Input / Output (I/O)

Systems Architecture

Animesh Trivedi
a.trivedi@vu.nl

Reading at home

- Chapter 3 and 7 from Carl Hamacher and Zvonko Vranesic, Computer Organization, 6th edition, McGraw-Hill Education, 2011. ISBN-13: 978-0073380650
- Operating Systems: Three Easy Pieces, Chapter 36 - I/O Devices (section 36.1 till 36.6) http://pages.cs.wisc.edu/~remzi/OSTEP/file-devices.pdf

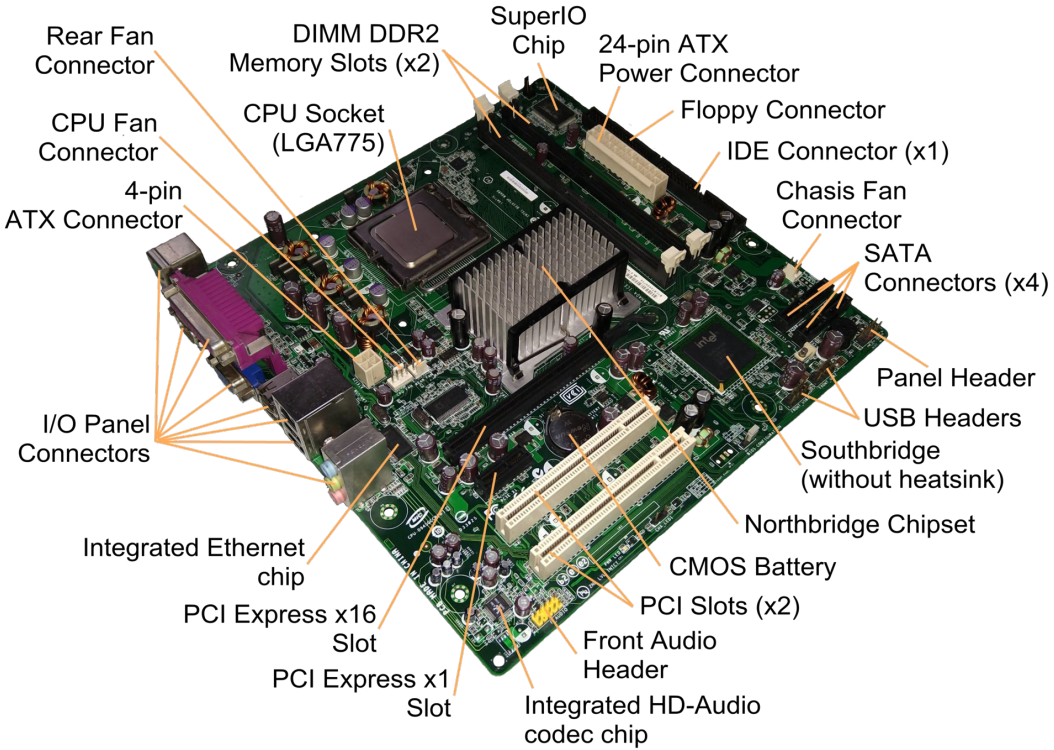
What is I/O - Input/Output

So far, in the course we have seen how the CPU does data processing, and how the main memory subsystem helps

Where does this data come from? Where does it go?

One fundamental CPU capability is to communicate with the outside world

Inside your computer



So many possibilities to connect different “external” devices or components to your computer!

On a practical side: Linux commands

https://opensource.com/article/19/9/linux-commands-hardware-information

lspci -tv : shows your PCIe tree inside the system

lsusb : shows all the connected USB root and devices

lscpu : CPU topology (a single CPU is not a single CPU)

lshw : all hardware information about your system

Example command: lsusb

```
atr@atr-XPS-13:~$ lsusb -t
/:  Bus 04.Port 1: Dev 1, Class=root_hub, Driver=xhci_hcd/2p, 10000M
    |__ Port 1: Dev 25, If 0, Class=Hub, Driver=hub/7p, 5000M
        |__ Port 2: Dev 26, If 0, Class=Vendor Specific Class, Driver=r8152, 5000M
/:  Bus 03.Port 1: Dev 1, Class=root_hub, Driver=xhci_hcd/2p, 480M
/:  Bus 02.Port 1: Dev 1, Class=root_hub, Driver=xhci_hcd/6p, 10000M
/:  Bus 01.Port 1: Dev 1, Class=root_hub, Driver=xhci_hcd/12p, 480M
    |__ Port 5: Dev 2, If 1, Class=Video, Driver=uvcvideo, 480M
    |__ Port 5: Dev 2, If 0, Class=Video, Driver=uvcvideo, 480M
    |__ Port 7: Dev 3, If 0, Class=Wireless, Driver=btusb, 12M
    |__ Port 7: Dev 3, If 1, Class=Wireless, Driver=btusb, 12M
    |__ Port 9: Dev 39, If 0, Class=Hub, Driver=hub/7p, 480M
        |__ Port 5: Dev 40, If 3, Class=Audio, Driver=snd-usb-audio, 480M
        |__ Port 5: Dev 40, If 1, Class=Audio, Driver=snd-usb-audio, 480M
        |__ Port 5: Dev 40, If 2, Class=Audio, Driver=snd-usb-audio, 480M
        |__ Port 5: Dev 40, If 0, Class=Audio, Driver=snd-usb-audio, 480M
        |__ Port 6: Dev 41, If 0, Class=Hub, Driver=hub/4p, 480M
            |__ Port 4: Dev 43, If 0, Class=Human Interface Device, Driver=usbhid, 1.5M
            |__ Port 4: Dev 43, If 1, Class=Human Interface Device, Driver=usbhid, 1.5M
        |__ Port 7: Dev 42, If 0, Class=Human Interface Device, Driver=usbhid, 1.5M
atr@atr-XPS-13:~$
```

The basic idea of a devices

I/O devices can:

- Provide data **input** to the CPU by putting data in the memory
  - From the sensors, voice commands, keyboard, mouse …
- **Output** the data from the CPU (from the memory)
  - To a display monitor, printer, actuators …

Some can do both: network, storage, touch screen…

Often in a single computer there are a mix of external (keyboard, mouse) and internal devices (WiFi, storage)

Devices come in a variety of shapes and sizes

Keyboard/mouse - as fast as we can type 10-100s bytes/sec (**once** in a while)

Bluetooth - headset, speakers ~ 10-100 Megabits/sec (**regular**)

Storage - 10-10,000 MBytes/sec with latencies in 10-1,000 microseconds (**high-speed**)

Ethernet/Infiniband - 100-200 Gbps with 1-2 microseconds (**very high-speed**)

CPU - nanoseconds clock

How do we do I/O then?

When thinking about I/O, there are a few interesting problems:

1. How do we connect devices to the CPU and memory (or the computer)?
2. How do we identify and address devices?
3. How do we communicate with devices?
4. How do devices transfer data between computer and themselves?
5. [...]

Connecting Devices

The idea of a bus, revisited

A bus is a communication system that connects multiple components

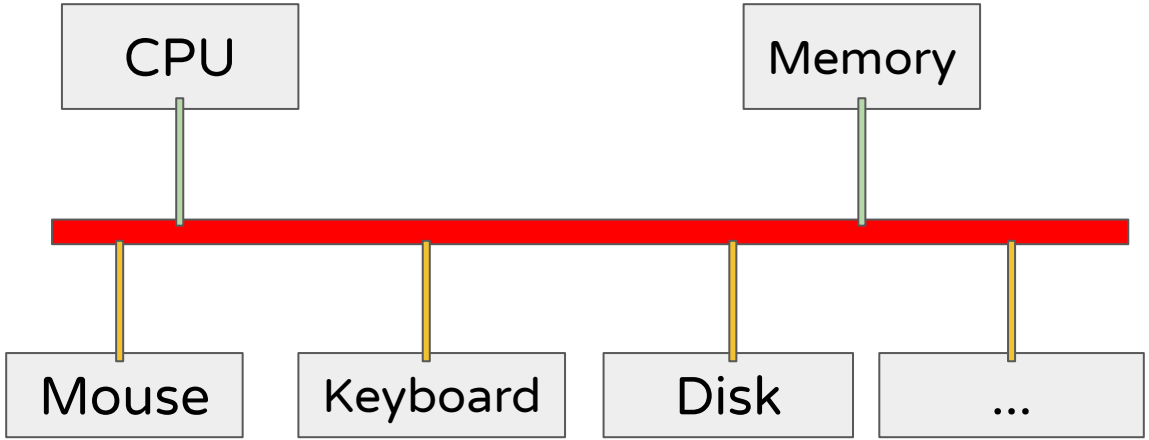
- Can be internal or external

Each bus has a way to

- Connect components (including devices)
- Give an address to connected components
- Send commands - i.e., a request to initiate an I/O operation
- Do data transfers between devices and memory

Can you think of examples of a bus already?

A single, shared bus structure



Can work, but

- Different devices have different speeds - how to take turns
- Cost - high speed links to slow devices is a waste
- New devices - how to expand to connect to new devices

Often, there are multiple buses

CPU to Memory connections, often known as the Front Side Bus (**FSB**)

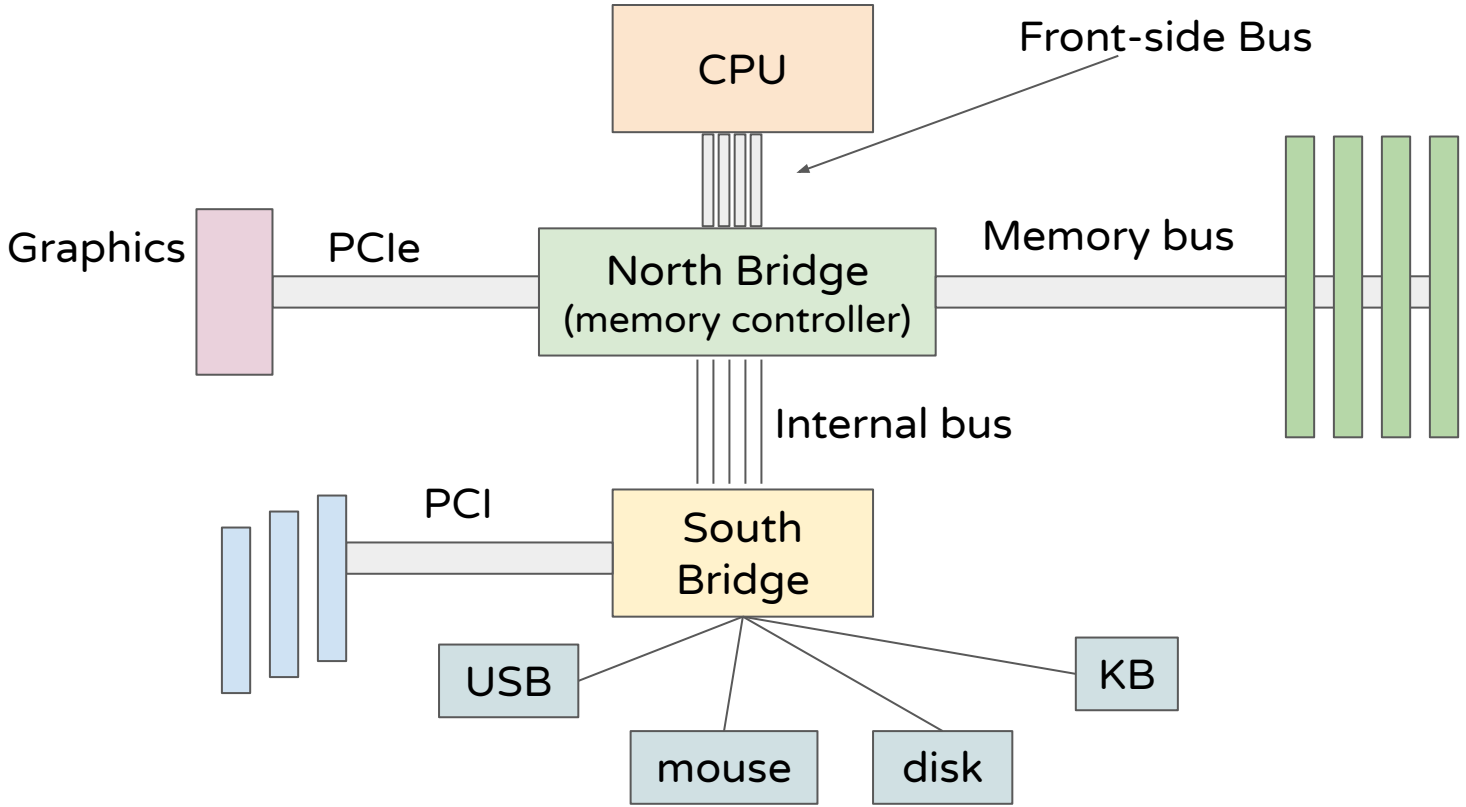
- More buses inside to connect other high performance components

I/O Buses:

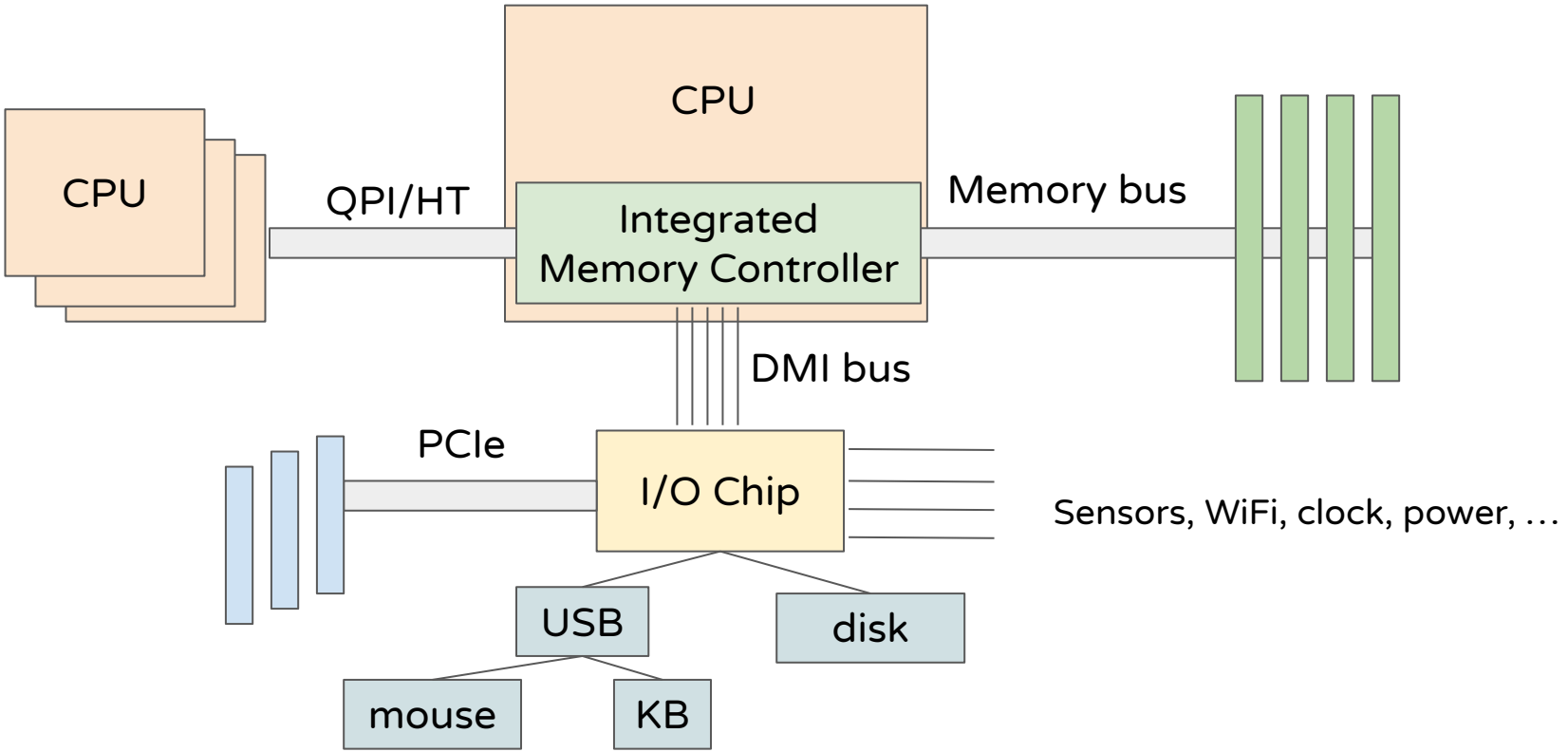
- Peripheral Component Interconnect Express (PCIe): point-to-point
- Universal Serial Bus (USB): point-to-point
- Storage: SAS, SCSI
- For displays: HDMI, DVI, VGA, …

Internal and external buses: distance, speed, cost, and expandability

A typical setup



A more modern setup



Bus: basic components

A bus (or also known as interconnect) generally has:

- **Data Lines:** used for transmitting data
- **Address Lines:** used for addressing or identifying from which device
- **Control Lines:** send commands (R/W), activate/schedule transfers

A bus has a set of rules called a ***bus protocol,*** that needs to be followed by the connected devices for a successful data transfer (taking turns, etc.)

