CS429: Computer Organization and Architecture

Introduction

Dr. Bill Young
Department of Computer Science
University of Texas at Austin

Last updated: January 22, 2020 at 14:37

Acknowledgement

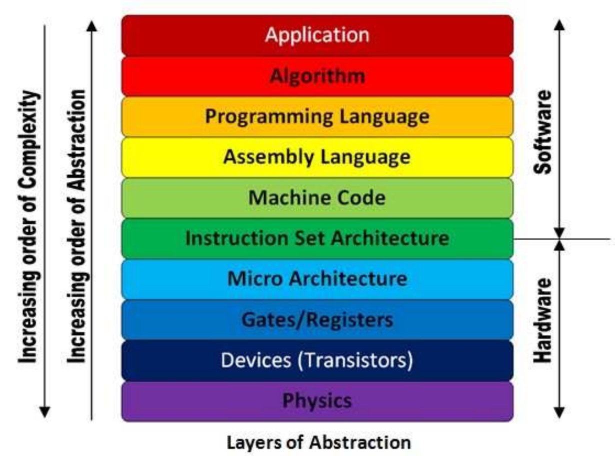
The slides used this semester are derived from slides originally prepared by the textbook authors, Randall Bryant and David O'Hallaron.

They were modified with permission and reformatted for use in our class. *You should consider them as copyright material; do not repost them anywhere.*

- Theme of the course
- Great realities of computer science
- How this class fits into the CS curriculum

Abstraction is our Friend

Most of what we study in Computer Science is really about a hierarchy of abstractions! *Without abstraction, we wouldn't be able to accomplish much.*



Abstraction vs. Reality

Abstraction is good, but don't forget reality!

Most of your courses to date have emphasized abstraction.

- High level programming languages
- Abstract data types
- Asymptotic analysis

These abstractions have limits!

- Especially in the presence of bugs
- Need to understand underlying implementations
- Need to have a working understanding of architecture

Desired Outcomes

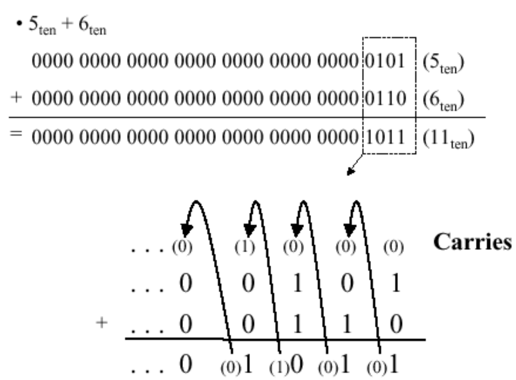
Useful outcomes!

- Know "stuff" that all computer scientists should know
- Become more effective programmers
  - Able to find and eliminate bugs efficiently
  - Able to tune program performance
- Prepare for later "systems" classes: Compilers, Operating Systems, Networks, Computer Architecture, Embedded Systems, many others.

Hint: Hang onto your book. You'll be using this same book (3rd edition) in CS439.

Great Reality 1

Ints are not Integers; Floats are not Reals.

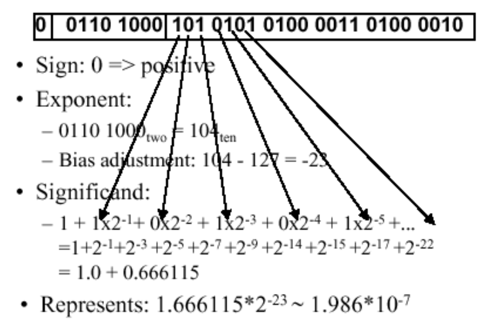


Is $x^2 \geq 0$? For floats, yes. For ints, not necessarily.

$$40000 * 40000 \rightarrow 1600000000$$

$$50000 * 50000 \rightarrow ??$$

Floats are not Reals



Is $(x + y) + z = x + (y + z)$? For int's: yes. For floats, maybe not.

