# CS 250: FOUNDATIONS OF COMPUTER SYSTEMS [COMPUTER ARCHITECTURE]

**Strings to Bits**
To execute
Strings of code must
Embark on a journey

That transforms and re-expresses
their semantics
Using opcodes in binary

A few lines of high-level code
Gets amplified into long sequences of
Ones and Zeros

Braided tightly together
so that the story
And what must be done
stays the same

SHRIDEEP PALLICKARA
Computer Science
Colorado State University

COMPUTER SCIENCE DEPARTMENT
COLORADO STATE UNIVERSITY

1

## Frequently asked questions from the previous class survey

- Placements of cache lines in fully associative caches?
- When a cache line is not "dirty", how is it evicted?
- Using virtual memory: performance implications?

COLORADO STATE UNIVERSITY Professor: SHRIDEEP PALLICKARA COMPUTER SCIENCE DEPARTMENT ARCHITECTURE L13.2

2

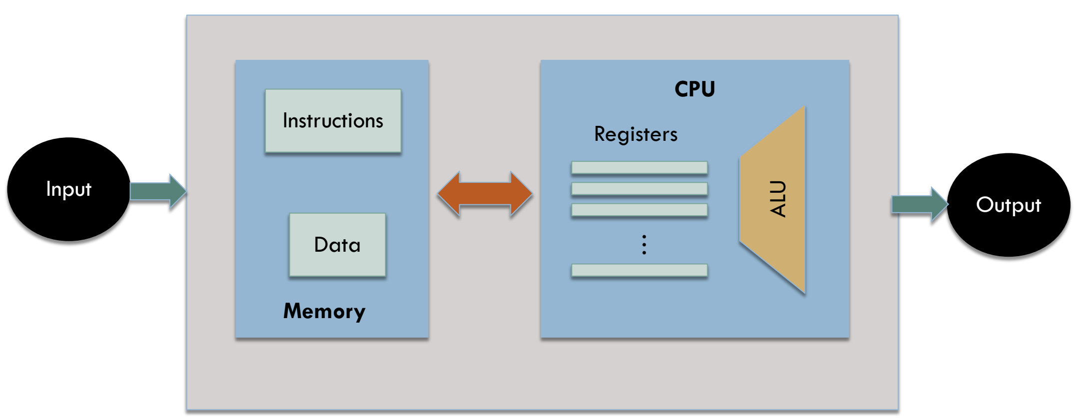
## Topics covered in this lecture

- Bus architectures
- Memory mapped I/O
- Layering & abstractions
- Machine Language



