Data Representation

Numerals

- A numeral is a symbolic representation of numbers.
  - Simply, a **sequence of digits**

- The value of a n-digit numeral $d_{n-1}d_{n-2}\ldots d_0$ in base b is $\sum_{i=0}^{n-1} d_i b^i$

  - Example 1: 0b1001 = $1 * 2^3 + 0 * 2^2 + 0 * 2^1 + 1 * 2^0 = 8 + 1 = 9$
  - Example 2: 0x16 = $1 * 16^1 + 6 * 16^0 = 16 + 6 = 22$

Binary	Decimal	Hex
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

Decimal to Hexadecimal

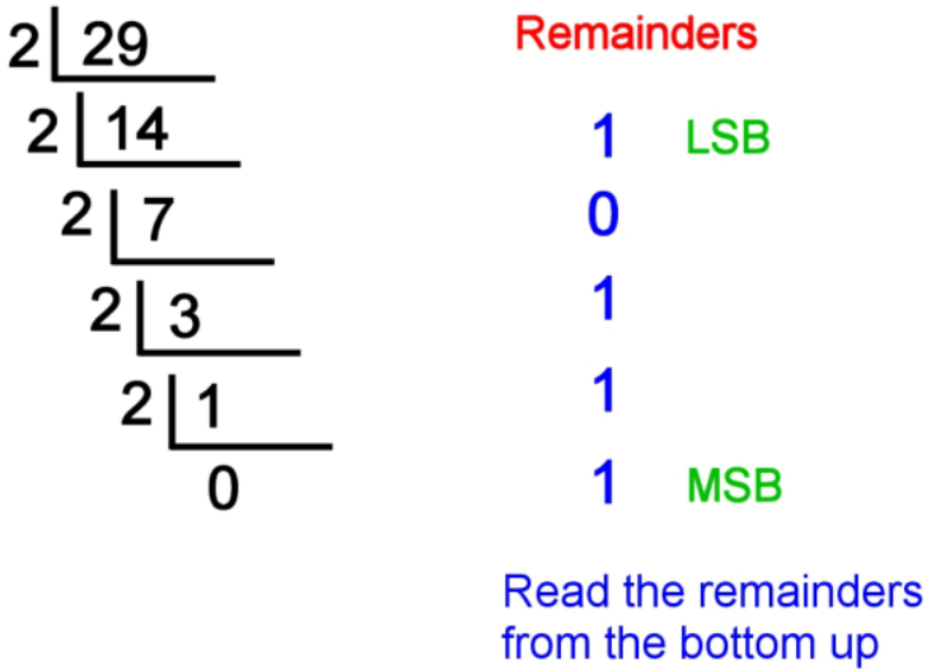
DIVISION	RESULT	REMAINDER (in HEX)
921 / 16	57	9
57 / 16	3	9
3 / 16	0	3
ANSWER		399

DIVISION	RESULT	REMAINDER (HEX)
590 / 16	36	E (14 decimal)
36 / 16	2	4 (4 decimal)
2 / 16	0	2 (2 decimal)
ANSWER		24E

