Computer Security Field Guide

Written by: Krishna Parashar
Published by: Atrus

All of the following uses the IA-32 (Intel 32-bit systems) Notation
Instructions are formatted at inst src dst

- Registers are prefixed with %
- Constants are prefixed with $
- ($exx) means accessing memory at register %exx
- l suffix for instructions that are 32-bit (long) instructions
- SFP is the saved %ebp on the stack
- OFP is the old %ebp from the previous stack frame
- RIP is the return address on the stack

Registers

There are six general purpose registers: %eax, %ebx, %ecx, %edx, %edi, %esi (%eax stores return value)

1. %ebp - base pointer, indicates start of stack frame (gen const for given function).
2. %esp - stack pointer, indicates bottom on stack (can and does change).
3. %eip - instruction pointer, indicates instruction to run.

Instructions

- mov a, b - copies value of a into b
- push a - pushes a onto the stack (decrements stack, copies value over)
- pop a - pop data from stack on a (copies value over and increments stack)
- call func - pushes address of next instruction onto stack and transfers control to func
- ret - pops return address of next instruction onto stack and transfers control to func
- leave - mov %ebp, %esp then pop %ebs (restores previous stack frame)
- push1 %ebp is part of the prologue for instructions that move the stack pointer to the current top of the stack. You then call movl %esp, %ebp to move $ebp to where %esp is. You do this at the beginning of each function call.
- You push arguments in memory in reverse order.
- Indexes in array are stored with the highest index immediately under the SFP, and lower values under that.