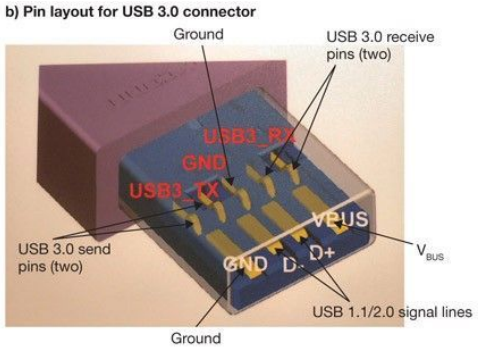
During data transfer, one device becomes **the master** (often the CPU)

From the device side

Devices are attached to a bus through an **interface** that has:

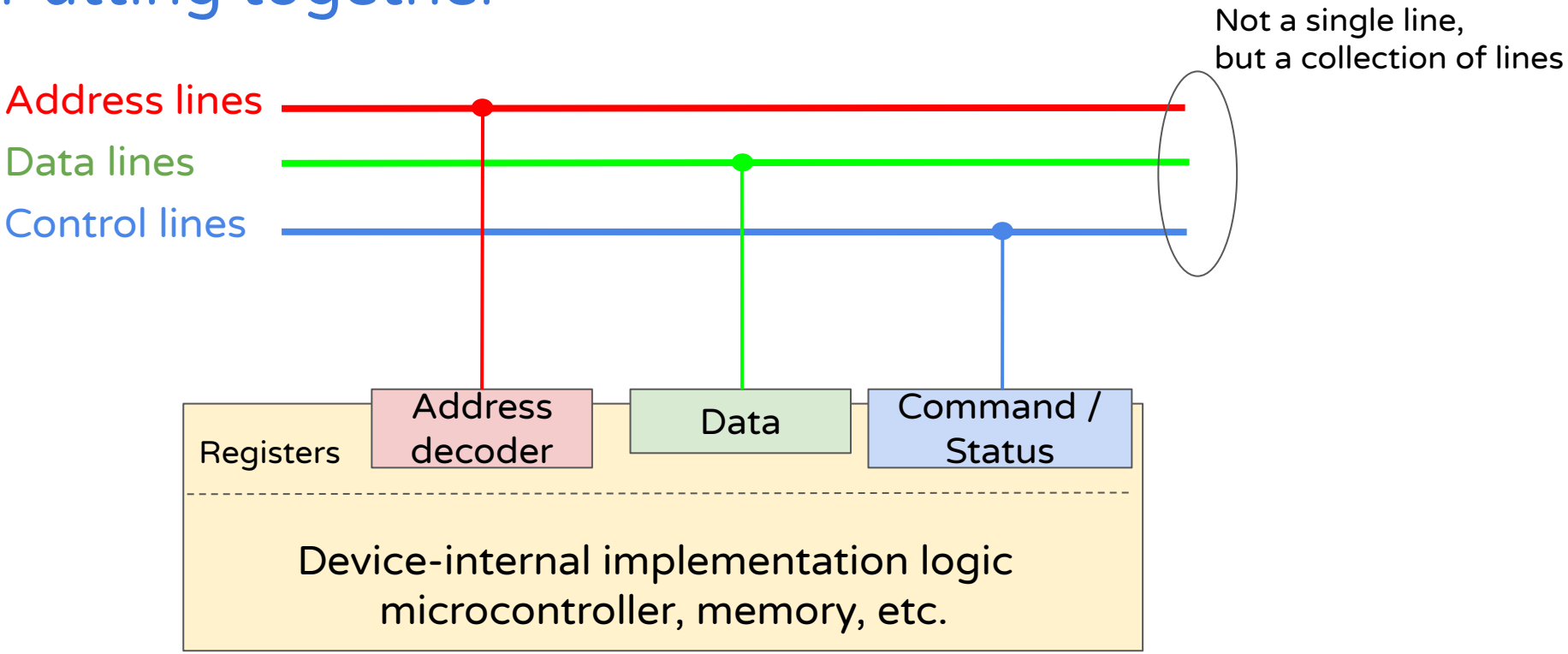
- **Address decoder:** for detecting if its address is being called for I/O
- **Data registers:** to store incoming and outgoing data (also device buffers)
- **Status and control registers:** to get commands and report device status

Often devices are connected electronically, but a wireless setup is also possible. In that case, there is some device component that is connected inside the computer.

