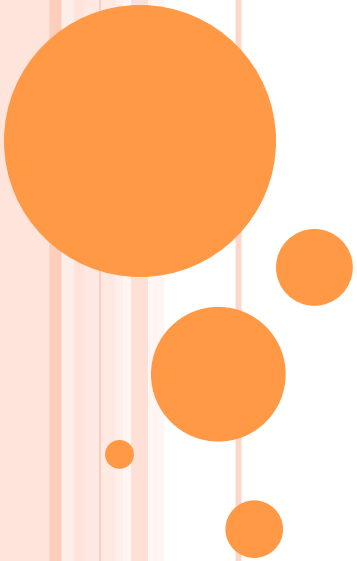


CS 24: Introduction to Computing Systems

Spring 2015

Lecture 1

Welcome to CS24!

- Introduction to Computing Systems

- How do modern digital computers work?
- What features, capabilities, and optimizations do processors provide?
- How do we translate programs to run on processors?
  - e.g. intermediate values, looping, subroutines, recursion
- How to provide common runtime support?
  - e.g. memory management
- What do operating systems do for us?!
  - e.g. process isolation, virtualization, input/output

CS24 Administrivia

- Course website: Caltech Moodle
  - http://courses.caltech.edu
  - Go to the CS section, then click CS24 (key = segfault)
- **Make sure to enroll in CS24 course today!**
  - Class announcements are made via Moodle
- All lectures, assignments posted on CS24 Moodle
  - Submit homeworks and receive grades via Moodle
  - (I will keep track of your overall grade separately)
- Assignment grading guidelines will be posted
  - **Correct programs are not sufficient!**
  - Style, clarity, commenting, etc. are also important

CS24 Administrivia (2)

- Approximate course weighting:
  - 8 assignments (70% of grade)
    - Each assignment is 8.75% of your grade
  - Midterm (15% of grade)
  - Final exam (15% of grade)

- I will sometimes curve individual assignments or exams, depending on the class' average grade
  - Tends to be done infrequently
  - On average, people do very well on assignments

CS24 Late Policy

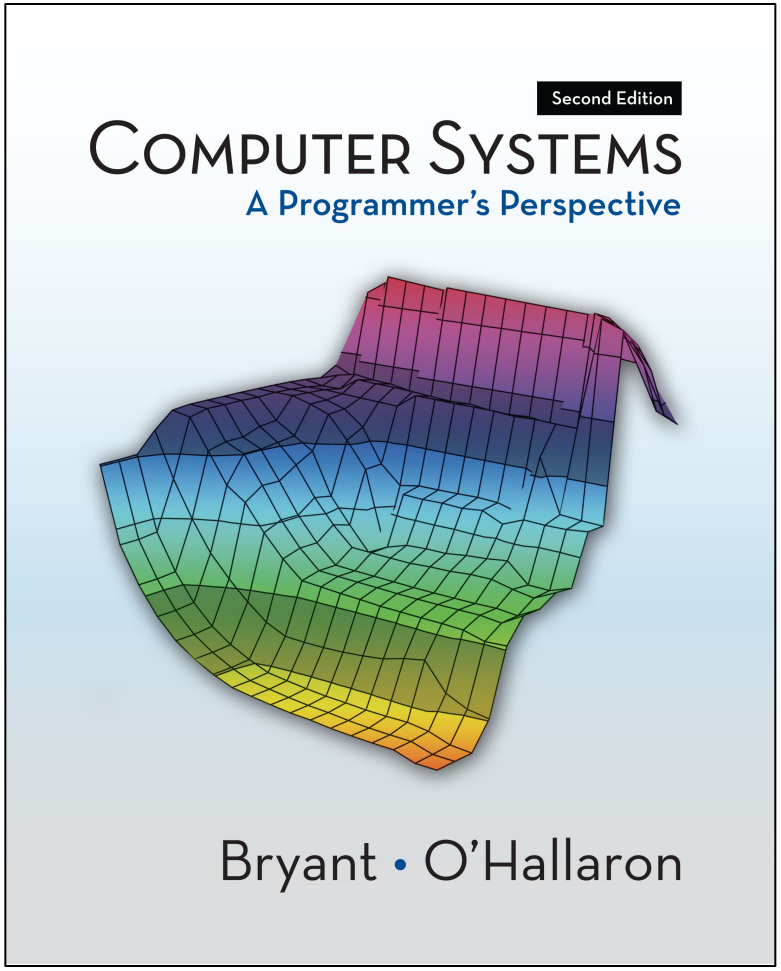
- Late assignments will be penalized:
  - Up to 1 day (24 hours) late: 10% deduction
  - Up to 2 days (48 hours) late: 10 + 20 = 30%
  - Up to 3 days (72 hours) late: 10 + 20 + 30 = 60%
  - After 3 days: Sorry, don't bother. ☹
- Every student has 4 "late tokens"
  - Each token is worth up to 24 hours of extension, "no questions asked"
  - State on your submission how many tokens you used
- Also, notes from Dean's Office or Health Center will almost always warrant an extension
  - (no tokens are consumed this way)