https://www.permadi.com/tutorial/numDecToHex/

Decimal to Binary

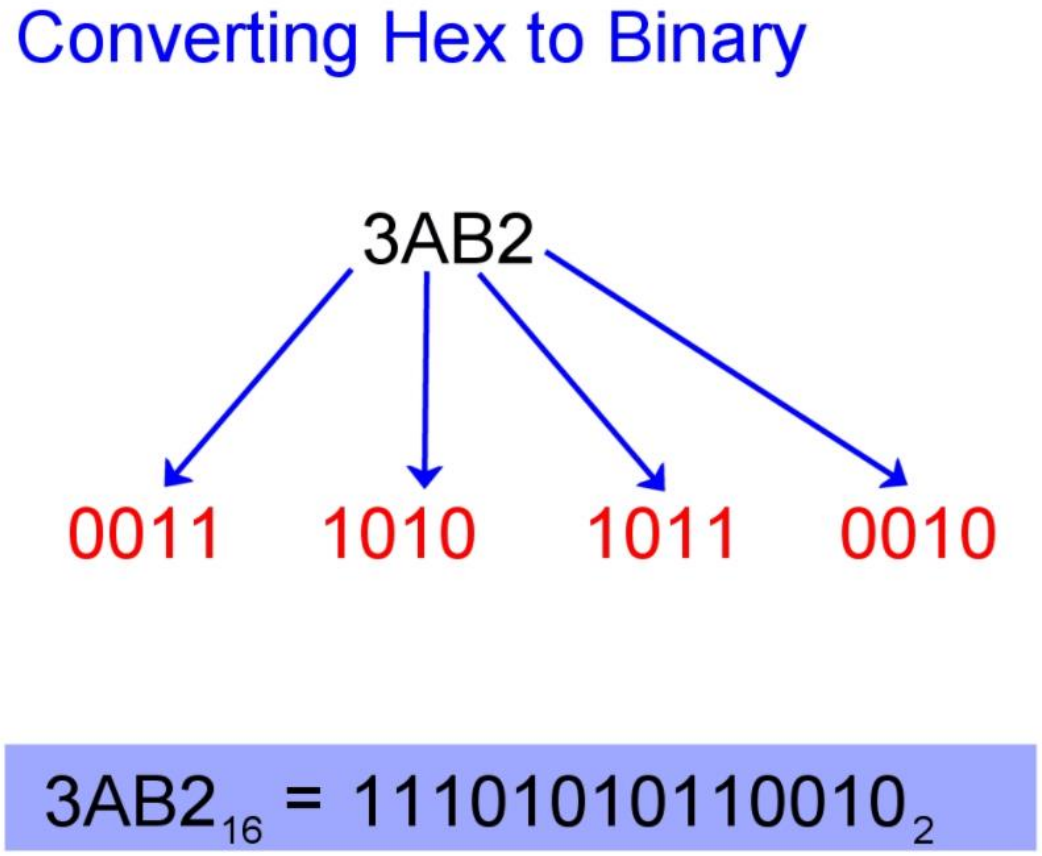
Successive Division by 2



29 decimal = 11101 binary

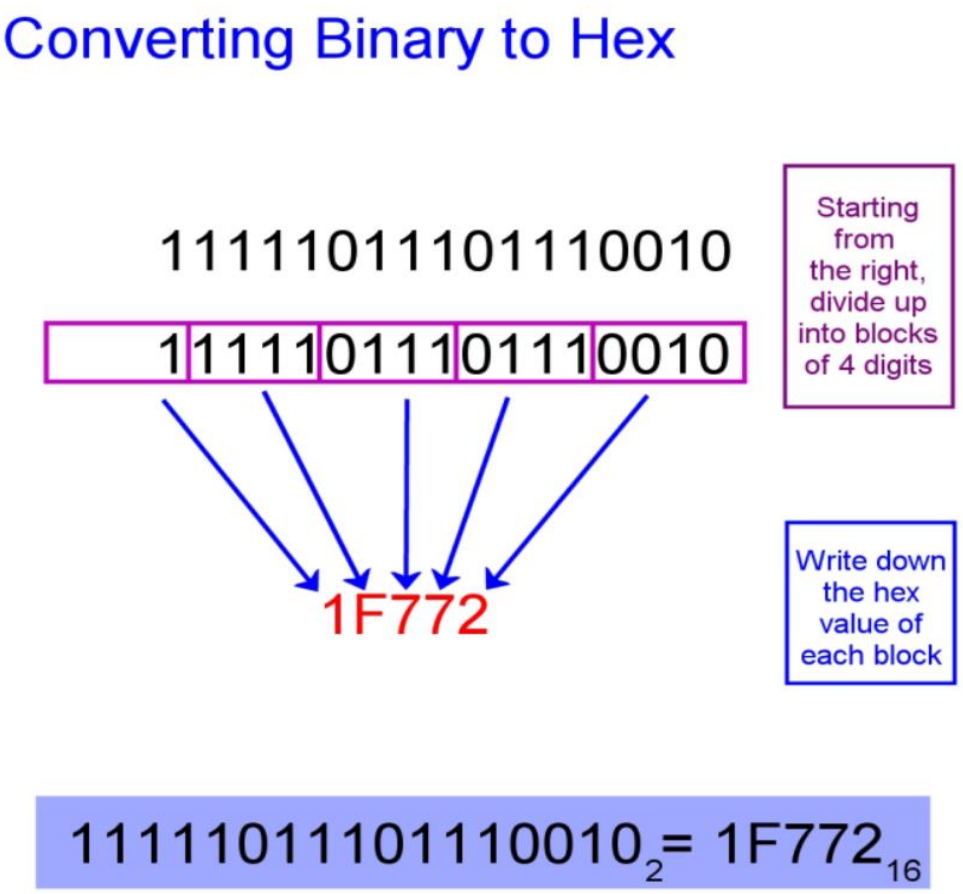
https://owlcation.com/stem/How-to-Convert-Decimal-to-Binary-and-Binary-to-Decimal

