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

1

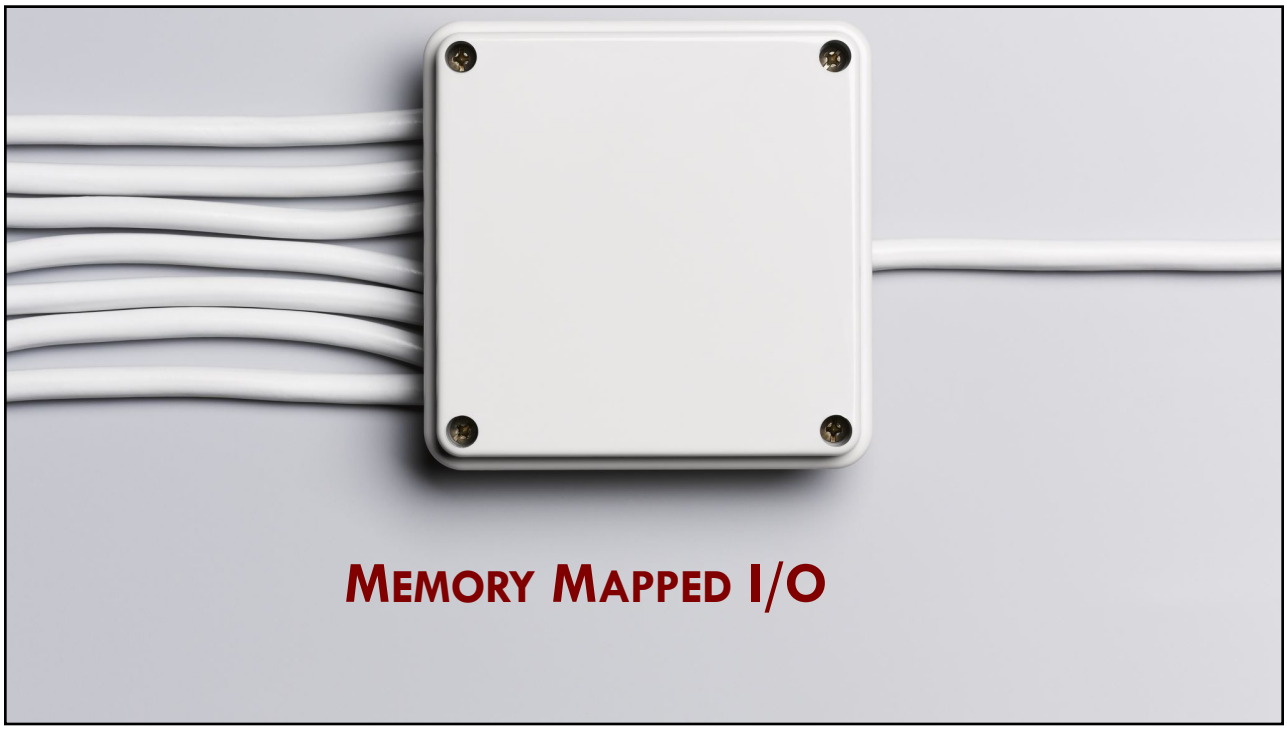
Frequently asked questions from the previous class survey

2

Topics covered in this lecture

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

3



4

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 and each requires 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

13

14

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 (e.g., C and Java)/ interpreter (e.g., Python and Javascript)
  - 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

Works of imagination should be written in very plain language; the more purely imaginative they are, the more necessary it is to be plain.
—Samuel Taylor Coleridge (1772–1834)

MACHINE LANGUAGE

The simplest program out there

```
/** The simplest program out there! */

public class HelloWorld {

    /** This does not even take an argument */
    public static void main(String[] args) {

        System.out.println("Hello World");
    }

}
```


What does it take to actually run this? [1/2]

- Let's look under the hood
- For starters, note that the program is nothing more than a **sequence of plain characters**, stored in a text file
- This abstraction is a complete mystery for the computer, which understands only instructions written in machine language

What does it take to actually run this? [2/2]

- The first thing we must do is **parse** the string of characters of which the high-level code is made, **uncover its semantics** *i.e., figure out* what the program seeks to do
  - And then generate low-level code that **reexpresses this semantics** using the **machine language** of the target computer
- The result of this elaborate translation process, known as **compilation**, will be an **executable sequence of machine language instructions**