CS24 Assignments

- CS24 is a very time-consuming class
- Always start assignments well before they're due!
  - Don't get caught by assignments you didn't expect to be hard for you
  - You will make the most of office hours this way, too
- Some assignments are more involved than others
  - I will warn you ahead of time
- Submit your assignments via Moodle
  - Instructions for packaging at top of each assignment
  - If you don't follow these instructions, you will lose points on your assignments
  - We will be happy to help if you need help with this

Textbook – CS:APP2e

- Computer Systems: A Programmer's Perspective
  - A *very* good textbook, from Carnegie Mellon ICS class
  - amazon.com: $126 new (eText edition: $105)
- Book is optional for the course
  - A great book, but too expensive
  - Copies of most relevant material (and HW problems) are provided
  - Chapters/sections to read are specified for each week



Programming Languages for CS24

- Assignments involve programming with C, and IA32 assembly language
  - You are expected to have a general familiarity with C (syntax, pointers, structs, memory management)
  - IA32 is introduced more gently, along the way
- CS:APP contains many helpful hints for C in each chapter
  - A great resource if not intimately familiar with C
  - TAs can help with nuances of using C, but they will not teach you C from scratch – that's not their job.
- IA32 assembly language:
  - Language used for programming Intel x86-family processors

Programming Environment

- All assignments must be completed on a 32-bit Linux platform
  - No 64-bit Linux. No MacOS X. No Cygwin.
- Multiple technical reasons:
  - Incompatibilities between 32-bit and 64-bit platforms
  - Variations in how OSes link and load C programs
  - Some assignments use low-level system APIs that are only provided on Linux, not MacOS X or Cygwin
- Use a CS cluster account (recommended)
- Or, install 32-bit Linux in a virtual machine on your computer (e.g. using Virtual Box)
  - We will provide a VM image in the next few days

Programming Environment (2)

- GNU toolset:
  - **`gcc`** for C programming
  - **`as`** (GNU assembler) for IA32 assembly
  - GNU **`make`** for building/running programs
  - **`gdb`** for finding your bugs ☺
- Will provide supplemental material for **`gcc`**, **`gdb`** and **`make`** on Moodle
- Will also provide recordings of two older lectures on how to debug programs with **`gdb`**
  - You should watch these by end of 2nd week of class
  - You want to learn **`gdb`** – it will shave *hours* off of your assignments!

Motivations for CS24

- Why study computing systems in the first place?

- Reason 1:
  - Understanding how the computer works will help you to use it more effectively.
  - You will be a better programmer if you understand the details of how the computer works.

Example: Molecular Dynamics

- Experiments involve simulating individual atoms

```
#define N_ATOMS 10000
#define DIM 2
/* Array of data for each atom being simulated. */
double atoms[N_ATOMS][DIM][DIM];
```

- Version 1:

```
for (i = 0; i < DIM; i++)
  for (j = 0; j < DIM; j++)
    for (n = 0; n < N_ATOMS; n++)
      atoms[n][i][j] = ... ;
```

- Version 2:

```
for (n = 0; n < N_ATOMS; n++)
  for (i = 0; i < DIM; i++)
    for (j = 0; j < DIM; j++)
      atoms[n][i][j] = ... ;
```

- *Why is version 2 significantly faster than version 1?*

Example: Financial Computations

- Candy Shop in the Math Department:
  - First candy costs 10¢
  - Each subsequent candy costs 10¢ more than previous one
  - You have one dollar to spend
  - How many candies can you purchase?
- Write a C program to solve it:

```
float fundsLeft = 1.0, price;
int numCandies = 0;
for (price = 0.1; price <= fundsLeft; price += 0.1) {
  numCandies++;
  fundsLeft -= price;
}
printf("%d candies; %f left over\n",
  numCandies, fundsLeft);
```

