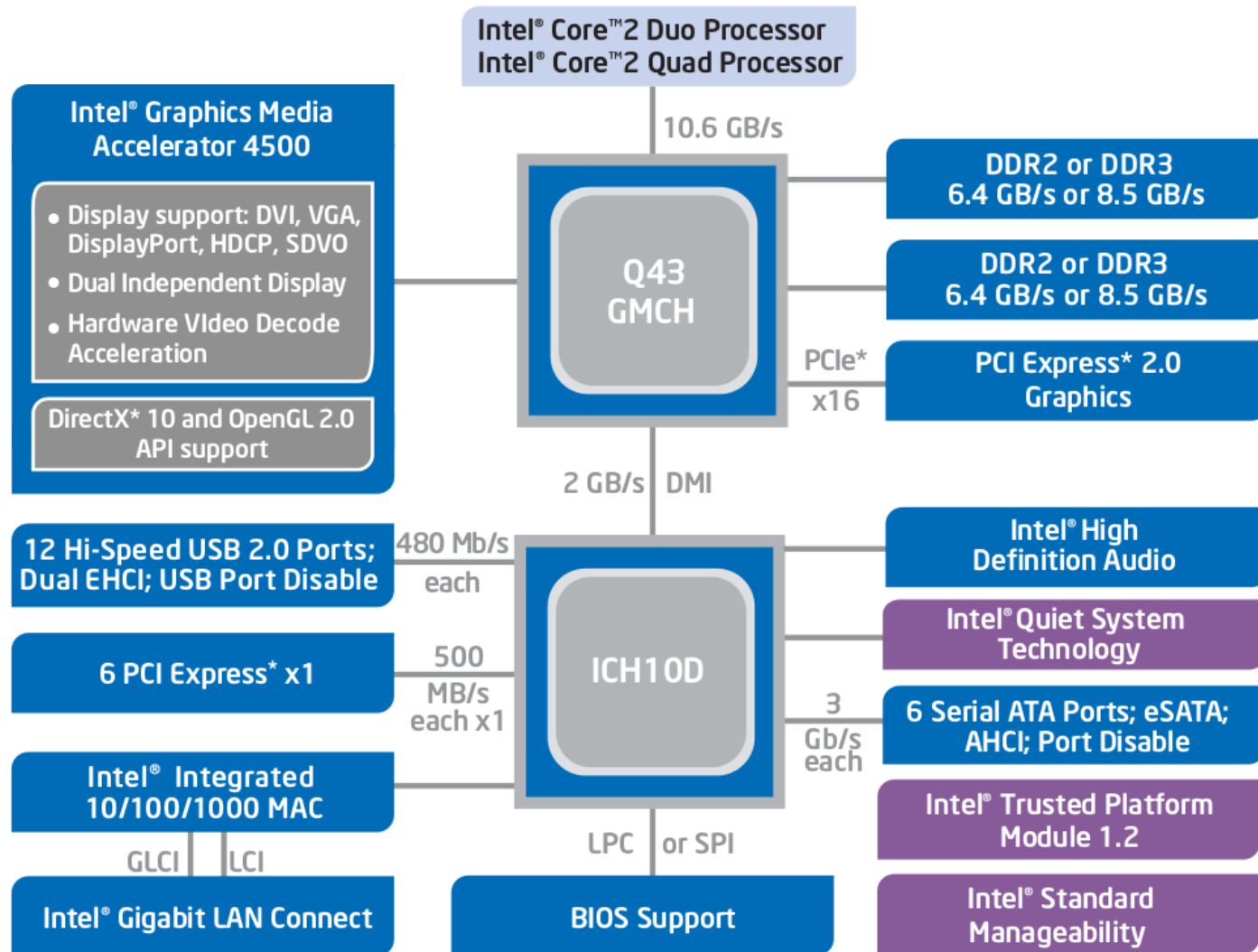


- System bus: speed and protocol depends on the CPU protocol
- I/O bus: constant, standardized interface