Machine language is also an abstraction

- An *agreed upon* set of binary codes
- To make this abstraction concrete
  - It must be realized by some hardware architecture
  - And this architecture, in turn, is implemented by a certain set of hardware: registers, memory units, adders, and so on
  - Now, every one of these hardware devices is constructed from lower-level, elementary logic gates
  - And these gates, in turn, can be built from primitive gates like Nand and Nor
    - These primitive gates are very low in the hierarchy, but they, too, are made of several switching devices, typically implemented by transistors

But this is so much easier on your computer

- On your computer, compiling and running programs is much easier
  - All you have to do is click this icon or write that command!
- Indeed, a modern computer system is like a submerged iceberg
  - Most people get to see only the top
  - Knowledge of computing systems is often sketchy and superficial

A machine language is an agreed-upon formalism designed to code machine instructions

- Using these instructions, we can instruct the computer's processor to:
  - Perform arithmetic and logical operations
  - Read and write values from and to the computer's memory
  - Test Boolean conditions
  - Decide which instruction to fetch and execute next

Design goals in high-level and machine languages differ

- Design goals in high-level languages
  - Cross-platform compatibility and power of expression
- Machine languages are designed to effect **direct execution** in, and total control of, a specific hardware platform
  - Of course, generality, elegance, and power of expression are still desired
  - But only to the extent that they support the basic requirement of direct and efficient execution in hardware

Machine language is the most profound interface in the computer enterprise

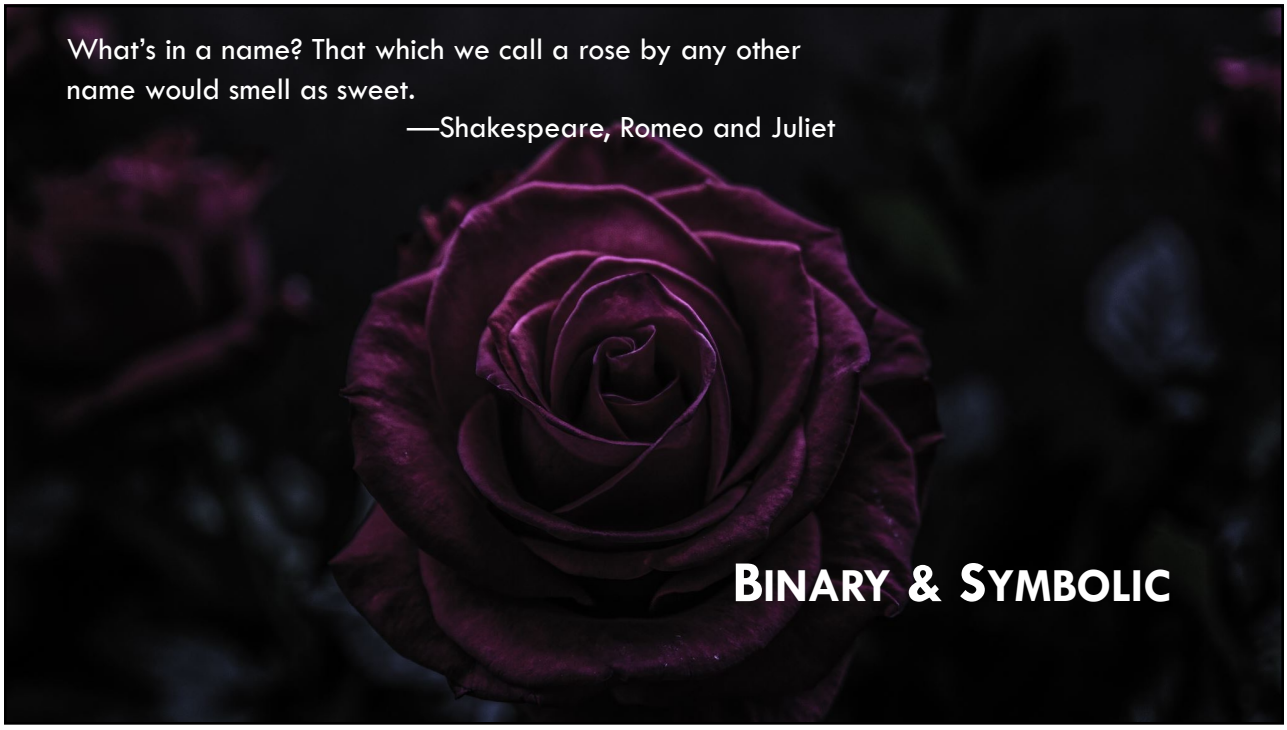
- The fine line **where hardware meets software**
- The point where the abstract designs of humans, as manifested in high-level programs, are finally reduced to physical operations performed in silicon

A machine language is both a programming artifact and an integral part of the hardware platform

- Just as we say that the machine language is **designed to control** a particular hardware platform
- We can say that the hardware platform is **designed to execute** instructions written in a particular machine language

Who writes machine language programs?