- *Why doesn't this blasted program work properly?!*
  - Output: **`3 candies; 0.400000 left over`**

Motivations for CS24 (2)

- Why study computing systems in the first place?
- Reason 1:
  - Understanding how the computer works will help you to use it more effectively.
  - You will be a better programmer if you understand the details of how the computer works.
- Both examples are very simple to understand…
  - …if you actually know how the computer works!
- Will see much more sophisticated examples as we go through the term

Motivations for CS24 (3)

- Why study computing systems in the first place?
- Reason 2:
  - The concepts we will cover are ubiquitous in modern computing systems
  - Have a profound impact on most hardware designs, and also on operating system design/implementation
- If you ever participate in hardware design, or in operating system design:
  - Need to understand the common challenges, and strengths/weaknesses of the common solutions
  - You might even devise new solutions that are better than what we presently use!

Example: Memory Management

- Operating systems provide a "process" abstraction
  - Allows multiple programs to share a single CPU "at the same time"
  - e.g. a web browser, text editor, and email client
- Want to isolate memory used by different processes
  - An incorrect program should not cause other programs to crash, or corrupt the operating system itself
- Want to provide a "virtual memory" abstraction
  - OS can allow programs to use more memory than the physical hardware actually provides

- *What features should the hardware provide, to make these features fast, secure, and easy to implement?*

Instruction Set Architectures

- Intel IA32 is a specific example of an Instruction Set Architecture (ISA)
  - A specific set of instructions that can be executed by a processor, along with their byte encodings
- Multiple vendors can implement a specific ISA
  - Intel and AMD both implement the IA32 ISA
- Different kinds of instruction set architectures
- CISC: Complex Instruction Set Computer
  - A large number of very powerful instructions
  - Programs require fewer instructions to implement a particular computation
  - Logic for supporting these instructions is more complicated
  - IA32 is a CISC architecture

Instruction Set Architectures (2)

- RISC: Reduced Instruction Set Computer
  - A relatively small number of simpler instructions
  - Programs require more instructions to implement a computation
  - Hardware implementing these instructions can provide more pipelining
- These days, line between RISC/CISC is often blurred
  - CISC processors can internally translate instructions into RISC-like steps to pipeline and execute
  - RISC processors often include more sophisticated CISC-like instructions
- More pure-RISC processors are seen primarily in embedded/mobile systems
  - Simple and low-power are critical requirements

How To Build a Programmable Computer?

- Computers are very complex systems!
- What basic concepts underlie programmable computers?
- How are they assembled into a usable system?

- This week: a brief tour of how a programmable computer works
  - What components make up a simple computer?
  - What do the instructions look like?
  - How do we implement a computation?

Abstraction Hierarchy

- Handle complexity in computing systems with an abstraction hierarchy
- A physical medium of computation
  - …including a way to represent information
  - We generally use semiconductors these days
  - Vacuum tubes, relays, gears, tinker toys and string
- Simple gates for processing signals
  - AND, OR, NOT, XOR, NAND, etc.
  - Implemented in the physical medium
  - Abstracts away the need to think about physics
- Build basic functional units from our gates
  - Counters, arithmetic/logic unit (ALU), memory, multiplexers, decoders, etc.
  - Don't need to think about gates anymore

Abstraction Hierarchy (2)

- From functional units, can construct a programmable ISA computer!
  - Provides a very simple, limited instruction set
  - We can program it to implement various computations
- This is good, but not very easy to use.
  - Continue extending abstraction hierarchy
- Runtime support to create larger programs:
  - Stacks, heaps, a means to dynamically allocate memory
  - Ability to create subroutines, implement recursion
- Operating systems:
  - Provide many useful abstractions for programming
  - File IO, processes, threads, isolation, virtual memory, networking, etc., etc.

Signals and Gates

- We are studying *digital* computers…
- Most fundamental piece of information is a bit
  - A single 0 or 1 value
- Logic gates allow us to process bits
- Simple examples:

AND Gate

A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1

OR Gate

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	1

NOT Gate

A	Output
0	1
1	0

More Complex Gates

- Construct more complex gates from simple gates
  - In fact, can construct OR from AND and NOT
  - $a \vee b = \neg (\neg a \wedge \neg b)$ (De Morgan's Law)
- Example: XOR, exclusive OR
  - Output is 1 iff exactly one input is 1
  - A XOR B = (A AND NOT B) OR (NOT A AND B)

XOR Gate

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0

- We can also construct XOR entirely from AND and NOT
  - We know how to make OR from AND and NOT…

Logic and Arithmetic

- With these simple gates, can actually implement addition, subtraction, etc.
- Need a way to represent numeric values with bits
- Need circuits that can manipulate these values

- Data Representation:
  - How do we represent various values in our digital computer?
  - Also, how do we represent different *kinds* of values?
    - Integers, decimal values, characters, etc.

