CS61: SYSTEMS PROGRAMMING AND MACHINE ORGANIZATION

LECTURE 3

1. Announcements

- Sectioning
  - Section times will be announced and the section tool will be open with the link on the website.
  - Sections will start next week.
- There will be office hours this week; times are on the calendar on the website.
- Assignment 0 is due this Friday.
- There will be a section on Git, the version control system, right after lecture in MD G115 on Thursday. Git will be used starting in assignment 1.
  - This section will be led by Rob Bowden, who volunteered his time!
  - This section is not mandatory (and no future sections will be.) They are recommend, though.
  - The section on git is especially recommended if you've never used git before.
- In response to a question about showing work on Assignment 0: Showing a few lines of work in the first question is useful; it's better to know how to convert from decimal to binary, even if it can be accomplished by a computer program. Even if it's possible to write a computer program to generate the work.
- Congrats to Nathaniel Hermann, who found an image file with assembly executable code for the sum challenge! Way to go.

2. Unsigned Arithmetic

Our discussion of unsigned arithmetic will involve the following C types:

- unsigned short, whose maximum value is $2^{16} - 1$
- unsigned char, whose maximum value is $2^8 - 1$
- unsigned int, whose maximum value is $2^{32} - 1$
- unsigned long, whose maximum value is $2^{32} - 1$ (See historical digression below.)
- unsigned long long, whose maximum value is $2^{64} - 1$

2.1. Historical Digression.

- The earliest digital computers sold had 18-bit, then 36-bit words.
- This makes sense, if we think about it in terms of approximations:
  $2^{36} = 2^{30} \cdot 2^6 \approx 10^9 \cdot 64 > 10^{10}$ (the smallest even power of 20 that can fit 10-decimal digits)

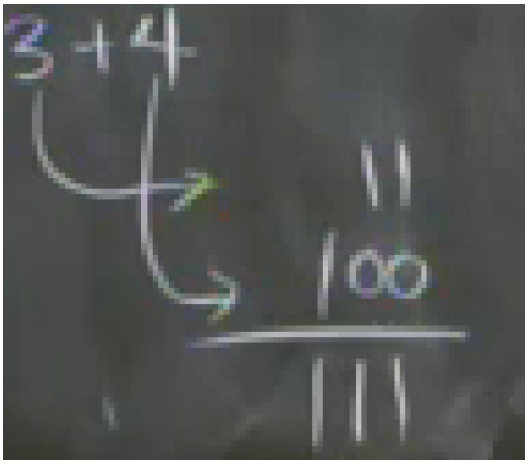
- Earliest digital calculators could manipulate 10-digit numbers, so early digital computers needed to fit 10-digit numbers to match up
- Case-in-point: Unisys (a "dinosaur" computer company)
  - From a technical specification: "The OS 2200 used 36 bits of every platform word (8 bytes) to minimize the complexity of the Unisys design"
  - Throws away a little less than half of every word in memory!

2.2. **Concepts.**

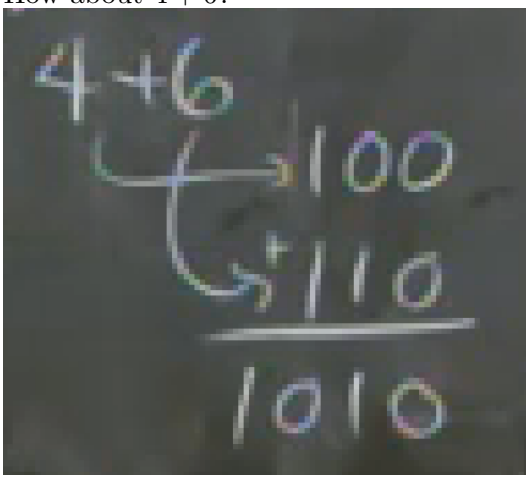
- One of the most important concepts of unsigned arithmetic is the following:

  > Unsigned arithmetic is regular, pencil-and-paper arithmetic modulo 2^w, where w is the number of bits that a type can hold. This is a very simple thing to remember, and it is always true by the definition of the C abstract machine.

- What's $3 + 4$ going to evaluate to, assuming a word size (w) of 4?



- How about $4 + 6$?



Note the carry.

- What about subtraction? Remember that unsigned computer arithmetic is the same as mathematical arithmetic modulo 2^w.
  - Using the laws of regular arithmetic, t $a-b$ should be the same as $a+-b$. That's just a mathematical law.
  - What does that say about $-b$? How do we represent it in unsigned arithmetic?
    - Describe it in words: it's the number that, when added to b gives 0.
    - $-b = c$ s.t. $b + c = 0$
    - $b + c = 0 mod 2^w$
  - What else is 0 mod 2^w? 2^w.
  - This says that $c = 2^w - b$. Doesn't look like we've solved anything, but this is mathematical subtraction, not unsigned, modulus subtraction, since we have bounds on c.
  - So what's -1? $2^w - 1$. Note that, $2^w - 1 + 1 = 2^w = 0$. That's the definition of -1: when we add 1 to it, we should get 0, and in computer arithmetic, we do!

