CS33 Project

Gear Up

Data
Project Overview

You will be solving a series of puzzles using your knowledge of data representations.
IMPORTANT: Collaboration

- This project has a different collaboration policy than most projects in CS 33. Please review it!
- As a result, this Gear Up Session will be a little different -- we will not discuss any implementations of any of the code you will write.
- Come ask us questions after if you’re unsure.
Topics You Should Review

- Data representation (lecture slides, ch. 2 textbook)
  - Integers (Two’s Complement)
  - Bitwise operations
  - Bit masks and sign extension
  - Floating point
Roadmap

- Read the Worked Example to get an idea of the problems you will be solving
- Read the stencil carefully to know limitations on operators you can use for each puzzle
- Solve each of the 9 puzzles in bits.c
- Use the provided testers to test your solutions as you go
Testers

- **btest**
  - In the directory for the project, run `make btest; ./btest`
  - Tests functionality, does not check if you followed the coding rules

- **dpc**
  - In the directory for the project, run `./dpc bits.c`
  - Checks that your code follows the rules for each puzzle, and counts the number of operators

- **driver.pl**
  - In the directory for the project, run `./driver.pl`
  - Combines dpc and btest to check correctness and performance of your solution
Logical Operators

**logical NOT**

<table>
<thead>
<tr>
<th>A</th>
<th>NOT A</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>
Logical Operators (C)

logical **NOT**: !

<table>
<thead>
<tr>
<th>A</th>
<th>!A</th>
</tr>
</thead>
<tbody>
<tr>
<td>non-zero</td>
<td>00...00</td>
</tr>
<tr>
<td>zero</td>
<td>00..01</td>
</tr>
</tbody>
</table>
**Bitwise Operators (C)**

**bitwise NOT:** ~

<table>
<thead>
<tr>
<th>A</th>
<th>~A</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>00</td>
</tr>
<tr>
<td>10</td>
<td>01</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
</tr>
<tr>
<td>00</td>
<td>11</td>
</tr>
</tbody>
</table>
## Logical Operators

### logical **AND**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A AND B</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>
### Logical Operators (C)

**logical AND: `&&`**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A &amp;&amp; B</th>
</tr>
</thead>
<tbody>
<tr>
<td>non-zero</td>
<td>non-zero</td>
<td>1</td>
</tr>
<tr>
<td>non-zero</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>non-zero</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Bitwise Operators (C)

bitwise AND: &

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A &amp; B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Example: 1100 & 1010 = 1000
## Logical Operators

**logical OR**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A OR B</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>
### Logical Operators (C)

**logical OR:** `||`

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>A</strong></td>
<td><strong>B</strong></td>
<td>**A</td>
</tr>
<tr>
<td>non-zero</td>
<td>non-zero</td>
<td>1</td>
</tr>
<tr>
<td>non-zero</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>non-zero</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Bitwise Operations (C)

bitwise OR: |

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Example: 1100 | 1010 = 1110
Logical Operators

**logical XOR**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A XOR B</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>
# Bitwise Operations (C)

**bitwise XOR:** ^

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>A ^ B</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>-------</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Example 1100 ^ 1010 = 0110
Two’s Complement Representation

- The last bit (all the way on the left) is 0 if the number is positive, and 1 if it is negative.
- To calculate the decimal representation for a \( w \)-bit word size (\( b_i \) is the bit at index \( i \)):

\[
-b_{w-1}2^{w-1} + \sum_{i=0}^{w-2} b_i \cdot 2^i
\]
Two’s Complement Examples

● For a 4-bit word size...