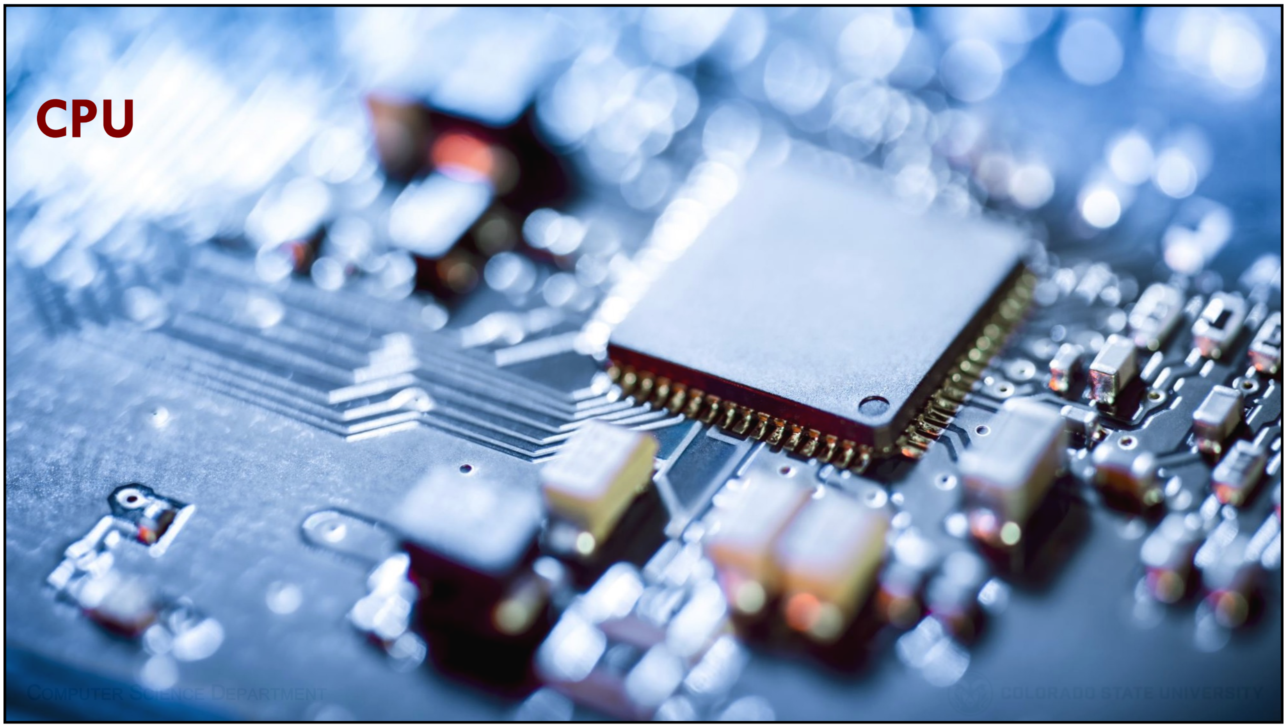
3

## A generic von Neumann computer architecture






4




5

## The CPU is the centerpiece of the computer's architecture

- The Central Processing Unit (CPU) is in charge of executing the instructions of the currently running program
- Each instruction tells the CPU which computation to perform, which registers to access, and which instruction to fetch and execute next
- The CPU executes these tasks using three main elements:
  - An Arithmetic Logic Unit (ALU)
  - A set of registers, and
  - A control unit



6

## Arithmetic Logic Unit (ALU)

- The ALU chip is built to perform all the low-level arithmetic and logical operations featured by the computer
- A typical ALU can add two given values, compute their bitwise And, compare them for equality, and so on …



## How much functionality should be packed into the ALU is a design decision

- In general, any function not supported by the ALU can be *realized later*, using system software running on top of the hardware platform
- The **trade-off** is simple:
  - Hardware implementations are typically more efficient but result in more expensive hardware
  - While software implementations are inexpensive and less efficient

## Why we use registers [1/2]

- While performing computations, the CPU is often required to store **interim values** temporarily
- In theory, we could have stored these values in the RAM, but this would entail **long-distance trips** between the CPU and the RAM
  - The CPU and the RAM are two separate chips



## Why we use registers [2/2]

- Long distance trips between the CPU and RAM result in **delays**
- These delays would frustrate the CPU-resident ALU, which is an ultra-fast combinational calculator
- The result will be a condition known as **starvation**
  - Happens when a fast processor depends on a **sluggish data store** for supplying its inputs and consuming its outputs

## To avert starvation and boost performance

- CPUs are equipped with a **small set** of high-speed (and relatively expensive) registers, acting as the processor's immediate memory
- These registers serve **various purposes**:
  - Data registers store interim values
  - Address registers store values that are used to address the RAM
  - The program counter stores the address of the instruction that should be fetched and executed next
  - The instruction register stores the current instruction

COLORADO STATE UNIVERSITY | Professor: SHRIDEEP PALLICKARA, COMPUTER SCIENCE DEPARTMENT | ARCHITECTURE | L13.11

11


12

## Control

- A computer instruction is a structured package of agreed-upon **micro-codes**
  - Sequences of one or more bits that signal to different circuitry what to do
- Thus, before an instruction can be executed, it must first be decoded into its micro-codes
- Next, each micro-code is **routed** to its designated hardware circuitry (ALU, registers) within the CPU
  - Where it tells the device *how to partake* in the overall instruction execution