Hexadecimal to Binary



https://owlcation.com/stem/How-to-Convert-Hex-to-Binary-and-Binary-to-Hexadecimal

Binary to Hexadecimal



https://owlcation.com/stem/How-to-Convert-Hex-to-Binary-and-Binary-to-Hexadecimal

Exercise 1

Binary	Decimal	Hexadecimal
0b10010011	147	0x93
0b00010110	22	0x16
0b00111111	63	0x3F
0b100100	36	0x24
0b110000110000	3120	0xC30
0b0	0	0x0
0b101110101101	2989	0xBAD
0b110110101	437	0x1B5

C Bitwise Operators

x	y	x|y	x&y	x^y
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	0

- **AND (&)** outputs 1 only when both input bits are 1
- **OR (|)** outputs 1 when either input bit is 1
- **XOR(^)** outputs 1 when either input is exclusively 1(i.e., when two input bits are different)
- **NOT(~)** inverts 0 to 1 and 1 to 0

https://fresh2refresh.com/c-programming/c-operators-expressions/c-bit-wise-operators/

NOT (~) vs Negation(!)

- Bitwise NOT (~) takes each bit in a number and inverts it
  - ~(0b001010) = 0b110101

- Logical Negation(!) operator reverses the meaning of its operand
  - !0 = 1
  - !(non-zero)= 0

Exercise 2

x & 0	
x|0	
x^0	
x&1	
x|1	
x^1	

Exercise 2

x & 0	0
x|0	x
x^0	x
x&1	x
x|1	1
x^1	~x

x ^ 1

x = 0: 0 ^ 1 = 1 = ~x

x = 1: 1 ^ 1 = 0 = ~x

Bit Masking

- ‘Using a mask, multiple bits can be set either on, off or inverted from on to off (or vice versa) in a single bitwise operation.’

https://en.wikipedia.org/wiki/Mask_(computing)#:~:text=This%20can%20be%20achieved%20using,ing%20them%20with%20a%201%20.

Masking to extract Bits

- Given binary number **0b1000 0001 1110 1011**, extract only the last 4 LSB bits

Bit 1	Bit 2	Bitwise AND
1	Y	Y
0	Y	0

Num 1 : **0b1000 0001 1110 1011**

& **0b0000 0000 0000 1111**

Result : **0b0000 0000 0000 1011**

Masking Bits to 1

- Given binary number **0b1000 0001 1100 1011**, convert the last 8 bits to 1

Bit 1	Bit 2	Bitwise OR
1	Y	1
0	Y	Y

Num 1 : **0b1000 0001** 1100 1011

| **0b0000 0000 1111 1111**

Result : **0b1000 0001 1111 1111**

Masking to toggle Bits

- Given binary number **0b1000 0001 1110 1011**, invert/toggle the last 8 bits

Bit 1	Bit 2	Bitwise XOR
1	X	~X

Num 1 : **0b1000 0001 1100 1011**

^ **0b0000 0000 1111 1111**

Result : **0b1000 0001 0011 0100**

Shift Operations

- Left shift (`x << n`)
  - Fill with `0`s on right
- Right shift (`x >> n`)
  - Logical shift
    - Unsigned values
    - Fill with 0s on left
  - Arithmetic shift
    - Signed values
    - Replicate most significant bit on left
- Notes:
  - Shifts by `n<0` or `n≥w` (`w` is bit width of `x`) are *undefined*
  - C: behavior of `>>` is determined by compiler
    - Typically depends on data type of `x` (signed/unsigned)
  - Java: logical shift is `>>>` and arithmetic shift is `>>`

	x	0010 0010
	x<<3	0001 0000
logical:	x>>2	0000 1000
arithmetic:	x>>2	0000 1000

	x	1010 0010
	x<<3	0001 0000
logical:	x>>2	0010 1000
arithmetic:	x>>2	1110 1000

Exercise

- What is x << n equivalent to?
  - x^n
  - 2^{nx}
  - n^x
  - $x.2^n$

Exercise

- Convert from 0011 0101 1001 0001 -> 0000 0000 0000 1011
  - Assume x = 0011 0101 1001 0001
  - Step 1: x << 4
    - x = 0101 1001 0001 0000
  - Step 2: x >> 11
    - x = 0000 0000 0000 1011