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 through 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 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 actually *does* cool things) in a pretty logical and shorthand script. Want to print "hello" in Python? Here it is: `print ("hello")`! (Beautiful isn't it?)
- 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);
  ```

  (Now it's not as fun as before.)
- 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 this guide.
- An **Instruction** is each one of those funky little fragments from the compiler.
- An **Assembler** is yet another program that takes in the assembly language 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 a collection of 8 *bits*. 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? That 500 GB hard drive? 500 * 1,000,000,000 (from the *Giga* part) **bytes** or 500 * 1,000,000,000 * 8 (from the byte part) **bits**!
- A **Transistor** is the physical representation of the 0 or 1 bit. It is a physical digital *switch*. When it is flipped *ON* (current going through it) it used to mean a *1*, and when it is *OFF* (no current) it represents a *0*.

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:

The Six Great Ideas in Computer Architecture

It is now that 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; an occurrence 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.** This idea relates to what we talked about in the previous section. Split up the responsibility of understanding your code by using a standardized hierarchy (High Level Language -> Assembly Language -> Machine Level) so that each layer takes in an input and passes it down to the next layer. This process makes it infinitely easier to find out where things went or can go wrong.
3. **Design so that the most common case is fast.** This one should be rather intuitive. Instead of wasting time, energy, and of course money trying to optimize so that every part of the program (down to the tiny edge cases) is blazing fast, why not just make the most used cases faster? Otherwise you'll end up with a lot of code that is probably only trivially faster than if you just made the most often used cases faster.
4. **Make dependable systems by using redundancy.** This one is also rather intuitive. Basically you should make backups of the data and repeat the work with other parts of the computer system to ensure everything is accurate and dependable (nothing fails).
5. **Use the capacities and speeds of different storage systems to make things fast.** This is actually one of the main ideas in this guide. We use this principal to optimize the usage of different kinds of memory (fast vs. slow, big vs. small) to cleverly and thriftily use the resources and make our programs fast and light.
6. **Find ways to improve performance** using techniques such as **Parallelism, Pipelining, and Prediction**. The techniques will also be discussed in greater detail later on. The basic idea for each is to have multiple parts of the computer to split up the work (**parallelism**), stage the processes so that no part of the computer has the excuse that it wasn't told what to do (**pipelining**), and lastly try to predict where along the pipeline things might fail and proactively prevent those failures (**prediction**).

With basically these ideas we have managed to come to where computers are today! Pretty impressive, isn't it?

The Hardware Structure of A Computer

Introduction

Now we want to try and understand how a modern computer is structured today. We will choose a relatively simple example, but don't fear if you don't understand what something is. All will come in due time! Thus we begin.
Talk about the components, CPU, SRAM, DRAM, Hard Disks, I/O

Memory Hierarchy

Introduction

One of the most fascinating ideas in the structure of computers is Memory. We know from the previous section that memory is the part of the computer that allows data to be stored and accessed rather quickly. It is a step down from Cache (in the CPU so super fast) and a step up from Hard Disks (tons of Capacity but takes longer to access). Ideally we would want an indefinitely large memory capacity such that any particular data we want would would be immediately available. However in order to reduce costs (memory is very expensive), we have developed a bunch of tricks and techniques to try to optimize the situation and get the most out of this bad bargain.

Memory Hierarchy

So now we come across our first trick *Memory Hierarchy*. This structure (depicted below) that uses multiple levels of memories with different speeds and sizes efficiently in order to present the user with as much memory as is available in the cheapest technology, while providing access at the speed offered by the fastest memory. Unfortunately as the distance from the processor increases, both the size and access time of memories increase.
Data is similarly hierarchical; a level closer to the processor is generally a subset of any level further away, and the entirety of the data is stored at the lowest level (generally a hard drive)

Principle of Locality

So in order to allow programs to run efficiently, a technique we take advantage of is the *Principles of Locality*. Programs accesses a relatively portion of their address space at any instant of time. We take advantage of the *Memory Hierarchy* from above. There are two different types of locality:

1. **Temporal Locality** (locality in time) *is the ideas that if an item is referenced we will assume it will tend to be referenced again soon.* An Example of this is *Loops* in which that variables and functions are likely to be accessed repeatedly so we save them closer to the processing.
2. **Spatial Locality** (locality in space) *is the idea that if an item is referenced, items whose addresses are close by will probably be referenced sometime soon.* Again we can use loops as an example, since we know that instructions are normally accessed sequentially so we can predict the next iteration and store it when we access the previous iteration. We can also think of this using array elements. We can save the nearby array elements when we get one of the elements with hopes that it will come in handy. It may seem a bit counterintuitive but it really does work!

In short by using these two ideas (*Memory Hierarchy* and the *Principle of Locality*) uses smaller and faster memory technologies close to the processor. Thus, accesses that hit in the highest level of the hierarchy can be processed quickly. Accesses that miss go to lower levels of the hierarchy which are larger but slower.