- Even the most sophisticated software systems are, at bottom, streams of simple instructions
  - Each specifying a primitive operation on the underlying hardware
- It should be noted that machine language programs are rarely written by humans
- Rather, they are typically written by **compilers**
- And a compiler can optionally bypass the symbolic instructions and generate binary machine code directly



33

Writing machine language programs

- Machine language programs can be written in two alternative, but equivalent, ways
  - Binary
  - Symbolic

34

Machine Language: Binary vs Symbolic

- Consider the abstract operation "set R1 to the value of R1 + R2"
- Language designers, can decide to represent
  - The addition operation using the 6-bit code 101011,
  - Registers R1 and R2 using the codes 00001 and 00010, respectively
- Assembling these codes left to right:
  - The 16-bit instruction 1010110000100010 can be used as the binary version of "set R1 to the value of R1 + R2"

In the early days of computer systems, computers were programmed manually

- When proto-programmers wanted to issue the instruction "set R1 to the value of R1 + R2"
  - They pushed up and down mechanical switches that stored a binary code like 1010110000100010 in the computer's instruction memory
- And if the program was a hundred instructions long?
  - They had to go through this ordeal a hundred times
- Of course, debugging such programs was a perfect nightmare

Symbolic codes to the rescue

- This led programmers to invent and use **symbolic codes**
  - Convenient way for documenting and debugging programs **on paper**, *before* entering them into the computer
- For example, the symbolic format `add R2,R1` could be chosen
  - For representing the semantics "set R1 to the value of R1 + R2" and the binary instruction 1010110000100010

It didn't take long before several people hit on the same idea

- Symbols like R, 1, 2, and + can also be represented using agreed-upon binary codes
- **Why not use symbolic instructions for writing programs**?
  - And then use another program (a *translator*) for translating the symbolic instructions into executable binary code?
- This innovation liberated programmers from the tedium of writing binary code
  - Paving the way for the subsequent onslaught of high-level programming languages

Symbolic machine languages

- Symbolic machine languages are called **assembly languages**
- The programs that *translate* them into binary code are called **assemblers**

High-level vs Assembly languages

- Syntax of high-level languages
  - Portable and **hardware independent**
- The syntax of an assembly language?
  - Tightly related to the low-level details of the target hardware
    - The available ALU operations, number and type of registers, memory size, and so on

But there is so much diversity in hardware

- Since different computers vary greatly in terms of any one of these parameters, there is a *Tower of Babel* of machine languages
  - Each with its obscure syntax
  - Each designed to control a particular family of CPUs
- Irrespective of this variety, though, **all machine languages are theoretically equivalent**
  - All of them support similar sets of generic tasks

Symbolic language & the Assembler [1/2]

- The symbolic version includes all sorts of things that humans are fond of seeing in computer programs
  - Comments, white space, indentation, symbolic instructions, and symbolic references
- None of these embellishments concern computers, which understand one thing only: bits

Symbolic language & the Assembler [2/2]

- The agent that bridges the gap between the symbolic code convenient for humans and the binary code understood by the computer is the **assembler**
- The assembler takes as input a stream of assembly instructions and generates as output a **stream of translated binary instructions**
  - The resulting code can be loaded as is into the computer memory and executed

INSTRUCTION SET ARCHITECTURES

If everything seems under control, you're just not going fast enough.
—Mario Andretti (b. 1940), race car champion

CPU instructions

- Although all CPUs implement arithmetic instructions, the specific instructions available on different processors vary
- Some instructions that exist for one type of CPU simply don't exist on other types of CPUs
- Even instructions that do exist on nearly all CPUs **aren't implemented in the same way**
  - For example, the specific binary sequence used to mean "add two numbers" is not the same across processor types

A family of CPUs that use the same instructions are said to share an **instruction set architecture (ISA)**