Exercise

Bit Extraction: Returns the value (0 or 1) of the 19th bit (counting from LSB). Allowed operators: >>, &, |, ~.

```
int extract19(int x) {

    return ____[          ]____;
}
```

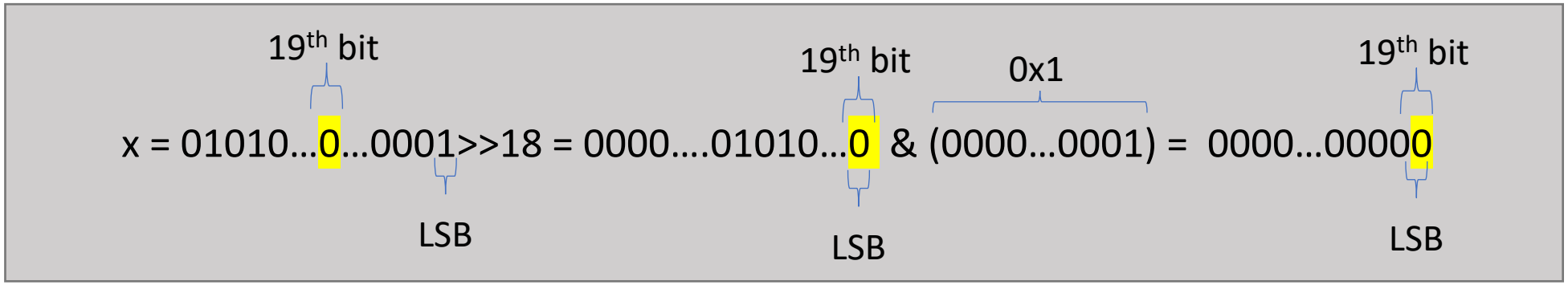
- **Extraction of bits** usually needs application of masking
- Requires extracting just the 19th bit while **masking** (blocking) the other bits

Exercise

Bit Extraction: Returns the value (0 or 1) of the 19th bit (counting from LSB). Allowed operators: >>, &, |, ~.

```
int extract19(int x) {
    return (x>>18)&0x1;
}
```

- **Extraction of bits** usually needs application of masking
- Requires extracting just the 19th bit while **masking** (blocking) the other bits



Exercise

Subtraction: Returns the value of `x-y`. Allowed operators: `>>`, `&`, `|`, `~`, `+`.

```
int subtract(int x, int y) {

    return x+((~y)+ 1)                                                    ;
}
```

- $x - y = x + (2's\ complement\ of\ y)$
  - $2's\ complement\ of\ y = \sim y + 1$

Exercise

Equality: Returns the value of `x==y`. Allowed operators: `>>`, `&`, `|`, `~`, `+`, `^`, `!`.

```
int equals(int x, int y) {

    return _____________________________________________;
}
```

- Use the property `x ^ x = 0`
- Check for equality usually means you need to use bitwise XOR operator

Exercise

Equality: Returns the value of `x==y`. Allowed operators: `>>`, `&`, `|`, `~`, `+`, `^`, `!`.

```
int equals(int x, int y) {
    return !(x ^ y);
}
```

Exercise

Divisible by Eight? Returns the value of (x%8)==0. Allowed operators: >>, <<, &, |, ~, +, ^, !.

```
int divisible_by_8(int x) {

    return _____;

}
```

- Check if the last 3 digits of the binary equivalent of x are 0
  - 8 = 0b1000, 16 = 0b10000, 24 = 0b11000, …

Exercise

Divisible by Eight? Returns the value of (x%8)==0. Allowed operators: >>, <<, &, |, ~, +, ^, !.

```
int divisible_by_8(int x) {
    return !(x<<29)                                                    ;
}
```

0011…001000 <<29 = 0000000…000

LSB MSB

Exercise

Greater than Zero? Returns the value of `x>0`. Allowed operators: `>>, &, |, ~, +, ^, !.`

```
int greater_than_0(int x) {

    return _____;

}
```