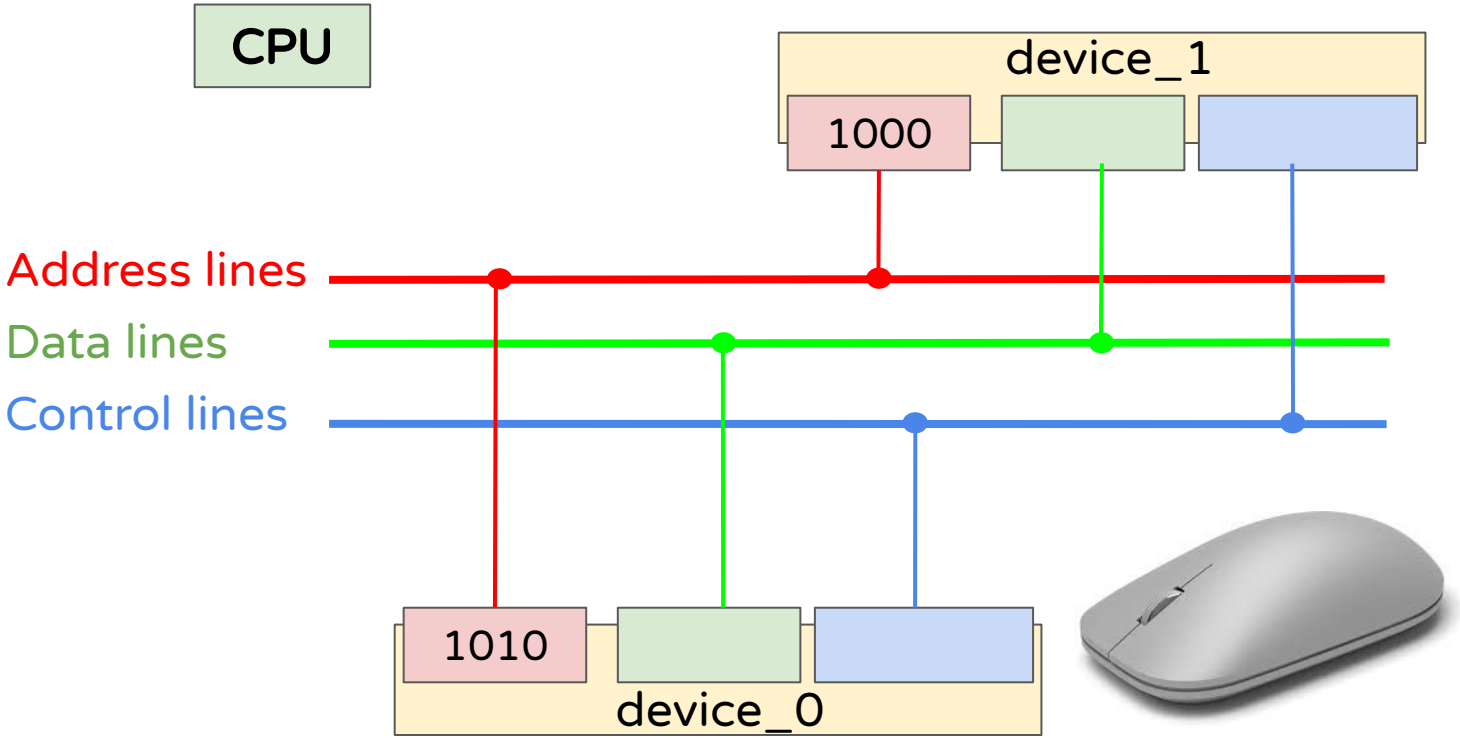


https://www.pinterest.com/pin/414401603184159650/

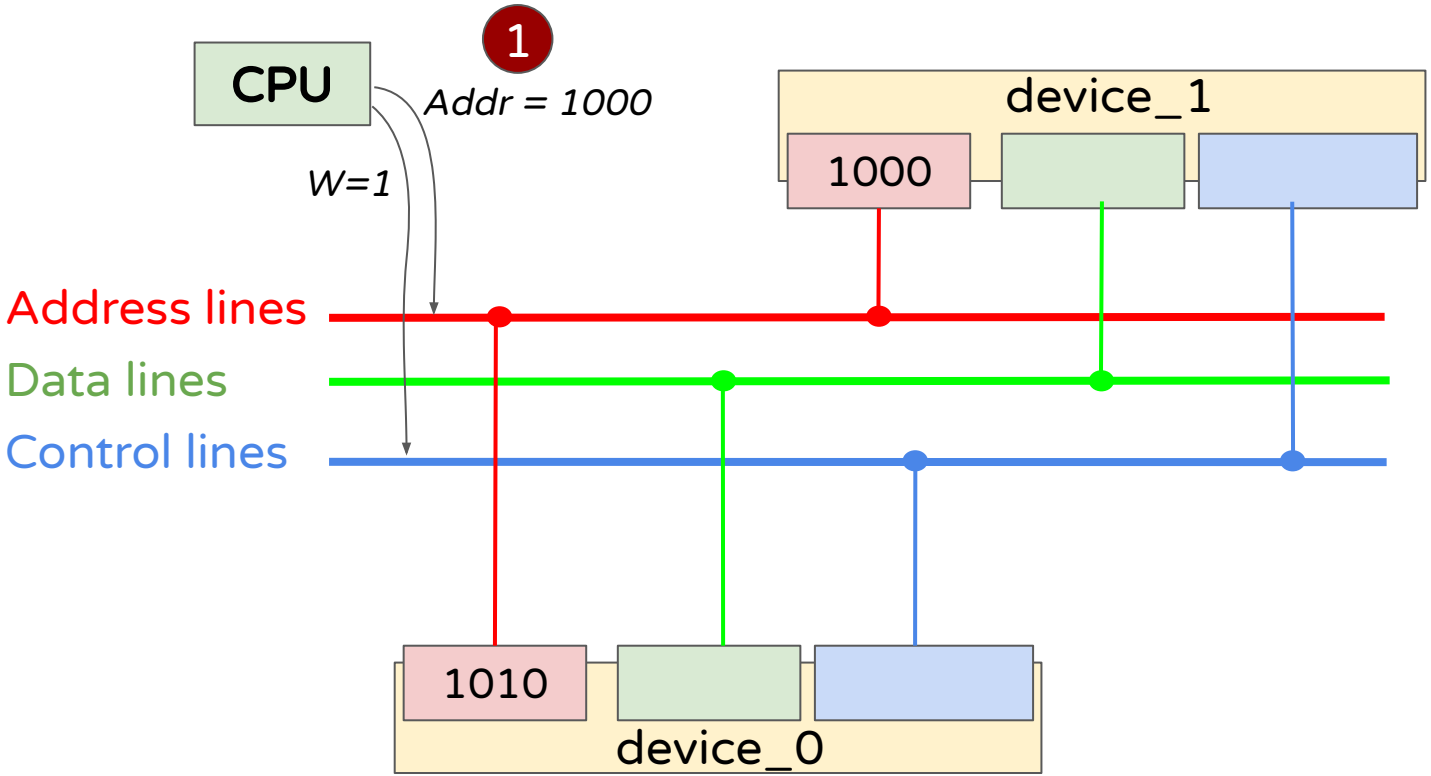
Putting together



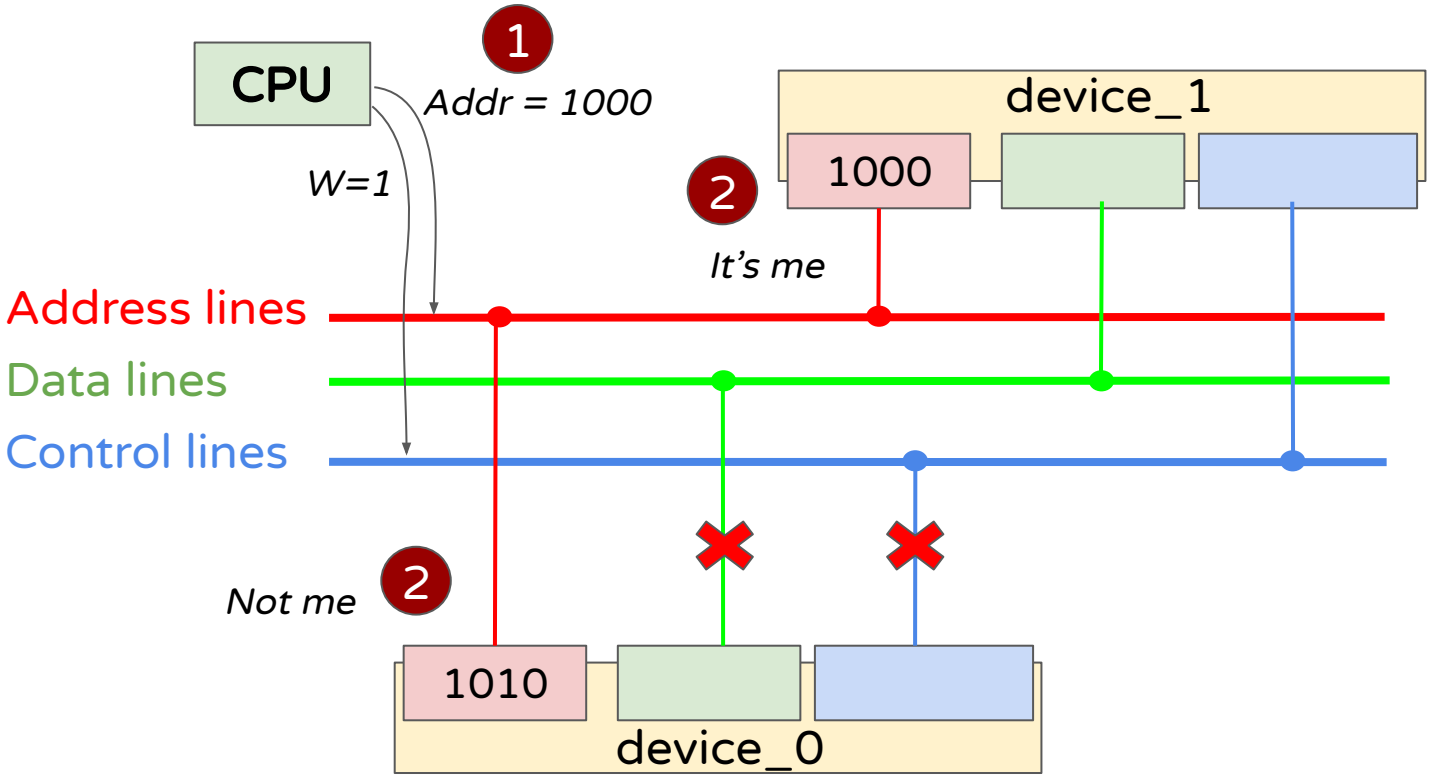
A simple example



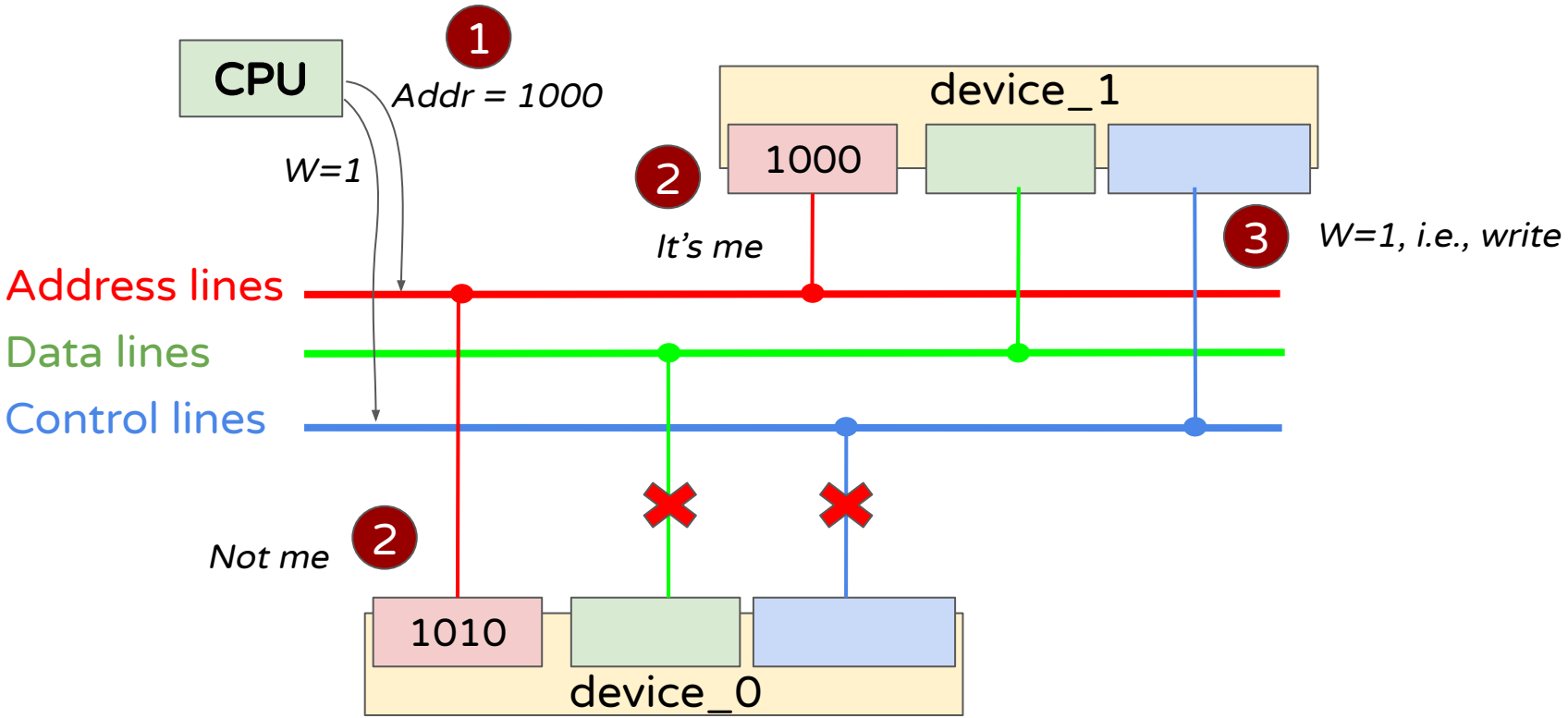
A simple example



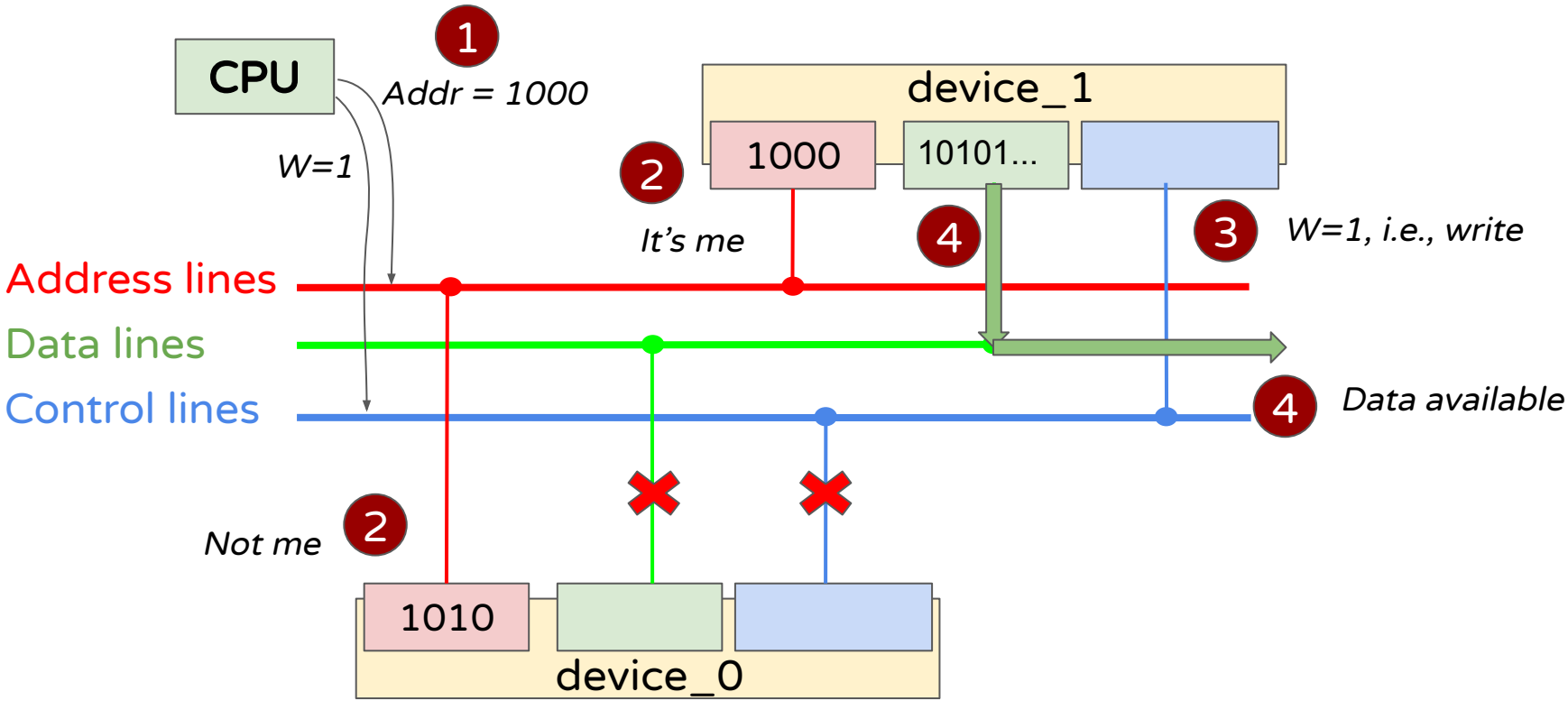
A simple example



A simple example



A simple example



Design choices with a bus

Goals: performance, standardization, flexibility (v1, v2, …), cost, expandability

Design choices:

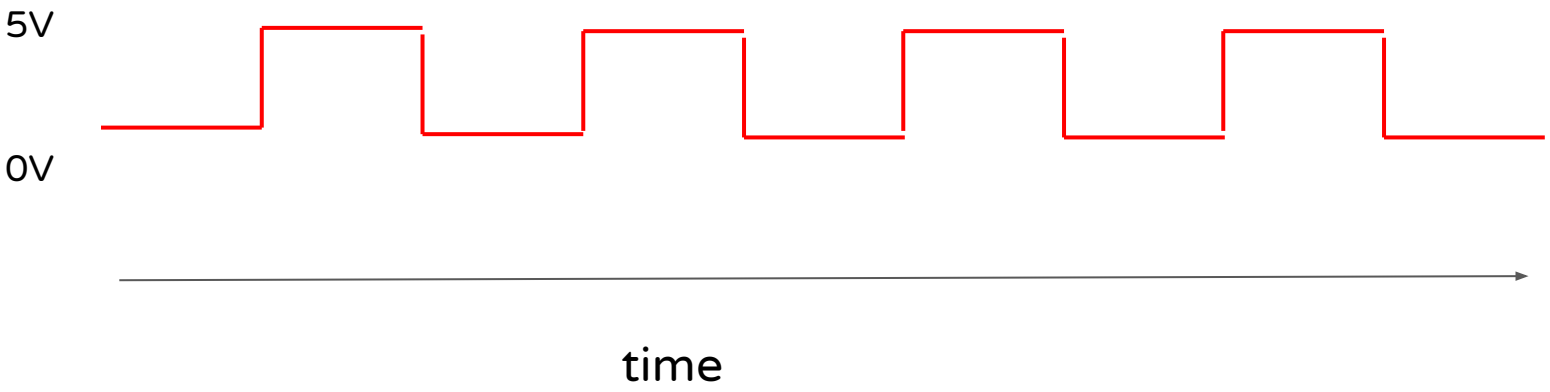
- Clocking: yes (synchronous) or not (asynchronous)
- Bus width in bits : serial or parallel bus
- Point-to-point or multiplexing (shared) bus
- Switching: how/when bus control is acquired and released
- Arbitration: which device gets the bus next

Synchronous bus

All devices derive timing from a control line called the **bus clock**

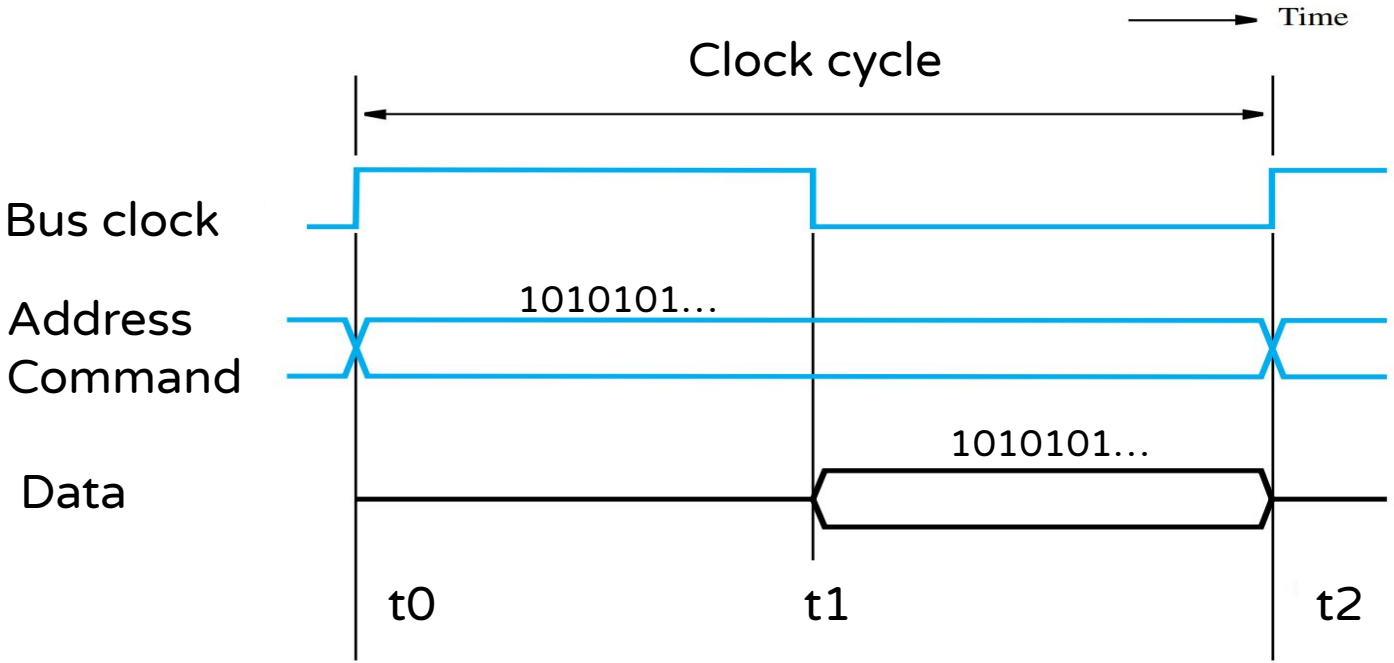
Clock is a special line which has a defined pattern with a rate or frequency

A 1 GHz clock frequency will have 10^9 such cycles per second



Synchronous bus

All devices derive timing from a control line called the **bus clock**



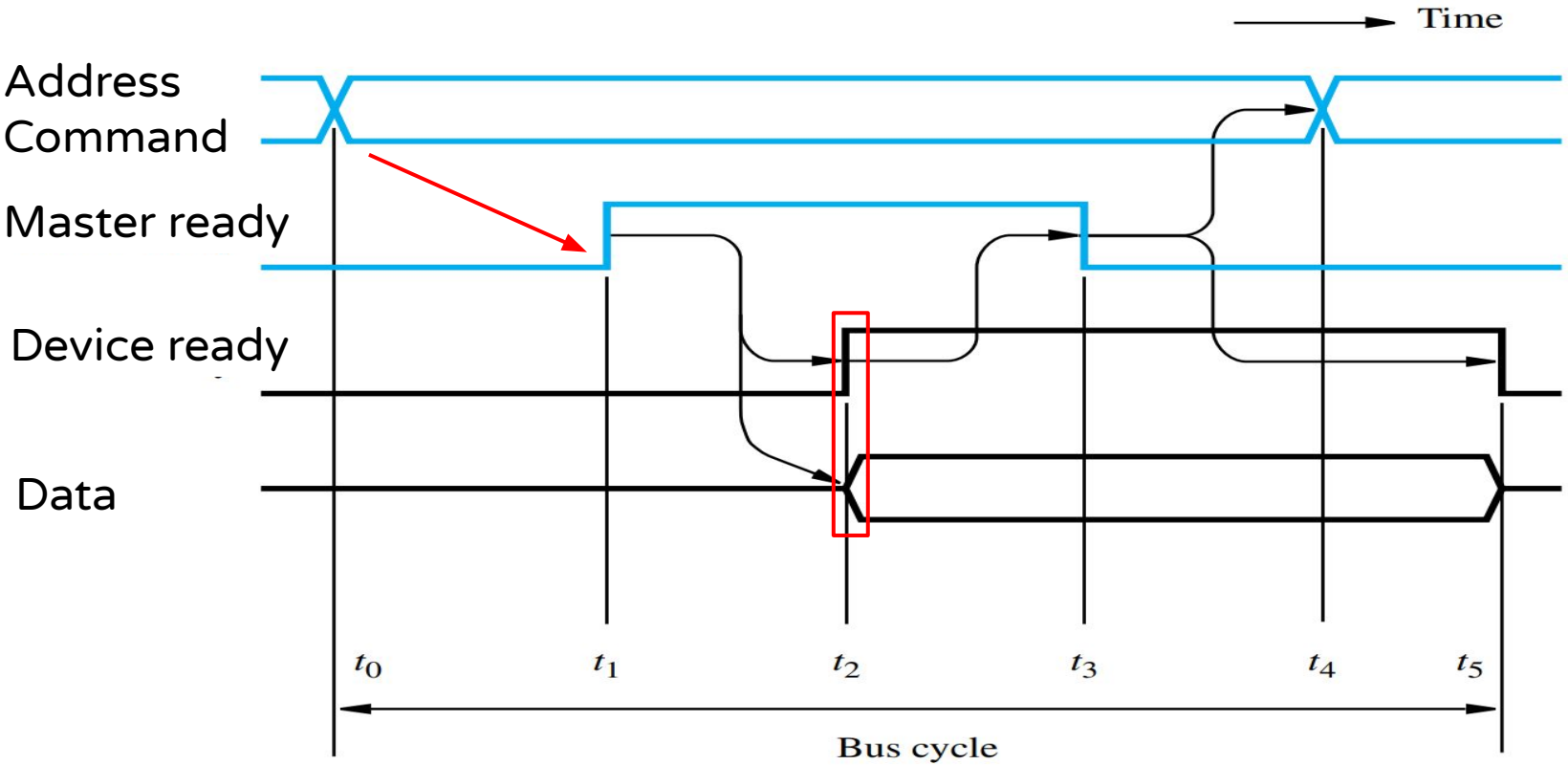
Synchronous bus

What does the clock cycle depend on:

- **Maximum propagation delay**: (t1 - t0) should be longer than the bus length. For example:
  speed of light (~300,000,000 m/s)
  bus distance of 30cm
  Propagation time => 10^{-9} sec, so the clock cannot be more than 1 GHz
- We also need to consider the time it takes for a device to **respond to a signal** (add to the frequency above, and calculate again)
  - So if a device takes 10 nanoseconds to respond, then...
  - ...1 + 10 = 11 nsec, or inversely (maximum) 90.9 MHz clock frequency

Maintaining the clock over a distance at a high-frequency is challenging

Asynchronous transfer

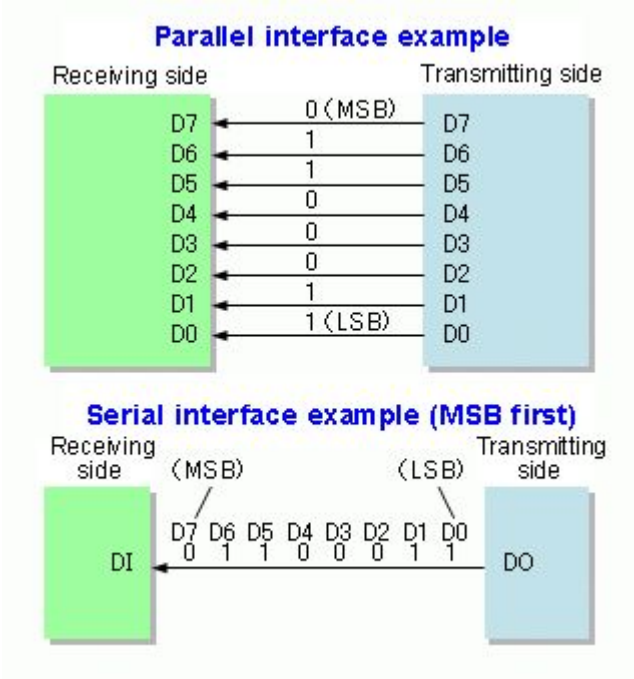


Serial vs. parallel data line design

Parallel buses can transmit more data in a single cycle, but maintaining clocking information and low interference on multiple links over a distance is a challenging task

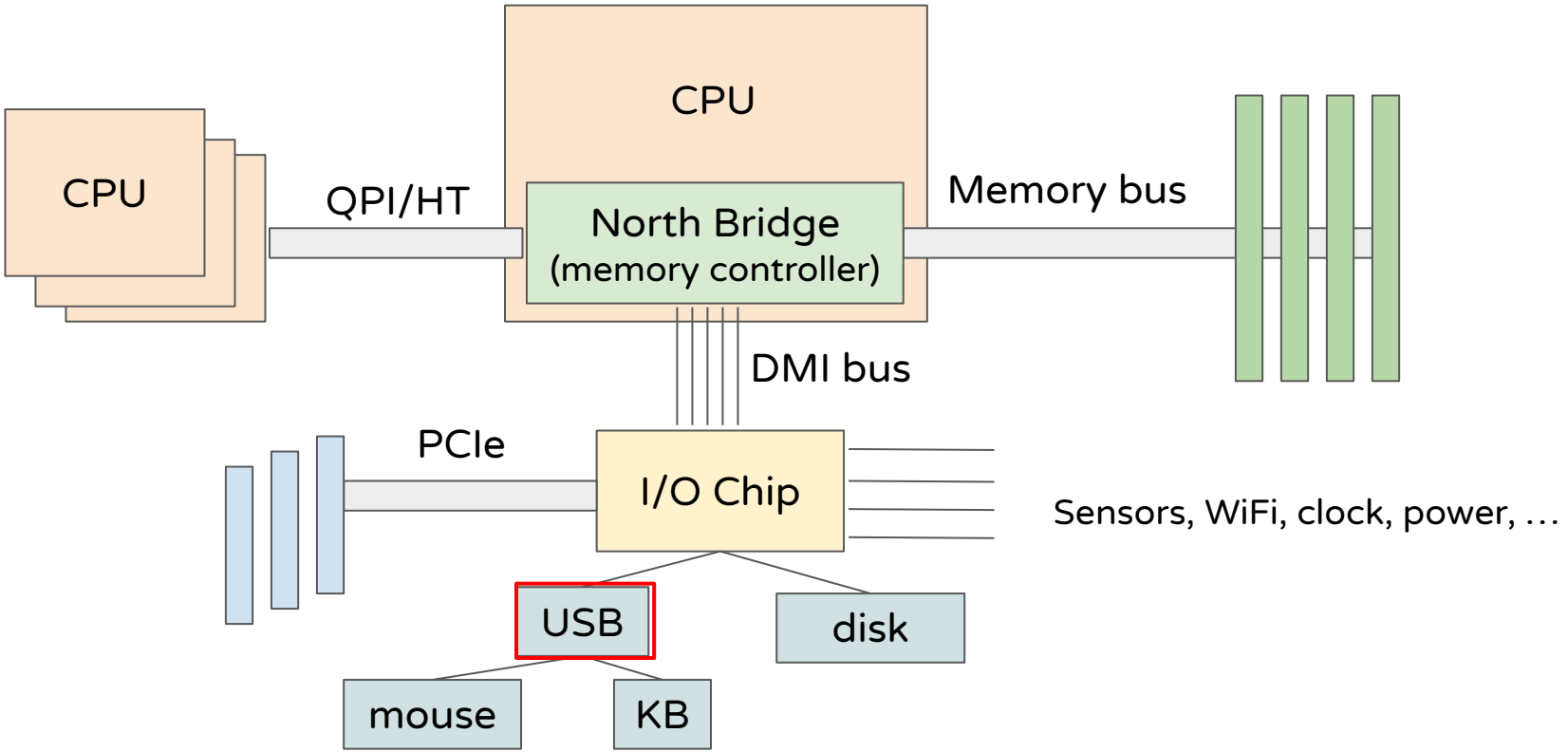
Serial interface

- Simple to implement
- Different encoding (8/10B) can be used to synchronize clocks implicitly
- Preferred design today



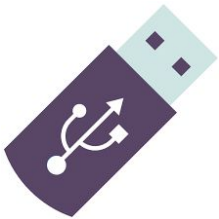
https://upload.wikimedia.org/wikipedia/commons/a/a6/Parallel_and_Serial_Transmission.gif

