# Introduction to Systems Programming and Computer Organization

CPSC 323, Yale University, Fall 2018

These are lecture notes for CPSC 323a, "Introduction to Systems Programming and Computer Organization," taught by Stanley Eisenstat at Yale University during the fall of 2018. These notes are not official, and have not been proofread by the instructor for the course. These notes live in my lecture notes respository at

https://github.com/jopetty/lecture-notes/tree/master/CPSC-323.

If you find any errors, please open a bug report describing the error, and label it with the course identifier, or open a pull request so I can correct it.

## Contents

# Syllabus

| | |
|---|---|
| **Instructor** | Stanley C. Eisenstat, sce@cs.yale.edu |
| **Lecture** | MW 1:00–2:15 PM, DL 220 |
| **Recitation** | TBA |
| **Textbooks** | John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* 6th ed. Morgan Kaufman, 2017; Neil Matthew and Richard Stones. *Beginning Linux Programming.* 4th ed. Wrox, 2007 |
| **Midterms** | Monday, October 15, 2018<br>Wednesday, December 5, 2018 |

## Coursework

The class will NOT meet during reading period. There will be 6 assignments requiring an average of 6-9 hours per week (or an average of 15-25 hours per C assignment, somewhat less per non-C assignment). There will be an in-class examination on Monday, October 15th, and a final on Wednesday, December 5th. Homework will constitute ∼70% of the final grade; the examinations will constitute the remainder.

Programs will be submitted electronically and checked using test scripts: a public script, which will generally be available at least one week before the assignment is due; and a more comprehensive private script, which will be used to assign a grade. C programs may also be evaluated for "style".

Programs should be submitted electronically by 2:00 AM on the day specified in the assignment. Late work not authorized by a Dean's excuse will be assessed a penalty of 5% per calendar day or part thereof and MAY not be graded at all if more than ten days late or after solutions are released. The submit-times of the sources determine when the program was completed.

## Schedule

| What | Points | Due | Spec | Script | Program |
|---|---|---|---|---|---|
| Homework #1 | 60 | 09/14 (F) | 08/29 | 09/07 | farthing (C) |
| Homework #2 | 50 | 09/28 (F) | 09/12 | 09/21 | bshParse (C) |
| Homework #3 | 30 | 10/12 (F) | 09/26 | 10/05 | cpp2018 (script) |
| Homework #4 | 60 | 11/02 (F) | 10/10 | 10/26 | LZW2018 (C) |
| Homework #5 | 60 | 11/16 (F) | 10/31 | 11/09 | bshShell (C) |
| Homework #6 | 40 | 12/13 (R) | 11/14 | 12/06 | Networking (script) |

## Topics Covered

**Systems programming in a high level language:** user-level interfaces to a typical operating system (Linux), writing programs (e.g., a shell) that interact with the operating system;

**Elementary machine architecture / computer organization:** computer arithmetic and general structure/organization of machines, approaches to parallelism (vector, SIMD, MIMD, networks), instruction set architectures and pipelining instruction execution;

**Operating systems:** implications of concurrency, implementation of semaphores at machine level (a la Dijkstra), implementation and ramifications of virtual memory and caches;

**Other:** data compression, error detection and correction, computer networks.

# References

[1] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. 6th ed. Morgan Kaufman, 2017.

[2] Neil Matthew and Richard Stones. *Beginning Linux Programming*. 4th ed. Wrox, 2007.

# 1 August 29, 2018

The beginning of this lecture was mostly administrata. Stan discussed the structure of the course and the various grading policies. Of particular importance are the facts that, unlike in CPSC 223, the public test cases for our assignments will only cover $\sim 25\%$ of the private ones. If you read between the lines of the public test cases, you can probably guess another 25% of the private tests, and a careful reading of the entire program specification will net you almost all of the private test cases, so long as your logic holds. It's also important to note that 323 has a "line allowance" for code, usually around 180 lines (after all the comments and lone braces have been remove). This means that if your code is 300+ lines, it might be best to simply scrap the assignment and start from scratch since you probably aren't implementing it in the correct or most efficient way.