## Fetch-Execute

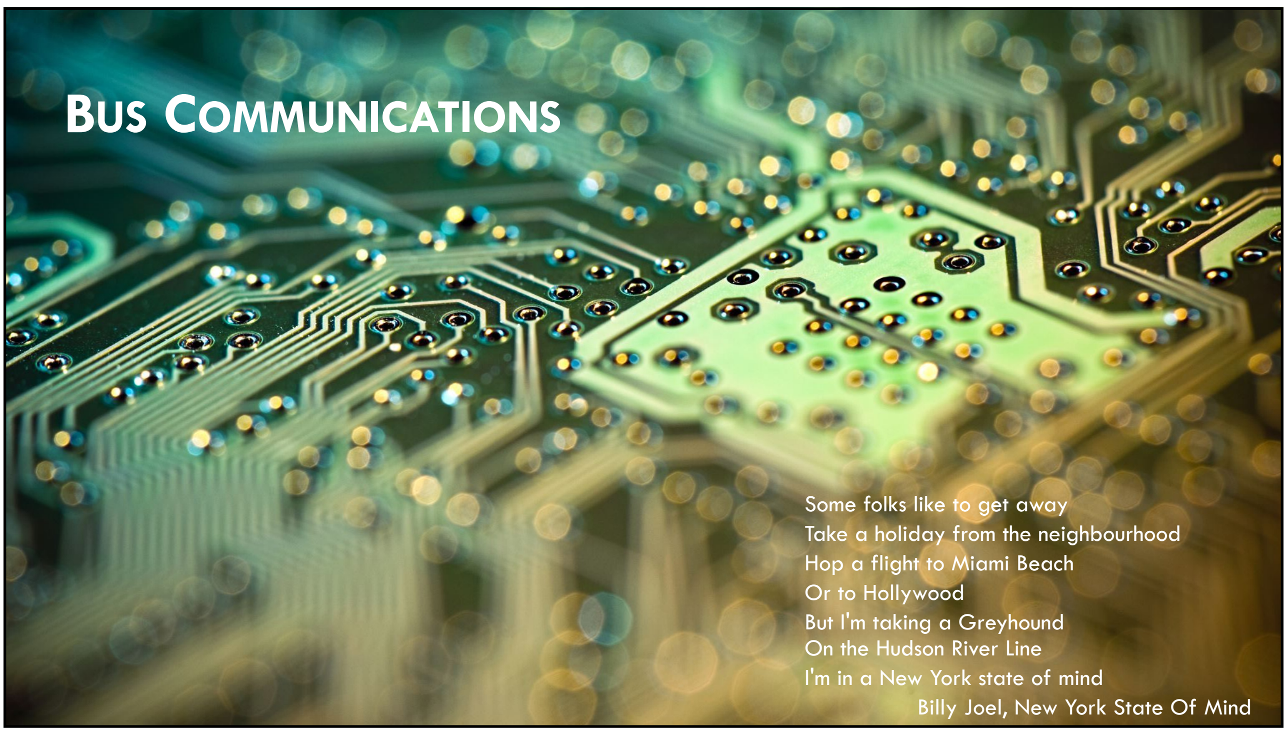
- In each step (cycle) of the program's execution:
  - The CPU fetches a binary machine instruction from the instruction memory,
  - Decodes it, and
  - Executes it
- As a side effect of the instruction's execution, the CPU also **figures out** *which instruction* to fetch and execute next
- This *repetitive process* is sometimes referred to as the **fetch-execute cycle**

## CPU trends

- Multiprocessor systems with multiple CPUs debuted in the 1980s to get higher performance than could be achieved with a single CPU
- As it turns out, though, it's not that easy
- Dividing up a single program so that it can be parallelized to make use of multiple CPUs is an **unsolved problem in the general case**
  - Although it works well for several classes of problems



15




16

## A bus is a hardware communication system used by computer components

- In the early days of computers, a bus was simply a set of **parallel wires**, each carrying an electrical signal
  - This allowed multiple bits of data to be transferred in parallel; the voltage on each wire represented a single bit
- Today's bus designs aren't always that simple, but the **intent is similar**



## There are **3** common bus types used in communication between the CPU, memory, and I/O devices [1/2]

- An **address bus** selects the memory address that the CPU wishes to access
  - For example, if a program wishes to write to address 0x2FE, the CPU writes 0x2FE to the address bus
- The **data bus** *transmits* a value read from (or to be written to) memory
  - If the CPU is reading data from memory, that value is read from the data bus
  - If the CPU wishes to write 25 to memory, then 25 is written to the data bus