A more modern setup



Modern buses: USB

Universal Serial Bus



- Simple low-cost, easy to use interconnect
- Wide variety of devices
- Plug-n-play mode

USB 1.0, 2.0, and 3.0 standards

- Backwards compatible
- 3.0 supports up to 5 Gigabits/s data transfer
- Needs a bit of memory and logic in devices

Modern buses: USB

There is a **root complex** and each device has a unique USB address (7bits)

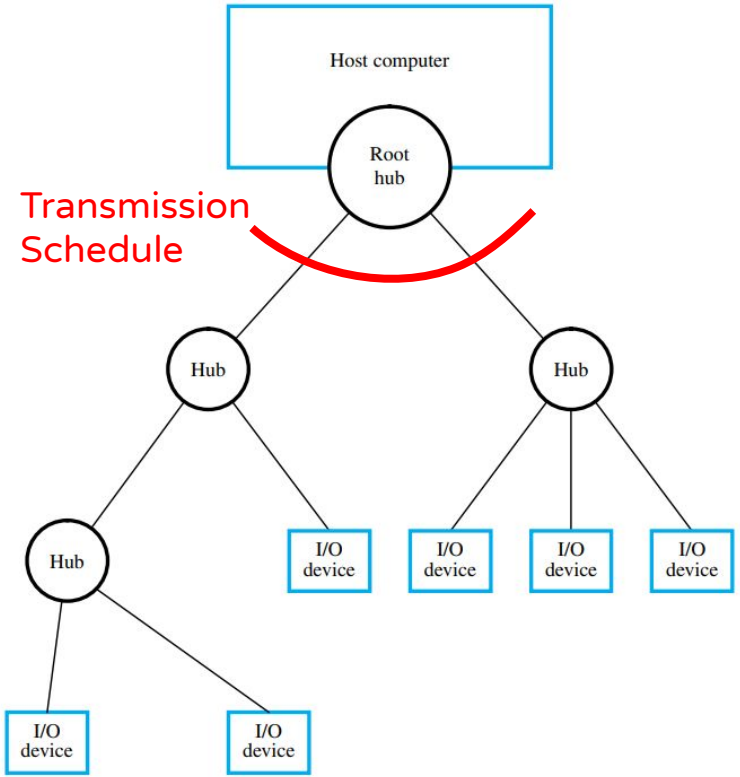
There are three types of traffic schedules

- **Asynchronous**: any time (e.g., keyboard)
- **Isochronous**: regular interval (e.g., sound sampling)
- **Bulk**: mass storage devices (USB sticks)

The USB root complex uses:

- Point-to-point links (not a shared bus)
- Serial transmission with a schedule and device polling
- CPU _only_ sees the USB hub as a single device

USB device tree

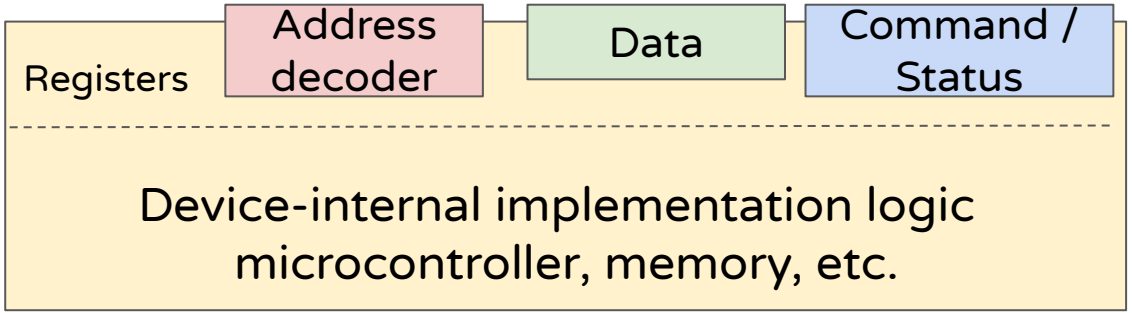



```
atr@atr-XPS-13:~$ lsusb -t
/:  Bus 04.Port 1: Dev 1, Class=root_hub, Driver=xhci_hcd/2p, 10000M
    |__ Port 1: Dev 25, If 0, Class=Hub, Driver=hub/7p, 5000M
        |__ Port 2: Dev 26, If 0, Class=Vendor Specific Class, Driver=r8152, 5000M
/:  Bus 03.Port 1: Dev 1, Class=root_hub, Driver=xhci_hcd/2p, 480M
/:  Bus 02.Port 1: Dev 1, Class=root_hub, Driver=xhci_hcd/6p, 10000M
/:  Bus 01.Port 1: Dev 1, Class=root_hub, Driver=xhci_hcd/12p, 480M
    |__ Port 5: Dev 2, If 1, Class=Video, Driver=uvcvideo, 480M
    |__ Port 5: Dev 2, If 0, Class=Video, Driver=uvcvideo, 480M
    |__ Port 7: Dev 3, If 0, Class=Wireless, Driver=btusb, 12M
    |__ Port 7: Dev 3, If 1, Class=Wireless, Driver=btusb, 12M
    |__ Port 9: Dev 39, If 0, Class=Hub, Driver=hub/7p, 480M
        |__ Port 5: Dev 40, If 3, Class=Audio, Driver=snd-usb-audio, 480M
        |__ Port 5: Dev 40, If 1, Class=Audio, Driver=snd-usb-audio, 480M
        |__ Port 5: Dev 40, If 2, Class=Audio, Driver=snd-usb-audio, 480M
        |__ Port 5: Dev 40, If 0, Class=Audio, Driver=snd-usb-audio, 480M
        |__ Port 6: Dev 41, If 0, Class=Hub, Driver=hub/4p, 480M
            |__ Port 4: Dev 43, If 0, Class=Human Interface Device, Driver=usbhid, 1.5M
            |__ Port 4: Dev 43, If 1, Class=Human Interface Device, Driver=usbhid, 1.5M
        |__ Port 7: Dev 42, If 0, Class=Human Interface Device, Driver=usbhid, 1.5M
atr@atr-XPS-13:~$
```

Communicating With Devices

How does the CPU give commands to I/O devices?
How do I/O devices execute data transfers?
How does CPU know when I/O devices are done?

An abstract device



Status registers: show the current status of the device

Command registers: tell device to perform a certain task

Data registers: get or put data to/from the device

How do you/the CPU program devices?

The CPU executes a series of instructions

There are two strategies:

1. Special CPU instructions to identify I/O operations explicitly
2. Memory-mapped I/O

Both are used, but today memory-mapped I/O are more common.

1. Doing I/O - using special instructions

Using special I/O instructions (also known as ***Port-mapped I/O***)

- Device/registers are enumerated, and given a special port address
- Special instructions for I/O (the bus knows it will go the device). E.g., in/out instructions on x86

```
IN    AL  19H // 8 bits are saved to register AL from the I/O port "19H"  (fixed!)
OUT   DX  EAX // 32 bits are written to port number in DX register (16 bits, variable)
```

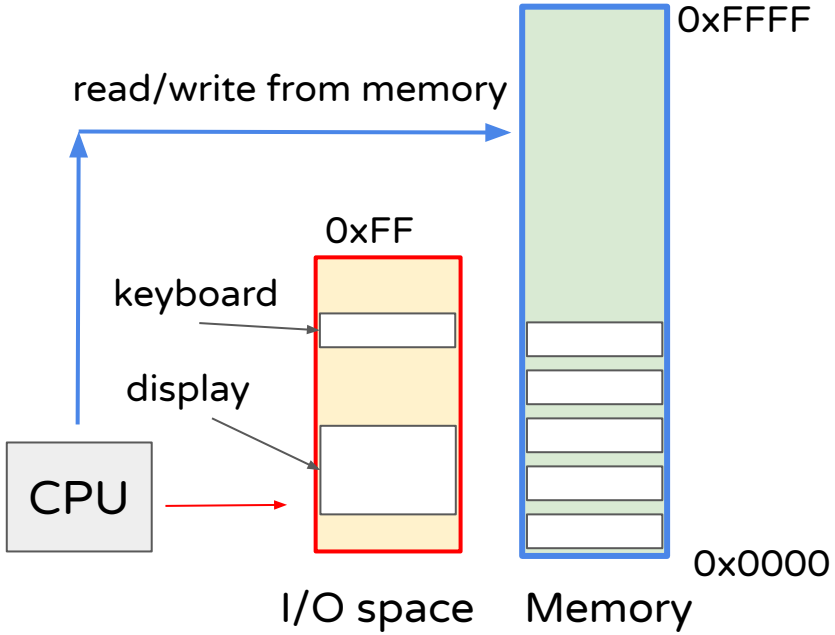
- Simple to implement
- That also means that you have to write it in assembly, compilers cannot generate this automatically

2. Doing I/O - Memory-mapped I/O

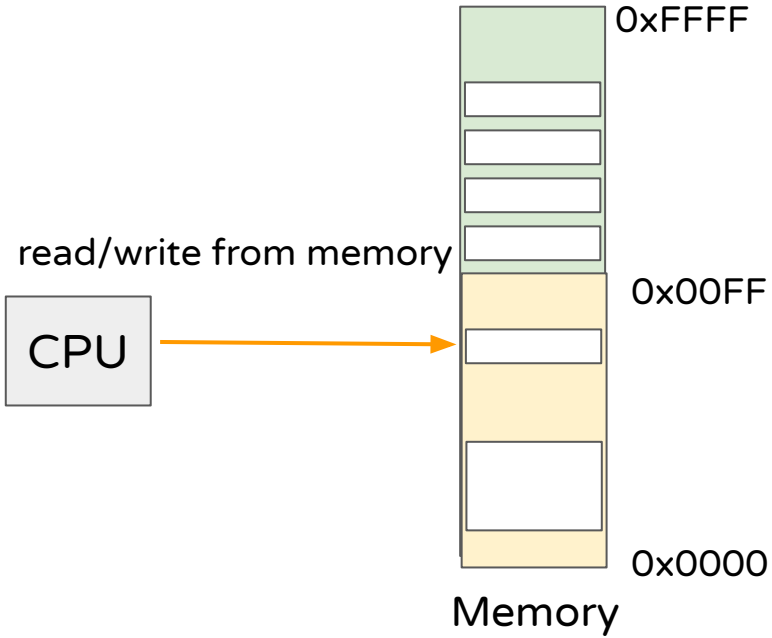
Memory-mapped I/O

- All device interfaces are mapped to a "**memory location**"
- Certain memory address ranges are reserved for this purpose
- From the CPU point of view no difference between accessing memory or a device register
- Just use any instruction that is used to access data
  - movl 0xABCD 0x1234
  - Luckily, a compiler can do this for you!
- Needs a bit of a priori setup

Port-Mapped vs. Memory-Mapped I/O



2 different address spaces



Single address spaces

Example: In Linux (sudo cat /proc/iomem)

Memory mapped addresses for devices

Notice how memory addresses are also there

```
atr@atr-XPS-13:~$ sudo cat /proc/iomem
00000000-00000fff : Reserved
00001000-0009dfff : System RAM
0009e000-0009efff : Reserved
0009f000-0009ffff : System RAM
000a0000-000fffff : Reserved
  000a0000-000bffff : PCI Bus 0000:00
  000c0000-000c3fff : PCI Bus 0000:00
  000c4000-000c7fff : PCI Bus 0000:00
  000c8000-000cbfff : PCI Bus 0000:00
  000cc000-000cffff : PCI Bus 0000:00
  000d0000-000d3fff : PCI Bus 0000:00
  000d4000-000d7fff : PCI Bus 0000:00
  000d8000-000dbfff : PCI Bus 0000:00
  000dc000-000dffff : PCI Bus 0000:00
  000f0000-000fffff : System ROM
00100000-2c8d1fff : System RAM
2c8d2000-2c8d2fff : ACPI Non-volatile Storage
2c8d3000-2c8d3fff : Reserved
2c8d4000-3d1a0fff : System RAM
3d1a1000-3dc0efff : Reserved
3dc0f000-3dc8bfff : ACPI Tables
3dc8c000-3dd56fff : ACPI Non-volatile Storage
  3dce5000-3dce5fff : USBC000:00
3dd57000-3fe22fff : Reserved
3fe23000-3fffefff : Unknown E820 type
3ffff000-3fffffff : System RAM
40000000-47ffffff : Reserved
  40200000-45f7ffff : INT0E0C:00
48000000-48dfffff : System RAM
48e00000-4f7fffff : Reserved
  4b800000-4f7fffff : Graphics Stolen Memory
4f800000-dfffffff : PCI Bus 0000:00
  4f800000-4f800fff : 0000:00:15.0
    4f800000-4f8001ff : lpss_dev
      4f800000-4f8001ff : lpss_dev
    4f800200-4f8002ff : lpss_priv
    4f800800-4f800fff : idma64.0
      4f800800-4f800fff : idma64.0
  4f801000-4f801fff : 0000:00:15.1
    4f801000-4f8011ff : lpss_dev
      4f801000-4f8011ff : lpss_dev
    4f801200-4f8012ff : lpss_priv
    4f801800-4f801fff : idma64.1
      4f801800-4f801fff : idma64.1
  50000000-5fffffff : 0000:00:02.0
  60000000-a9ffffff : PCI Bus 0000:03
    60000000-a9ffffff : PCI Bus 0000:04
```

How does data get transferred?

Option 1: The CPU does it all, easy and simple (assume memory-mapped I/O)

1. while (STATUS == BUSY)
   ; // wait until device is not busy
2. write data to DATA register
3. write command to COMMAND register (the device executes the command - write data)
4. while (STATUS == BUSY)
   ; // wait until device is done with your request

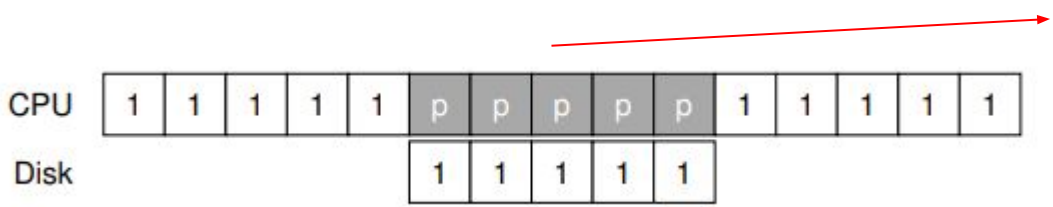
How does data get transferred?

Option 1: The CPU does it all, easy and simple (assume memory-mapped I/O)

1. while (STATUS == BUSY)
   ; // wait until device is not busy
2. write data to DATA register
3. write command to COMMAND register (the device executes the command - write data)
4. while (STATUS == BUSY)
   ; // wait until device is done with your request

How does data get transferred?

Option 1: The CPU does it all, easy and simple



Repeated checking of a device status by a CPU in a loop is called "**polling**"

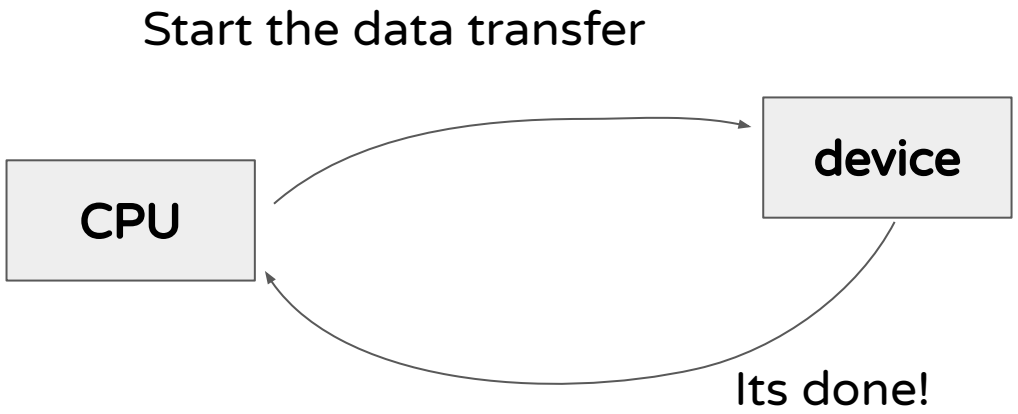
The CPU is typically very faster than a device (disks ~ms, network ~usecs, CPU ~ns)

Waste of CPU resources

Can we do better?

Interrupts: let the device tell the CPU

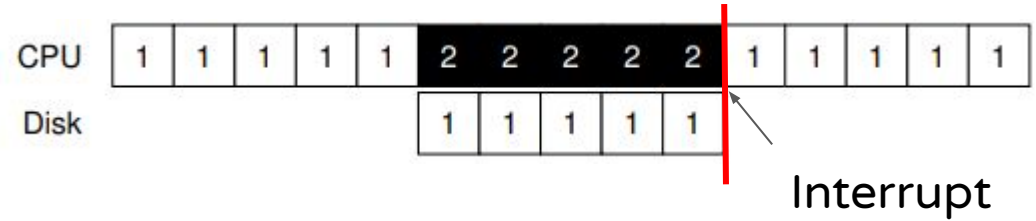
Option 2: The device can generate an interrupt when I/O is finished



The CPU has a special circuit and interface to raise interrupts

Interrupts: let the device tell the CPU

Option 2: The device can generate an interrupt when I/O is finished

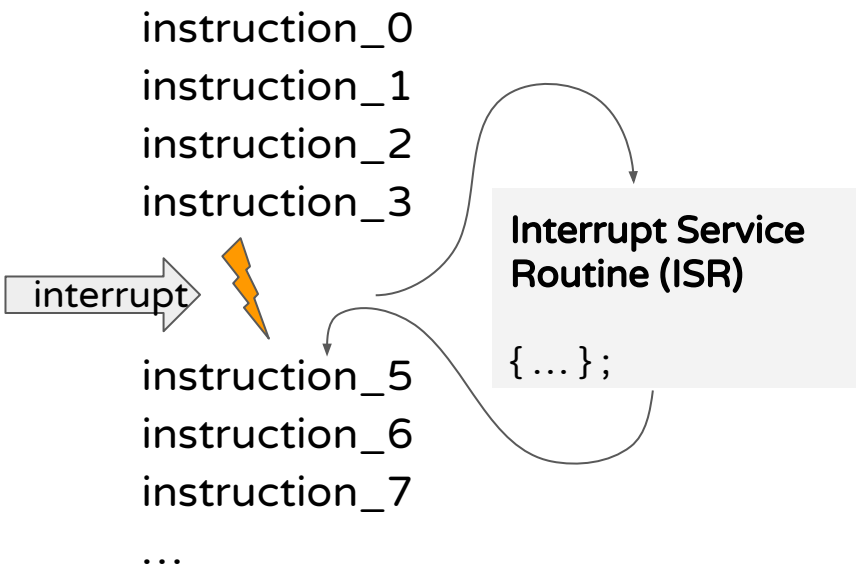


Interrupt is a special case or exception, where the CPU is notified to:

- Please stop whatever you were doing
- Check what has happened around

Exception is a broader term, which includes software exceptions as well, e.g. segmentation fault.