Memory Access <- NEEDS WORK

A memory hierarchy usually consist of many different levels as shown in the *Memory Hierarchy* section. However Data can only be copied between only two adjacent levels at a time. Thus to show how this data is accessed we will show how one particular instance between two levels is done.
Block (Line): The minimum unit of information that can be either present or not present in cache. This is shown in the image above as the red square.
Every pair of levels in the memory hierarchy can be thought of as having an:

- **Upper Level**:
  - Closer to the processor
  - Smaller, faster, and more expensive than the lower level
  - Data in this level is generally a subset of the lower level
- **Lower Level**

SRAM Technology

- Static Random Access Memory
- Integrated circuits that form memory arrays
- Usually has a single access port that can provide either a read or a write
- Fixed access time to any datum (read/write access times may differ)
- Typically uses 6-8 transistors per bit to prevent information disruption when read
- As long as power is applied the value can be kept indefinitely

DRAM Technology

- Dynamic Random Access Memory
- Value keep in a cell is stored as a charge in a capacitor
- Single transistor is used to access stored charge (either to read or overwrite the stored charge)
  - Because it uses one transistor per bit of storage, it is much denser and cheaper than SRAM
- Stores charge on a capacitor: it cannot be kept indefinitely and must be periodically refreshed
- *Refresh*: contents from an entire row is read and immediately written back to the same row

Performance Specifications <- Why? Seems Unnecessary.

- Buffer Rows:
  - Acts like a SRAM: changing the address enables random bits access to be accessed until the next row is accessed
- Wider Chips:
  - Improves memory bandwidth
- Organization:
  - Modern DRAMs are organized in banks
  - *Bank*: Series of rows
    - Pre-charge opens/closes a bank
  - DDR3: 4 banks
  - *Act* : Signal sent with row addresses that activates the transfer of the row to the buffer
  - When row is in the buffer, it can be transferred by
    - Successive column addresses at whatever the width of the DRAM is (typically 4, 8 or 16)
    - Specifying a block transfer and the starting address
- Clocks Added:
  - SDRAM (Synchronous DRAM)
  - Eliminates synchronization time between memory and processor
  - Speed Advantage: transfers bits in the burst without having to specify additional address bits
  - DDR SDRAM (Double Data Rate):
    - Data transfers on both the rising and falling edge of the clock (twice bandwidth)
    - Latest version: DDR4 can do 3200 million transfers per second (1600 MHz clock)
- Address Interleaving
  - Instead of just a faster row buffer, DRAM can read from or write to multiple banks, each having its own row buffer
  - Accesses rotation: Enables sending addresses to several banks to read/write simultaneously
  - Bandwidth = Bandwidth x (# of banks)

Flash Memory

Disk Memory

- Wear Leveling
  - Like other EEPROM technologies, writes can wear out flash memory bits
  - To handle this a controller is used to spread the writes by remapping blocks that have been written many times to less used blocks
  - With this technology mobile devices are very unlikely to exceed flash's write limits
  - To improve performance even more, incorrectly manufactured memory cells are mapped out
  - Wear Leveling lowers flash's potential performance, but is needed unless higher-level software monitors block wear
- Magnetic Hard Disks consist of a collection of platters
- Metal platters are covered with magnetic recording material on both sides
- Track: One of thousands of concentric circles that makes up the surface of a magnetic disk
- Sector:
  - One of the segments that make up a track on a magnetic disk
  - The smallest amount of information that is read/written on a disk
- Read/Write mechanism
  - Read-Write Head: Movable arm containing a small electromagnetic coil
  - Each surface has one arm containing two RW-Heads, one facing up and one facing down

Virtual Memory

Caches

Structure

Registers, Etc.

Hitting and Missing <- NEEDS WORK

We know from the *The Hardware Structure of A Computer* section that caches live in the CPU and live near the top of Memory Hierarchy pyramid (registers are faster). So before we move on and discuss caches, we want to clear up some important terminology.

- *Hit*: When data requested by the processors is found in some block in the upper level
- *Hit Rate*: Fraction of memory accesses found in a level of the memory hierarchy
- *Hit Time*: Time required to access a level of the memory hierarchy, including the time needed to determine whether the access time is a hit or a miss
- *Miss*: When data requested by the processor is not found in the upper level -> lower level accessed to retrieve the block containing the requested data
- *Miss Rate*: Fraction of memory accesses not found in a level of the memory hierarchy (1 - Hit Rate)
- *Miss Penalty*: Time required to fetch a block from a lower level of the memory hierarchy, including — time to access the block — transmit it from one level to the other — insert it in the level that experienced the miss — pass the block to the requestor ### AMAT

Direct Mapped Caches

Multilevel Caches

Set Associative Caches

Machine Instructions

Introduction

MIPS

Basics MIPS stuff e.g. how it works, instructions, link to green sheet, and uses Converting Binary to MIPS, Parity Bits, and those table things and everything. CS61C Winston's Discussion will be helpful.

Binary Representations

Introduction

Signed, Unsigned, Two's Complement, Hex, Conversions, Extensions etc.

Hardware Level

Introduction

Transistors and Logic Gates

Boolean Algebra

Assembly to Machine, Logic Gates, Useful Processor Stuff

The CPU

Structure

Purpose? Multiprocessing ALU, shifters, Reg files, Muxing, demuxming, FSM ### Performance CPU Time, CPI, Instruction Count, Clock Cycle time?

Code Optimization Techniques

Introduction

Cache Blocking

Pipelining

Lead to next section : Parallelism

Parallelism

Introduction

Amdahl's Law

Request Level Parallelism

Application: MapReduce

Data Level Parallelism

Flynn Taxonomy

Thread Level Parallelism

Shared Memory

Application: OpenMP (?)

Instruction Level Parallelism

Application: Warehouse Scale Computing

Redundancy

ECC, RAID, Stuff from Professor Katz's lecture.

Virtual Machines (?)

Colophon

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