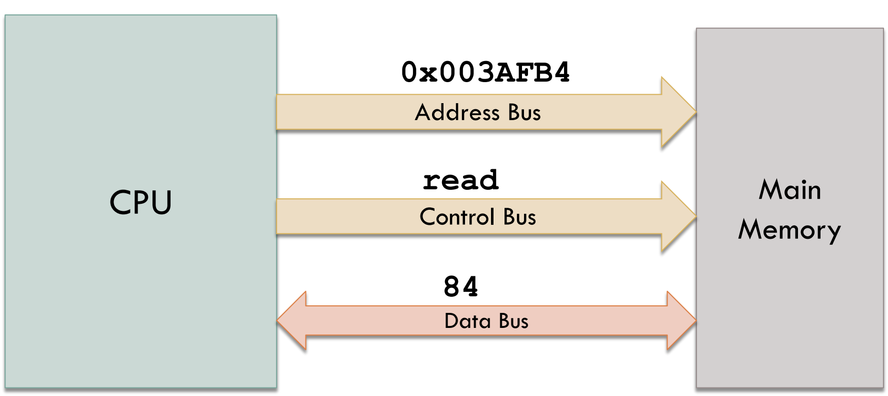
## There are **3** common bus types used in communication between the CPU, memory, and I/O devices [2/2]

- A **control bus** manages the operations happening over the other two buses
  - For e.g., the CPU uses the control bus to indicate that a write operation will occur
  - Or the control bus can carry a signal indicating the status of an operation

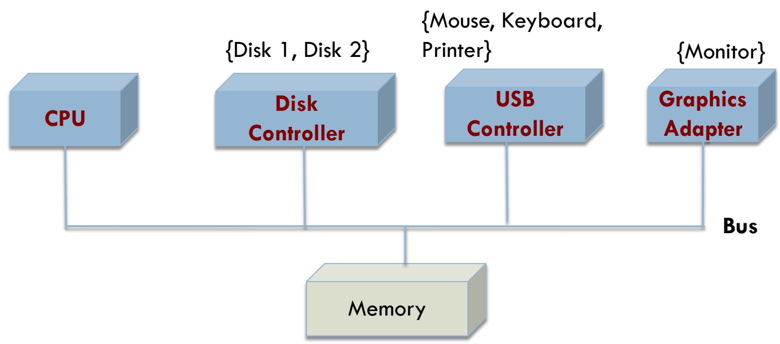


## Bus Interactions: Example

The CPU requests a read of the value at memory location **`0x003AFB4`** which returns the value **`84`**



## A simple bus-based structure

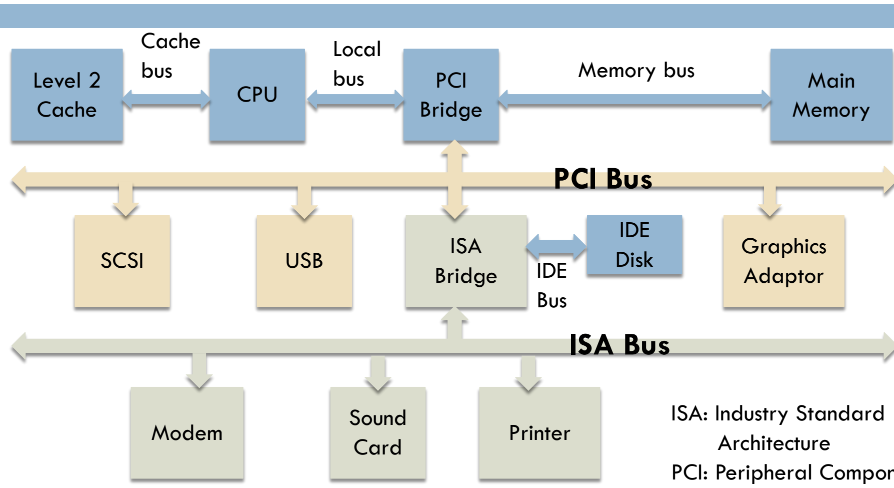





## Limitations of the bus structure from the earlier slide

- As processors and memories got faster
  - Ability of a single bus to handle *all traffic* strained considerably
- Result?
  - Additional buses were added
  - For faster I/O devices and CPU-memory traffic

# What a modern bus architecture looks like




ISA: Industry Standard Architecture
PCI: Peripheral Component Interconnect



# There are two main BUS standards

- Original IBM PC ISA (Industry Standard Architecture)
- PCI (Peripheral Component Interconnect)
  - From Intel

## The IBM PC ISA bus

- Runs at 8.33 MHz
- Transfers 2 bytes at once
- Maximum speed = 16.67 MB/sec
- Included for backward compatibility
  - Older and slower I/O cards