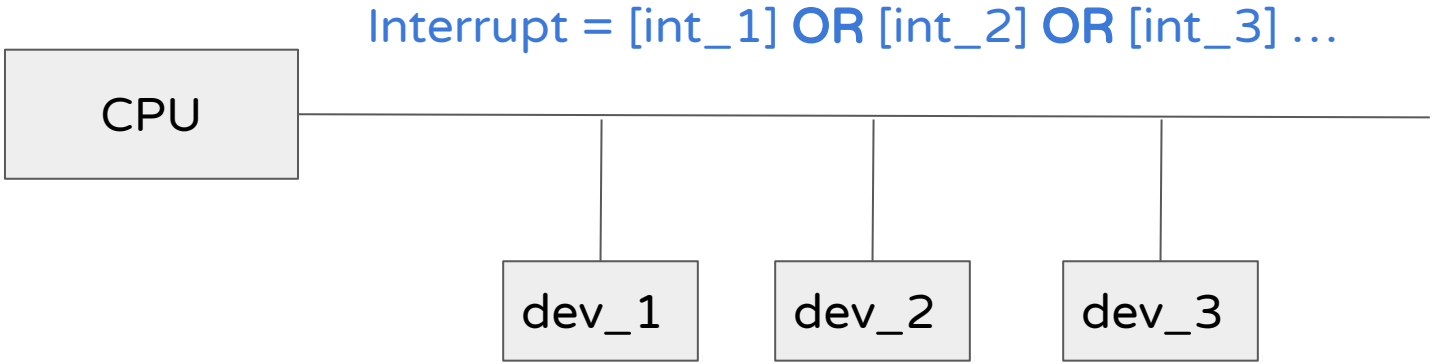
What happens when there is an interrupt



1. Device raises interrupt request
2. Processor interrupts program in execution
3. Interrupts are disabled
4. Device is informed of acceptance and, as a consequence, lowers interrupt
5. Interrupt is handled by **service routine**
6. Interrupts are enabled
7. Execution of interrupted program is resumed

Logically ISR are the same as calling a function call in the code, but there are subtle differences

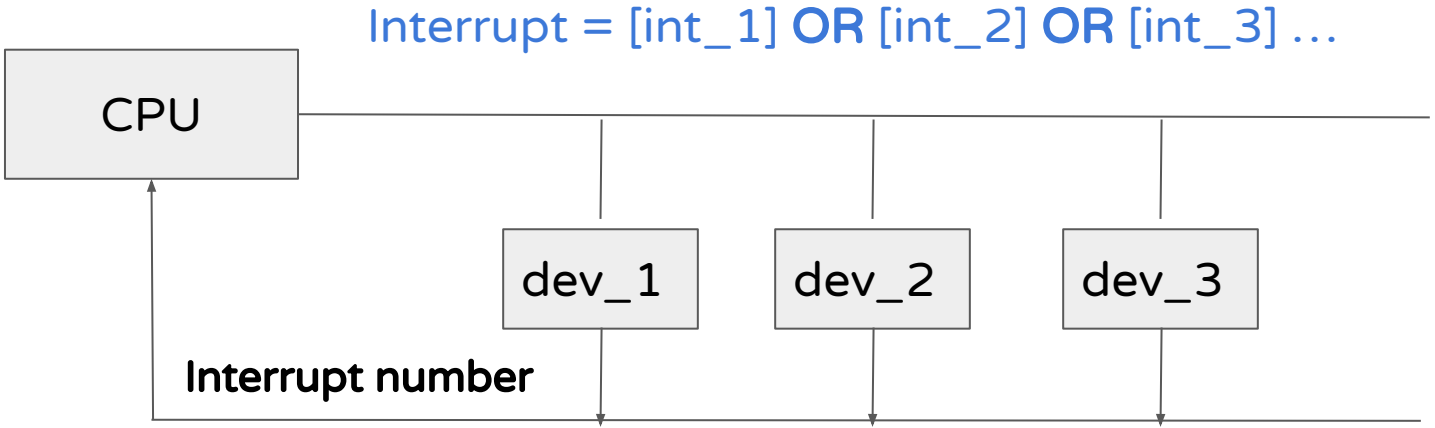
Who called an interrupt?



Check each device one by one

- Check if their status register has interrupt (IRQ) bit set
- Might have multiple devices that need servicing

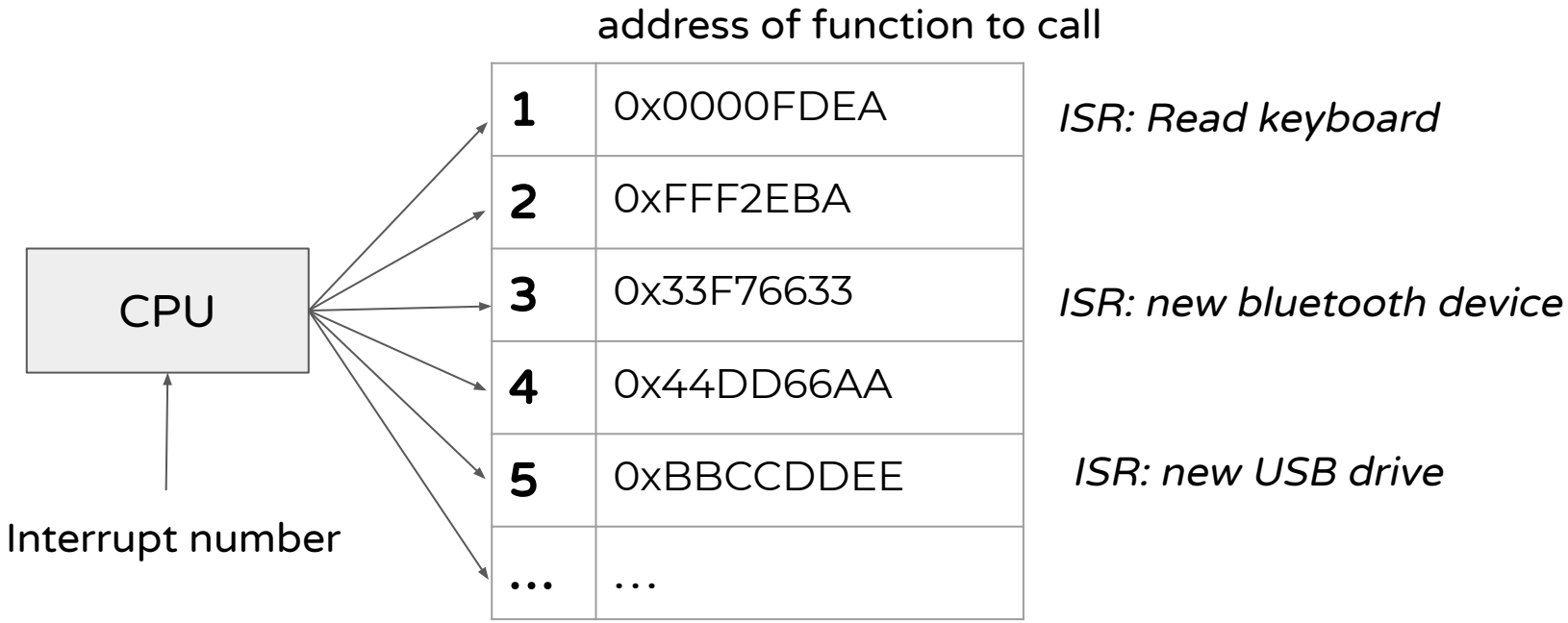
Vectored interrupts : Interrupts with numbers



Each device is assigned a unique interrupt number

With interrupt, the number is put on a data line

Invoking ISRs with vectored interrupts



Interrupt vector table - IVT (each CPU has one!)

Example: IRQs in Linux (cat /proc/interrupts)

```
atr@atr-XPS-13:~$ cat /proc/interrupts
           CPU0       CPU1       CPU2       CPU3       CPU4       CPU5       CPU6       CPU7
  0:          8          0          0          0          0          0          0          0  IR-IO-APIC    2-edge      timer
  1:          0          0          0          0          0        103          0          0  IR-IO-APIC    1-edge      i8042
  8:          0          0          0          0          0          0          1          0  IR-IO-APIC    8-edge      rtc0
  9:          0      53422          0          0          0          0          0          0  IR-IO-APIC    9-fasteoi   acpi
 12:          0          0          0          0        239          0          0          0  IR-IO-APIC   12-edge      i8042
 14:          0          0          0          0          0          0          0          0  IR-IO-APIC   14-fasteoi   INT34BB:00
 16:          0          0          0          0        649          0          0          0  IR-IO-APIC   16-fasteoi   idma64.0, i2c_designware.0
 17:          0          0          0          0          0     171114          0        862  IR-IO-APIC   17-fasteoi   idma64.1, i2c_designware.1
 39:          0          0          0          0          0          0        809          0  IR-IO-APIC   39-fasteoi   ELAN2934:00
 51:       4667          0          0          0          0          0          0          0  IR-IO-APIC   51-fasteoi   DELL08AF:00
120:          0          0          0          0          0          0          0          0  DMAR-MSI    0-edge      dmar0
121:          0          0          0          0          0          0          0          0  DMAR-MSI    1-edge      dmar1
122:          0          0          0          0          0          0          0          0  IR-PCI-MSI 458752-edge      PCIe PME, pcie-dpc
123:          0          0          0          0          0          0          0          0  IR-PCI-MSI 471040-edge      PCIe PME, pcie-dpc
124:          0          0          0          0          0          2          0          0  IR-PCI-MSI 475136-edge      PCIe PME, pcie-dpc, pciehp
125:          0          0          0          0          0          0          0          0  IR-PCI-MSI 483328-edge      PCIe PME, pcie-dpc
126:          0          0          0          0          0          0          0          0  IR-PCI-MSI 2113536-edge      pciehp
127:          0          0          0          0          0          0          0          0  IR-PCI-MSI 2162688-edge      pciehp
128:      52160          0          0          0          0          0          0          0  IR-PCI-MSI 57671680-edge      nvme0q0, nvme0q1
129:       2874       1844     159921      14248          0          0          0          0  IR-PCI-MSI 327680-edge      xhci_hcd
130:          0          0          0          0     401273          0          0          0  IR-PCI-MSI 29884416-edge      xhci_hcd
131:          0      45210          0          0          0          0          0          0  IR-PCI-MSI 57671681-edge      nvme0q2
132:          0          0      34524          0          0          0          0          0  IR-PCI-MSI 57671682-edge      nvme0q3
133:          0          0          0      40579          0          0          0          0  IR-PCI-MSI 57671683-edge      nvme0q4
134:          0          0          0          0      32822          0          0          0  IR-PCI-MSI 57671684-edge      nvme0q5
135:          0          0          0          0          0      52326          0          0  IR-PCI-MSI 57671685-edge      nvme0q6
136:          0          0          0          0          0          0      42306          0  IR-PCI-MSI 57671686-edge      nvme0q7
137:          0          0          0          0          0          0          0      34522  IR-PCI-MSI 57671687-edge      nvme0q8
138:          0          0          0          0          0          0          0         66  IR-PCI-MSI 524288-edge      rtsx_pci
139:          0        398          0          0          0          0          0          0  IR-PCI-MSI 2621440-edge      thunderbolt
140:          0          0        398          0          0          0          0          0  IR-PCI-MSI 2621441-edge      thunderbolt
155:          0          0          0       2827          0    1148333      13794          0  IR-PCI-MSI 32768-edge      i915
156:          0          0          0          0     126790       2792       3800     146655  IR-PCI-MSI 1048576-edge      ath10k_pci
157:          0          0          0         34          0          0          0          0  IR-PCI-MSI 360448-edge      mei_me
158:          0          0          0        877          0          0          0          0  IR-PCI-MSI 514048-edge      snd_hda_intel:card0
```

Example: IRQ in Linux

Which CPU?

Which device?

IRQ numbers

How many

```
atr@atr-XPS-13:~$ cat /proc/interrupts
           CPU0       CPU1       CPU2       CPU3       CPU4       CPU5       CPU6       CPU7
  0:          8          0          0          0          0          0          0          0   IR-IO-APIC    2-edge      timer
  1:          0          0          0          0          0        103          0          0   IR-IO-APIC    1-edge      i8042
  8:          0          0          0          0          0          0          1          0   IR-IO-APIC    8-edge      rtc0
  9:          0      53422          0          0          0          0          0          0   IR-IO-APIC    9-fasteoi   acpi
 12:          0          0          0          0        239          0          0          0   IR-IO-APIC   12-edge      i8042
 14:          0          0          0          0          0          0          0          0   IR-IO-APIC   14-fasteoi   INT34BB:00
 16:          0          0          0          0        649          0          0          0   IR-IO-APIC   16-fasteoi   idma64.0, i2c_designware.0
 17:          0          0          0          0          0     171114          0        862   IR-IO-APIC   17-fasteoi   idma64.1, i2c_designware.1
 39:          0          0          0          0          0          0        809          0   IR-IO-APIC   39-fasteoi   ELAN2934:00
 51:       4667          0          0          0          0          0          0          0   IR-IO-APIC   51-fasteoi   DELL08AF:00
120:          0          0          0          0          0          0          0          0   DMAR-MSI    0-edge      dmar0
121:          0          0          0          0          0          0          0          0   DMAR-MSI    1-edge      dmar1
122:          0          0          0          0          0          0          0          0   IR-PCI-MSI 458752-edge      PCIe PME, pcie-dpc
123:          0          0          0          0          0          0          0          0   IR-PCI-MSI 471040-edge      PCIe PME, pcie-dpc
124:          0          0          0          0          0          2          0          0   IR-PCI-MSI 475136-edge      PCIe PME, pcie-dpc, pciehp
125:          0          0          0          0          0          0          0          0   IR-PCI-MSI 483328-edge      PCIe PME, pcie-dpc
126:          0          0          0          0          0          0          0          0   IR-PCI-MSI 2113536-edge      pciehp
127:          0          0          0          0          0          0          0          0   IR-PCI-MSI 2162688-edge      pciehp
128:      52160          0          0          0          0          0          0          0   IR-PCI-MSI 57671680-edge      nvme0q0, nvme0q1
129:       2874       1844     159921      14248          0          0          0          0   IR-PCI-MSI 327680-edge      xhci_hcd
130:          0          0          0          0     401273          0          0          0   IR-PCI-MSI 29884416-edge      xhci_hcd
131:          0      45210          0          0          0          0          0          0   IR-PCI-MSI 57671681-edge      nvme0q2
132:          0          0      34524          0          0          0          0          0   IR-PCI-MSI 57671682-edge      nvme0q3
133:          0          0          0      40579          0          0          0          0   IR-PCI-MSI 57671683-edge      nvme0q4
134:          0          0          0          0      32822          0          0          0   IR-PCI-MSI 57671684-edge      nvme0q5
135:          0          0          0          0          0      52326          0          0   IR-PCI-MSI 57671685-edge      nvme0q6
136:          0          0          0          0          0          0      42306          0   IR-PCI-MSI 57671686-edge      nvme0q7
137:          0          0          0          0          0          0          0      34522   IR-PCI-MSI 57671687-edge      nvme0q8
138:          0          0          0          0          0          0          0         66   IR-PCI-MSI 524288-edge      rtsx_pci
139:          0        398          0          0          0          0          0          0   IR-PCI-MSI 2621440-edge      thunderbolt
140:          0          0        398          0          0          0          0          0   IR-PCI-MSI 2621441-edge      thunderbolt
155:          0          0          0       2827          0    1148333      13794          0   IR-PCI-MSI 32768-edge      i915
156:          0          0          0          0     126790       2792       3800     146655   IR-PCI-MSI 1048576-edge      ath10k_pci
157:          0          0          0         34          0          0          0          0   IR-PCI-MSI 360448-edge      mei_me
158:          0          0          0        877          0          0          0          0   IR-PCI-MSI 514048-edge      snd_hda_intel:card0
```

Interrupt vs. Polling

Interrupt breaks the current program flow, and jumps to the ISR execution (this jumping takes time!)

Interrupts are good for *slow* devices: when the ISR setup time is less than the device I/O servicing time . Otherwise, a very high speed device can freeze a system with interrupts. This is called **Interrupt Livelock or Interrupt Storm**

Polling is good for high-speed devices, which can finish I/O at a similar pace as the CPU. No need to generate interrupts

A hybrid strategy: start with interrupt, then switch to polling based on the load and speed of the device

How does data get transferred?

Who does this?

1. while (STATUS == BUSY)
   ; // wait until device is not busy
2. write data to DATA register
3. write command to COMMAND register (the device executes the command - write data)
4. while (STATUS == BUSY)
   ; // wait until device is done with your request

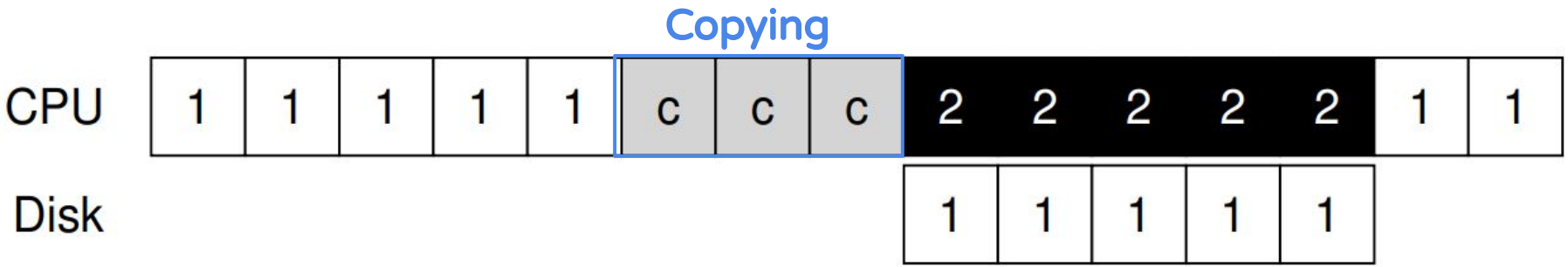
Polling and interrupts

Freeing the CPU from copying data

The CPU is faster than the I/O devices

CPU copying data is also very inefficient

But if there is a data copy engine that can transfer data between devices and memory, how will it look?