## (Binary) Number Representation

Stan began the actual academic part of the lecture by discussing how arithmetic works on an arbitrary binary machine. Because this is computer science and not mathematics, we are often constrained by limits that do not constrain mathematicians. Notably, we have to worry about whether or not a number is representable by our machine.

*Representable Numbers*

> **Definition 1** (Representable Numbers)**.** The *representable numbers* is the set of all possible bit patters for a given level of precision. An element of this set is said to be representable since it can be stored in a computer's memory and can be operated upon. Since we are discussing classical computers in this course, the representable numbers are a proper subset of $\mathbb{R}$. In practice, representable numbers are either integral (and so they are a proper subset of $\mathbb{Z}$) or they are floating-point numbers with a fixed degree of precision, and so there exists an injective map between these representable numbers and $\mathbb{Z}$.

Unlike in mathematics, representable numbers are not closed under the standard arithmetic operations, and so the result of arithmetic expressions is not guaranteed to be representable. In practice, programmers give special names to the ways in which the representable numebers fail to be closed under these operations:

- Overflow: When $a \star b$ is too large to be represented. Consider `INT_MAX` $+ 1$;
- Underflow: When $a \star b$ is too small to be represented. Consider $1/10^{30}$, which is approximately zero but definitely not equal to zero.

- Inexactness: When $a \star b$ requires too much precision. Consider $1/3 = 0.33333333\ldots$, which naïvely requires an infinite precision to represent accurately.

This leads to expressions which break the traditional axioms of field mathematics. Namely,

- Operations need not be associative. Consider that if $a < 0$ and $b+c >$ `INT_MAX`, then $a + (b + c) \neq (a + b) + c$;
- Operations need not be distributive. Compilers can expand code in mysterious ways and this can bite you, especially with floating point multiplication;
- It's not guaranteed that $x < y \iff x - y < 0$, since $x - y$ may not be representable.

Incongruency between the worlds of Math and CS leads to losses in the billions of dollars. The moral of the story is that it's important and not trivial to check that our software works the way we think it ought to. Never assume that machine arithmetic is correct.

### Nonnegative Numbers

Consider a number $N$, represented with $m$ bits of precision by

$$N = \beta_{m-1}\beta_{m-2}\cdots\beta_2\beta_1\beta_0 \equiv \sum_{i=0}^{m-1} \beta_i \cdot 2^i,$$

where $\beta_i$ are the place value digits of $N$.

If $\beta_0$ is on the right, we say the representation is *Little Endian*. Otherwise, we say that it is *Big Endian*. Notice that if $N \geq 2^m$, then $N$ is not representable since it is too big.

### Nonnegative Numbers in the C Standard

For the scope of this discussion, we'll assume that all types today are *unsigned*, so that $1 +$ `INT_MAX` $= 0$. Here are the several integral types in C, along with their size on the Zoo machines and their specified size in the C standard.

*The Zoo is Yale's network of UNIX machines used by the CS department*

| Type | Zoo Size | C Standard Size |
|---|---|---|
| `char` | 8 bits | $\geq 8$ bits |
| `short` | 16 bits | $\geq 16$ bits |
| `int` | 32 bits | $\geq 16$ bits |
| `long` | 64 bits | $\geq 32$ bits |
| `long long` | 64 bits | $\geq 64$ bits |

C also requires that each of these be less than or equal to the one succeding it, so

$$\texttt{char} \leq \texttt{short} \leq \texttt{int} \leq \texttt{long} \leq \texttt{long long}.$$

If you want to write portable code, never violate these limits regardless of your machine's implementation. There is a hierarchy of how portable your code is.

- Portable: Will run the same on anything.
- Transportable: Use `#define/typedefs` to mix your code around based on the machine. You can also include `limits.h` and `stdint.h` which define custom types with specific bit lengths, like `int_64` for an integer which is guaranteed to have 64 bits of precision.

Ideally, your code will be portable, but that can be very difficult since there are some *odd* machines out there which will fuck with your program and with common sense in general.