## The PCI bus

- Can run at 66 MHz
- Transfer 8 bytes at once
- Data transfer rate: 528 MB/sec
- Most high-speed I/O devices use PCI
- Newer computers have an updated version of PCI
  - **PCI Express**

## Other specialized buses: IDE (Integrated Drive Electronics) bus

- For attaching peripheral devices
  - CD-ROMs and Disks
- Grew out of the disk controller interface



## Other specialized buses: USB (Universal Serial Bus)

- Attach **slow** I/O devices to the computer
  - Keyboard, mouse etc
- Uses a small **4-wire** connector
  - **Two** supply electrical power to the USB devices
- Centralized bus
  - Root device polls I/O devices every millisecond
    - Check if they have any traffic

## Some more information about USB

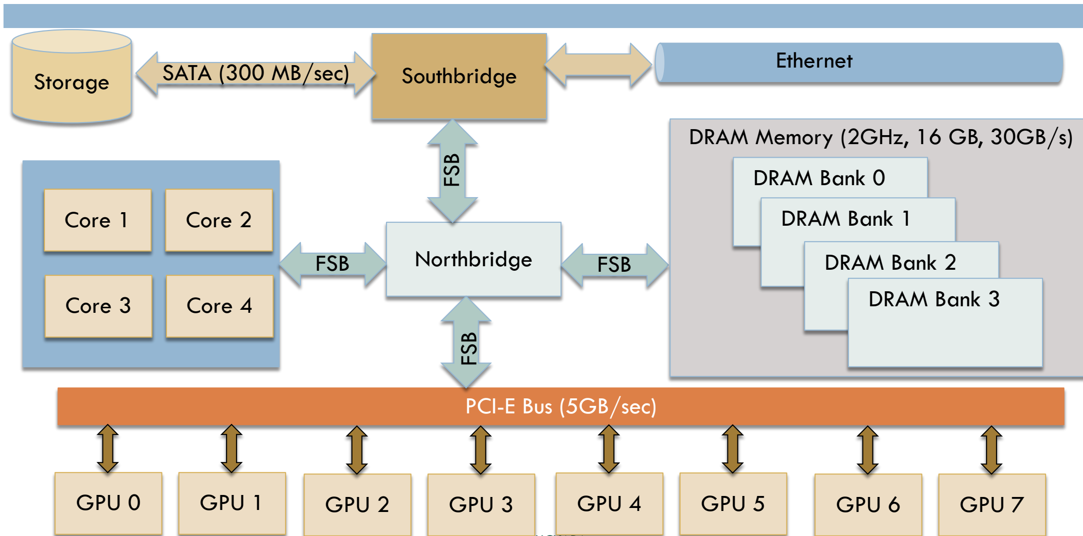
- All USB devices share a **single** USB device driver
  - *No need to install* a driver for each device
  - Can be added to computer *without need to reboot*
- USB 1.0 had a transfer rate of 1.5 MB/sec
- USB 2.0 went up to 60 MB/sec
- USB 3.0
  - Specification ready on 17 November 2008
  - Theoretical signaling rate: 600 MB/sec (4.8 Gbps)
  - USB 3.1: Jan 2013 goes up to 10 Gbps
  - US 3.2 released in September 2017 transfer modes 10 and 20 Gbps
- USB-C 24 pins: 16 for data transfer, 4 for power, and 4 ground



## Other buses

- SCSI (Small Computer System Interface)
  - High performance bus
  - For devices that need high bandwidth
    - Fast disks, scanners
  - Up to 320 MB/sec
- IEEE 1394
  - Sometimes called FireWire (used by Apple)
  - Transfer speeds of up to 100 MB/sec
    - Camcorders and similar multimedia devices
  - No need for a central controller (unlike USB)

## Typical PC architecture






31

## Typical desktop CPU/GPU layout

- All GPU devices are connected to the processor via the PCI-E bus
  - To get data from the processor, we need to go through the Northbridge device over the slow FSB (front-side bus)
- The FSB can run anything up to 1600 MHz clock rate, although in many designs it is much slower



