# BIT TRICKS

Different uses & applications of Bit operations!

# THE BASIC BIT OPERATIONS

- Recall CSC/EEE 120 – Digital Design and Logic:
  - AND → '&' like `a & b`
  - OR → '|' like `a | b`
  - XOR → '^' like `a ^ b`
  - NOT → '~' like `~a`
- Bit operations in programming languages apply to ALL bits for a particular integer type. [IE `int` is 32 bits so a & b will do an AND operation on all 32 bits.]

# QUICK REVIEW OF THE TRUTH TABLES

### AND Truth Table

| A | B | Y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

### OR Truth Table

| A | B | Y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

### XOR Truth Table

| A | B | Y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

### NOT Truth Table

| A | B |
|---|---|
| 0 | 1 |
| 1 | 0 |

# AND, OR, XOR OPERATIONS IN ACTION

```
      6 & 11
    0110 (decimal 6)
AND 1011 (decimal 11)
  = 0010 (decimal 2)
```

```
      5 | 3
   0101 (decimal 5)
OR 0011 (decimal 3)
 = 0111 (decimal 7)
```

```
      5 ^ 3
    0101 (decimal 5)
XOR 0011 (decimal 3)
  = 0110 (decimal 6)
```

```
      6 & 1
    0110 (decimal 6)
AND 0001 (decimal 1)
  = 0000 (decimal 0)
```

```
      2 | 8
   0010 (decimal 2)
OR 1000 (decimal 8)
 = 1010 (decimal 10)
```

```
      2 ^ 10
    0010 (decimal 2)
XOR 1010 (decimal 10)
  = 1000 (decimal 8)
```

# GETTING STARTED WITH BASICS

- Setting a bit:
  - `a |= b;` (b can be either a number, a variable, or expression)
  - Ex: `a |= 2;` → this sets the 2nd bit index to 1, even if it's already 1.
  - `a |= 3;` → sets 2nd and 1st bit indexes to 1 at the same time.
- Testing/Checking a bit:
  - `a & b;` → if 0, the bit is 0. if non-zero, the bit is 1.
  - Usually used in if-statements with boolean condition: `if( (a & b) > 0 )`
  - Ex: `a & 2` → value > 0 because I set 'a''s 2nd bit index to 1.
- Toggling a bit:
  - `a ^ b;`
  - Recall that XORing a number with the same number always yields 0.
  - `a ^ a = 0`
- Clearing a bit:
  - `a &= ~b;` → Combination of using AND with NOT.
  - Ex: `a &= ~2;` This will set the 2nd bit index to 0 even if it's already 0, exact opposite of setting a bit.
  - *Go* programming language (Golang) has special operator for clearing bits: `&^` → `a &^ b` .

# EXTENDED BASICS

- `a & 1` → Checks if a number is even or odd.
  - Result of 0 means it's even. Result > 0 means it's odd.
  - `a &= ~1;` → make the data even.
  - `a |= 1;` → make the data odd.
- `a & 0x80000000` → Checks if a 32-bit integer/float is negative [is the sign bit on]. Result > 0 → sign bit is on.
  - `a & 0x8000` → checks for 16-bit integer/float.
  - `a & 0x80` → checks for byte integer/float.
- `a & (a - 1)` → Checks if a number is a power of 2. If result is 0, it's a power of 2.
- `a & (2^n - 1)` → `a % 2^n, a mod 2^n`.
  - Modulo'ing with a power of 2 is the same as ANDing with the power of 2 minus 1.
  - `a % 32` is the same as doing `a & 31`.
- Alphabet bit twiddling:
  - ORing with space character (`a | ' '`) converts letter to lowercase.
  - ANDing with underscore (`a & '_'`) converts letter to uppercase.
  - XORing with space character (`a ^ ' '`) toggles the letters case.
  - ANDing with 31 (`a & 31`) gives letter's position in alphabet (regardless of letter case).
  - ANDing with question mark (`a & '?'`) gets uppercase letter's position in alphabet.
  - XORing with backtick (``a ^ '`'``) gets lowercase letter's position in alphabet.

# BITSHIFTING

- *Not in CSC120*:
- Bit Shifting → **`<<, >>, >>>`** (Java uses >>>)
  - Example: **`a >> b`**, **`a << b`**, **`a >>> b`**
  - Moves/Shifts over bits to the right or to the left.
  - Three kinds: Arithmetic, Logical, Circular/Rotational.

# BASIC BIT-SHIFTING

- `a << b;` → left logical & arithmetic shift.
- `1 << n` → $2^n$(very useful for quickly making powers of 2)
- `a << n` →$a * 2^n$ , `a >> n` → $\frac{a}{2^n}$ aka $a * 2^{-n}$ (right <u>logical</u> shift)
  - For Java + others, **>>>** is for right logical shifting where **>>** is an arithmetic shift.
  - Recall: If last bit is 1, arithmetic shift copies the 1 down over.
- **`a << 1;`** → $2a$ , **`a >> 1;`** → $\frac{a}{2}$
- Circular Shifts:
  - Circular Shift Left: `(a << b) | (a >> (-b & (bits_in_type - 1)));`
  - Circular Shift Right: `(a >> b) | (a << (-b & (bits_in_type - 1)));`

# MORE COMPLEX TRICKS

- Aligning a number to a certain multiple of a number.
  - `(a + (b - 1)) & ~(b - 1);` where a is the number and b is the multiples you want to align a to. Works best when aligning numbers to multiples of powers of 2.
- Bitwise Ceiling: flips all trailing bits to 1.
  - If you add 1 to the result, it's a power of 2!
  - Bitwise ceiling result + 1 = gets next power of 2.

```
x |= x >> 1;
x |= x >> 2;
x |= x >> 4;
x |= x >> 8;
x |= x >> 16;
```

- Bit-Scan Reverse (start at highest set bit, count its bit index)
  - XOR result with bits-1 gives Number of Leading Zeroes.
  - Performs log base-2!