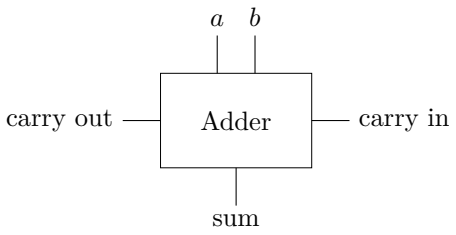
## Operaitons on Nonnegative Numbers

Some quick key highlights about operations on representable numbers:

- Subtraction leads to negatice numbers very quickly
- Multiplicaiton is repeated addition and bit shifting if you don't care about efficiency. Otherwise, there are more advanced multiplication algorithms.
- Division is repeated subtraction and shifting, so it caries the same caveats as multiplication.
- Therefore, we'll focus mostly on addition.

### Addition of Nonnegative Numbers

Addition is based on a type of circuit known as a *Full Adder*, which we'll treat as a black box. There are two value inputs, a carry input, a carry output, and a sum output. Each input and output is one bit in size.

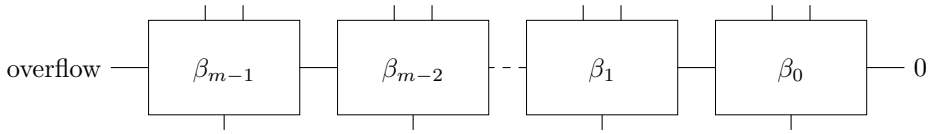



Here, the "carry in" bit tells us whether or not the previous sum resulted in a magnitude large enough that we must "carry" a digit over to the current sum.

Similarly, the "carry out" bit tells the next calculation whether the current sum was too large to fit in a single bit. The bits $a$ and $b$ are the input bits of the two numbers we're adding, and "sum" is their sum. Mathematically,

$$\text{sum} = a \oplus b \oplus \text{carry in}.$$

In a half adder, carry in is assumed to be zero. To add two $m$ bit numbers, we use a *Ripple Carry Adder*, which is just $m$ full adders daisy-chained together, connecting the "cary out" bit of the previous sum to the "carry in" bit of the next. This is a very




simplistic way to add numbers — it's exactly the algorithm learned in grade school for adding two numbers together. The final carry out bit determines whether or not the number will overflow — that is, is the number representable by our machine. Unfortinately, it's not particularly efficient; The time and space complexity of the ripple carry algorithm is $\mathcal{O}(m)$.

**Fast Adders**

Of course, we could do the whole process with only one adder if we store the intermediate results, but that doesn't get us any improvement in the time or space complexity. A change in thinking is needed if we want to find a faster method of adding. If we think about it for a bit, we realize that the slow part of the algorithm isn't the part which adds the bits together; rather, it's getting each carry sent off to the next circuit. If we could somehow know the carries ahead of time, the entire problem could be done in parallel in constant time.

In some cases, we can determine the "carry out" of a given digit without considering what the "carry in" bit is. For example, if $a_i = b_i = 1$, than "carry out"$_i = 1$ regardless of the "carry in". We can use this to build a divide-and-conquer approach which has significant time advantages over the ripple carry adder, but pays a heavy price in terms of space. The time for this is $\text{Time}(m/2) + 1 = \mathcal{O}(\log_2 m)$, but the space is $3 \cdot \text{Space}(m/2) + \mathcal{O}(m) = \mathcal{O}(m^{\log_2 3})$. Since $\log_2 3 > 1$, the space complexity grows superlinearly in $m$.

# 2 August 31, 2018

## Linux File System

The Linux file system has a user interface which consists of a hierarchical name structure based in a root directory, usually called `/` or `root`. There are two kinds of names, *absolute* and *relative*. Absolute names, like `/c/cs323/Hwk1/...`, always begin with a `/` in the root directory, and are often quite long. Relative names begin with some character other than `/`, and is implicitly prefixed by the name of the current working directory. This directory can change based on how you're using the computer. These make names much shorter. We can change the current working directory using the `cd` bash command, or the `chdir()` syscall in C.

A *file* is a stream of bytes with *no* implied formatting. Each file has associated with it a unique number called an *inode number* which is an index into an array of structs. There is one array per file system. The inode structs have several fields, including

- the `length` of the file in bytes;
- the `type` of the file/directory/symbolic links/devices/named pipes/etc;
- the `userid` of the owner of the file;
- the `groupid` of the file group;
- `protection` bits, essentially the read/write/exec permissions — there are separate protection bits for the owner, group, and everyone else;
- `timestamps` for the access, create, and modify times of the file;
- the number of `hard links` to the file;
- a block of pointers to the the disk block.

