Lecture #4-5: Computer Hardware (Overview and CPUs)

CS106E Spring 2018, Young

In these lectures, we begin our three-lecture exploration of Computer Hardware. We start by looking at the different types of computer components and how they interact during basic computer operations.

Next, we focus specifically on the CPU (Central Processing Unit). We take a look at the Machine Language of the CPU and discover it's really quite primitive. We explore how Compilers and Interpreters allow us to go from the High-Level Languages we are used to programming to the Low-Level machine language actually used by the CPU.

Most modern CPUs are multicore. We take a look at when multicore provides big advantages and when it doesn't. We also take a short look at Graphics Processing Units (GPUs) and what they might be used for.

We end by taking a look at Reduced Instruction Set Computing (RISC) and Complex Instruction Set Computing (CISC). Stanford President John Hennessy won the Turing Award (Computer Science's equivalent of the Nobel Prize) for his work on RISC computing.

Hardware and Software:

Hardware refers to the physical components of a computer.
Software refers to the programs or instructions that run on the physical computer.

- We can entirely change the software on a computer, without changing the hardware and it will transform how the computer works. I can take an Apple MacBook for example, remove the Apple Software and install Microsoft Windows, and I now have a Window's computer.

- In the next two lectures we will focus entirely on Hardware.

Computer hardware components can generally be broken down into three categories:

Processing – Processing components are responsible for actually carrying out actions in the computer. The main processing component is the ***Central Processing Unit (CPU)***. We will take an extensive look at the CPU later in this lecture. In addition, modern consumer laptops, desktop computers, and smartphones including a separate ***Graphics Processing Unit (GPU)***, which we will take a brief look at.

Memory – As the term suggests, memory components remember information. We can generally divide memory into two components: ***primary memory*** and ***secondary memory***.

Primary memory is fast, but ***volatile*** – it loses all information stored in it when the power goes off (a laptop maintains a trickle of energy to the primary memory in sleep mode, but if the battery completely dies, the laptop's primary memory contents will be lost. Primary memory is primarily RAM (Random Access Memory). Secondary memory is much slower and used for permanent, ***non-volatile*** storage. Secondary memory will usually be either a Solid State Drive (SSD) or a Hard Drive (sometimes abbreviated HDD).

Smartphones have a similar divide between primary and secondary memory and include volatile RAM and non-volatile Flash Memory.

We'll study Memory in more depth next lecture.

Input/Output – A variety of devices are used to get information to and from the computer. On a consumer laptop or desktop, these are primarily the keyboard, mouse, and computer display (also sometimes referred to on desktop computers as the computer monitor). In addition, devices like printers and scanners also fall into this category.

Input/Output is commonly abbreviated as ***I/O*** or simply ***io*** (pronounced as "eye-oh").

While all computers have some sort of processing unit, the actual components do vary. The most common computers in the world are neither desktop nor laptop computers; instead they are ***embedded computers*** or embedded systems, which do not have a keyboard or mouse, and in some cases don't even have a display for output. These are the computers that are embedded in mundane everyday devices from microwave ovens, coffee makers, and vending machines up to automobiles and airplanes. Their inputs come from sensor devices, and their outputs are electronic signals, which control the devices in which they are embedded.

Common Tasks on the Computer

- Let's take a look at how some common tasks on the computer relate to the Computer Components we've just listed.

Installing a Program on the Computer

- Typically, the installation files will be compressed, particularly if the application installer is downloaded from the Internet, so they will need to be decompressed before the installation progress can begin.
- The installation process copies the instructions for your new program onto your computer's secondary storage device (e.g., Solid State Drive or Hard Drive).
- At this point, the instructions are now decompressed and in permanent, non-volatile storage.

Running a Program on the Computer

- When we execute a program, the instructions for the program are copied from our secondary storage device into primary memory.
- The instructions must be in primary memory for the CPU to access and execute them.
- In addition, storage space is set aside in primary memory for the program's variables and other information

Saving a File from a Program

- When we create data or documents in our program, the data is stored just in primary memory unless we explicitly save the documents.
- As previously noted, primary memory is volatile and if the power goes out, your data will be lost.
- Saving a file copies the data from primary memory to secondary memory where it is permanently saved.

Opening a File from a Program

- Opening a file from our program copies the data in the file from secondary storage into primary memory where it can be operated on by the program.
- Generally, a program will want all data in primary memory before it can manipulate it.

