CS61C – Machine Structures
Lecture 2 – Number Representation

8/27/2003
⇐ Brian Harvey
(www.cs.berkeley.edu/~bh)

John Wawrzynek ⇒
(Warznek)
(www.cs.berkeley.edu/~johnw)

www-inst.eecs.berkeley.edu/~cs61c/
From Lecture 1

° Pick up and read class info document.
° Get class account form and login by Friday.
° Look at class website often!

° Posted / handouts:
  • Reading for this week: P&H Ch1 & 4.1-2, K&R Ch1-4
  • Reading for next: K&R Ch5&6
  • HW1 due Wednesday
  • Lab this week
Decimal Numbers: Base 10

Digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Example:

3271 =  

\((3 \times 10^3) + (2 \times 10^2) + (7 \times 10^1) + (1 \times 10^0)\)
Numbers: positional notation

° Number Base $B \Rightarrow B$ symbols per digit:
  • Base 10 (Decimal): 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
  • Base 2 (Binary): 0, 1

° Number representation:
  • $d_{31}d_{30} \ldots d_2d_1d_0$ is a 32 digit number
  • value $= d_{31}x B^{31} + d_{30}x B^{30} + \ldots + d_2x B^2 + d_1x B^1 + d_0x B^0$

° Binary: 0, 1 (In binary digits called “bits”)
  • $1011010 = 1x2^6 + 0x2^5 + 1x2^4 + 1x2^3 + 0x2^2 + 1x2 + 0x1 = 64 + 16 + 8 + 2 = 90$
  • Notice that 7 digit binary number turns into a 2 digit decimal number
  • A base that converts to binary easily?
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 from the 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

° Example:
  • 1010 1100 0101 (binary) = ? (hex)
Decimal vs. Hexadecimal vs. Binary

Examples:

1010 1100 0101 (binary)
= AC5 (hex)

10111 (binary)
= 0001 0111 (binary) = 17 (hex)

3F9(hex)
= 11 1111 1001 (binary)

How do we convert between hex and Decimal?
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
Comparison

How do you tell if $X > Y$?
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:
    \[ 32_{10} == 0x20 == 100000_2 \]
  - Use subscripts “ten”, “hex”, “two” in book, slides when might be confusing
Limits of Computer Numbers

° 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 ⇒ 16 bits (“unicode”)

° Logical values?
  • 0 ⇒ False, 1 ⇒ True

° colors? Ex: Red (00) Green (01) Blue (11)

° locations / addresses? commands?

° but N bits ⇒ only \(2^N\) things
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:
  \[
  \begin{array}{c}
  000000000000000000000000001 \\
  \end{array}
  \]
- And $-1_{ten}$ in sign and magnitude would be:
  \[
  \begin{array}{c}
  1000000000000000000000000001 \\
  \end{array}
  \]
Shortcomings of sign and magnitude?

° Arithmetic circuit complicated
  • Special steps depending whether signs are the same or not

° Also, Two zeros
  • 0x00000000 = +0_{ten}
  • 0x80000000 = -0_{ten}
  • What would 2 0s mean for programming?

° Therefore 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 leading 1s.

  00000 00001 \ldots 01111

  10000 \ldots 11110 11111

- What is \(-00000\)? Answer: 11111

- How many positive numbers in N bits?

- How many negative ones?
Shortcomings of One’s complement?

° Arithmetic still a somewhat complicated.

° Still two zeros
  • $0x00000000 = +0_{\text{ten}}$
  • $0xFFFFFFFF = -0_{\text{ten}}$

° Although used for awhile on some computer products, one’s complement was eventually abandoned because another solution was better.
Standard Negative Number Representation

° 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
  - \(3 - 4 \Rightarrow 00...0011 - 00...0100 = 11...1111\)

• With no obvious better alternative, pick representation that made the hardware simple

• As with sign and magnitude, leading 0s \(\Rightarrow\) positive, leading 1s \(\Rightarrow\) negative
  - 000000...xxx is \(\geq 0\), 111111...xxx is \(< 0\)
  - except 1...1111 is -1, not -0 (as in sign & mag.)