2.3. **Fun.**

- Strictly speaking, in math, if $x > 0$, then $-x < x$.
- However, what about computer math, for unsigned numbers? We know that $-x$ is representable. But they're not in any specific relationship.
- Is there a time when $-x == x$ for $x! = 0$? Yes, solve by finding an x so that $2x == 0$. That can be $\frac{2^w}{2}$, or 2^{w-1}, another "interesting" number!

2.4. **Operations on C Integers.**

- Multiplication
- Division
- Modulus
- Addition
- Subtraction
- Bit shifting operators
  - These operators are "crazy!"
  - Order the following values:
    - $min(x, y)$
    - max(x,y)
    - x & y
    - $x|y$
    - x
    - y
    - Answer: $x|y >= max(x, y) \geq x, y, >= min(x, y) >= x\&y$
  - Idea: Let x_i represent the bit of x in position i. Then, x (say 3) is expressible as $\sum_{i=0}^{2} 2^i x_i = 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 3$. Similarly, y (say 5) is expressible as

$\sum_{i=0}^{2} 2^i x_i = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 5$. The value at the bit position in the "or" is at least than the value of either x or y. The same logic applies for the others: for the $x\&y$ value, zero "consumes" 1, so we minimize specific to each bit position, and that will result in something at most $min(x, y)$.

- Comparison operators
- Logical operators

2.5. **More Fun.**

- What is $(x|y) - (x\&y)$? Answer: $(x \underline{\vee} y)$. One explanation: Take the bits that all the bits that are lit up, and then take away the bits that are lit up in both places. You have left only the bits in positions that are lit up in exactly one number.
- Scribe's explanation: Imagine this in terms of probabilities: if you calculate the probability of events x or y occurring, you've found the probability that x occurs, y occurs, or x and y occurs. To find the probability that exactly one of these events occurs, just subtract the intersection, $x\&y$!
- When is $x\&(x-1) == 0$?
  - 0 is a possiblity.
  - All powers of 2 are possibilities. Let's think about how a power of two is represented in binary: $100000 \cdots 0$. Now, let's think about how this power of two minus 1 is represented in binary $011111 \cdots 1$. "And"ing these, we get 0!
- $\neg$ x flips all of the bits in x.
- What's $x|\neg x$ equal to? Answer: -1.
- What is $x + \neg x$? Answer: -1.
- Why do + and | have the same effect here? We know there's no carrying, because there are no bit positions where both are lit up!
- Now let's try $\neg x + 1$!
- Gosper's Hack: A blog entry from CS207 (http://read.seas.harvard.edu/cs207/2012/?p=64)

3. A Bit About Sets

- Let's see if we can use bits to construct a representation of a set that supports the following operations:
  - Membership?
  - Set union
  - Set intersection
  - Set difference
- Suggestion: An array of bits: where 0 means it's not in the set, and 1 means it's in the set.
- How many letters are there? 26! What fits in 26 - an unsigned int!
- We can represent sets of up to 32 objects in terms of bits. Very cheap operations!
- Empty set: 0
- Set containing only a character C, where C is between 'a' and 'z'. Let's use some bitwise operators! We're going to represent that set as $1 << (c - 97)$, which is equivalent to $1 << (c - `a')$ (ASCII). (Left shift takes a number, and adds zeroes

to the lower end of the number, shifting everything to the left!) So we represent $\{'a'\}$ by 1, and $\{'b'\}$ by 2.
- Set intersection: &
- Set union: |
- Membership: 'c' is in the set S iff $\{'c'\} \cap S$ is non-empty. Or, in computer arithmetic: $(1 << ('c' - 'a'))\&S! = 0$
- Set difference: $x - (x\&y)$, or $(x|y) - y$, or $x \underline{\vee} (x\&y)$, or $x\&\neg(x\&y)$
- Singleton? $x! = 0\&\&(x\&(x-1)) == 0$. Power of 2 test from before!

4. Signed Arithmetic

- Idea: What if implementation of signed arithmetic was the same as unsigned?
- Bit pattern for signed -1 same as bit pattern for unsigned -1!
- Bit pattern for $-x$: $2^w - x$: two's complement!
- We know that the number is negative if the left-most bit (sign bit) is a 1. If it is a zero, then it's non-negative. This is why the maximum is $2^(w-1)$, since a one bit is used in signs.
- Anomaly:$8 = 1000_b = -8$. So there's no way to represent $+8$ in signed arithmetic with this word size. Not symmetric about zero!