- Requirements
  - Required to check the value of **sign bit** of the integer x (Most Significant bit (leftmost bit))
    - If sign bit is 1, the number is negative and we return false
  - Also, required to check if the number is zero. If the number is zero, then we return false

Exercise

Greater than Zero? Returns the value of `x>0`. Allowed operators: `>>`, `&`, `|`, `~`, `+`, `^`, `!`.

```
int greater_than_0(int x) {
    return !((x&(1<<31)) | !x);
}
```

- `x &(1 << 31)` checks if the number is negative
  - `1<<31` is a number with all 0s except the MSB being 1
  - So, `x &(1 << 31)` masks `x` with `1<<31` to extract the value of x's MSB (sign bit)
- `!x` is to check if `x` is 0
- `x>0` if it is not negative and not zero

Exercise

Greater than Zero? Returns the value of `x>0`. Allowed operators: `>>, &, |, ~, +, ^, !.`

```
int greater_than_0(int x) {
    /* invert and check sign; we need the third operand for the T_min case */
    return ((~x + 1) >> 31) & 0x1 & ~(x >> 31) _OR_ !!x & ~(x >> 31);
}
```

Take care of operator precedences

- Take care of operator precedence
  - In *c&a>>b*, *a>>b* operation takes place first, followed by the operation using *&*
  - *c&a>>b* is equivalent to *c&(a>>b)*
  - **Using parenthesis to separate individual operations is a safe practice.**

- Find the full operator precedence list of C here.

Data Lab Puzzles

Rating	Name	Description
1	bitXor	Compute `x ^ y` using only `~` and `&`
2	allEvenBits(x)	Return 1 if all even-numbered bits in word set to 1
2	logicalShift(x)	Shift x to the right by n bits, performing a logical shift, but using arithmetic shifts
3	logicalNeg(x)	Implement the `!` operator without using `!`
1	tmax()	Return maximum two's complement integer
3	twosBits(x)	Return 1 if x can be represented as an n-bit, two's complement integer
3	floatFloat2Int(f)	Return bit-level equivalent of expression `(int) f` for floating point argument `f`

Puzzle *bitXOR*

DeMorgan's Law:

```
~(A|B) = ~A & ~B
~(A&B) = ~A | ~B
```

- Example: **Puzzle *bitAND***
  - Compute x & y using only **~** and **|**
    - **x & y**

      = ~(~(x & y))

      =~(~x | ~y)

      = ~(~x) & ~(~y)

      = x & y

Puzzles

- logicalNeg(x)
  - !x will return 1 only if x == 0
    - How to determine if x == 0?

- logicalShift(x)
  - Recall what arithmetic right shift does for signed integers
    - How to take care of the extended sign bits?
    - If x = 1101
      - For arithmetic right shift, x>>2 = 11 1101
      - For logical right shift, x>>2 = 00 1101
      - How to ensure this?

allEvenBits(x)

- Return 1 if all even-numbered bits in word set to 1

- allEvenBits(0x55555555) = 1
  - 0x5555 5555 = 0101 0101 … 0101
  - What is allEvenBits(0xFFFFFFFF)?

  - Rough idea: x & (0101 0101 … 0101) = 0101 0101 … 0101
    - But this solution will NOT be accepted!
    - Try to find a way to check if the even bits are all 1 without making use of such a big constant
      - Keep in mind the limit of the number of operators that can be used.

Puzzle *tmax()*

- What is the Maximum value of unsigned int?
  - 0xFFFFFFFF
  - How can you get this number?
    - Not allowed to use big constants

- What is the Maximum value of 2’s complement integer?
  - Maximum value of signed int?
    - How to get this with the help of bit manipulation discussed previously?

Puzzle *twosBits(x, n)*

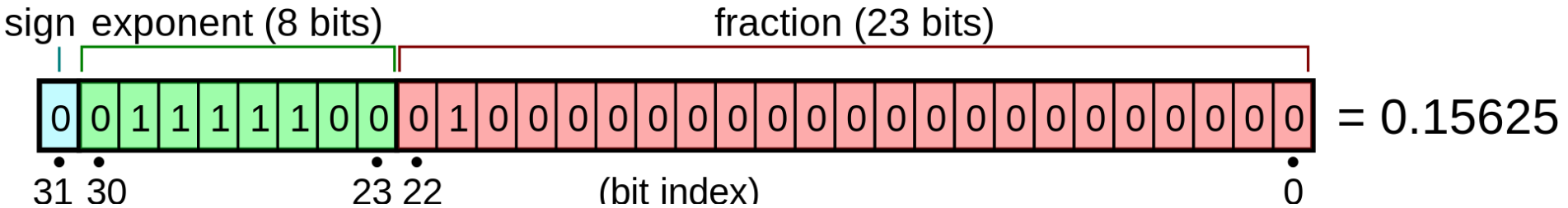
- Return 1 (i.e. True) if x can be represented as a *n-bit 2's complement integer*

