# CSE 351 Lecture 4 – Data III & Integers I

## Boolean Algebra

Boolean Algebra is an algebraic representation of logic, where True is represented by 1 and False by 0. The C ***bitwise operators*** apply Boolean operations to the *bits* of the operand(s) and can only be used on integral data types (*e.g.* `char`, `short`, `int`, `long`).

| & | 0 | 1 |
|---|---|---|
| 0 | **0** | **0** |
| 1 | **0** | **1** |

← **AND** (`&`) outputs a 1 only when both input bits are 1.

| \| | 0 | 1 |
|---|---|---|
| 0 | **0** | **1** |
| 1 | **1** | **1** |

**OR** (`|`) outputs a 1 when either input bit is 1. →

| ^ | 0 | 1 |
|---|---|---|
| 0 | **0** | **1** |
| 1 | **1** | **0** |

← **XOR** (`^`) outputs a 1 when either input is *exclusively* 1.

| ~ | |
|---|---|
| 0 | **1** |
| 1 | **0** |

**NOT** (`~`) outputs the opposite of its input. →

In contrast, C ***logical operators*** (AND: `&&`, OR: `||`, NOT: `!`) apply Boolean operations to the *value* of the operand(s). C treats a value of `0` (in any form, including `NULL` and `'\0'`) as False, and *anything else* as True. Despite this, the logical operators will always return the value of `1` for True. The logical operators exhibit ***short-circuit evaluation***, meaning that the 2nd operand is never computed if the outcome is determined by the 1st operand (*e.g.* `0 && x` or `1 || x`).

---

## Unsigned Integers

***Unsigned integers*** represent non-negative (*i.e.* counting up from `0`) integers. The bit pattern used matches regular binary numerals, just in a fixed-width based on the data type used. The different integer widths (assuming a 64-bit machine) are:

| | |
|---|---|
| `unsigned char` | 1 byte |
| `unsigned short` | 2 bytes |
| `unsigned int` | 4 bytes |
| `unsigned long` | 8 bytes |

For any given integer width (assuming $n$ bits), we can only represent $2^n$ things, so for unsigned integers that covers the range 0 to $2^n - 1$.

We can perform addition and subtraction in binary much the same way that we do in decimal, except that the *carry* and *borrow* happen at the value of 2 instead of 10. See the following 8-bit examples:

$$\begin{array}{r} {\scriptstyle 1\phantom{1}} \\ 63 \\ +\ 8 \\ \hline 71 \end{array} \Leftrightarrow \begin{array}{r} {\scriptstyle 11\phantom{1}} \\ 00111111_2 \\ +00001000_2 \\ \hline 01000111_2 \end{array} \qquad \begin{array}{r} {\scriptstyle 4\,10} \\ 55 \\ -\ 8 \\ \hline 47 \end{array} \Leftrightarrow \begin{array}{r} {\scriptstyle 02\phantom{1}} \\ 00110111_2 \\ -00001000_2 \\ \hline 00101111_2 \end{array}$$

## Two's Complement Integers

***Signed integers*** represent both positive and negative integers. There are many possible encodings, but the accepted implementation in all modern computers is ***Two's Complement***. This is achieved by making the weight of only the most significant bit *negative* ($-2^{n-1}$ instead of $2^{n-1}$) while keeping the weight of the other bits as in standard binary (or unsigned). This representation has the following advantages:

- Zero has the encoding of all zeros (`0b0…0`).
- It represents roughly the same number of positive and negative numbers ($-2^{n-1}$ to $2^{n-1}-1$).
- The encodings for the positive numbers exactly match the encodings in unsigned.
- It has the following negation procedure: **`-x == ~x + 1`** ("flip the bits and add one").

See the following 8-bit example:

`0b10001110` $= -2^7+2^3+2^2+2^1 =$ **`-114`**

`-(0b10001110) = ~(0b10001110) + 1 = 0b01110001 + 1 = 0b01110010`

$= 2^6+2^5+2^4+2^1 =$ **`114`**