Exploring the CPU

We will now take a closer look at the CPU. The most important things to take away from this discussion are:

1) The primitive nature of the CPU in comparison to how we view computers (either as programmers or as users).
2) Some understanding of what Machine Language and Assembly Language are.
3) The distinction between High-Level Languages and Low-Level Languages.
4) How Compilers and Interpreters allow us to execute High-Level Languages on a computer that actually only understands Machine Language.

Components of the CPU include:

Registers – Storage in the CPU is limited to a small number of registers. For example, the ARM architecture (used in the iPhone among other places) has 31 general-purpose 64-bit registers (the 32nd is for special purposes only) and 32 registers that are 128-bits for handling floating-point operations.

Instruction Register – A special register stores the binary code of the current instruction that is being executed.

Instruction Counter – A special register keeps track of the current address in Main Memory of the instruction that is currently being executed.

Arithmetic Logic Unit (ALU) – The ALU can take the contents of two different registers and perform either an arithmetic operation on them (such as Add or Multiply) or a Boolean logic operation on them (such as And or Or).

Control Unit – This contains the logic circuitry controlling the other parts of the CPU.

Machine Language

- Each type of computer processor has its own internal machine language. So an ARM processor used in an Apple iPhone speaks an entirely different language than a MacBook Pro, which has an Intel x86 processor in it.
- The actual language of all CPUs is quite primitive. Let's take a look at some sample instructions from the MIPS processor.[1]

Example Arithmetic and Boolean Instructions

- ADD $d, $s, $t
    - Takes the contents of the $s register adds it to the $t register, using the Arithmetic Logic Unit (ALU), and stores the results in the $d register.

[1] The MIPS processor was one of the first commercial RISC processors. We'll see the significance of this later in this lecture. It was developed by John Hennessy, current Stanford Computer Science Professor and Stanford's President from 2000-2016.

- OR $d, $s, $t
    - Takes the contents of the $s register and perform a bitwise OR operation with the contents of the $t register then stores the results in the $d register.

Example Jump and Branch Instructions

- JR $s
    - Jump-based on Register: jump to the address specified in register $s. This changes the next instruction which will be executed to the new memory address specified in register $s.
- BGEZ $s, offset
    - Branch-if Greater than or Equal to Zero: if the contents of register $s are greater than or equal to zero, jump to a new location determined by offset—this changes the contents of the Instruction Counter based on the offset amount, and thus changes the next instruction to execute.
- BEQ $s, $t, offset
    - Branch-if Equal: if the contents of register $s and register $t are the same, then jump to the new location determined by offset.

Example Memory Instructions

- LBU $d, address
    - Load Byte: loads the byte at a given memory address into the $d register.
- SB $s, address
    - Store Byte: takes the least significant byte of the $s register and stores it at the address given.
- LW $d, address
    - Load Word: loads the 32-bits at a given memory address into the $d register.
- SW $s, address
    - Store Word: takes the contents of the $s register and stores it at the address given.

There are more instructions then I have listed here, but this gives a good sense of what they look like. There are arithmetic instructions to subtract, multiple, and divide, for example, as well as arithmetic instructions for the floating point registers; more branch include testing if something is less than zero or greater than zero; and the memory instructions allow transfers in 16-bit chunks in addition to the 8-bit and 32-bit instructions I have listed.

General Operation of the CPU

- Instructions for the current program are stored in Main Memory.
- The Instruction Counter keeps track of the address in Main Memory for the current instruction to execute.

Here is the basic cycle the CPU uses to carry out a program. This is sometimes referred to as the *Fetch-Decode-Execute* cycle:

1) The instruction in Main Memory corresponding to the current value of the Instruction Counter will be fetched from Main Memory and placed in the Instruction Register.
2) The Control Unit contains logic circuitry, which will decode the contents of the Instruction Register to determine the instruction's type and the registers it operates on; it will then send the contents of those registers into the Arithmetic Logic Unit; finally it will send the resulting value to the destination register.
3) The Instruction Counter will be incremented.
4) We go back to 1) where the instruction in Main Memory associated with the Instruction Counter is retrieved from Main Memory.

There are some exceptions. Branching will require changing the instruction counter. Loading and storing operations will use different circuitry than the ALU. However, this gives you a basic idea of how the CPU works.

Machine Language and Assembly Language

There are two languages that work at the CPU level. They are:

Machine Language – This is the actual language of the CPU. It consists of binary codes.