- An ISA is also a **model** of how a CPU works
- Software that's built for a certain ISA works on *any* CPU that implements that ISA
  - It's possible for multiple processor models, even those from different manufacturers, to implement the same architecture
  - Such processors **may work very differently internally**, but by adhering to the same ISA, they can run the same software

Today, there are two prevalent instruction set architectures

- x86
  - Personal computers, desktops, and servers
- ARM (Advanced RISC Machines)
  - Hand-held devices

The majority of desktop computers, laptops, and servers use x86 CPUs

- The term **x86** refers to a set of related architectures
- The name comes from Intel Corporation's naming convention for its processors (each ending in 86)
  - Beginning with the 8086 released in 1978, and continuing with the 80186, 80286, 80386, and 80486
  - After the 80486 (or more simply the 486), Intel began branding its CPUs with names such as Pentium and Celeron
    - These processors are still x86 CPUs despite the name change
- Other companies besides Intel also produce x86 processors
  - Most notably Advanced Micro Devices, Inc. (AMD)

Over time, new instructions have been added to the x86 architecture

- But each generation has tried to **retain backward compatibility**
- This generally means that software developed for an older x86 CPU runs on a newer x86 CPU
  - But software built for a newer x86 CPU that takes advantage of new x86 instructions won't be able to run on older x86 CPUs
    - Older x86 CPUs don't understand the new instructions: **no forward compatibility**

49

50

Generations of the x86 architecture

- The x86 architecture includes three major generations of processors:
  - 16-bit
  - 32-bit
  - 64-bit

What do we mean when we say that a CPU is a 16-bit, 32-bit, or 64-bit processor?

- The number of bits associated with a processor refers to the number of bits it can deal with at a time
  - Also known as its **bitness** or **word size**
- So, a 32-bit CPU can operate on values that are 32 bits in length
  - More specifically, this means that the computer architecture has 32-bit registers, a 32-bit address bus, or a 32-bit data bus
  - Or all three may be 32-bit

Bitness

- The original 8086 processor, released in 1978, was a 16-bit processor
  - Intel's subsequent x86 processors were also 16-bit until the 80386 processor
- The 80386 released in 1985, brought with it a new 32-bit version of the x86 architecture
  - This 32-bit version of x86 is sometimes called **IA-32**
- Thanks to backward compatibility, modern x86 processors still fully support IA-32

Interestingly, it was AMD, and not Intel, that brought x86 into the 64-bit era

- In the late 1990s, Intel's 64-bit focus was on a new CPU architecture called IA-64 or Itanium
  - This was not an x86 ISA, and ended up as a niche product for servers
- With Intel focused on Itanium, AMD seized the opportunity to extend the x86 architecture
  - In 2003, AMD released the Opteron processor, the first 64-bit x86 CPU

AMD's architecture was originally known as AMD64

- Later Intel adopted this architecture and called its implementation Intel 64
- The two implementations are mostly **functionally identical**
- Today 64-bit x86 is generally referred to as x64 or **x86-64**
  - **981** unique mnemonics and **3,684** instruction variants
  - Only 81 instructions in the 8086
  - Apple Silicon chips (M1, M2, M3, M4) use the AArch64 (ARM64) ISA
    - Supports around 350–400+ distinct instructions
  - NVIDIA A100 GPU: the native SASS (streaming assembler has 350–400+ opcodes)

Although x86 rules the PC/server world, ARM processors command the realm of mobile devices

- Multiple companies manufacture ARM processors
- A company called ARM Holdings develops the ARM architecture and licenses their designs to other companies to implement

It's common for ARM CPUs to be used in **system-on-chip (SoC)** designs

- Where a single integrated circuit contains not only a CPU, but also memory and other hardware
- The ARM architecture originated in the 1980s as a 32-bit ISA
- A 64-bit version of the ARM architecture was introduced in 2011

ARM processors are favored in mobile devices

- Due to their **reduced power consumption** and lower cost as compared to x86 processors
- ARM processors can be used in PCs as well
  - But that market largely remains focused on x86, to retain backward compatibility with existing x86 PC software

ARMs foray into desktops and laptops

- Started in earnest in 2020 when Apple announced their intention to move macOS computers from x86 to ARM CPUs
- The first line of computers, MacBook Pros were released with this ARM-based SoC design in 2021
- Includes mechanisms to work with executables that use legacy x86-64 and x86-32 codes
  - **Rosetta** software to use apps built for a Mac with an Intel processor

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]