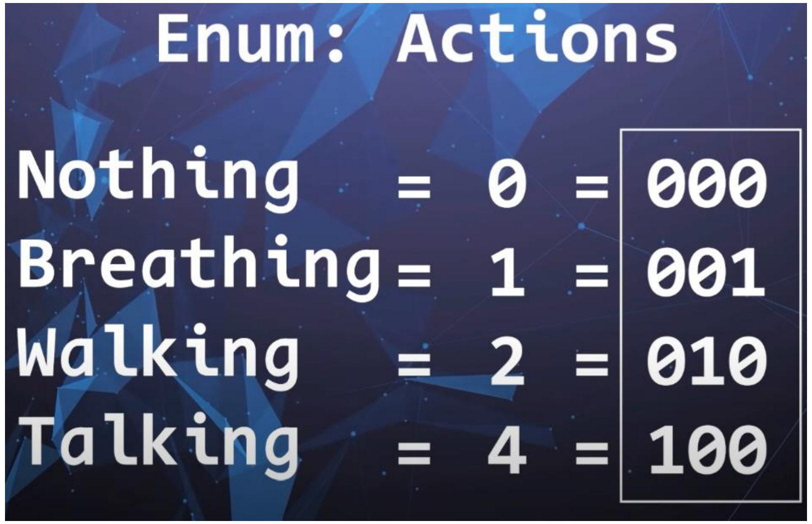
Secret: `1 << BitScanReverse(x);`
Gives previous power of 2!

```c
/// C
int BitScanReverse(int n) {
	int bit_index = 0;
	for( int i = (unsigned)(n) >> 1; i > 0; i >>= 1 ) {
		bit_index++;
	}
	return bit_index;
}
```

```java
/// Java
public static int BitScanReverse(int n) {
	int bit_index = 0;
	for( int i = n >>> 1; i > 0; i >>>= 1 ) {
		bit_index++;
	}
	return bit_index;
}
```

# MORE COMPLEX TRICKS PT. 2

- `x = a ^ b ^ x;` → `if( x==a ) x=b; else if( x==b ) x=a;` equivalent
- `(x ^ y) >= 0;` → checks if two values have the same sign.
- `(x + y) >> 1;` → gets the average of two integer numbers.
  - Likewise: `(x + y + z + w) >> 2;` → get average of four integer numbers.
- Using bits as individual, simultaneous Boolean values (Bit Flags):
  - Good for representing states → good for making state machines.
  - 8-bit NES used bit flags to check what buttons were pressed.
  - Chess games use an array of size 8 of 64-bit integers called a Bit Board.
  - Used in RPGs for quest steps.




- What if Bit Flags were combined with arrays of integers?
  - Bit Sets.
  - Can be treated as a giant variable of N*(number of bits in an `int`)
  - Bit index can be converted to an array index (slot) and bitflag using bitwise ops.
  - `bit_index & (bits_in_integer_type - 1);` → gives us the bitflag (Notice anything here?)
  - `bit_index (logical right shift) log2(bits_in_integer_type);` → gives us the array slot
  - Array length can be statically computed by the desired number of bits divided by the number of bits in integer type.
  - Converting bitflag and array slot back to the bit index → `(slot << log2(bits_in_integer_type)) + bitflag;`

# FAST INTEGER DIVISION

- Integer division → <u>*Slow*</u>…
  - Can be Faster but takes a lot of physical space for the processor. This means more $$$
  - Taking up less space means slower division, cheaper but processor takes more cycles to complete.
  - Space-Time tradeoff in physical hardware.
- How can we do faster integer division?
  - Not much you can do unfortunately.
  - However,… You can do faster integer division <u>if you're dividing by a constant</u>.
- Here's how to do it.
  - Figure out what constant you want to make a reciprocal of, we'll call it **b**.
  - Follow this formula: $[2^{16}/b] + 1$. Using 3 as an example. $\left(\frac{2^{16}}{3}\right) + 1 = 21846$.
  - Whatever you want to divide by 3, multiply it with the constant you made: $a * 21846 = c$
  - Divide **c** by $2^{16}$ (by shifting **c** with 16) and it'll give you the integer division of $\frac{a}{3}$

# CONCLUSION

- In your career, you'll come across many situations where mathematically, you will have to implement formulas and operations in code or make it work from a computer.
- Bitwise operations will help make those formulas perform faster.
- **Have a good break & see you guys next year.**
- Next Semester – 1st Workshop:
- Text input calculator:
  - We will make a calculator read: '1 + ln e' and reply '2'.