$$(1e20 + -1e20) + 3.14 \rightarrow 3.14$$

$$1e20 + (-1e20 + 3.14) \rightarrow ??$$

Treat CS as an Experimental Science

Get into the habit of writing programs to experiment with the architecture:

```
void main() {
  printf("40000 * 40000 = %d\n", 40000 * 40000);
  printf("50000 * 50000 = %d\n", 50000 * 50000);
  printf("1e20 + (-1e20 + 3.14) = %f\n", 1e20 + (-1e20
     + 3.14));
  printf("(1e20 + -1e20) + 3.14 = %f\n", (1e20 + -1e20)
      + 3.14);
}
```

```
> gcc tester.c
> a.out
40000 * 40000 = 1600000000
50000 * 50000 = -1794967296
1e20 + (-1e20 + 3.14) = 0.000000
(1e20 + -1e20) + 3.14 = 3.140000
```


Computer Arithmetic

Computer arithmetic does not generate random values. Arithmetic operations have important mathematical properties.

But not the "usual" properties of arithmetic.

- Due to finiteness of representations.
- Integer operations satisfy "ring" properties: commutativity, associativity, distributivity.
- Floating point operations satisfy "ordering" properties: monotonicity, values of signs.

Observation:

- Need to understand which abstractions apply in which contexts.
- Important issues for compiler writers and serious application programmers.

Great Reality 2

Computer scientists should understand assembly language!

You won't often program in assembly. Compilers are much better at it and more patient than you are.

Understanding assembly is key to understanding what *really* happens on the machine.

- Behavior of programs in presence of bugs; high-level language model breaks down.
- Tuning program performance and understanding sources of program inefficiency.
- Implementing system software
  - Compiler has machine code as target
  - Operating systems must manage process state
- Creating / fighting malware: x86 is the language of choice for attackers.

Great Reality 3

Memory Matters!

Memory is not unbounded!

- It must be allocated and managed.
- Many applications are memory dominated.

Memory referencing bugs are especially pernicious. The effects may be distant in both time and space.

Memory performance is not uniform.

- Cache and virtual memory effects can greatly affect program performance.
- Adapting your programs to characteristics of memory system can lead to major speed improvements.

Memory Referencing Bug Example

```
double fun(int i)
{
   int a[2];
   double d[1] = {3.14};
   a[i] = 1073741824;
   return d[0];
}
```

Assume x86 (double is 8 bytes; int is 4 bytes). This will be different on other systems, and may cause segmentation fault on some.

Call	Result
fun(0)	→ 3.14
fun(1)	→ 3.14
fun(2)	→ 3.1399998664856
fun(3)	→ 2.00000061035156
fun(4)	→ 3.14, then segmentation fault

What can you infer about how the memory is laid out?

Memory Referencing Bug Explanation, Little Endian

```
double fun(int i)
{
   int a[2];
   double d[1] = {3.14};
   a[i] = 1073741824;
   return d[0];
}
```

Modified	Call	Result
$a[0]$	fun(0)	→ 3.14
$a[1]$	fun(1)	→ 3.14
$d_3 \dots d_0$	fun(2)	→ 3.1399998664856
$d_7 \dots d_4$	fun(3)	→ 2.00000061035156
saved state	fun(4)	→ 3.14, then seg fault

Memory Referencing Errors

C and C++ do not provide much memory protection.

- Out of bounds array references
- Invalid pointer values
- Abuses of malloc/free

This can lead to nasty bugs.

- Whether or not bug has any effect depends on system and compiler.
- Action at a distance
  - Corrupted object logically unrelated to one being accessed.
  - Effect of bug may be first observed long after it is generated.

How can I deal with this?

- Program in Java, Lisp, or ML
- Understand what possible interactions may occur
- Use or develop tools to detect referencing errors

Memory Performance Example

The following copies an $n \times n$ matrix:

```
/* ij */
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    b[i][j] = a[i][j];
  }
}
```

This one computes precisely the same result.

```
/* ji */
for (j=0; j<n; j++) {
  for (i=0; i<n; i++) {
    b[i][j] = a[i][j];
  }
}
```

But the performance may be much (could be 10X) slower, particularly for large arrays. Can you guess why that may be?