- For now: simple unsigned integers

Unsigned Integers in Binary

- We're used to representing integers as vectors of decimal digits
  - Each digit is 0..9
- Represent unsigned integers as vectors of bits
  - Each digit is 0 or 1
- Also, constrain ourselves to a specific number of bits w for representing values
  - e.g. 4 bits = 1 nibble, 8 bits = 1 byte, 16 bits, 32 bits
  - Obviously limits the range of values we can represent
- A vector of bits $\boldsymbol{x}$ maps to an unsigned integer:

$$\mathrm{B2U}_w(\boldsymbol{x}) = \sum_{i=0}^{w-1} x_i \, 2^i$$

  - Individual bits are numbered 0 to w-1

Unsigned Integers (2)

- $42_{10} = 00101010_2$
  - $0\times2^7 + 0\times2^6 + 1\times2^5 + 0\times2^4 + 1\times2^3 + 0\times2^2 + 1\times2^1 + 0\times2^0$
  - $= 32 + 8 + 2$
- Adding integers in base 2 is also straightforward

$$\boldsymbol{a} + \boldsymbol{b} = \sum_{i=0}^{w-1} a_i\, 2^i + \sum_{i=0}^{w-1} b_i\, 2^i$$

  - Important detail: need to carry in base 2, just like in base 10!
- Example: $106_{10} + 105_{10}$
  - $= 01101010_2 + 00101001_2$
  - $= 11010011_2 = 211_{10}$

```
C:  011010000
A:    01101010
B: + 01101001
S:   11010011
```

Adding Unsigned Integers

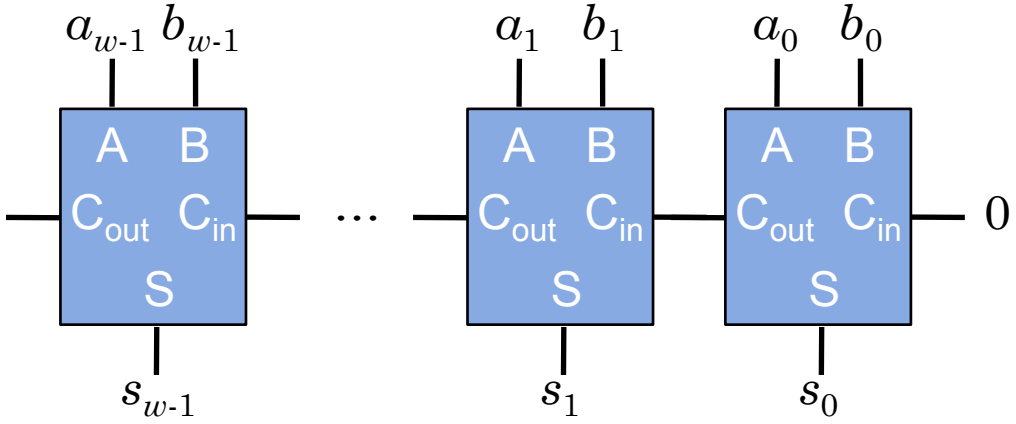
- Simply need to construct necessary machinery for adding bits, using our gates
- Full adder:
  - Takes inputs: A, B, C_{in}
  - Produces outputs: S, C_{out}
- Logic for full adder?
- Sum S is relatively easy:
  - S = A XOR B XOR C_{in}
- Carry-out is more complicated:
  - Carry-out if A and B are 1, or if (A + B) and C_{in} are 1
  - Cout = (A AND B) OR (A XOR B) AND C_{in}

A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

Adding Unsigned Integers (2)

- Simply need to construct necessary machinery for adding bits, using our gates
- Full adder:
  - Takes inputs: A, B, C_{in}
  - Produces outputs: S, C_{out}
- To add two *w*-bit unsigned values, hook together *w* full adders:

