Review

- CS61C: Learn 6 great ideas in computer architecture to enable high performance programming via parallelism, not just learn C
  1. Layers of Representation/Interpretation
  2. Moore’s Law
  3. Principle of Locality/Memory Hierarchy
  4. Parallelism
  5. Performance Measurement and Improvement
  6. Dependability via Redundancy

Putting it all in perspective…

“If the automobile had followed the same development cycle as the computer,

– Robert X. Cringely

Data input: Analog → Digital

- Real world is analog!
- To import analog information, we must do two things
  - Sample
    - E.g., for a CD, every 44,100ths of a second, we ask a music signal how loud it is.
  - Quantize
    - For every one of these samples, we figure out where, on a 16-bit (65,536 tic-mark) “yardstick”, it lies.

Digital data not nec born Analog…

- BIG IDEA: Bits can represent anything!!
  - Characters?
    - 26 letters → 5 bits (2^5 = 32)
    - upper/lower case + punctuation → 7 bits (in 8) (“ASCII”)
    - standard code to cover all the world’s languages → 8,16,32 bits (“Unicode”)
    - www.unicode.com
  - Logical values?
    - 0 → False, 1 → True
  - colors? Ex: Red (650), Green (0), Blue (168)
  - locations / addresses? commands?
  - MEMORIZE: N bits → at most 2^N things

BIG IDEA: Bits can represent anything!!

- Characters?
  - 26 letters → 5 bits (2^5 = 32)
  - upper/lower case + punctuation → 7 bits (in 8) (“ASCII”)
  - standard code to cover all the world’s languages → 8,16,32 bits (“Unicode”)
    - www.unicode.com
- Logical values?
  - 0 → False, 1 → True
- colors? Ex: Red (650), Green (0), Blue (168)
- locations / addresses? commands?
- MEMORIZE: N bits → at most 2^N things
How many bits to represent $\pi$?

a) 1
b) 9 ($\pi = 3.14$, so that's .011 ".” 001 100)
c) 64 (Since Macs are 64-bit machines)
d) Every bit the machine has!
e) $\infty$

What to do with representations of numbers?

• Just what we do with numbers!
  • Add them 1 1
  • Subtract them 1 0 1 0
  • Multiply them + 0 1 1 1
  • Divide them
  • Compare them

• Example: 10 + 7 = 17 1 0 0 0 1
  • ...so simple to add in binary that we can build circuits to do it!
  • subtraction just as you would in decimal
  • Comparison: How do you tell if $X > Y$?

What if too big?

• Binary bit patterns above are simply representatives of numbers. Abstraction! Strictly speaking they are called “numerals”.
• Numbers really have an $\infty$ number of digits
  • with almost all being same (00...0 or 11...1) except for a few of the rightmost digits
  • Just don’t normally show leading digits
• If result of add (or -, *, /) cannot be represented by these rightmost HW bits, overflow is said to have occurred.

0000 0001 0010 1111

unsigned 1111 1111

How to Represent Negative Numbers?

(C’s unsigned int, C99’s uintN_t)

• So far, unsigned numbers

00000 00001 ... 01111 10000 ... 11111

Binary odometer

• Obvious solution: define leftmost bit to be sign!
  • 0 $\Rightarrow$ +
  • $\overline{0}$ $\Rightarrow$ –
  • Rest of bits can be numerical value of number
• Representation called sign and magnitude

0000 0001 ... 0111

Binary odometer

1111 ... 10001 1000

META: Ain’t no free lunch

Shortcomings of sign and magnitude?

• Arithmetic circuit complicated
  • Special steps depending whether signs are the same or not
• Also, two zeros
  • 0x00000000 = +0$_{bn}$
  • 0x80000000 = $\overline{0}_{bn}$
  • What would two 0s mean for programming?
• Also, incrementing “binary odometer”, sometimes increases values, and sometimes decreases!

Therefore sign and magnitude abandoned

Administrivia

• Upcoming lectures
  • Next few lectures: Introduction to C
• Lab overcrowding
  • Remember, you can go to ANY discussion (none, or one that doesn’t match with lab, or even more than one if you want)
  • Overcrowded labs - consider finishing at home and getting checkoffs in lab, or bringing laptop to lab
  • If you’re checked off in 1st hour, you get an extra point on the labs!
• TAs get 24x7 cardkey access (and will announce after-hours times)
• Enrollment
  • It will work out, don’t worry
• Soda locks doors @ 6:30pm & on weekends
• Look at class website, piazza often!
  \[\text{http://inst.eecs.berkeley.edu/~cs61c/piazza.com}\]

Iclickerskinz.com
in terms of the bit value times a power of 2:

Example:

`1101` in a nibble?

- `1 x 2^3` + `1 x 2^2` + `0 x 2^1` + `1 x 2^0`
- `-8 + 4 + 0 + 1`
- `-4`
- `-3_{ten}`

Another try: complement the bits

- Example: `-7_{10} = 001112, 7_{10} = 110002`
- Called **One's Complement**
- Note: positive numbers have leading 0s, negative numbers have leadings 1s.