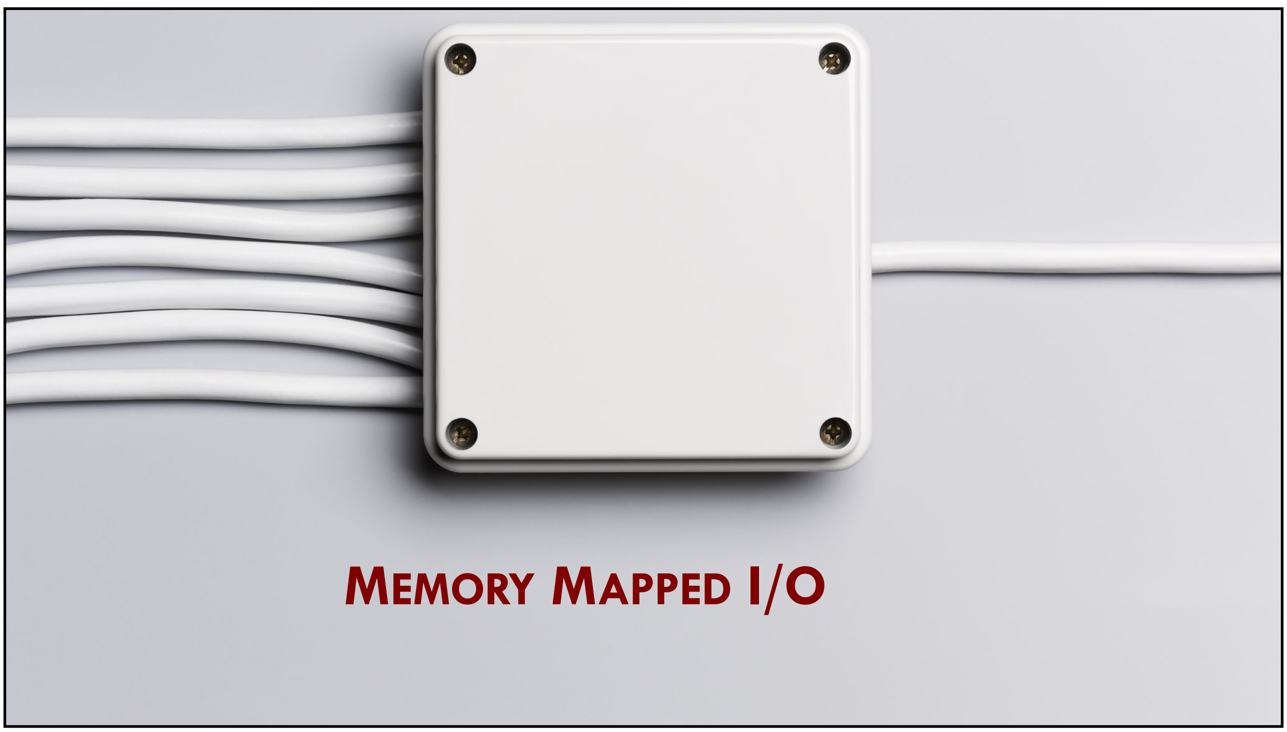
32

## The Northbridge/Southbridge chipsets

- The **Northbridge** chipset deals with all the high-speed components
  - Memory, CPU, PCI-E bus connections, etc.
- The **Southbridge** chip deals with the slower devices such as hard disks, USB, keyboard, network connections, etc.
- Since the 2010s, functions performed by northbridges are now often incorporated into other components
  - Die shrink and improved transistor density allow for chipset integration
  - Notably in 2019, both Intel and AMD released chipsets where all northbridge functions were integrated into the CPU

COLORADO STATE UNIVERSITY Professor: SHRIDEEP PALLICKARA COMPUTER SCIENCE DEPARTMENT ARCHITECTURE L13.33

33




34

## Computers interact with their external environments using a great variety of I/O devices

- Examples include screens, keyboards, storage devices, printers, microphones, speakers, network interface cards, and so on
- Not to mention the bewildering array of sensors and activators
  - Embedded in automobiles, cameras, hearing aids, alarm systems, and all the gadgets around us



## There are **two reasons** why we don't concern ourselves with these I/O devices

- Every one of them represents a unique piece of machinery, requiring a unique knowledge of engineering
- For that very same reason, computer scientists have devised clever schemes
  - For **abstracting** away this complexity and
  - Making all I/O devices **look exactly the same** to the computer
- The key element in this abstraction is called **memory-mapped I/O**

## Memory mapped I/O

- The basic idea is to create a **binary emulation** of the I/O device
  - *Making it appear* to the CPU as if it were a **regular linear memory segment**
- How?
  - By allocating, *for each I/O device*, a **designated area** in the computer's memory that acts as its **memory map**



