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

Frequently asked questions from the previous class survey

- Cache Evictions
  - In a direct mapped cache: Whatever is there in the current cache line
  - Associative cache: Least Recently Used (LRU)
- Von Neumann Architecture: Alternatives?
- Is machine language binary?
Topics covered in this lecture

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

A generic van Neumann computer architecture
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.
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

- 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

☐ 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.
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

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)
The simplest program out there

```java
/** 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—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 chips — 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 [1/2]

- 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 and
  - Decide which instruction to fetch and execute next
A machine language is an agreed-upon formalism designed to code machine instructions

- 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 bitwise 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 — being an automaton — can optionally bypass the symbolic instructions and generate binary machine code directly.
What’s in a name? That which we call a rose by any other name would smell as sweet.
—Shakespeare, Romeo and Juliet

Writing machine language programs

- Machine language programs can be written in two alternative, but equivalent, ways
  - Binary
  - Symbolic
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 1010110001000001 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 1010110001000001 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 `1010110001000001`.

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

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.

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