Machine Representation/Numbers
Lecture 3
CS 61C Machines Structures
Fall 00
David Patterson
U.C. Berkeley

http://www-inst.eecs.berkeley.edu/~cs61c/

Overview
- Recap: C v. Java
- Computer representation of “things”
- Unsigned Numbers
- Administrivia
- Free Food 5PM Thursday, Sept. 7
- Computers at Work
- Signed Numbers: search for a good representation
- Shortcuts
- In Conclusion

Decimal Numbers: Base 10
- Digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
- Example:
  3271 =
  (3x10^3) + (2x10^2) + (7x10^1) + (1x10^0)

Numbers: positional notation
- Number Base B => B symbols per digit:
  - Base 10 (Decimal): 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
  - Base 2 (Binary): 0, 1
- Number representation:
  - d_3 d_2 d_1 d_0 is a 32 digit number
  - value = d_3 x B^3 + d_2 x B^2 + d_1 x B^1 + d_0 x B^0
- Binary: 0, 1
  - 1011010 = 1x2^6 + 0x2^5 + 1x2^4 + 1x2^3 + 0x2^2 + 1x2 + 0x1
  - Notice that 7 digit binary number turns into a 2 digit decimal number
  - A base that converts to binary easily?

From last time: C v. Java
- C Designed for writing systems code, device drivers
- C is an efficient language, with little protection
  - Array bounds not checked
  - Variables not automatically initialized
- C v. Java: pointers and explicit memory allocation and deallocation
  - No garbage collection
  - Leads to memory leaks, funny pointers
  - Structure declaration does not allocate memory; use malloc() and free()
Hexadecimal Numbers: Base 16

- **Hexadecimal:**
  0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F
  - Normal digits + 6 more: picked alphabet

- **Conversion: Binary <-> Hex**
  - 1 hex digit represents 16 decimal values
  - 4 binary digits represent 16 decimal values
  - → 1 hex digit replaces 4 binary digits

- **Examples:**
  - 1010 1100 0101 (binary) = ? (hex)
  - 10111 (binary) = 0001 0111 (binary) = ?
  - 3F9(hex) = ? (binary)

Decimal vs. Hexadecimal vs. Binary

- **Examples:**
  - 00 0 0000
  - 01 1 0001
  - 02 2 0010
  - 03 3 0011
  - 04 4 0100
  - 05 5 0101
  - 06 6 0110
  - 07 7 0111
  - 08 8 1000
  - 09 9 1001
  - 10 A 1010
  - 11 B 1011
  - 12 C 1100
  - 13 D 1101
  - 14 E 1110
  - 15 F 1111

What to do with representations of numbers?

- **Just what we do with numbers!**
  - Add them
  - Subtract them
  - Multiply them
  - Divide them
  - Compare them

- **Example:**
  - 10 + 7 = 17
  - so simple to add in binary that we can build circuits to do it
  - subtraction also just as you would in decimal

Which base do we use?

- **Decimal:** great for humans, especially when doing arithmetic
- **Hex:** if human looking at long strings of binary numbers, its much easier to convert to hex and look 4 bits/symbol
  - Terrible for arithmetic; just say no
- **Binary:** what computers use; you learn how computers do +, -, *, /
  - To a computer, numbers always binary
  - Doesn’t matter base in C, just the value: 32_{10} == 0x20 == 100000_2
  - Use subscripts “ten”, “hex”, “two” in book, slides when might be confusing

Administrivia

- **Grading:** fixed scale, not on a curve
- To try to switch sections - email request to cs61c
- Viewing lectures again: tapes in 205 McLaughlin
- Read web page: Intro, FAQ, Schedule
  - TA assignments, Office Hours
  - Project 1 due Friday by Midnight

- **Tu/Th section 5-6PM; 18/118**
  - “Mark Chew” is most recent TA
  - He quit, so lab/discussion in canceled
Free Food 5PM Thursday, Sept. 7
- "The Importance of Graduate School"
  - Professor Katherine Yelick, UC Berkeley (Moderator)
  - Professor Mary Gray Baker, Stanford University
  - Dr. Serap Savari, Lucent Technology
  - Kris Hildrum, CS Current Graduate Student
- 5:30 p.m. PANEL DISCUSSION, Hewlett-Packard Auditorium, 306 SODA
- 5:00 p.m. REFRESHMENTS in the Hall, Fourth Floor, Soda Hall

Bicycle Computer (Embedded)
- P. Brain
  - wireless heart monitor strap
  - record 5 measures: speed, time, current distance, elevation and heart rate
  - Every 10 to 60 sec.
  - 8KB data => 33 hours
  - Stores information so can be uploaded through a serial port into PC to be analyzed

Limits of Computer Numbers
- Bits can represent anything!
- Characters?
  - 26 letter => 5 bits
  - upper/lower case + punctuation => 7 bits (in 8)
  - rest of the world’s languages => 16 bits (unicode)
- Logical values?
  - 0 => False, 1 => True
- colors?
- locations / addresses? commands?
- but N bits => only 2^N things

Comparison
- How do you tell if X > Y?
  - See if X - Y > 0

How to Represent Negative Numbers?
- So far, unsigned numbers
- Obvious solution: define leftmost bit to be sign!
  - 0 => +, 1 => -
  - Rest of bits can be numerical value of number
- Representation called sign and magnitude
- MIPS uses 32-bit integers. +1_{ten} would be: 0000 0000 0000 0000 0000 0000 0000 0001
- And - 1_{ten} in sign and magnitude would be: 1000 0000 0000 0000 0000 0000 0000 0001