Assembly Language – This is a language used by programmers, which translates directly into Machine Language, with each individual instruction in Assembly having a direct equivalent in Machine Language.

Here's an example:

As a programmer, I might write the instruction:

ADD t1, t2, t3

This means take the contents of register t2 add it to the contents of register t3 and store it in register t1. The equivalent machine language code is:

00000001010010110100100000100000

We can break this down into the following parts

000000 01010 01011 01001 00000 100000

Going from right-to-left the 100000 at the end is the code for ADD. The code for SUB (subtract) would be 100010, the code for DIV would be 011010. The CPU uses these last six bits to determine what operation is being performed.

The next set of bits (again going right-to-left) is 00000. These bits would be used for SHIFT operations (operations shifting bits leftward or rightward within a register) and is not needed for our ADD operation.

The next three sets of bits 01001, 01011, and 01010 designate the registers used by our ADD operation. 01001 is the destination register. The 01011 is the first source register, and the 01010 is the second source register. Note that these register codes are five bits long $2^5 = 32$ and MIPS has 32 registers for integer values.[2]

The final six bits set to 000000 are not needed for basic arithmetic operations.

- Each line of Assembly Code can be translated directly into the binary Machine Code that the MIPS processor actually uses. However "ADD t1, t2, t3" is much easier for a human to write and much harder to make a mistake with than writing "00000001010010110100100000100000".

[2] If you're wondering how t1 translates into 01001 instead of just 00001, MIPS designates different sets of registers for different purposes. The register t1 is designated for storing temporary results and corresponds to register 9, not register 1.

- A human programmer working at this level of programming would write the program in Assembly Code. They would then use a program called an *Assembler*, which translates Assembly Code to its Machine Code equivalent. The Machine Code generated is what would actually be loaded onto the computer.

High-Level Languages and Low-Level Languages

- Computer Scientists distinguish between two types of computer languages:

  Low-Level Languages – Which are Machine Language and Assembly Language.

  High-Level Languages – Which is basically everything else, including whatever you learned to program in. This includes Java, C++, Python, C, JavaScript, anything that isn't either Machine Language or Assembly Language.

- As you may have noticed Machine Languages and their equivalent Assembly Language are missing a lot of things we take for granted. To point out just a few examples:

  - Data Types are very limited. There are no Strings; there are no Characters; and there are no Objects.
  - Control Structures are very limited. There are no "for" loops and no "while" loops.
  - We don't even have Functions, much less Classes and Methods.

- Most programmers do not work at the Assembly or Machine Language level.
  - these languages are very difficult to program with
  - it is very easy to make mistakes when working at this level
  - it takes a lot of code to get even simple things done

- Most programmers program at the High-Level Language level because it's much more productive.

Compilers and Interpreters

- So this raises a question, if the internal computer language of a CPU is Machine Language, and we're doing all our programming in a High-Level Language like Java or C++, how does our program written in a High-Level Language actually get executed by a processor that doesn't understand that language?

- There are two basic methods for executing a High-Level Language.

  Compilation – In this approach, we take the original file written in the High-Level Language, which we refer to as the ***Source File*** or the ***Source Code***. This file is input into a program called a ***Compiler***. The Compiler completely translates the Source Code into Machine Language Binary Code for our target computer. This binary machine code is referred to as the ***Object Code***. (Note that this use of the word Object has nothing to do with the word Object used in "Object-Oriented Programing (OOP)", if you're not familiar with OOP, we'll be discussing it during our Programming Languages lecture later in the quarter.)

  In many cases, there will be multiple Source Files, each which has a corresponding Object Code file. In addition, a program may require one or more library files, which are generally provided as

Object Code files. A program called a ***Linker*** combines the Object Code into a single ***Executable File***. The Executable File is the file that the user needs on their computer to run the program.[3]

You can think of Compilation as similar to Translating a book. For example, I'm reading *Harry Potter and the Philosopher's Stone* in Japanese. JK Rowling wrote the book and gave it to Yuko Matsuoka, the translator. Ms. Matsuoka proceeded to create an entirely new book in Japanese, and that is the copy I am reading. This is what happens with Compilation. We start with a program written in one language and end up with the program written in an entirely different language. the consumer ends up with the machine language version and we keep the source code version to ourselves. We'll see this contrasts with the next approach for executing a High-Level Language.

Languages that are typically compiled include C++ and C.