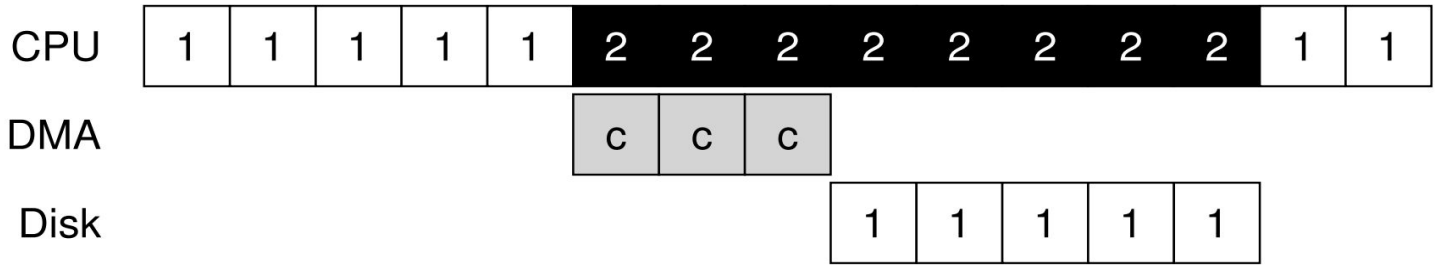
Direct Memory Access (DMA)

Modern high-speed bulk transfer devices (storage, network) have a DMA engine in them

CPU programs the DMA engine

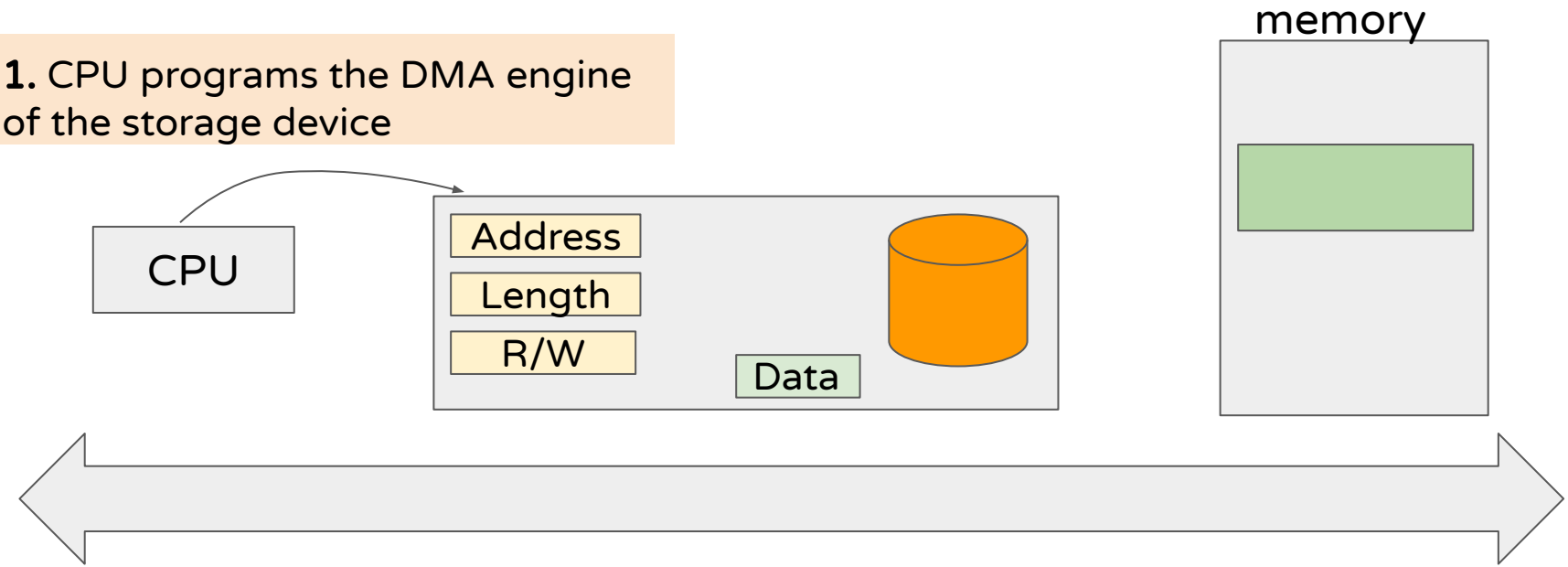
DMA engine does the data transfer

DMA engine either (1) generates an interrupt; (2) CPU can poll and check

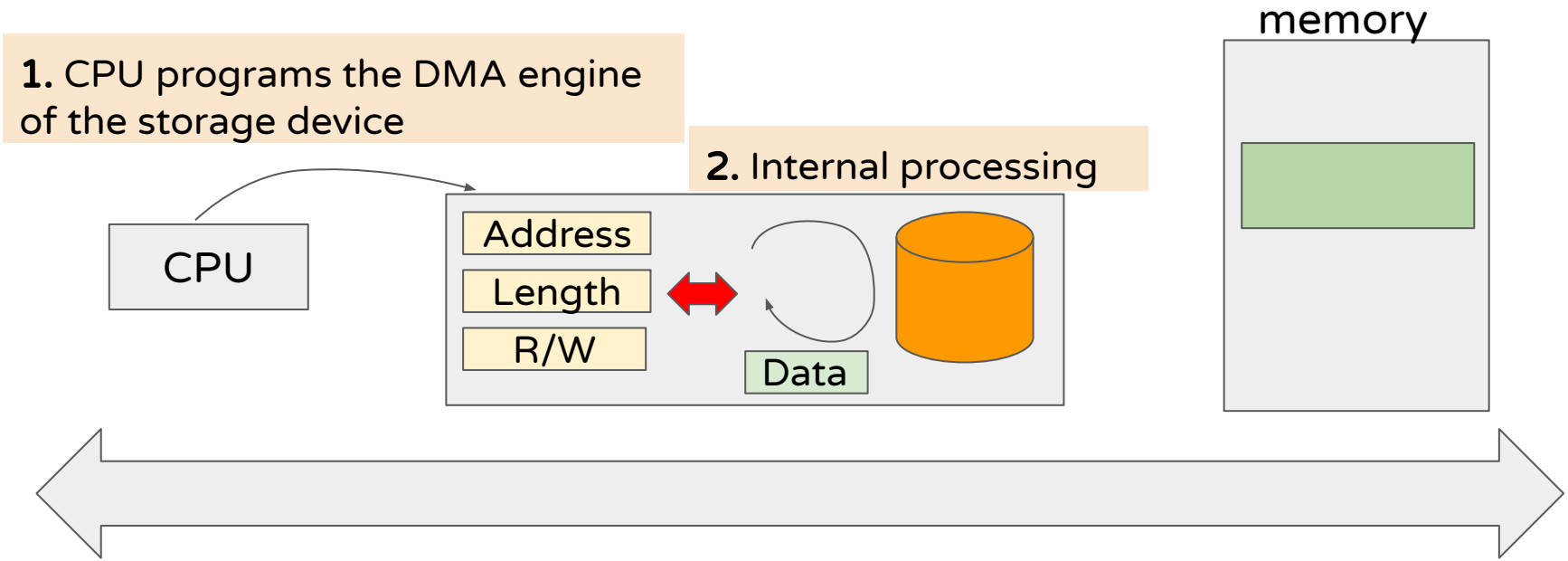


DMA architectural view

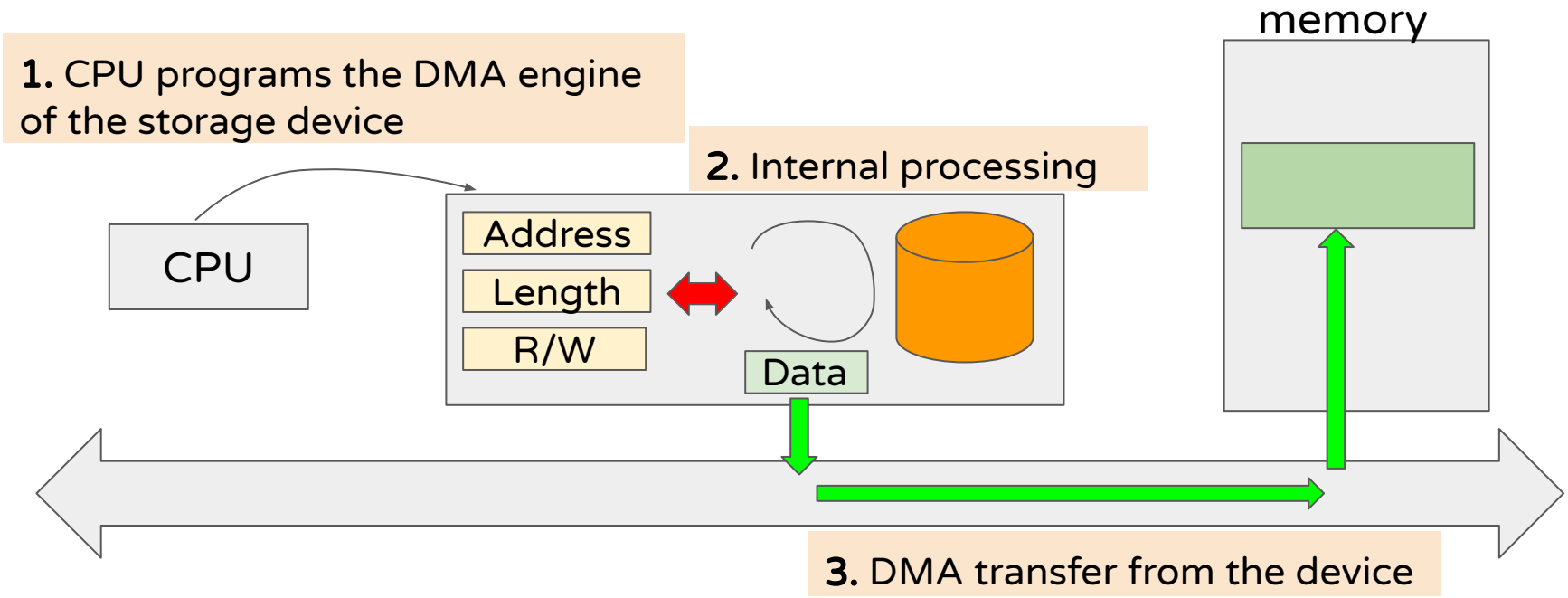
1. CPU programs the DMA engine of the storage device



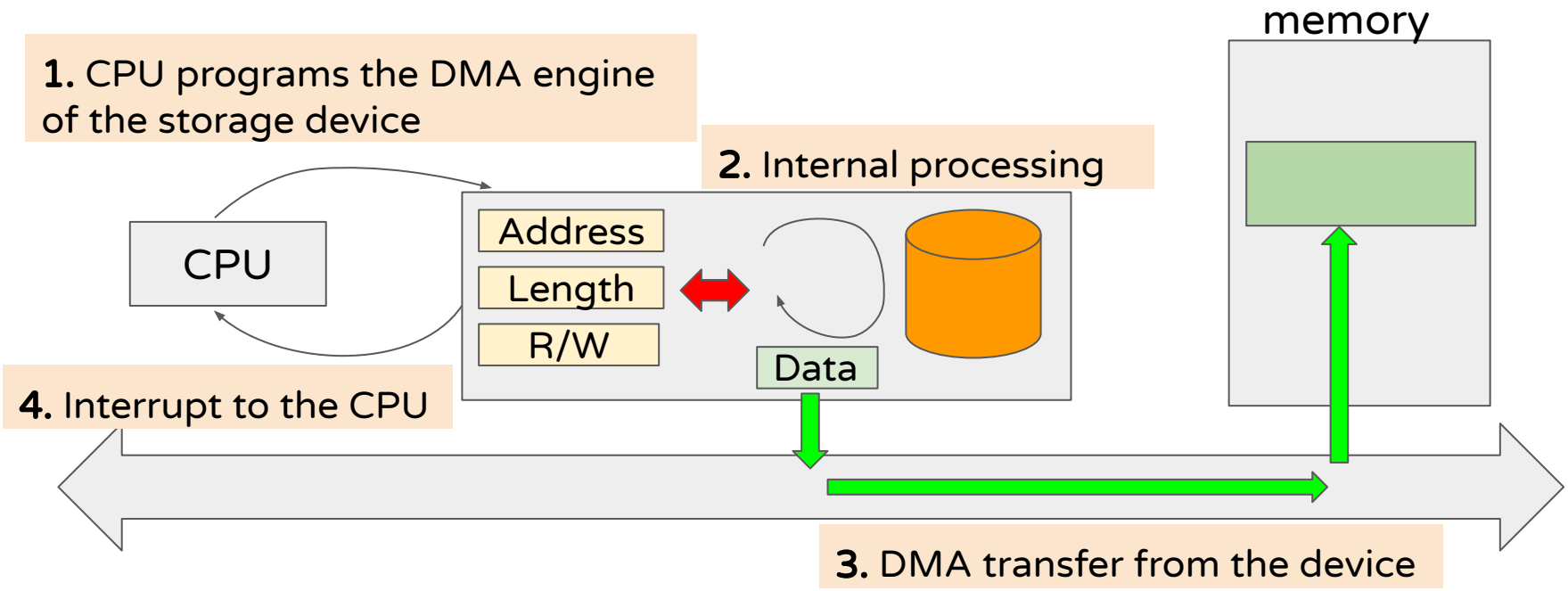
DMA architectural view



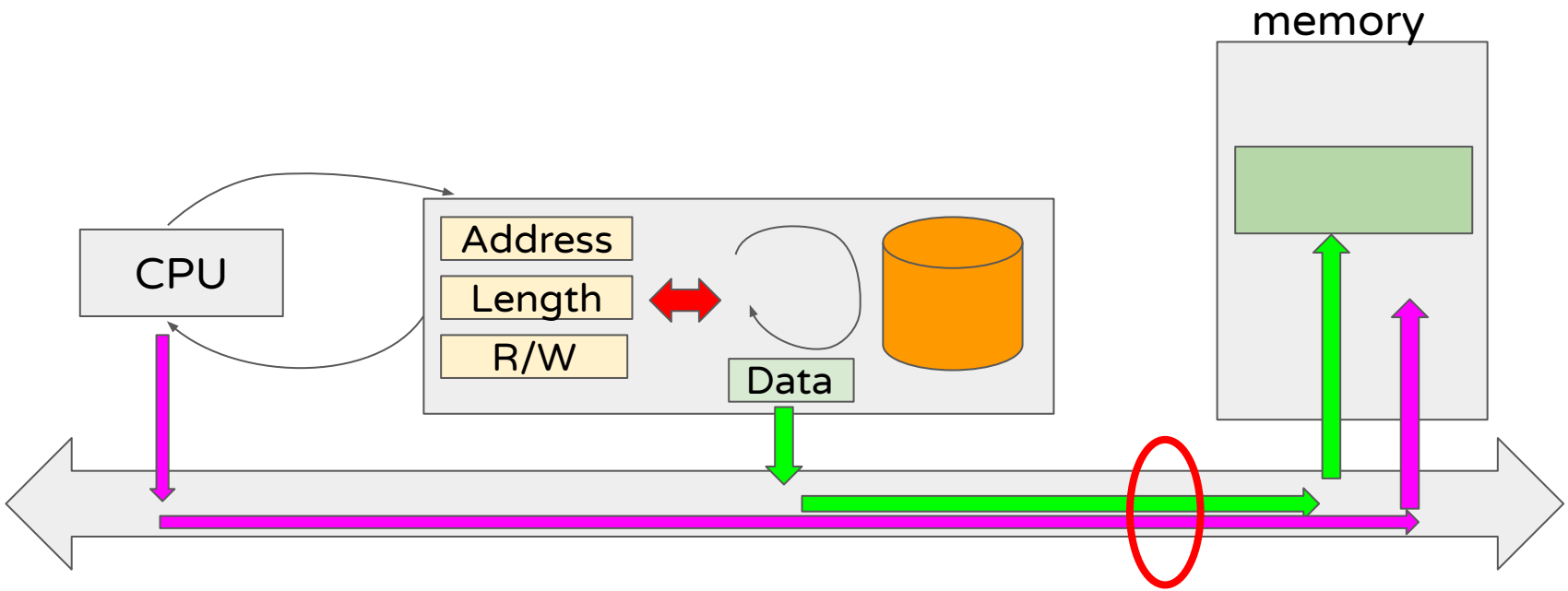
DMA architectural view



DMA architectural view

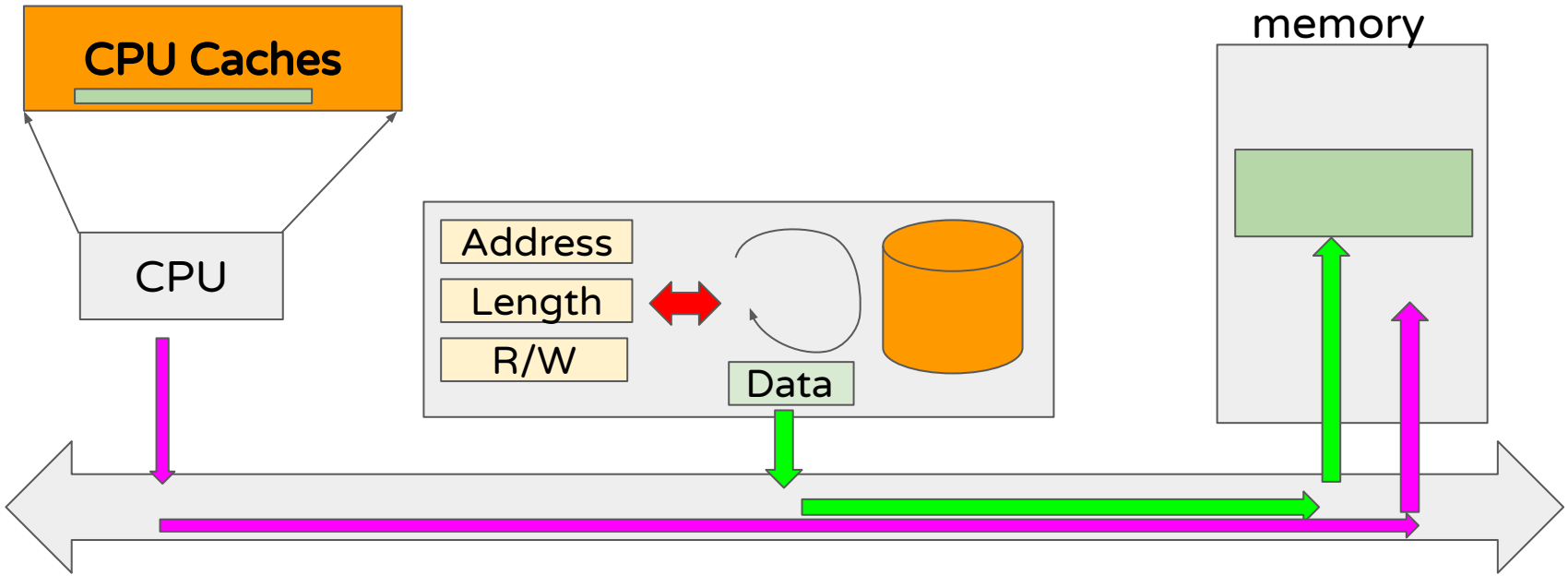


DMA architectural view



Competition on the memory bus between the CPU and the DMA engine can lead to slow CPU memory access. Thus, the DMA engine steals CPU's memory cycles - also called **Cycle Stealing**

DMA architectural view



Question: What happens if the CPU has some data cached in CPU caches?

