Bitwise Data Manipulation

Bitwise operations
More on integers

**Aside: sets as bit vectors**

**Representation:** $n$-bit vector gives subset of $\{0, ..., n-1\}$.

$a_i = 1 \equiv i \in A$

- $01101001$ \{0, 3, 5, 6\}
- $01010101$ \{0, 2, 4, 6\}

**Bitwise Operations**
- $\&$ 01000001 \{0, 6\}
- $|$ 01111101 \{0, 2, 3, 4, 5, 6\}
- $^\wedge$ 00111100 \{2, 3, 4, 5\}
- $\sim$ 10101010 \{1, 3, 5, 7\}

<table>
<thead>
<tr>
<th>Set Operations?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intersection</td>
</tr>
<tr>
<td>Union</td>
</tr>
<tr>
<td>Symmetric diff.</td>
</tr>
<tr>
<td>Complement</td>
</tr>
</tbody>
</table>

**bitwise operators**

**Bitwise operators** on fixed-width **bit vectors**.

AND $\&$   OR $|$   XOR $^\wedge$   NOT $\sim$

<table>
<thead>
<tr>
<th>01101001 &amp; 01010101</th>
<th>01101001 &amp; 01010101</th>
<th>01101001 $^\wedge$ 01010101</th>
<th>01010101 $\sim$ 01010101</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000001</td>
<td>01010101</td>
<td>01010101</td>
<td>01010101</td>
</tr>
</tbody>
</table>

Laws of Boolean algebra apply bitwise.

*e.g.*, DeMorgan’s Law: $\sim(A \mid B) = \sim A \& \sim B$

**bitwise operators in C**

$\&$ $|$ $^\wedge$ $\sim$ apply to any **integral** data type

*long, int, short, char, unsigned*

Examples (**char**)

$\sim 0x41 =$

$\sim 0x00 =$

$0x69 \& 0x55 =$

$0x69 \mid 0x55 =$

Many bit-twiddling puzzles in upcoming assignment
**logical operations in C**

& & || !

apply to any "integral" data type

long, int, short, char, unsigned

0 is false nonzero is true result always 0 or 1

early termination a.k.a. short-circuit evaluation

Examples (**char**)

!0x41 =
!0x00 =
!!0x41 =

0x69 & & 0x55 =
0x69 || 0x55 =

**Encode playing cards.**

52 cards in 4 suits

How do we encode suits, face cards?

What operations should be easy to implement?

Get and compare rank
Get and compare suit

**Two possible representations**

52 cards – 52 bits with bit corresponding to card set to 1

52 bits in 2 x 32-bit words

“One-hot" encoding

Hard to compare values and suits independently
Not space efficient

4 bits for suit, 13 bits for card value – 17 bits with two set to 1

Pair of one-hot encoded values

Easier to compare suits and values independently
Smaller, but still not space efficient

**Two better representations**

Binary encoding of all 52 cards – only 6 bits needed

Number cards uniquely from 0
Smaller than one-hot encodings.
Hard to compare value and suit

Binary encoding of suit (2 bits) and value (4 bits) separately

Number each suit uniquely
Number each value uniquely
Still small
Easy suit, value comparisons
**Compare Card Suits**

*mask:* a bit vector that, when bitwise ANDed with another bit vector \( v \), turns all but the bits of interest in \( v \) to 0

```c
#define SUIT_MASK 0x30

int sameSuit(char card1, char card2) {
    return !((card1 & SUIT_MASK) ^ (card2 & SUIT_MASK));
    // same as (card1 & SUIT_MASK) == (card2 & SUIT_MASK);
}
```

```c
char hand[5];       // represents a 5-card hand
char card1, card2;  // two cards to compare
... if ( sameSuit(hand[0], hand[1]) ) { ... }
```

---

**Compare Card Values**

*mask:* a bit vector that, when bitwise ANDed with another bit vector \( v \), turns all but the bits of interest in \( v \) to 0

```c
#define VALUE_MASK

int greaterValue(char card1, char card2) {
    return !((card1 & VALUE_MASK) ^ (card2 & VALUE_MASK));
    // same as (card1 & VALUE_MASK) == (card2 & VALUE_MASK);
}
```

```c
char hand[5];       // represents a 5-card hand
char card1, card2;  // two cards to compare
... if ( greaterValue(hand[0], hand[1]) ) { ... }
```

---

**Bit shifting**

- **Logical shift left:**  
  - `x << 2`  
  - Result: `01100100`  
  - Lose bits on left, fill with zeroes on right

- **Logical shift right:**  
  - `x >> 2`  
  - Result: `10011001`  
  - Lose bits on right, fill with zeroes on left

- **Arithmetic shift right:**  
  - `x >> 2`  
  - Result: `11101101`  
  - Lose bits on right, fill with copies of MSB on left

---

**unsigned shifting and arithmetic**

- **Logical shift left:**  
  - `x = 27;`  
  - Result: `00011011`

- **Logical shift right:**  
  - `y = x >> 2;`  
  - Result: `00110110`

- **Arithmetic shift right:**  
  - `x = 237;`  
  - Result: `11101101`

- **Logical shift right:**  
  - `y = x >> 2;`  
  - Result: `00111011`

- **Arithmetic shift right:**  
  - `x = 11;`  
  - Result: `11101101`  
  - Result: `00111011`
two's complement shifting and arithmetic

**Signed**

\[ x = -101; \]
\[ y = x << 2; \]
\[ y == 108 \]

Logical shift left

\[ 1 0 0 1 1 0 1 1 \]

**Arithmetic**

Shift right

\[ 1 1 1 0 1 1 0 1 \]

**Signed**

\[ x = -19; \]
\[ y = x >> 2; \]
\[ y == -5 \]

Logical or arithmetic shift right: how do we tell?