Interpretation – In this approach, we have a program called an ***Interpreter*** that executes our High-Level Language. The Interpreter reads our High-Level Language program line by line. When it gets to a particular line, it considers what that line means and executes that line, and then it moves to the next line and does the same thing.

Let's look at some examples of how this might work:

- If a line in our High-Level Language file says to declare a variable, the Interpreter reads that line, allocates a storage location for the variable, and makes an entry in a symbol table associating the name of the variable with the storage location.

- If we have a line in our code to add 1 to a variable, the Interpreter uses the symbol table to determine where that variable is stored, retrieves the value of the variable and adds one to it.

No object code is produced. The equivalent Machine Language instructions are not generated. I actually give my users a copy of the original High-Level Language code.

Interpreted languages include JavaScript and Python.

Strengths and Weaknesses of Compilers and Interpreters

- Let's take a quick look at why one might use one of these techniques or the other.

Compilers:

- Compiled code runs much faster than interpreted code. If your objective is execution speed, you'll want to use a compiled language. Most high-end games, for example, are written in C++, which is a compiled language. They want the code to run as fast as possible so that the game runs smoothly.

Interpreters:

- Code written with an interpreted language can run on any platform that has an interpreter for that language. For example, JavaScript is an interpreted language. Every modern web browser acts as a JavaScript interpreter. I can have my web server send the same JavaScript code to a PC, a Macintosh, an iPhone, and an Android Phone.

[3] There are some cases where Object Code files for libraries are not included in the Executable File. Typically, this is because these libraries will be used by a lot of programs, and including the library code in each executable will lead to a lot of duplicate code. However, if the user does not have the library installed, they may be required to somehow get a copy of the library object code.

In contrast, if I were to write C++ code for the web, I would need to send different copies to a PC vs. to an iPhone, since the PC's CPU runs x86 machine code and the iPhone runs a variant of ARM machine code.

- Interpreters do require sending the original High-Level Language code to the end-user. This does make it easier for someone to see exactly how your program works. There are different techniques for making your code more difficult to reverse engineer, but ultimately interpreted code is much easier to reverse engineer than compiled code.

In the programming languages lecture and in the software engineering lecture, both of which will be given later in the quarter, we'll discover some other issues associated with Compiled and Interpreted languages that might cause someone to choose one or the other. However, these are the most important issues.

While most programming languages are designed for one approach or the other, it is possible to develop a compiler for a normally interpreted language or an interpreter for a normally compiled language.

- For example, BASIC traditionally has been interpreted. However, Microsoft's version of BASIC became quite popular with corporate IT departments, so Microsoft developed a compiler for it. This allowed BASIC developers to choose to either continue to run their programs with an Interpreter or take advantage of a BASIC compiler, which allowed their programs to run faster and more efficiently.

A Hybrid Approach (Java Virtual Machine)

- The creators of Java wanted the benefits of *portability* that an interpreted language provided (i.e., the ability to run the same code on many different types of machines), at the same time, they wanted the speed provided by compiled languages.

- They came up with a hybrid approach.
  - They reasoned that one reason interpreters were slow is that the high-level languages that are being interpreted are quite different from the actual machine language run by the interpreter.
  - If the language being interpreted was much closer to machine language, the interpreter would run much faster.

  - They designed a fake computer architecture called the *Java Virtual Machine (JVM).* The language instructions for this fake architecture were essentially at the machine language level.
  - The Machine Language for this new Java Virtual Machine architecture is called *Java Bytecode.*
  - They then built interpreters for Java Bytecode on any machine where they wanted Java to run.

  - The Java source code we write is compiled to Java Bytecode.
  - That Java Bytecode can be distributed to any computer, which has the JVM interpreter code installed on it.
  - Those computers execute the Java Bytecode using the JVM interpreter.
  - The JVM interpreter is faster than a traditional interpreter is because Java Bytecode is very simple and low level.

- This all has had an interesting side effect. Java has been popular enough that most Desktop and Laptop computers have the Java Virtual Machine installed on it.

- New language developers don't have the resources to write compilers or interpreters for all the different types of computer they might want to support.
- Instead, they write a single compiler for their language, which converts their High-Level Language code to Java Bytecode, and their language can now run on any computer, which has the JVM on it.

- When talking about new and upcoming computer languages one natural question to ask is what *platforms* (i.e., what computers) does the computer language run on? One common response is it runs on JVM, which means it runs on any computer that can run Java.