## Memory mapped I/O: Examples

- In the case of an input device like a keyboard, the memory map is made to **continuously reflect the physical state of the device**:
  - When the user presses a key on the keyboard, a binary code representing that key appears in the keyboard's memory map
- In the case of an output device like a screen, the screen is made to **continuously reflect the state of its designated memory map**
  - When we write a bit in the screen's memory map, a respective pixel is turned on or off on the screen

## How?

- The I/O devices and the **memory maps are refreshed**, or synchronized, many times per second
  - So, the response time from the user's perspective appears to be instantaneous
- Programmatically, the key implication is that low-level computer programs can **access any I/O device**
  - By **manipulating its designated memory map**



## The memory map convention is based on several agreed-upon contracts [1/2]

- The data that drives each I/O device must be **serialized, or mapped,** onto the computer's memory
  - Hence the name memory map
- For example, the screen, which is a two-dimensional grid of pixels, is mapped on a one-dimensional block of fixed-size memory

## The memory map convention is based on several agreed-upon contracts [2/2]

- Each I/O device is required to support an **agreed-upon interaction protocol**
  - So that programs will be able to access it in a predictable manner
  - For example, it should be decided which binary codes should represent which keys on the keyboard
- Given the multitude of computer platforms, I/O devices, and different hardware and software vendors
  - Agreed-upon, industry-wide standards play a crucial role in realizing these low-level interaction contracts



## The practical implications of memory-mapped I/O are significant

- The computer system is **totally independent** of the number, nature, or make of the I/O devices that interact, or may interact, with it
- Whenever we want to connect a **new I/O device** to the computer, all we have to do is **allocate to it a new memory map** and take note of the map's base address
  - These **one-time configurations** are carried out by installer programs

## What else?

- Another necessary element is a **device driver** program, which is added to the computer's operating system
- The device driver program **bridges the gap**
  - Between the I/O device's memory map data and the way this data is actually rendered on, or generated by, the physical I/O device

COLORADO STATE UNIVERSITY Professor: SHRIDEEP PALLICKARA COMPUTER SCIENCE DEPARTMENT ARCHITECTURE L13.43

43


44

## The journey from high level to machine level [1/2]

- All high-level languages rely on a suite of **translators** for reducing high-level code all the way down to machine-level instructions
- The translators could be
  - Compiler/ interpreter
  - Virtual machine
  - Assembler



## The journey from high level to machine level [2/2]

- Some high-level languages are interpreted rather than compiled, and some don't use a virtual machine
  - But the big picture is essentially the same
- This observation is a manifestation of a fundamental computer science principle, known as the **Church-Turing conjecture**
  - At its core, all computers are essentially equivalent

## Abstractions vs Implementations [1/2]

- The cognitive ability to "**divide and conquer**" a complex system into manageable modules is key
- Empowered by yet another cognitive gift:
  - Our ability to discern between the abstraction and the implementation of each module



## Abstractions vs Implementations [2/2]

- In computer science, we take these words concretely
- **Abstraction** describes *what* the module does
- **Implementation** describes *how* it does it
- With this distinction in mind, here is the most important rule in system design:
  - *When using any module* as a building block you are to focus exclusively on the module's abstraction, ignoring its implementation details

## To recap …

- Whenever your implementation uses a lower-level hardware or software module
  - You are to treat this module as an **off-the-shelf, black box** abstraction



## All you need is the documentation of the module's interface, describing what it can do, and off you go

- You are to pay no attention whatsoever to **how** the module performs what its interface advertises
- This abstraction-implementation paradigm helps developers **manage complexity and maintain sanity**:
  - By dividing an overwhelming system into well-defined modules, we create manageable chunks of implementation work
    - And **localize error detection and correction**
  - This is the most important design principle in hardware and software construction projects

## The abstractions are often built layer upon layer

- Resulting in higher and higher levels of functionality
- If the system architect designs a good set of modules, the implementation work will flow like clear water
  - If the design is slipshod, the implementation will be doomed!
- Modular design is an **acquired art**
  - Honed by seeing and implementing many well-designed abstractions



## The contents of this slide-set are based on the following references

- Noam Nisan and Shimon Schocken. *The Elements of Computing Systems: Building a Modern Computer from First Principles*. 2nd Edition. ISBN-10/ ISBN-13: 0262539802 / 978-0262539807. MIT Press. [Preface, Chapter 4-5]