<table>
<thead>
<tr>
<th>2’s Complement Number “line”: N = 5</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>00001</code></td>
</tr>
<tr>
<td><code>-2</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2^k-1</th>
<th>2^k-2</th>
<th>2^k-3</th>
<th>2^k-4</th>
<th>2^k-5</th>
</tr>
</thead>
<tbody>
<tr>
<td>non-negatives</td>
<td>negatives</td>
<td>one zero</td>
<td>how many positives?</td>
<td></td>
</tr>
<tr>
<td><code>00001</code></td>
<td><code>00010</code></td>
<td><code>00110</code></td>
<td><code>01010</code></td>
<td><code>11110</code></td>
</tr>
<tr>
<td><code>0</code></td>
<td><code>1</code></td>
<td><code>-1</code></td>
<td><code>-2</code></td>
<td><code>-4</code></td>
</tr>
</tbody>
</table>

Shortcomings of One’s complement?

- Arithmetic still a somewhat complicated.
- Still two zeros
  - `0x00000000 = +0_{ten}
  - 0xFFFFFFFF = -0_{ten}
- Although used for a while on some computer products, one’s complement was eventually abandoned because another solution was better.

Standard Negative # Representation

- Problem is the negative mappings “overlap” with the positive ones (the two 0s). Want to shift the negative mappings left by one.
- Solution! For negative numbers, complement, then add 1 to the result.
- As with sign and magnitude, & one’s compl. leading 0s ⇨ positive, leading 1s ⇨ negative
  - `00000...xxx` is ≥ 0, `111111...xxx` is < 0
  - except `11111` is -1, not -0 (as in sign & mag.)
- This representation is **Two’s Complement**
- This makes the hardware simple!
  - (C’s `int`, aka a “signed integer”)
  - (Also C’s `short`, long long, ..., C99’s `intN_t`)

Two’s Complement Formula

- Can represent positive and negative numbers in terms of the bit value times a power of 2:
  - `d_3 x (2^3) + d_2 x 2^2 + d_1 x 2^1 + d_0 x 2^0`
- Example: `1101` in a nibble?
  - `1 x (2^3) + 1 x 2^2 + 0 x 2^1 + 1 x 2^0`
  - `-8 + 4 + 0 + 1`
  - `-3_{ten}`
  - `-3_{ten}`

Example: `-3` to `-3` (again, in a nibble):

- `x : 1101_{two}
- +1 : 0011_{two}
- +1 : 1101_{two}
- +1 : 0011_{two}`
Bias Encoding: N = 5 (bias = -15)

- # = unsigned + bias
- Bias for N bits chosen as $-2^{(N-1)}$
- one zero
- how many positives?

How best to represent -12.75?

a) 2s Complement (but shift binary pt)

b) Bias (but shift binary pt)

c) Combination of 2 encodings

d) Combination of 3 encodings

e) We can’t

Shifting binary point means “divide number by some power of 2. E.g.,

$11_{10} = 1011.0_2 \rightarrow 10.110_2 = (11/4)_{10} = 2.75_{10}$

And in summary...
META: We often make design decisions to make HW simple

- We represent “things” in computers as particular bit patterns: N bits ⇒ 2^N things
- These 5 integer encodings have different benefits: 1s complement and sign/mag have most problems.
- unsigned (C99’s uintN_t):
  00000 ... 0000 000001 ... 01111 100000 ... 11111
- 2's complement (C99’s intN_t) universal, learn!
  0000000001 ... 01111
10000 ... 11111 11111
- Overflow: numbers we; computers finite, errors!

REFERENCE: 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 on paper
- Binary: what computers use; you will learn how computers do +, -, *, /
  - To a computer, numbers always binary
  - Regardless of how number is written:
    - $32_{ten} \Rightarrow 32_{hex} = 0x20 = 1000000$
    - Use subscripts “ten”, “hex”, “two” in book, slides when might be confusing

Two’s Complement for N=32

| 0000 ... 0000 0000 0000 | = 0_{ten} |
| 0000 ... 0000 0000 0001 | = 1_{ten} |
| 0111 ... 1111 1111 1111 | = 2,147,483,647_{ten} |
| 0111 ... 1111 1111 1110 | = 2,147,483,646_{ten} |
| 0111 ... 1111 1111 1101 | = 2,147,483,645_{ten} |
| 0111 ... 1111 1111 1100 | = 2,147,483,644_{ten} |
| 0111 ... 1111 1111 1111 | = 2,147,483,643_{ten} |
| 0111 ... 1111 1111 1110 | = 2,147,483,642_{ten} |
| 0111 ... 1111 1111 1101 | = 2,147,483,641_{ten} |
| 0111 ... 1111 1111 1100 | = 2,147,483,640_{ten} |
| 0111 ... 1111 1111 1111 | = 2,147,483,639_{ten} |
| 0111 ... 1111 1111 1110 | = 2,147,483,638_{ten} |
| 0111 ... 1111 1111 1101 | = 2,147,483,637_{ten} |
| 0111 ... 1111 1111 1100 | = 2,147,483,636_{ten} |

- One zero; 1st bit called sign bit
- 1 “extra” negative:no positive 2,147,483,648_{ten}

Two’s comp. shortcut: Sign extension

- Convert 2’s complement number rep. 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
  - Binary representation hides leading bits; sign extension restores some of them
  - 16-bit -4_{ten} to 32-bit: 1111 1111 1111 1100\_two

To a computer, numbers always binary
Regardless of how number is written:
- $32_{ten} \Rightarrow 32_{hex} = 0x20 = 1000000$
- Use subscripts “ten”, “hex”, “two” in book, slides when might be confusing

META: Ain’t no free lunch