A Hybrid Approach (JavaScript)

- Just as new language developers have taken advantage of the JVM, JavaScript has been used for similar purposes. Any consumer computer runs JavaScript, because all web browsers run JavaScript.
- Some new language developers write compilers, which convert their High-Level Language code to JavaScript. Once in JavaScript it will run on any computer with a web browser.

- The main downside of this approach is that in contrast to Java Bytecode, JavaScript is a high-level language, and interpreting JavaScript code is much slower than interpreting Java Bytecode.

Cross Compilers

- Note that a compiler does not have to run on the actual machine the code is intended to run on. For example, when developing an application for iOS, I would write the code on my Apple Computer which has x86 as its native language. On my Apple Computer, I might run a compiler which generates ARM machine code, the machine code for the CPU of an iOS device. I would then take the machine code output by the compiler running on my Apple Computer and place that code on the iOS device.
- I would not run a compiler directly on the iOS device itself.
- This approach is called cross-compiling.

Multicore Processors

- For many years, processors increased in speed every year. Unfortunately, Electrical Engineers are having difficulty further increasing the speed of the CPU. So as an alternative, we have been getting multiple Cores in each CPU.
- You can think of each Core as consisting of an entirely separate CPU with their own registers, their own Arithmetic Logic Unit, and their own instruction register and instruction counter.

- Having multiple cores doesn't necessarily lead to faster programs however. Let's take a look at some of the issues related to them:
    - Multiple cores do allow for simultaneous execution of different programs. I can assign one core to one program, and a different core to another program. (We'll take a look at how programs are scheduled in another lecture this week.)
    - We can assign *multiple cores to the same program*, but that depends on the program being written to take advantage of multiple cores.

    - Some programs are very easy to divide up into parts that can be run on multiple cores. Some programs that work well for multiple cores include:
        - Web servers with multiple cores can assign different cores to different visitors. The visitors don't generally interact, so the cores can operate more or less independently.
        - Photo processing can assign different cores to different parts of the image. For example, if I have four cores, I can assign one to the top-left, one to the top-right, one to the bottom-left, and the last to the bottom-right.

- Video processing can assign cores to different parts of a movie.
- Spreadsheets, such as Excel, can take advantage of multiple cores.
- Simulations can assign different processors to simulate different parts of the simulation. For example, a weather simulation can assign a processor to simulate one region, and a different processor for another region.
    - In fact, supercomputers, which have many, many CPUs, are often used for simulation work, and the Department of Energy is one of the biggest users of supercomputers. Among other things they use them to simulate nuclear weapons (which allows us to not actually have to run nuclear weapons tests).

- Unfortunately, not all programs can be easily divided to take advantage of multiple cores. While having a few cores is probably useful (because as we'll see next lecture you always have the Operating System running in addition to anything else executing on your computer), the ability for someone to really take advantage of four, six, eight or more cores really depends on the type of programs that you use.

Graphics Processing Units (GPUs)

- Modern desktop, laptop, and mobile devices all include a Graphics Processing Unit (GPU) in addition to the CPU.
- While CPUs with 2, 4, 6, and 8 cores are common, GPUs have many more cores. For example, NVidia's GeForce GTX 960, a consumer-level GPU, has 1024 cores and their high end-consumer models have as many as 3,584 cores as of this writing.

- What is the difference between the many, many cores in a GPU and the much smaller number of cores in a CPU?

- GPU cores have much more limited utility. Typically, we will load data into a GPU and all the cores will perform the same action simultaneously in all their cores.
    - This allows GPUs to perform operations on vectors (1-dimensional arrays) and on matrices.
    - These type of parallel operations are very useful for graphics, but can be used for many other purposes including Neural Networks, Code Cracking, and Bitcoin Mining.
- For the right type of operation, a GPU gives is tremendous power, but it doesn't replace the more general-purpose functionality of the CPU.

RISC and CISC

John Hennessy, current Stanford Computer Science Professor and Stanford's President from 2000-2016, earned the Turing Award (Computer Science's equivalent of a Nobel Prize) in 2018. Let's take a look at the work he earned it for.

As computers became more advanced, they gained increasingly complex instructions. So one might have an instruction that retrieved the contents of a memory address specified in one register, added it to the contents of another register and stored it back in the memory location specified by a third register. This trend toward more complex instructions is called ***Complex Instruction Set Computing (CISC)***.

