Bitwise Data Manipulation

Bitwise operations
More on integers
**bitwise operators**

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

**AND &**  **OR |**  **XOR ^**  **NOT ~**

\[
\begin{array}{c}
01101001 \\
& 01010101 \\
01000001 \\
\end{array}
\quad
\begin{array}{c}
01101001 \\
| 01010101 \\
\end{array}
\quad
\begin{array}{c}
01101001 \\
^ 01010101 \\
\end{array}
\quad
\begin{array}{c}
~ 01010101 \\
\end{array}
\]

Laws of Boolean algebra apply bitwise.

*e.g.*, DeMorgan’s Law:  \(~(A | B) = ~A & ~B\)
Aside: *sets as bit vectors*

**Representation:** An *n*-bit vector gives a subset of \( \{0, \ldots, n-1\} \).

\[
a_i = 1 \equiv i \in A
\]

<table>
<thead>
<tr>
<th>Bit</th>
<th>Bitwise Notation</th>
<th>Equivalent Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>01000001</td>
<td>{ 0, 6 }</td>
</tr>
<tr>
<td></td>
<td>01111101</td>
<td>{ 0, 2, 3, 4, 5, 6 }</td>
</tr>
<tr>
<td>^</td>
<td>00111100</td>
<td>{ 2, 3, 4, 5 }</td>
</tr>
<tr>
<td>~</td>
<td>10101010</td>
<td>{ 1, 3, 5, 7 }</td>
</tr>
</tbody>
</table>
**bitwise operators in C**

& | ^ ~

apply to any *integral* data type
long, int, short, char, unsigned

Examples *(char)*
~0x41 =

~0x00 =

0x69 & 0x55 =

0x69 | 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

“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

52 bits in 2 x 32-bit words
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) {
    ...}

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

\[ x \ll 2 \]

\[ \begin{array}{c}
0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\
\end{array} \]

logical shift left 2

lose bits on left

fill with zeroes on right

\[ x \gg 2 \]

\[ \begin{array}{c}
1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\
0 & 1 \\
\end{array} \]

logical shift right 2

fill with zeroes on left

arithmetic shift right 2

fill with copies of MSB on left

\[ 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\
0 & 1 \\
\]
unsigned **shifting** and **arithmetic**

**unsigned**

\[ x = 27; \]
\[ y = x \ll 2; \]
\[ y == 108 \]

\[ x = 237; \]
\[ y = x \gg 2; \]
\[ y == 59 \]
two's complement shifting and arithmetic

**signed**
x = -101;
y = x << 2;
y == 108

**logical shift left**

**arithmetic shift right**

**signed**
x = -19;
y = x >> 2;
y == -5
**shift-and-add**

Available operations

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

Implement \( y = x \times 24 \) using only \( \ll, +, \) and integer literals.
Shift gotchas

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`: \( x << 34 = x << 2 \)
Shift and Mask: extract a bit field

Write C code:

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

given 
\[ x = \begin{array}{c}
01100001 & 01100010 & 01100011 & 01100100 \\
\end{array} \]

should return:
\[ \begin{array}{c}
00000000 & 00000000 & 00000000 & 01100010 \\
\end{array} \]

All other bits are zero. Desired bits in least significant byte.
What does this function compute?

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;
}
multiplication

\[
\begin{array}{c c}
2 & 0010 \\
x 3 & \times 0011 \\
6 & 00000100 \\
\hline \\
-2 & 1110 \\
x 2 & \times 0010 \\
-4 & 11111100 \\
\end{array}
\]
multiplication

\[
\begin{array}{c}
5 \\
\times 4 \\
\hline
20 \\
\end{array}
\quad 0101
\]

\[
\begin{array}{c}
0001 \\
\times 0100 \\
\hline
00010100 \\
\end{array}
\]

\[
\begin{array}{c}
-3 \\
\times 7 \\
\hline
-21 \\
\end{array}
\quad 1101
\]

\[
\begin{array}{c}
-2 \\
\times 0111 \\
\hline
11101011 \\
\end{array}
\]

Modular Arithmetic
multiplication

\[
\begin{array}{c}
5 \\
\times 5 \\
\hline
25 \\
\hline
\text{Red} -7 \\
\hline
\text{Red} -2 \\
\times 6 \\
\hline
\text{Red} -12 \\
\hline
4 \\
\end{array}
\]

\[
\begin{array}{c}
0101 \\
\times 0101 \\
\hline
00011001 \\
\hline
1110 \\
\times 0110 \\
\hline
111101000 \\
\hline
0001 \\
\end{array}
\]

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: \texttt{37} is signed, \texttt{37U} is unsigned

Integer Casting: \textit{bits unchanged, just reinterpreted.}

Explicit casting:

\begin{verbatim}
int tx = (int) 73U; // still 73
unsigned uy = (unsigned) -4; // big positive #
\end{verbatim}

Implicit casting: Actually does

\begin{verbatim}
tx = ux; // tx = (int)ux;
uy = ty; // uy = (unsigned)ty;
void foo(int z) { ... }
foo(ux); // foo((int)ux);
if (tx < ux) ... // if ((unsigned)tx < ux) ...
\end{verbatim}
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>Argument&lt;sub&gt;1&lt;/sub&gt;</th>
<th>Op</th>
<th>Argument&lt;sub&gt;2&lt;/sub&gt;</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>unsigned</td>
<td>1</td>
</tr>
<tr>
<td>2147483647U</td>
<td>&lt;</td>
<td>-2147483648</td>
<td>unsigned</td>
<td>1</td>
</tr>
<tr>
<td>-1</td>
<td>&lt;</td>
<td>-2U</td>
<td>unsigned</td>
<td>0</td>
</tr>
<tr>
<td>(unsigned)-1</td>
<td>&lt;</td>
<td>-2U</td>
<td>unsigned</td>
<td>0</td>
</tr>
<tr>
<td>2147483647</td>
<td>&lt;</td>
<td>2147483648U</td>
<td>unsigned</td>
<td>0</td>
</tr>
<tr>
<td>2147483647</td>
<td>&lt;</td>
<td>(int)2147483648U</td>
<td>unsigned</td>
<td>0</td>
</tr>
</tbody>
</table>

**Note:** $T_{\text{min}} = -2,147,483,648$  \hspace{1cm} $T_{\text{max}} = 2,147,483,647$