Notice that there is no name! In Linux, files don't really have absolute names. Instead, we assign names to files using *directories*. A directory is a special file containing ordered pairs of an inode number and a name which doesn't contain a slash[1]:

`(inode #, name)`

[1] *slashes are used to separate directories*

These pairs are called *hard links*. In an inode struct, if the number of hard links ever reaches zero, then the file is "deleted." This gives the relation

`(inode #, name)` $\in$ `dir` $\iff$ `dir/name` is a valid name for the file.

The operating system is responsible for updating the number of hard links for every file. All of this really means that there isn't a one-to-one correspondence between inodes (files) and names.

In every directory, there are two special files: `.`, which is the current directory, and `..`, which is the parent directory. These let us walk around the file system.

## Links

We know that a hard link is just a pair of an inode and a local file name. The `rm` command removes the hard link from the directory and decrements the inode's hard link count. The command `ln` does the opposite. These are mirrored by the syscalls `link()` and `unlink()`.

There are also *soft links*. The contents of a soft link inode is a string containing a semantically valid file name. They are created by the `ln -s` command or the `symlink()` syscall. As an example, on the Zoo `ln -l /c` reveals that the `/c` directory is actually a soft link to the directory `/home/classes`.

## Directories

If you want to create a directory, use the `mkdir` command or the `mkdir()` syscall. To remove one, issue the `rmdir` command or the `rmdir()` syscall. To read a directory, use the `ls` command or several related commands, as in `opendir()`, `readdir()` $\times n$, `closedir()`.

## Machine Arithmetic, Round Two

[fill in an explanation of the fast adder from last lecture, or maybe just put it in Lec 1.]

The time to add two $m$ bit numbers is $\mathrm{Time}(m) = \mathrm{T}(m/2) + \mathcal{O}(1)$, where we have logarithmically many called for $\mathrm{T}(m/2)$ to do all of the calculations, so our total time is $\mathcal{O}(\log_2 m)$. This is great! That's the best we could hope for!

Unfornately, our space is $\mathrm{Space}(m) = 3 \cdot \mathrm{S}(m/2) + \mathcal{O}(m)$, since we have three distinct adders for each $m/2$ bit numbers, which is actually $\mathcal{O}(m^{k \log_2 3})$ for $k$ repeated additions. Since $k \log_2 3 > 1$, this means that our space is growing superlinearly. This isn't so great.

## Carry Look-Ahead Adder

Let $c_k$ be the carry-in bit for the $k$ bit position, which is the carry-out for the $k-1$ bit position. Then $c_0 = 0$ is the initial carry in and $c_m$ is the overflow bit. The following relation is true

$$c_{k+1} = (a_k \wedge b_k) \vee (a_k \vee b_k) \wedge c_k.$$

Let $g_k = a_k \wedge b_k$ be the generator, and let $p_k = a_k \vee b_k$ be the propogator. Then we can write this more generally as

$$c_k = g_k \vee p_k \wedge c_k.$$

As it turns out, we can calculate all $g_k$ and $p_k$ in parallel in only a single gate delay. This means that we can compute all carries in

$$\underbrace{1}_{pkgk} + \underbrace{2m}_{g_k \vee p_k \wedge c_k} + \underbrace{1}_{computesk} \quad \text{steps.}$$

Can we do any better?

## Recursive Doubling

Suppose I want to calculate $c_{2\ell+2} = g_{2\ell+1} \vee p_{2\ell+1} \wedge c_{2\ell+1}$. Notice that $c_{2\ell+1} = g_{2\ell} \vee p_{2\ell} \wedge c_{2\ell}$. If we treat this logic like arithmetic, we can simplify the above into

$$\begin{aligned} c_{2\ell+2} &= g_{2\ell+1} \vee p_{2\ell+1} \wedge c_{2\ell+1} \\ &= (g_{2\ell+1} \vee p_{2\ell+1} \wedge g_{2\ell}) \vee (p_{2\ell} \wedge p_{2\ell+1} \wedge c_{2\ell}), \end{aligned}$$

which can be computed in $6 + m = \mathcal{O}(m)$ steps. *fix this section up a lot*