DMA recap

Extra intelligence in devices (there are also standalone chips) to access memory

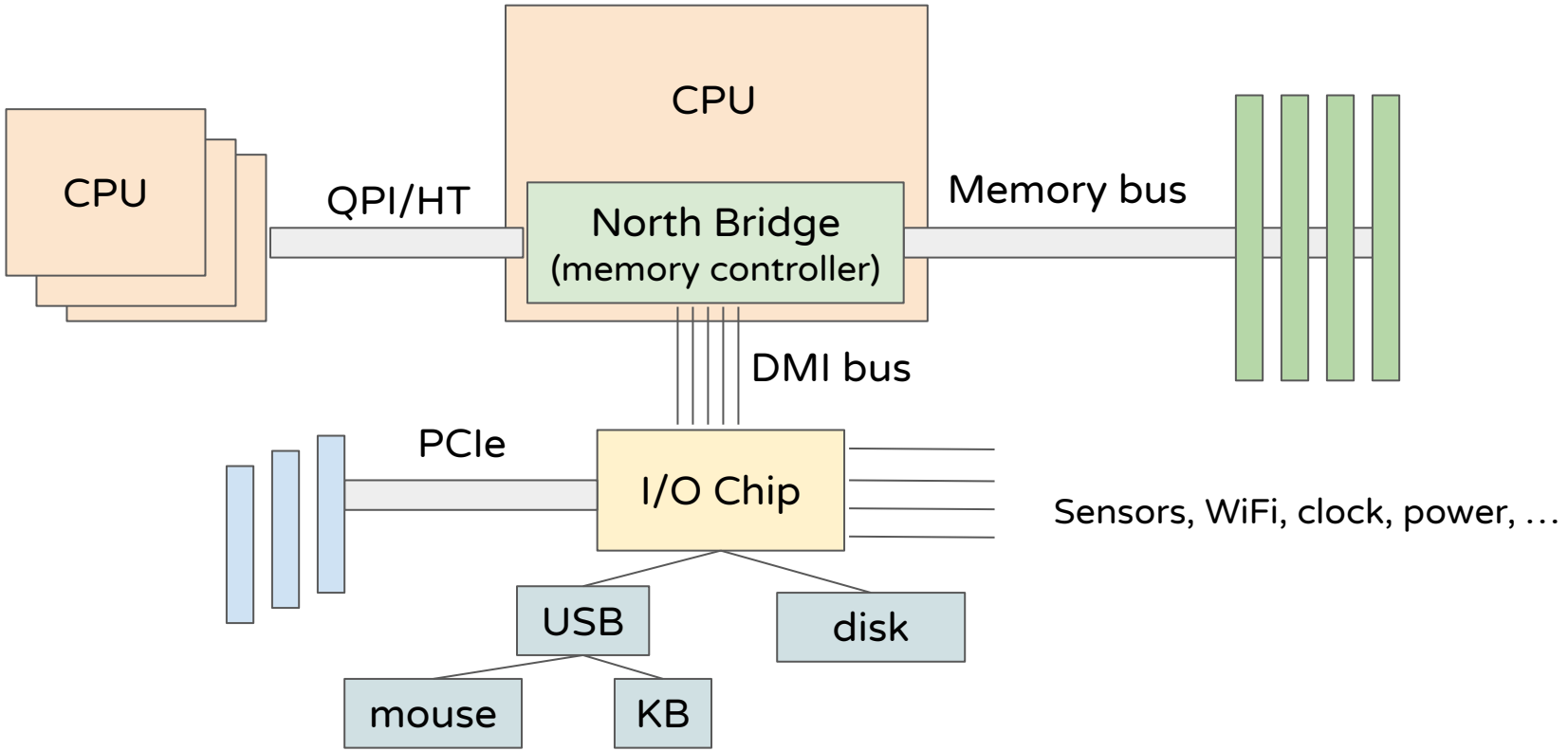
Frees the CPU from doing bulk data transfers from devices to memory

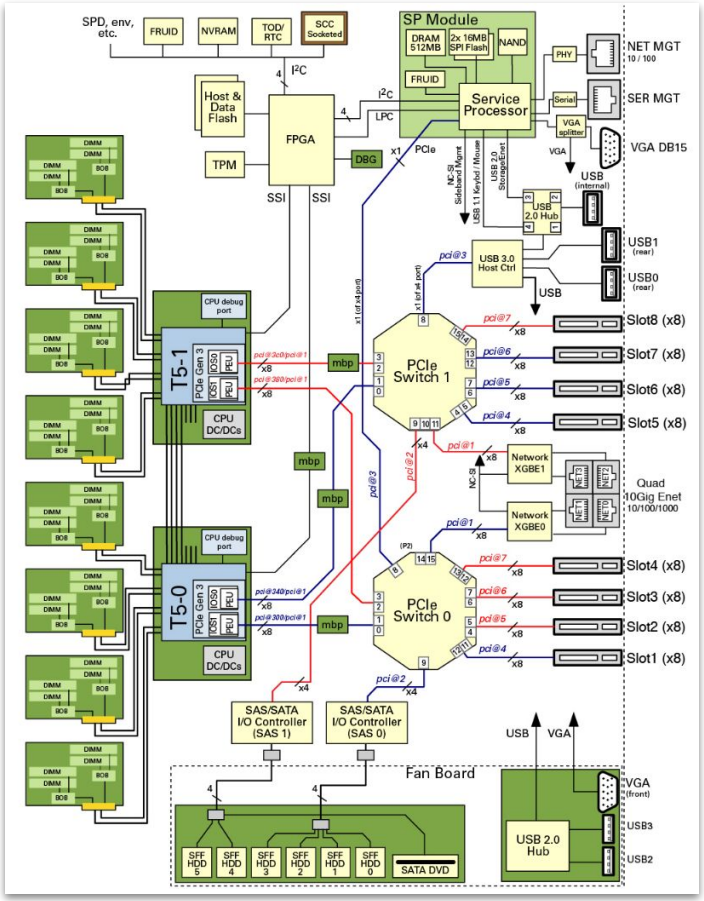
Delivers very high bulk data performance

Can interfere with the CPU memory access (but with modern systems with very high memory bandwidth, it is less of an issue)

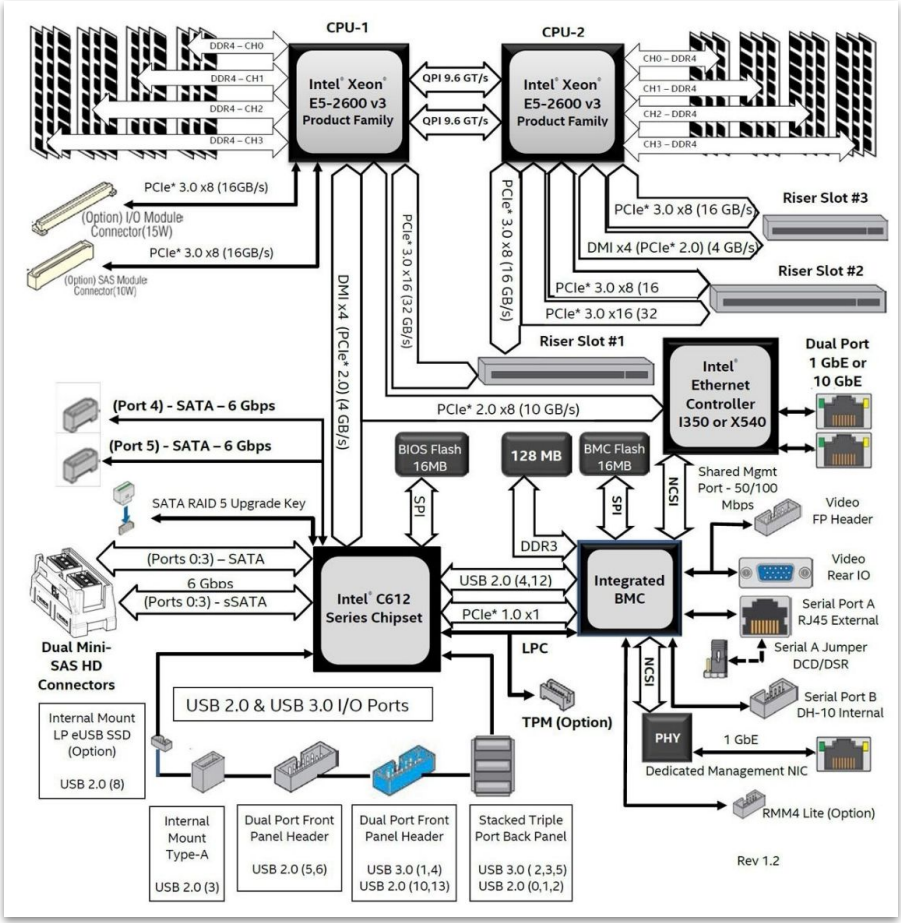
System implementation can choose to keep caches coherent (x86) or not (ARM)

How modern is this **modern** setup?



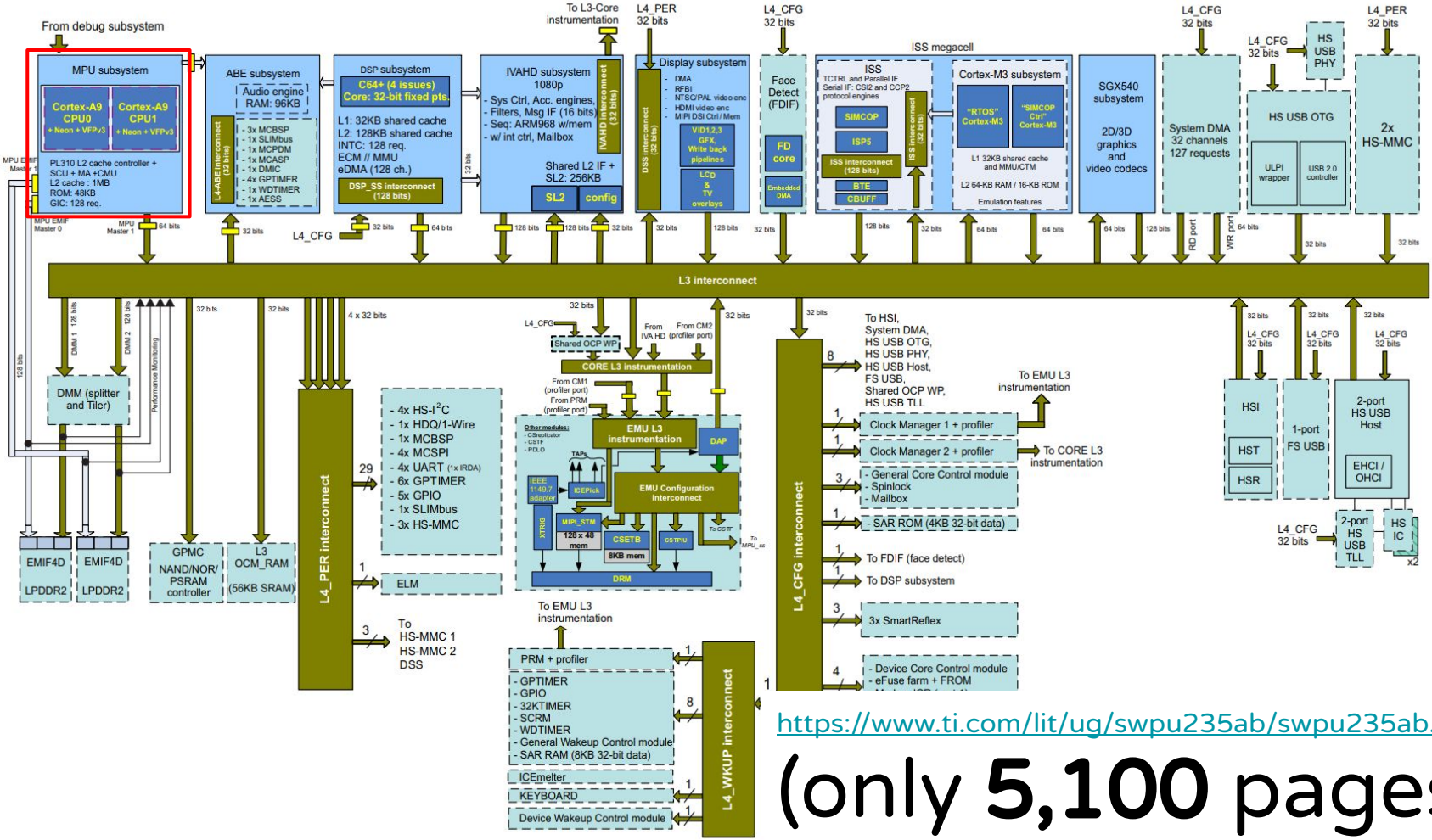


https://docs.oracle.com/cd/E28853_01/html/E28856/z40010911519112.html



https://www.tweaktown.com/reviews/7058/intel-server-r2208wt2ys-system-review/index.html

Figure 1-2. OMAP4460 Block Diagram



https://www.ti.com/lit/ug/swpu235ab/swpu235ab.pdf

(only **5,100** pages)

Summary: You should know …

The idea of the system bus

Address, commands, status interfaces and signalling lines

Synchronous, asynchronous, serial and parallel links*

Port-mapped and memory-mapped I/O techniques

Interrupts, DMA, and polling

* Read from the backup slides at the end

References

- Chapter 3, 7, and 8.10 Carl Hamacher and Zvonko Vranesic, Computer Organization, 6th edition, McGraw-Hill Education, 2011. ISBN-13: 978-0073380650
- Operating Systems: Three Easy Pieces, Chapter 36 - I/O Devices (section 36.1 till 36.6) http://pages.cs.wisc.edu/~remzi/OSTEP/file-devices.pdf

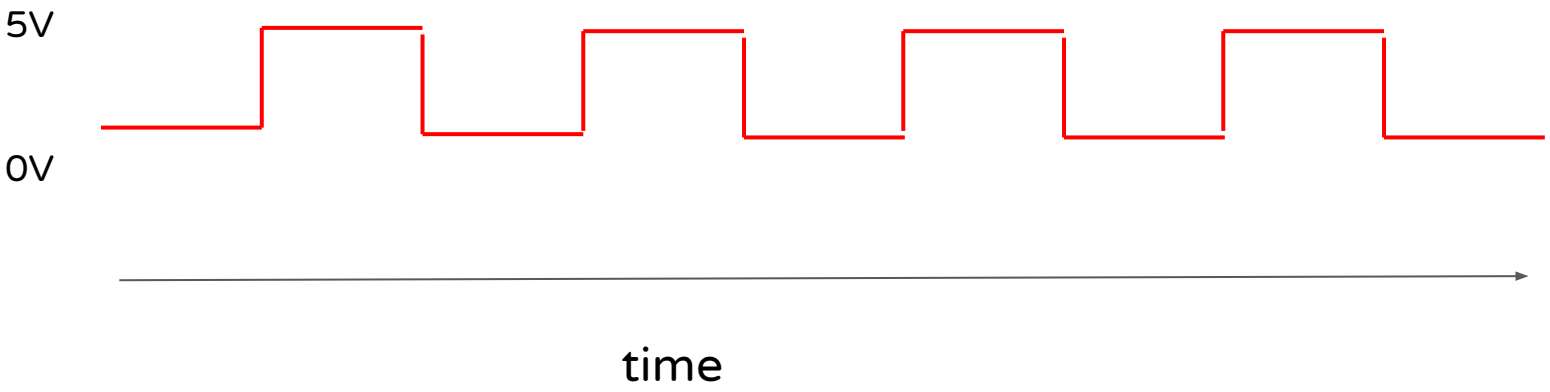
Backup

Synchronous bus

All devices derive timing from a control line called the **bus clock**

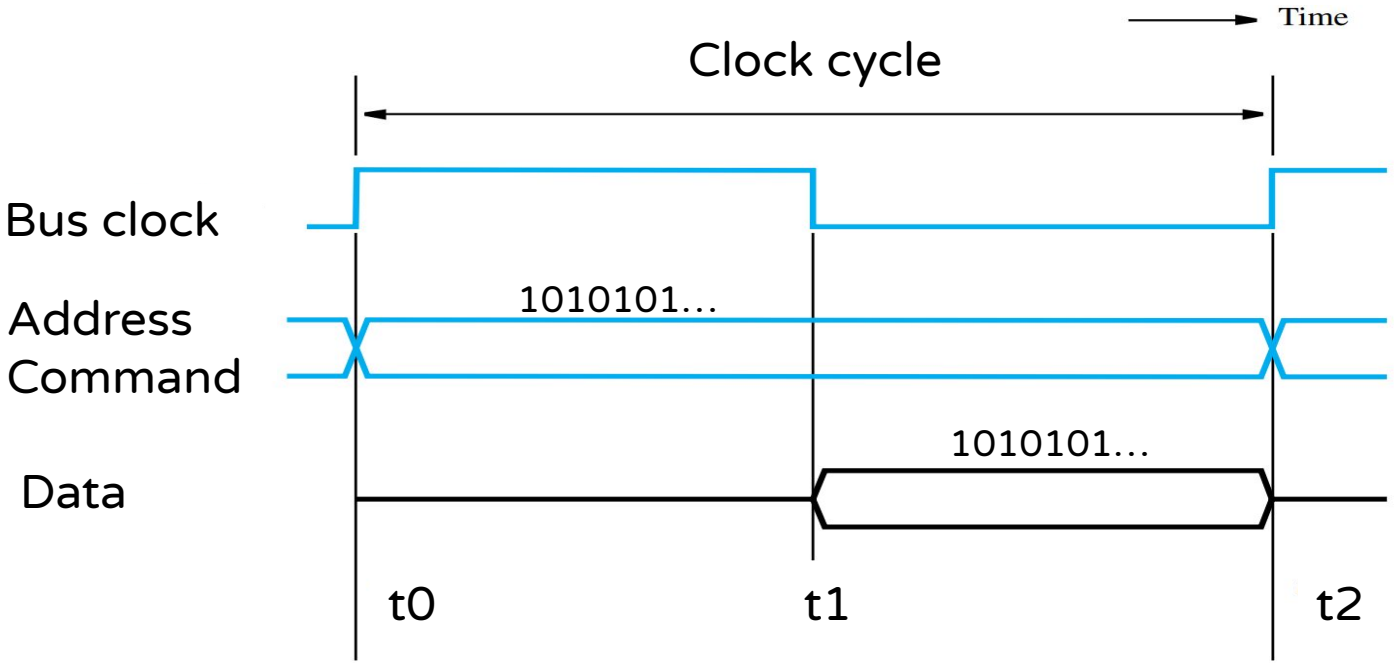
Clock is a special line which has a defined pattern with a rate or frequency