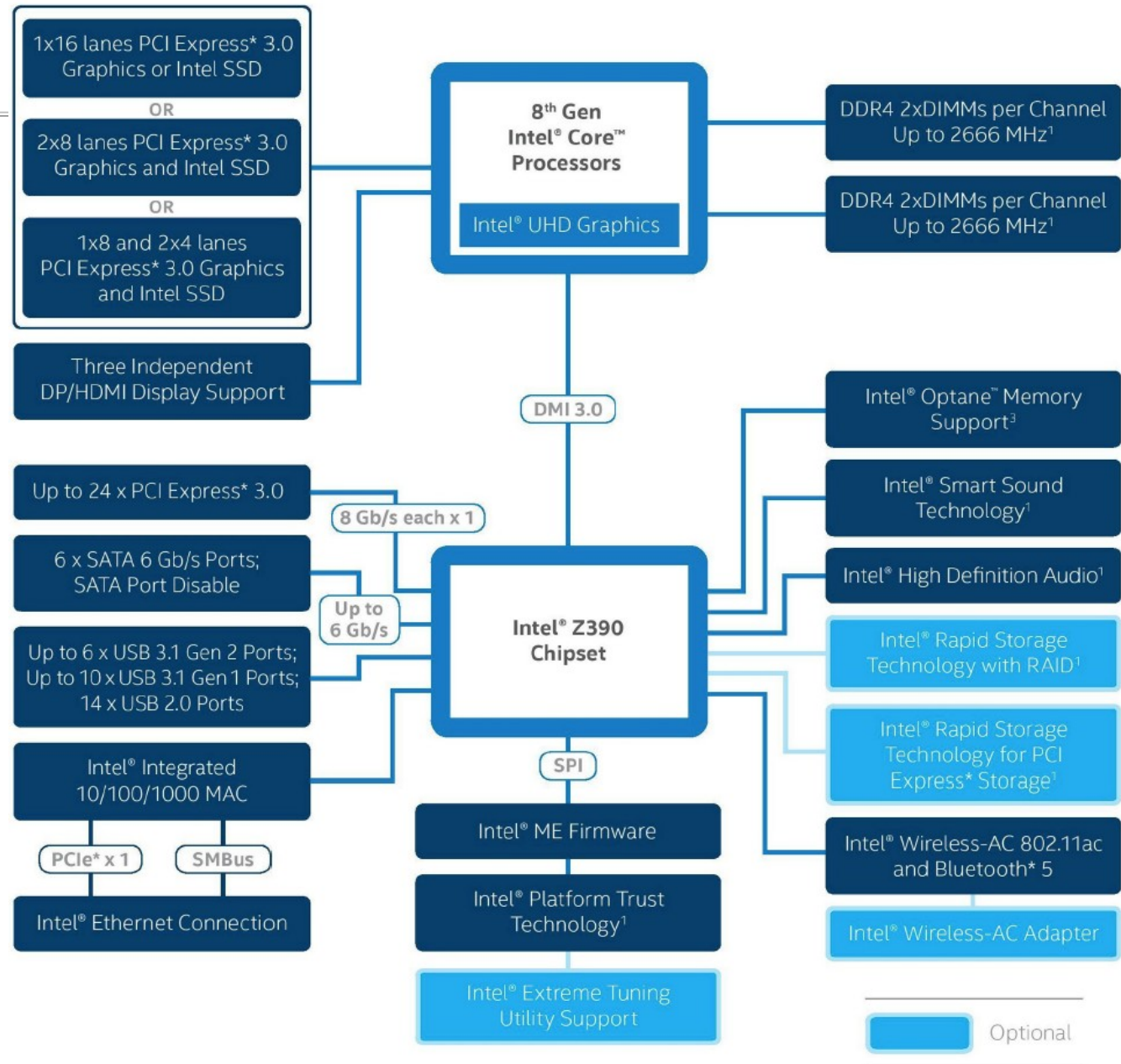
- Separate buses for:
 - CPU
 - Memory
 - I/O devices
- System bus:
 - CPU dependent
- Memory bus:
 - Standardized
 - CPU independent
- I/O bus (PCI):
 - Standardized
 - CPU independent



BRIDGE BASED SYSTEMS



BRIDGE BASED SYSTEMS





DEPARTMENT OF
NETWORKED SYSTEMS
AND SERVICES