Great Reality 4

There's more to performance than asymptotic complexity.

Constant factors matter too!

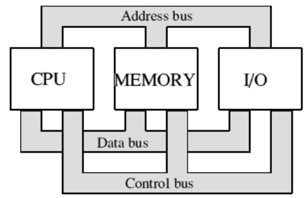
- Even an exact op count does not predict performance.
- Easily see 10:1 performance range depending on how code is written.
- Must optimize at multiple levels: algorithm, data representations, procedures, and loops.

Must understand the system to optimize performance.

- How programs are compiled and executed.
- How to measure program performance and identify bottlenecks.
- How to improve performance without destroying code modularity and generality.

Great Reality 5

Computers do more than execute programs.



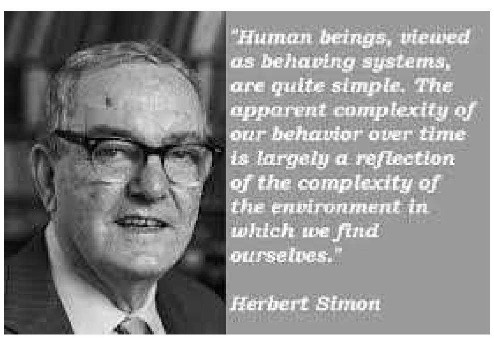
They need to get data in and out. The I/O system is critical to program reliability and performance.

They communicate with each other over networks. Many system-level issues arise in the presence of networking.

- Concurrent operations by autonomous processes
- Coping with unreliable media
- Cross platform compatibility
- Complex performance issues

Great Reality 6 (I Added this One)

Computers do a lot with very simple primitives.

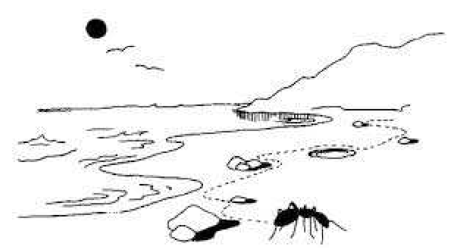


Nobel Prize winner (in Economics) Herbert Simon used an ant to explain how simple actions can explain complex results. (*Sciences of the Artificial*, 1969)

Imagine an ant walking along a beach. You notice that the ant is tracing a very intricate path. Must be executing a pretty complex algorithm, right?

Simon's Ant

Actually not! If you look closer, you notice that there are small pebbles in the ant's path. The ant responds to each by turning either right or left, possibly randomly.



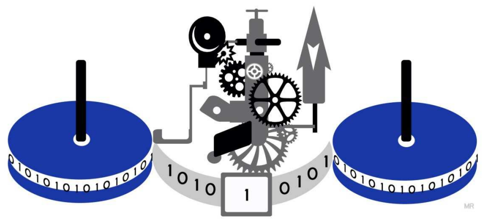
Lesson: You can generate very complex results using only very simple tools.

Church-Turing Thesis

A **Turing Machine** is a very simple computing device that can look at a symbol on a tape, write another symbol, and move right or left one square, under the direction of a simple program.

Everything that can be computed can be computed by a Turing Machine.

The most powerful computer you'll ever use in your life is *no more powerful than a Turing Machine.* We say that a machine is **Turing complete** if it can emulate a Turing machine.



Course Perspective

Most systems courses are "builder-centric."

- Computer Architecture: Design pipelined processor in Verilog.
- Operating Systems: Implement large portions of operating system.
- Compilers: Write compiler for simple language.
- Networking: Implement and simulate network protocols.

Course Perspective

This course is programmer-centric.

- The purpose is to show how by knowing more about the design of the underlying system, one can be more effective as a programmer.
- Enable you to
  - Write programs that are more reliable and efficient
  - Incorporate features that require hooks into OS: concurrency, signal handlers, etc.
- Not just a course for dedicated hackers. We bring out the hidden hacker in everyone.
- Cover material in this course that you won't see elsewhere.

Our Subject: Computer Organization