A 1 GHz clock frequency will have 10^9 such cycles per second



Synchronous bus

All devices derive timing from a control line called the **bus clock**

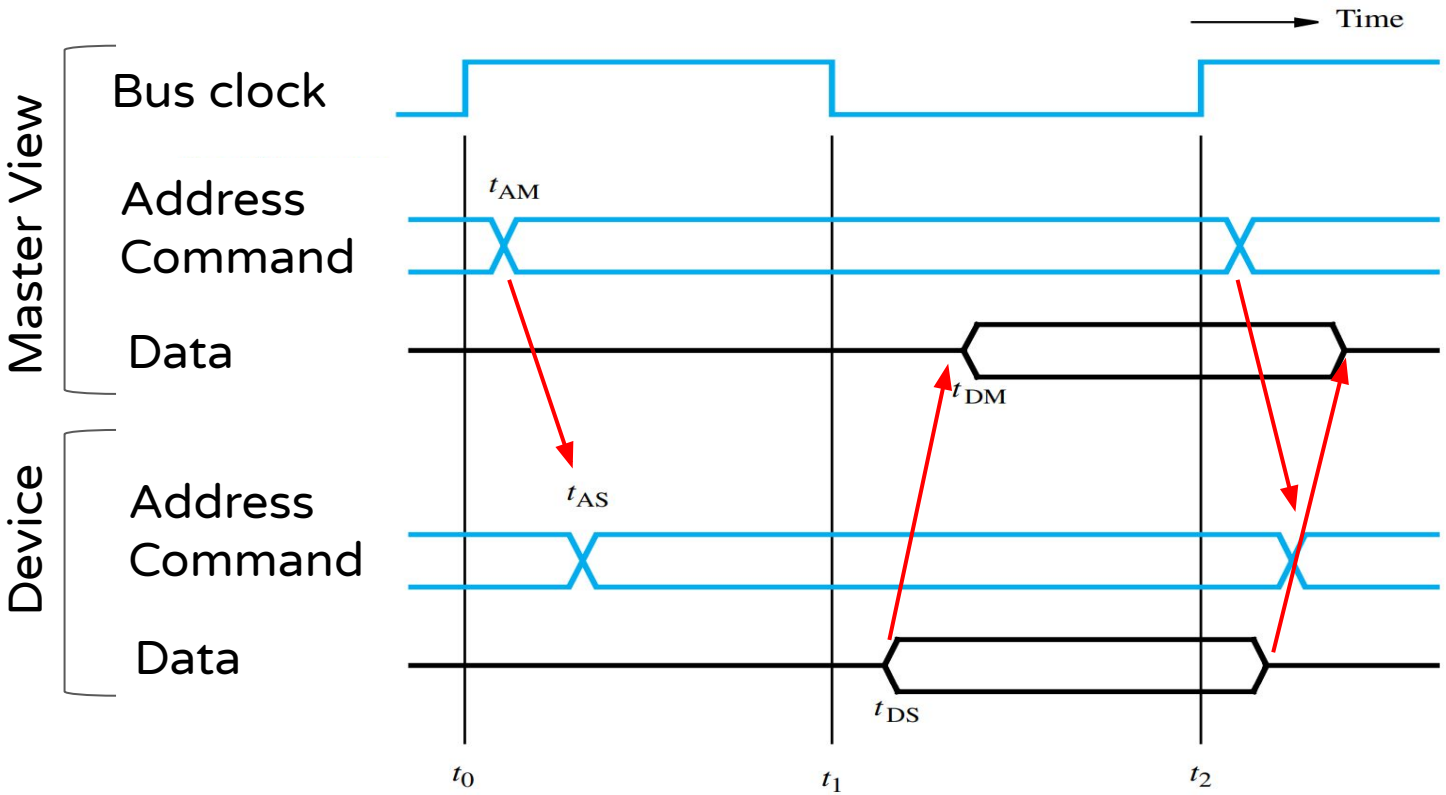


Synchronous bus

What does the clock cycle depend on:

- **Maximum propagation delay**: (t1 - t0) should be longer than the bus length. For example:
  speed of light (~300,000,000 m/s)
  bus distance of 30cm
  Propagation time => 10^{-9} sec, so the clock cannot be more than 1 GHz
- We also need to consider the time it takes for a device to **respond to a signal** (add to the frequency above, and calculate again)
  - So if a device takes 10 nanoseconds to respond, then...
  - ...1 + 10 = 11 nsec, or inversely (maximum) 90.9 MHz clock frequency

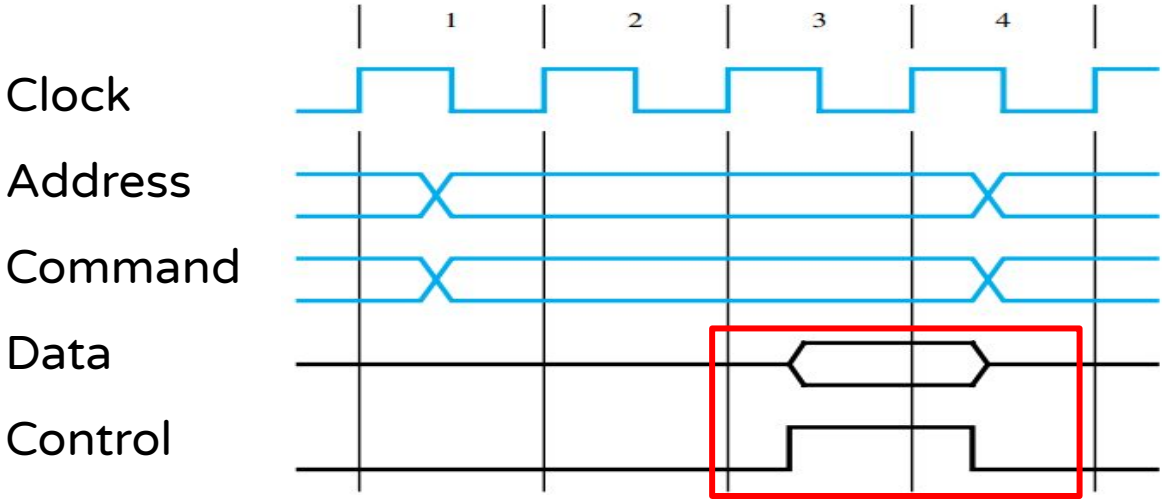
A realistic timing diagram with clock delays



How to do multi-cycle data transfer

How to tell the bus master that the data is available? How to tell the bus master that there is a multiple cycle transfer?

Use the control signal from the device, keep the signal 1 while a transfer is happening



Asynchronous transfer

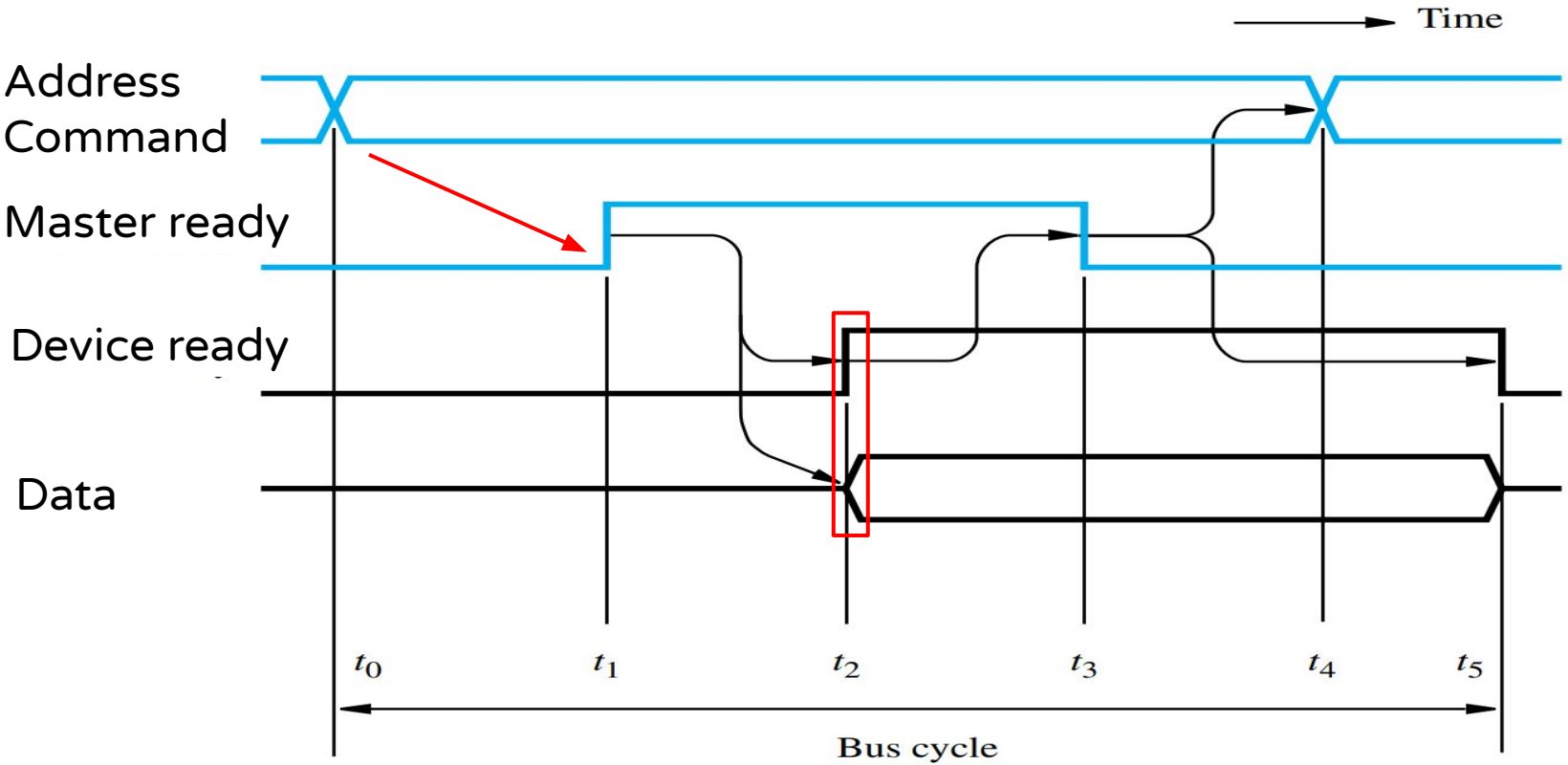
Another way of doing data transfer is **without clocks**

Simplified design: no need to ensure that all devices _must_ see the clock around the same time (with bounded delays)

Synchronization is done using master and device “readiness” signals with a handshake protocol

- The bus master sets the control lines up after putting the address and command
- The “right” device responds by setting its ready control line
- When both lines are ready, the transfer can be initiated

Asynchronous transfer



Synchronous vs. asynchronous buses

Synchronous

+ Fast
- Clock management

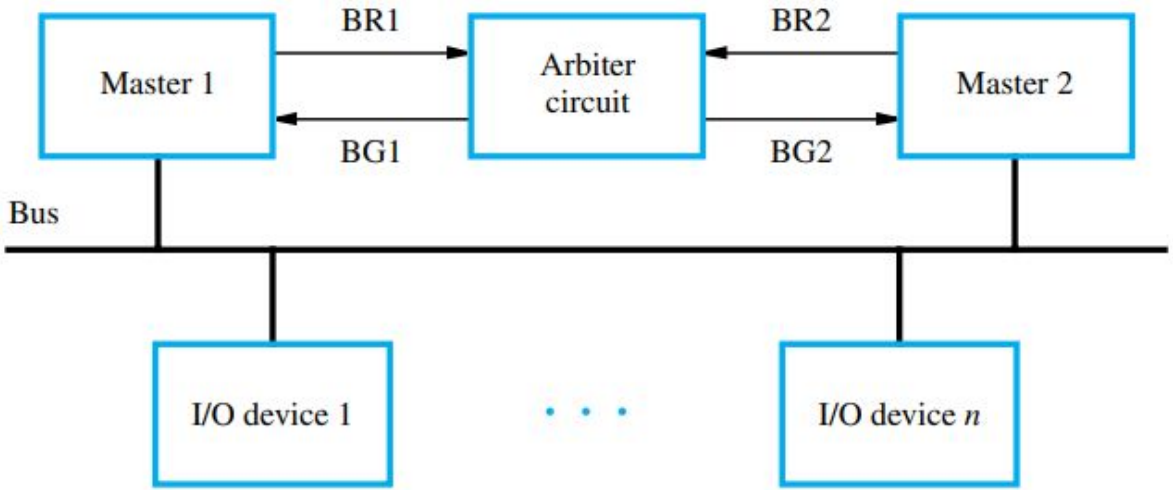
Asynchronous

+ Easy management with devices with different speeds
- Longer handshake protocol (4x steps in a single transaction)
    - Master ready
    - Device ready
    - Device done
    - Master done

Bus arbitration

There can be multiple devices trying to talk simultaneously

There is a bus arbiter - either using addresses, or first-come-first-service basis, or a daisy chaining



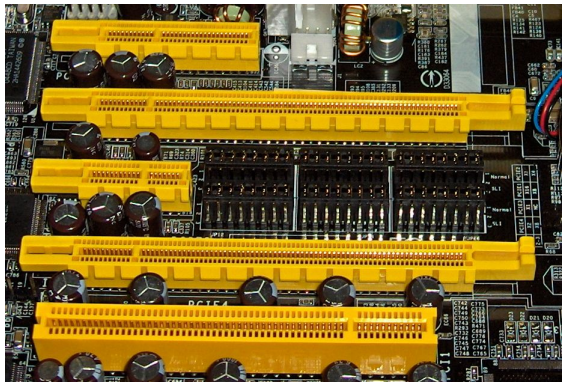
Example: PCIe

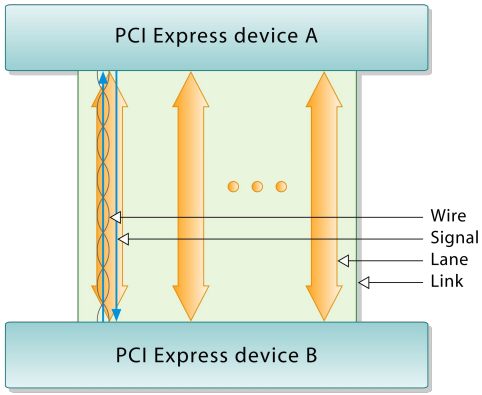
Peripheral Component Interconnect Express (PCIe) - serial, point-to-point

Most popular modern bus to connect devices

Developed by Intel, Dell, HP, IBM

Structure: multi-lane connectivity (x1, x2, x4, x16, x32)





By V4711This W3C-unspecified vector image was created with Adobe Illustrator. - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=37527913

PCIe specification tells what these different pins do
https://en.wikipedia.org/wiki/PCI_Express#Pinout

PCIe bandwidth calculation example

PCI Express version	Introduced	Line code	Transfer rate[i]	Throughput[i]				
				×1	×2	×4	×8	×16
1.0	2003	8b/10b	2.5 GT/s	250 MB/s	0.500 GB/s	1.00 GB/s	2.0 GB/s	4.0 GB/s
2.0	2007	8b/10b	5.0 GT/s	500 MB/s	1.000 GB/s	2.00 GB/s	4.0 GB/s	8.0 GB/s
3.0	2010	128b/130b	8.0 GT/s	984.6 MB/s	1.969 GB/s	3.94 GB/s	7.88 GB/s	15.75 GB/s
4.0	2017	128b/130b	16.0 GT/s	1969 MB/s	3.938 GB/s	7.88 GB/s	15.75 GB/s	31.51 GB/s
5.0	2019	128b/130b	32.0 GT/s[ii]	3938 MB/s	7.877 GB/s	15.75 GB/s	31.51 GB/s	63.02 GB/s
6.0 (planned)	2021	128b/130b	64.0 GT/s	7877 MB/s	15.754 GB/s	31.51 GB/s	63.02 GB/s	126.03 GB/s

https://en.wikipedia.org/wiki/PCI_Express#History_and_revisions

Shared or multiplexed bus

Wider buses (multiple parallel bit lines)

+ More bandwidth
- More expensive and more susceptible to skew

Multiplexed: address and data on same lines/or devices sharing buses

+ Cheaper
- Less bandwidth