○ 0000
  \[ 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 = 0 \]

○ 1000
  \[ -1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 = -8 \]

○ 1111
Two’s Complement Values

- Values to keep in mind:
  - All 0’s is 0
  - All 1’s is -1
  - 0 followed by all 1’s is the most positive integer
  - 1 followed by all 0’s is the most negative integer
Bit Shifts (C)

- Shifting the bits in some representation either left or right
- Left Shift: \( a \ll b \)
- Right Shift: \( a \gg b \)
Left Shift: $a \ll b$

1. Shift $a$ left $b$ bits, throwing away leading bits.
   a. How many bits are thrown away?

2. Fill in right bits with 0s.
   a. How many bits are 0s?
Left Shift: Examples

- For a 4-bit word size...
  - 1 << 1 = 2
    - 0001 << 1 = 0010
  - 1 << 3 = 8
    - 0001 << 3 = 1000
  - 7 << 2 = 12
    - 0111 << 2 = 1100
  - 8 << 1 = 0
    - 1000 << 1 = 0000
Right Shift: \( \gg \) (C)

\[ a \gg b \]

1. Shift \( a \) right \( b \) bits, throwing away trailing bits.
   a. How many bits are thrown away?
2. Logical shift or arithmetic shift?
   a. Logical: Fill in left bits with 0s.
      i. Used for unsigned integers
   b. Arithmetic: Fill in left bits with sign bit.
      i. Used for signed integers
Right Shift: Examples

● For a 5-bit word size, using two’s complement...
  ○ 00001 >> 1
    ■ unsigned (logical): 1 >> 1 = 00000 = 0
    ■ signed (arithmetic): 1 >> 1 = 00000 = 0
  ○ 00111 >> 2
    ■ unsigned (logical): 7 >> 2 = 00001 = 1
    ■ signed (arithmetic): 7 >> 2 = 00001 = 1
  ○ 11001 >> 2
    ■ unsigned (logical): 25 >> 2 = 00110 = 6
    ■ signed (arithmetic): -7 >> 2 = 11110 = -2
Bit Masking

A bit mask is an integer whose binary representation is intended to combine with another value using $\&$, $|$ or $^\wedge$ to extract or set a particular bit or set of bits.
Bit Masking with &

mask = 00000001
value1 = 10011011
value2 = 10011100

mask & value1 == 00000001
mask & value2 == 00000000
Bit Masking with $\mid$

mask = 00101000
value = 11000101

mask $\mid$ value == 11101101
Bit Masking with ^

mask = 11111111

value = 10101010

mask ^ value == 01010101
Multiplication using $\ll$

- You can multiply a number by a power of 2 by using left-shift ($\ll$).
  
  - Ex: $4 \times (2^3) = 00000100 \ll 3 = 00100000 = 32$
Dividing using `>>`

- You can use `>>` to divide an integer by a power of 2
- **Positive integers**
  - Ex: \(48/(2^4) = 00110000 \gg 4 = 00000011 = 3\)
  - Ex: \(48/(2^5) = 00110000 \gg 5 = 00000001 = 1\)
    - Throwing away trailing bits rounds down
- **Negative integers**
  - Ex: \(-48/(2^4) = 11010000 \gg 4 = 11111101 = -3\)
  - Ex: \(-48/(2^5) = 11010000 \gg 5 = 11111110 = -2\)
    - **WRONG!** Just using `>>` on a negative number rounds away from zero (bad)
Correct Rounding for Negative Ints

● Problem: we want to increment our answer only in the case where there’s rounding away from 0

● $-48 / (2^5) = \underbrace{11010000}_{\text{trailing bits}} >> 5 = \underbrace{11111111}_{\text{leading bits}}$

  ○ Only if there is a one in the trailing bits that will be cut off (ie: the answer will be rounded), then the leading bits need to be incremented by 1.

  ○ How tho?? Hint: check the lecture slides ;)}
Tips

- Try to follow the process used in the worked example.
- Understanding the lectures on bit manipulation and two’s complement will be very important for this project.
- Making a bit mask using the sign bit will be very useful: \((x \gg 31)\)
- Come to TA hours if you have questions! TAs will try to help without giving away the answer.
So Why Am I Doing This?

- You will gain a better understanding of bit-level representation of data.
- You will gain a better understanding of how the computer computes functions like negating a number or checking if numbers are equal.
- You will understand the various bit-level operations
Questions?

"Says we can buy the extra table leg for £5.99"

"Fantastic"

@jeromecaple