- If n = 3, what is the range of 2's complement integers
  - $-2^{3-1}\ to + 2^{3-1} - 1 = -4\ to + 3$

- What is the 32-bit representation of -4 in 2's complement ?

Single Precision Floating Point Numbers

- Representation of decimal numbers as a IEEE 754 single-precision floating point value



- Resources:
  - Decimal to IEEE 754 Floating point representation
    - https://www.youtube.com/watch?v=8afbTaA-gOQ
  - IEEE 754 Floating Point Representation to its Decimal Equivalent
    - https://www.youtube.com/watch?v=LXF-wcoeT0o

https://en.wikipedia.org/wiki/Single-precision_floating-point_format

Puzzle *floatFloat2Int(f)*

- Argument is passed as unsigned int, but it is to be interpreted as the bit-level representation of a single-precision floating point value.
- Hints
  - Extract the components of a float (sign, exponent, etc)
  - Check if the input is valid (i.e. within range)
  - Check if the exponent is valid
    - For instance, it should not make the corresponding integer overflow.
  - Follow the procedure for converting from IEEE 754 Floating Point Representation to its Decimal Equivalent

The function `floatFloat2Int` implements a single-precision floating-point operation. Here, you are allowed to use standard control structures (conditionals, loops), and you may use both `int` and `unsigned` data types, including arbitrary unsigned and integer constants. You may not use any unions, structs, or arrays. Most significantly, you may not use any floating point data types, operations, or constants. Instead, any floating-point operand will be passed to the function as having type `unsigned`, and any returned floating-point value will be of type `unsigned`. Your code should perform the bit manipulations that implement the specified floating point operations.

We have included the `ishow` and `fshow` programs to help you decipher integer and floating point representations respectively. Each takes a single decimal or hex number as an argument. To build them type, switch to the handout directory and type:

```
$ make
```

Example usages:

```
$ ./ishow 0x27
Hex = 0x00000027,        Signed = 39,        Unsigned = 39

$ ./ishow 27
Hex = 0x0000001b,        Signed = 27,        Unsigned = 27

$ ./fshow 0x15213243
Floating point value 3.255334057e-26
Bit Representation 0x15213243, sign = 0, exponent = 0x2a, fraction = 0x213243
Normalized.  +1.2593463659 X 2^(-85)

$ ./fshow 15213243
Floating point value 2.131829405e-38
Bit Representation 0x00e822bb, sign = 0, exponent = 0x01, fraction = 0x6822bb
Normalized.  +1.8135598898 X 2^(-126)
```

Lab 1 - Logistics

2 Logistics

Each person in the class *works alone* and submits the work for the lab electronically via Gradescope. Clarifications and corrections will be given during recitation sections. We strongly advise you to test your code on the Thoth Linux machine before submitting it. Also, please test your submission on Gradescope. We will not return any points lost because you did the lab in a different environment and have not tested it on Gradescope before the deadline.

Before you begin, please take the time to review the course policy on academic integrity at the course web site. Remember that (for this and all labs in this course):

- You are not allowed to search for help online!
- You are not allowed to ask other students for help, show them your code, or discuss the specifics of the solution.

Be aware that you may be asked to explain your code to a member of our course staff using only what you have submitted: your comments in the code should be such that you can determine what your code does and why.

DON'Ts for Integer based puzzles (in *bits.c*)

- You are expressly forbidden to:

  You are expressly forbidden to:

  - Use any control constructs such as if, do, while, for, switch, etc.
  - Define or use any macros.
  - Define any additional functions in this file.
  - Call any functions.
  - Use any other operations, such as &&, ||, -, or ?:
  - Use any form of casting.
  - Use any data type other than int. This implies that you cannot use arrays, structs, or unions.

Evaluation and Submission

Thank you!