**C:** compiler chooses

- Usually based on type: rain check!

**Java:** `>>` is arithmetic, `>>>` is logical

Shift an \( n \)-bit type by at least 0 and no more than \( n-1 \).

**C:** other shift distances are undefined.
- anything could happen

**Java:** shift distance is used modulo number of bits in shifted type

Given int \( x \): \( x << 34 == x << 2 \)

Ex: Shift and Mask: extract a bit field

Write C code:

extract 2\textsuperscript{nd} most significant byte from a 32-bit integer.

Given \( x = \)

should return:

All other bits are zero. Desired bits in least significant byte.

### Shift-and-add

Available operations

\[ x << k \quad \text{implements} \quad x \times 2^k \]
\[ x + y \]

Implement \( y = x \times 24 \) using only \(<, +, \) and integer literals
What does this function compute?

```c
unsigned puzzle(unsigned x, unsigned y) {
    unsigned result = 0;
    for (unsigned i = 0; i < 32; i++) {
        if (y & (1 << i)) {
            result = result + (x << i);
        }
    }
    return result;
}
```

**ex**

**multiplication**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0010</td>
</tr>
<tr>
<td>x 3</td>
<td>x 0011</td>
</tr>
<tr>
<td>6</td>
<td>00000100</td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-2</td>
<td>1110</td>
</tr>
<tr>
<td>x 2</td>
<td>x 0010</td>
</tr>
<tr>
<td>-4</td>
<td>11111100</td>
</tr>
</tbody>
</table>

**Modular Arithmetic**

**multiplication**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>x 4</td>
<td>x 0100</td>
</tr>
<tr>
<td>20</td>
<td>00010100</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-3</td>
<td>1101</td>
</tr>
<tr>
<td>x 7</td>
<td>x 0111</td>
</tr>
<tr>
<td>-21</td>
<td>11101011</td>
</tr>
<tr>
<td>-2</td>
<td></td>
</tr>
</tbody>
</table>

**Modular Arithmetic**

**multiplication**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0101</td>
</tr>
<tr>
<td>x 5</td>
<td>x 0101</td>
</tr>
<tr>
<td>25</td>
<td>00011001</td>
</tr>
<tr>
<td>-7</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-2</td>
<td>1110</td>
</tr>
<tr>
<td>x 6</td>
<td>x 0110</td>
</tr>
<tr>
<td>-12</td>
<td>11110100</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

**Modular Arithmetic**
Convert/cast signed number to larger type.

0 0 0 0 0 0 1 0 8-bit 2

_ _ _ _ _ _ _ _ _ _ 0 0 0 0 0 0 1 0 16-bit 2

1 1 1 1 1 1 0 0 8-bit -4

_ _ _ _ _ _ _ _ _ _ 1 1 1 1 1 1 0 0 16-bit -4

Rule/name?

Casting Integers in C

Number literals: 37 is signed, 37U is unsigned

Integer Casting: bits unchanged, just reinterpreted.

Explicit casting:

```c
int tx = (int) 73U;  // still 73
unsigned uy = (unsigned) -4;  // big positive #
```

Implicit casting:

```c
if (tx < uy) ... // if ((unsigned) tx < ux) ...
```

More Implicit Casting in C

If you mix unsigned and signed in a single expression, then signed values are implicitly cast to unsigned.

<table>
<thead>
<tr>
<th>Argument₁</th>
<th>Op</th>
<th>Argument₂</th>
<th>Type</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>==</td>
<td>0U</td>
<td>unsigned</td>
<td>1</td>
</tr>
<tr>
<td>-1</td>
<td>&lt;</td>
<td>0</td>
<td>signed</td>
<td>1</td>
</tr>
<tr>
<td>-1</td>
<td>&lt;</td>
<td>0U</td>
<td>unsigned</td>
<td>0</td>
</tr>
<tr>
<td>2147483647</td>
<td>&lt;</td>
<td>-2147483648</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2147483647</td>
<td>&lt;</td>
<td>-2147483648</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>&lt;</td>
<td>-2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(unsigned)-1</td>
<td>&lt;</td>
<td>-2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2147483647</td>
<td>&lt;</td>
<td>2147483648U</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2147483647</td>
<td>&lt;</td>
<td>(int)2147483648U</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: \( T_{\text{min}} = -2,147,483,648 \quad T_{\text{max}} = 2,147,483,647 \)