President Hennessy, along with his co-Turing Award winner Professor Dave Patterson (at Berkeley), concluded that this was the wrong approach. They advocated that instructions should become simpler, not more complex. This approach is called ***Reduced Instruction Set Computing (RISC)***. According to the Association for Computing Machinery (ACM), the primary Computer Science professional organization, 99% of modern CPUs are based on RISC architectures.

A small instruction set allows us to easily optimize and overlap instructions in ways that CISC cannot do without difficulty. Two important innovations that followed were Pipelined and Superscalar CPUs.

Pipelined CPUs are based on the insight that during the execution of a CPU instruction, different parts of the CPU are used at different times. We've previously seen a CPU cycle includes fetching an instruction, decoding an instruction, and executing an instruction. In a traditional approach, while we are fetching an instruction, the decoding circuits and the ALU (Arithmetic Logic Unit) sit fallow; and while the ALU runs, the fetching and decoding circuits are wasted.

Pipelining will overlap these parts to look like something like this:

Instruction Number	Pulse One	Pulse Two	Pulse Three	Pulse Four	Pulse Five
1	Fetch	Decode	Execute		
2		Fetch	Decode	Execute	
3			Fetch	Decode	Execute

Here while we are decoding an instruction, we are simultaneously fetching the next instruction. When we execute an instruction in the ALU, we are decoding the next instruction and fetching the third instruction.

We can continue to overlap all instructions until we hit a condition branch (e.g., an if-statement), which does cause issues with pipelining, since we've already started fetching and decoding an instruction which might not actually be executed depending on what happens with our condition. There are a variety of ways of handling this, which we will not go into in CS106E.

Superscalar CPUs include multiple CPU components, such as two or more ALUs, so that we can simultaneously execute some instructions. For example if I have the code:

```
a = b + c;
x = y + z;
```

A Superscalar CPU would run these both simultaneously as there is no interaction between the two lines of code. We can look at execution of a Superscalar CPU like this:

Pulse One	Pulse Two	Pulse Three	Pulse Four	Pulse Five
Fetch (Unit One)	Decode (Unit One)	Execute (Unit One)		
Fetch (Unit Two)	Decode (Unit Two)	Execute (Unit Two)		
	Fetch (Unit One)	Decode (Unit One)	Execute (Unit One)	
	Fetch (Unit Two)	Decode (Unit Two)	Execute (Unit Two)	
		Fetch (Unit One)	Decode (Unit One)	Execute (Unit One)
		Fetch (Unit Two)	Decode (Unit Two)	Execute (Unit Two)

Additional CPU-Related Terms

Here are some more CPU-Related Terms you may run into:

Microprocessor – Back in the old days, a CPU consisted of many different separate electronic chips connected together. So for example, the CPU registers might be on one (or more) chips, the Arithmetic Logic Unit might be another chip. A Microprocessor was a CPU that was fit entirely on a single chip, really an astonishing achievement at the time. Nowadays all CPUs are Microprocessors.

System-on-a-Chip (SoC) – Just as a Microprocessor took the disparate components of a CPU and put them all on the same chip, an SoC takes different components that are typically found on different chips and places them all on a single chip. For example, the Apple processors powering iPhones such as the A9, A9X, and A10 Fusion are System-on-a-Chips. They all contain both a multi-core CPU and a GPU both on the same physical computer chip.

32-bit Computing vs. 64-bit Computing – As we'll discover next lecture all bytes in the computer are given a numerical number, which is their address. If I set aside 32 bits for each address, I can access 2^{32} different bytes (roughly 4.29 billion or 4.29×10^9). If I set aside 64-bits for each address, I can access 2^{64} different bytes (1.84×10^{19} – if you're not up on scientific notation, 10^{19} is a 1 followed by 19 zeroes – which is to say it's an extremely large number).

Going from CPUs, which handled data and addresses in 32-bit chunks to ones that could handle 64-bit chunks potentially, allows us to write programs which can process much more data. However, we need to distinguish between three separate items:

- What size data does the CPU handle?
- What size data does the Operating System handle (essentially the system software, we'll take a close look at this later this wee).
- What size data does a particular program handle?

We need a 64-bit processor to run a 64-bit Operating System. However, it is still possible to run a 32-bit Operating System on a 64-bit processor.

We need a 64-bit Operating System to run a 64-bit program, however it is still possible to run a 32-bit program using a 64-bit Operating System.

However, if possible, it's best to run a 64-bit program, since that will be able to take advantage of the amount of memory in a modern computer. As we just saw, this does require using a 64-bit operating system, which in turn requires a 64-bit CPU.