A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

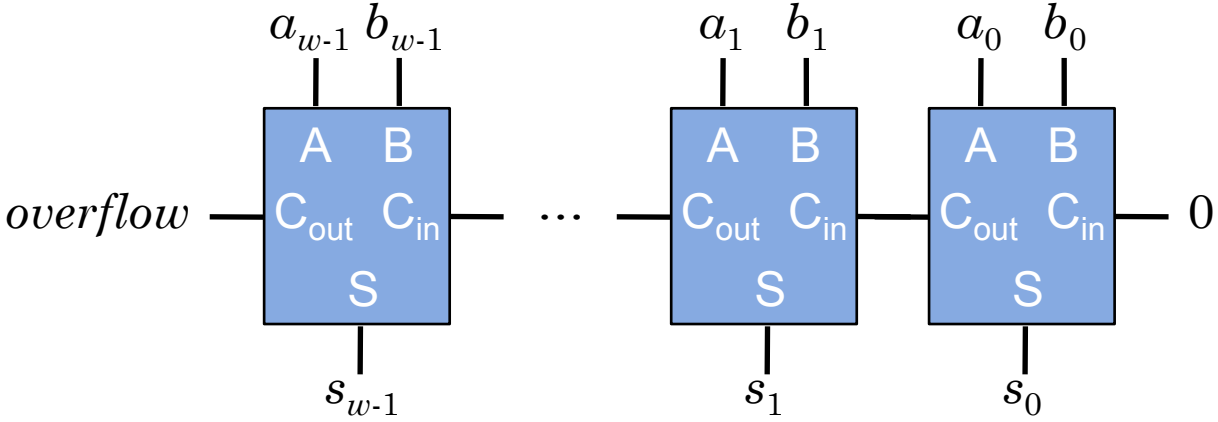


Ranges and Overflow

- For a given w, can only represent unsigned integer values in range 0 .. 2^w-1
  - e.g. for w = 8, can represent 0 .. 2^8-1, or 0 .. 255
- What happens if we add these values:
  - $175_{10} + 114_{10} = 10101111_2 + 01110010_2$
  - Result is 289 (100100001_2). This is a problem.
  - Computer adds these values and gets 33 (00100001_2)
- Best case scenario:
  - The computer will tell us when this happens
- Worst case scenario:
  - No way of telling that this problem has occurred

Ranges and Overflow (2)

- Can the computer tell us there was a problem?



  - **Yes:** topmost carry-out will be 1 when we overflow
- Label topmost carry-out "overflow"
  - When overflow is detected, we can handle the error
- *overflow* is a status value
  - It describes additional details of the computation
- One example of how computers can be designed to be more resilient to errors

Signed Integer Representation

- Often need to represent signed values as well
- Most common representation: two's complement
- Most significant bit x_{w-1} becomes the sign bit
  - 0 = positive value
  - 1 = negative value

$$\mathrm{B2T}_w(\boldsymbol{x}) = -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i\, 2^i$$

- Given w bits, can represent $-2^{w-1} \,..\, 2^{w-1} - 1$
  - e.g. for $w = 8$, can represent values -128 to +127
- Smallest negative value: $10000000_2 = -128$
- Largest positive value: $01111111_2 = 127$

Signed Integer Representation (2)

- Easy trick for converting an integer into its two's complement representation:
  - Invert the bits, then add one
- Example:
  - Find two's complement representation for -42
  - Unsigned representation for 42 is 00101010_2
  - Invert the bits: 11010101_2
  - Add one: 11010110_2
- Converting back, following $B2T_{w=8}$ function:
  - $= -1\times2^7 + 1\times2^6 + 1\times2^4 + 1\times2^2 + 1\times2^1$
  - $= -128 + 64 + 16 + 4 + 2$
  - $= -42$

Signed Integers and Overflow

- Rules for overflow flag clearly have to change.
- -1 in two's complement representation ($w = 8$):
  - 11111111
- Adding 1 to this value clearly results in a carry-out!

```
C:  111111110
A:    11111111
B: + 00000001
S:    00000000
```

- Need to redefine overflow test for signed integers:
  - e.g. for addition, if inputs are same sign, and output is opposite sign, then a signed overflow has occurred

Summary

- Have a data representation for signed and unsigned numbers now…
- Next time, begin discussing basic processor components
  - What they provide
  - How to assemble them into a simple processor
  - How to program the simple processor

- **Your action items:**
  - Enroll in CS24 Moodle course.
  - If using Annenberg Lab, get your CS account set up.
  - If you want, get a copy of CS:APP2e; read Chapter 1.