Shortcomings of sign and magnitude?
- Arithmetic circuit more complicated
  - Special steps depending whether signs are the same or not
- Also, Two zeros
  - 0x00000000 = +0_{ten}
  - 0x80000000 = -0_{ten}
  - What would it mean for programming?
- Sign and magnitude abandoned
Another try: complement the bits

- Example: \( 7_{10} = 00111_2 \), \(-7_{10} = 11000_2 \)
- Called one’s Complement
- Note: positive numbers have leading 0s, negative numbers have leadings 1s.

```
00000 00001 ... 01111
10000 ... 11111 11111
```

- What is \(-00000\)?
- How many positive numbers in \(N\) bits?
- How many negative ones?

Shortcomings of ones complement?

- Arithmetic not too hard
- Still two zeros
  - \(0x00000000 = +0_{10} \)
  - \(0xFFFFFFF = -0_{10} \)
- What would it mean for programming?
- One’s complement eventually abandoned because another solution was better

Search for Negative Number Representation

- Obvious solution didn’t work, find another
- What is result for unsigned numbers if tried to subtract large number from a small one?
  - Would try to borrow from string of leading 0s, so result would have a string of leading 1s
  - With no obvious better alternative, pick representation that made the hardware simple: leading 0s \(\Rightarrow\) positive, leading 1s \(\Rightarrow\) negative

```
00000...xxx \(\Rightarrow\geq0\), 11111...xxx \(<0\)
```

- This representation called two’s complement

2’s Complement Formula

- Can represent positive and negative numbers in terms of the bit value times a power of 2:
  
  \[
  -d_1 \times \underbrace{-2^{31}}_{\text{sign bit}} + d_2 \times 2^{30} + \ldots + d_i \times 2^2 + d_i \times 2^1 + d_i \times 2^0
  \]

- Example

```
1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
0111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
1000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111
```

- One zero, 1st bit \(=\) \(\geq0\) or \(<0\), called sign bit
  - but one negative with no positive

Two’s Complement Number line

- \(2^{N-1}\) non-negatives
- \(2^{N-1}\) negatives
- one zero
- how many positives?
- comparison?
- overflow?
Two's complement shortcut: Negation

- Invert every 0 to 1 and every 1 to 0, then add 1 to the result
  - Sum of number and its one's complement must be \(111\ldots1\)\text{two}
  - \(-1\)\text{ten} = \(-1\)\text{two}
  - Let \(x'\) mean the inverted representation of \(x\)
  - Then \(x + x' = 0 \Rightarrow x' + 1 = -x\)

- Example: -4 to +4 to -4

\[
\begin{align*}
  x : & \quad 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1100\text{two} \\
  x' : & \quad 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0011\text{two} \\
  +1: & \quad 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0000 \ 0100\text{two} \\
  ()': & \quad 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1011\text{two} \\
\end{align*}
\]

Signed vs. Unsigned Numbers

- C declaration \texttt{int}
  - Declares a signed number
  - Uses two's complement

- C declaration \texttt{unsigned int}
  - Declares an unsigned number
  - Treats 32-bit number as unsigned integer, so most significant bit is part of the number, not a sign bit

Signed v. Unsigned Comparisons

- \(X = 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1100\text{two}\)
- \(Y = 0011 \ 1011 \ 1001 \ 1010 \ 1000 \ 1010 \ 0000 \ 0000\text{two}\)

- Is \(X > Y\)?
  - unsigned: YES
  - signed: NO

- Converting to decimal to check
  - Signed comparison: \(-4\)\text{ten} < 1,000,000,000\text{ten}?
  - Unsigned comparison: \(-4,294,967,292\)\text{ten} < 1,000,000,000\text{ten}?

Numbers are stored at addresses

- Memory is a place to store bits
- A word is a fixed number of bits (eg, 32) at an address
  - also fixed no. of bits
- Addresses are naturally represented as unsigned numbers

What if too big?

- Binary bit patterns above are simply \texttt{representatives} of numbers
- Numbers really have an infinite number of digits
  - with almost all being zero except for a few of the rightmost digits
  - Just don’t normally show leading zeros
- If result of add (or \(-,\div\)) cannot be represented by these rightmost HW bits, \texttt{overflow} is said to have occurred

Two’s comp. shortcut: Sign extension

- Convert 2’s complement number using \(n\) bits to more than \(n\) bits
- Simply replicate the most significant bit (sign bit) of smaller to fill new bits
  - 2’s comp. positive number has infinite 0s
  - 2’s comp. negative number has infinite 1s
  - Bit representation hides leading bits; sign extension restores some of them
  - 16-bit -4\text{ten} to 32-bit:

\[
\begin{align*}
  1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1100\text{two} \\
  1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1100\text{two} \\
\end{align*}
\]
And in Conclusion...

- We represent “things” in computers as particular bit patterns: N bits $\Rightarrow 2^N$
  - numbers, characters, ...

- Decimal for human calculations, binary to understand computers, hex to understand binary

- 2’s complement universal in computing: cannot avoid, so learn

- Computer operations on the representation correspond to real operations on the real thing

- Overflow: numbers infinite but computers finite, so errors occur