° This representation is **Two’s Complement**
2’s Complement Number “line”: N = 5

- $2^{N-1}$ non-negatives
- $2^{N-1}$ negatives
- one zero
- how many positives?
## Two’s Complement for N=32

<table>
<thead>
<tr>
<th>Binary</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 ... 0000</td>
<td>0</td>
</tr>
<tr>
<td>0000 ... 0000</td>
<td>1</td>
</tr>
<tr>
<td>0000 ... 0000</td>
<td>2</td>
</tr>
<tr>
<td>0111 ... 1111</td>
<td>2,147,483,645</td>
</tr>
<tr>
<td>0111 ... 1111</td>
<td>2,147,483,646</td>
</tr>
<tr>
<td>0111 ... 1111</td>
<td>2,147,483,647</td>
</tr>
<tr>
<td>1000 ... 0000</td>
<td>-2,147,483,648</td>
</tr>
<tr>
<td>1000 ... 0000</td>
<td>-2,147,483,647</td>
</tr>
<tr>
<td>1000 ... 0000</td>
<td>-2,147,483,646</td>
</tr>
<tr>
<td>1111 ... 1111</td>
<td>-3</td>
</tr>
<tr>
<td>1111 ... 1111</td>
<td>-2</td>
</tr>
<tr>
<td>1111 ... 1111</td>
<td>-1</td>
</tr>
</tbody>
</table>

- One zero; 1st bit called **sign bit**
- 1 “extra” negative: no positive 2,147,483,648<sub>ten</sub>
Two’s Complement Formula

° Can represent positive and negative numbers in terms of the bit value times a power of 2:
  \[ d_{31} \times (-2^{31}) + d_{30} \times 2^{30} + \ldots + d_2 \times 2^2 + d_1 \times 2^1 + d_0 \times 2^0 \]

° Example: 1111 1100\text{two}

\[
= 1 \times -2^9 + 1 \times 2^8 + 1 \times 2^7 + \ldots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 \\
= -2^9 + 2^8 + 2^7 + \ldots + 2^2 + 0 + 0 \\
= -128 + 64 + 32 + 16 + 8 + 4 \\
= -128 + 12 \\
= -4_{\text{ten}}
\]
Two’s complement shortcut: Negation

- Change every 0 to 1 and 1 to 0 (invert or complement), then add 1 to the result.

- Proof: Sum of number and its (one’s) complement must be $111\ldots111_{\text{two}}$
  
  However, $111\ldots111_{\text{two}} = -1_{\text{ten}}$
  
  Let $x'$ $\Rightarrow$ one’s complement representation of $x$
  
  Then $x + x' = -1 \Rightarrow x + x' + 1 = 0 \Rightarrow x' + 1 = -x$

- Example: -4 to +4 to -4

| $x$ : 1111 1111 1111 1111 1111 1111 1111 1100 two |
| $x'$: 0000 0000 0000 0000 0000 0000 0000 0011 two |
| +1: 0000 0000 0000 0000 0000 0000 0000 0100 two |
| ()': 1111 1111 1111 1111 1111 1111 1111 1011 two |
| +1: 1111 1111 1111 1111 1111 1111 1111 1100 two |

Harvey / Wawrzynek Fall 2003 © UCB
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 $-4_{10}$ to 32-bit:

```
  1111 1111 1111 1100_{two}
  1111 1111 1111 1111 1111 1111 1111 1100_{two}
```
Signed vs. Unsigned Variables

- Java just declares integers `int`
  - Uses two’s complement

- C has declaration `int` also
  - Declares variable as a signed integer
  - Uses two’s complement

- Also, C declaration `unsigned int`
  - Declares a unsigned integer
  - Treats 32-bit number as unsigned integer, so most significant bit is part of the number, not a sign bit
Numbers represented in memory

° Memory is a place to store bits

° A word is a fixed number of bits (e.g., 32) at an address

° Addresses are naturally represented as unsigned numbers in C

11111 = 2^k - 1
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
What if too big?

- Binary bit patterns above are simply **representatives** of numbers. Strictly speaking they are called “numerals”.

- Numbers really have an infinite 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.

```
00000 00001 00010 00011 00100 00101 00110 00111 11110 11111
```

Unsigned
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, hexadecimal 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