Machine Structures Field Guide

Written by Krishna Parashar

Contents

Why This Exists

While taking my Machine Structures class I found it very difficult to conceptually understand and network the plethora of new found concepts. Thus I wrote up this brief synopsis of the concepts I found useful to understanding the core ideas. This is of course by no means **comprehensive** but I do hope it will provide you with a somewhat better understanding computer architecture. Please feel free to email me if you have an questions, suggestions, or corrections. Thanks and enjoy!

Introduction

Okay, so you want to *understand* Machine Structures. But why in heaven's name to you want to take on this rather insurmountable task? I'll take a wild guess you may be forced into this by your universities' curriculum task force -namely your professor. Despite the pain in frustration you *may* go thorugh as you dive deeper, believe or not the ideas in this realm are actually quite useful in your everyday life. In fact the advances we have made in machine structures in the past thirty years are the reason the internet exists in the capacity we have grown to love. Because of this progress you can use things like *parallelism* and *pipelining* run an intensive Google search in milliseconds or execute massive projects like mapping the Human Genome to tailor medical care specifically to you.

The Big Picture

So chances are you have already tried a tiny bit of coding. But how does that virtual code turn in to physical phenomenon? Well let's start of by defining a few ideas in the computing lexicon:

- An **Operating System (OS)** is a interface between a your program and the hardware that manages the resources and ensures you can do things like use a keyboard, store data in memory, and handle many applications at once.

- A **Scripting Language** is probably what you did or want to learn first. Python or Ruby or Java are pretty fun examples. These scripting languages are named so because they try to look like you are writing an essay (that actualy *does* cool things) in a pretty logical and shorthand script. Want to print "hello" in Python? Here it is: `print ("hello")` Pretty easy huh?

- A **High Level Language** is something you may or may not have written before. Lisp, C, C++ are all higher level languages. They may not be as beautiful as the Python code, but boy oh boy can you make the program run really really fast. That same hello statement from Python? Well in C it looks like:

```
char string[] = "Hello World";

printf("%s \n", string);
```

  Not as fun as before. Ah, tradeoffs.

- A **Compiler** is a program that parses (goes through) your complex C or C++ code and turns it into something that's harder for you to read, but easier for a computer to read. FYI though, a **compiler** is an umbrella term that can also mean turning that beautiful Python code to a more complicated C version, but more generally used to mean from C to an Assembly Language.

- An **Assembly Language** is the output from the compiler and looks like a funky short fragments such as:

```
multi $t2 $t1 4
```

  Don't worry if you don't understand what the above means. It is written in a language called **MIPS** that we will discuss later. It is worth mentioning that Intel has a very popular assembly language called **x86**, which can get quite complex and unfortunately will not be discussed in much detail in thsi guide.

- An **Instruction** is each one of those funky little fragments from the compiler.

- An **Assembler** is yet another program that takes in the machine code and interprets that into something the *CPU* can read and execute.

- The **Machine Language** is the output from the assembler looks quite intimidating. Here is an example of what adding two things looks like:

```
1000110010100000
```

  Yep! You guessed it. It's binary! The CPU's structure (discussed lated) need the format to be in just 1's and 0's for reasons in the realm of mathematics and logic. Feel free to look it up, there is a lot of cool information about that.

- A **Binary Digit (Bit)** is well, the each one of those 1's and 0's. Each spot where you can have a digit contains a *bit* so if you have the above machine language output, that would be 16 **bits**. Now you may be wondering, "hey I have heard of a **Byte**, is that the same as a *bit*?" Good question!

- A **Byte** the what you call when you have 8 *bits* together. So if you have the 16 bits of machine language we were talking about earlier, you can alternately say you have 2 **bytes** of machine language. Pretty cool huh? But wait, remeber when you bought you computer and it had 500 GB of space? Thats *just* 500 * 1,000,000,000 (from the *Giga* part) **bytes** or 500 * 1,000,000,000 * 8 (from the byte part) **bits**.

Phew! Now that that's out of the way, we can starting talking about some really useful and brilliant uses of these things. Here is a quick visual summary:
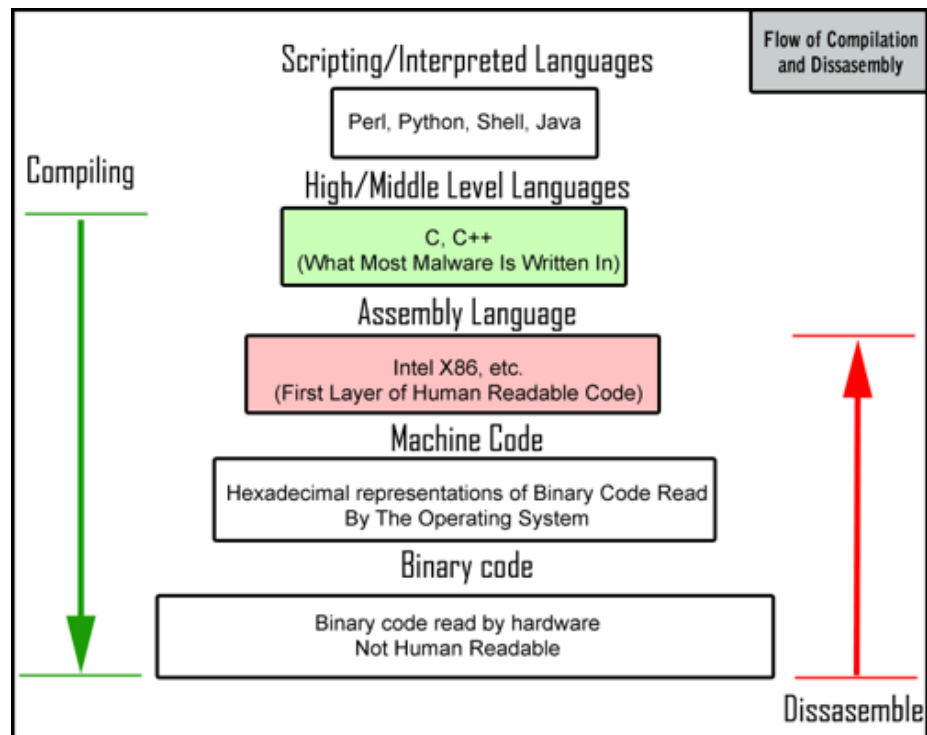

Figure 1: The Big Picture

The Six Great Ideas in Computer Architecture

Thus we come upon the The Six Great Ideas in Computer Architecture (so named due to their greatness). These topics will form the basis for the rest of this guide.

1. **Design processors using Moore's Law** which states processing speed and memory capacity will double every two years; a happenstance that is related to the number of transistors in the chip doubling. It was predicted by Gordon Moore in 1965, and has held roughly true thus far.

2. **Abstract as much as possible in order to simplify the design.**

3. **Design so that the most common case is fast.**

4. **Make dependable systems by using redundancy.**

5. **Use the capacities and speeds of different storage systems to make things fast.**

6. **Improve performance** using techniques such as **Parallelism, Pipelining, and Prediction**.

The Hardware Structure of A Computer

Colophon

Written by Krishna Parashar in Markdown on Byword. Used Pandoc to convert from Markdown to Latex.