CS356: Discussion #2

Integer Operations & Floating-Point Operations

USC University of Southern California

Integers in C (64-bit architecture)

Type	Size (bytes)	Unsigned Range	Signed Range
`char`	1	0 to 255	-128 to 127
`short`	2	0 to 65535	-32,768 to 32,767
`int`	4	0 to 4G	-2G to 2G
`long`	8	0 to 18×10^{18}	-9×10^{18} to 9×10^{18}

- Rule: **0 to 2^n-1** (unsigned) and **-2^{n-1} to 2^{n-1}-1** (signed) using *n* bits
- Signed integers are represented using **2's complement**:
  `0x80 == -128, 0xFF == -1, 0x00 == 0, 0x01 == 1, 0x7F == 127`

1	0	0	0	0	0	0	0
-128	64	32	16	8	4	2	1

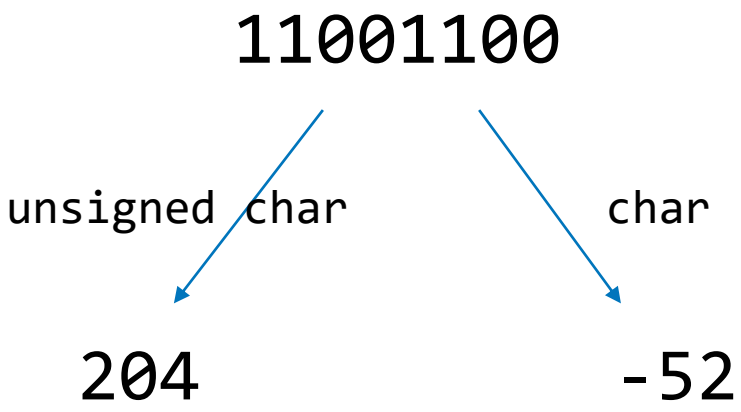
Integers in C (64-bit architecture)

Type	Size (bytes)	Unsigned Range	Signed Range
`char`	1	0 to 255	-128 to 127
`short`	2	0 to 65535	-32,768 to 32,767
`int`	4	0 to 4G	-2G to 2G
`long`	8	0 to 18×10^{18}	-9×10^{18} to 9×10^{18}

- Rule: **0 to 2^n-1** (unsigned) and **-2^{n-1} to 2^{n-1}-1** (signed) using *n* bits
- Signed integers are represented using **2's complement**:
  `0x80 == -128, 0xFF == -1, 0x00 == 0, 0x01 == 1, 0x7F == 127`

```
C Tips:
Hex value    Octal value
0x12 == 18    012 == 10
```

Signed and Unsigned



Integer Operations

- **Addition** / **Subtraction** (reduces to addition using 2's complement): `+` `-`
  - Unsigned addition overflow: result smaller than inputs
  - Unsigned subtraction overflow: result larger than minuend
  - Signed addition overflow: pos + pos = neg or neg + neg = pos
- **Multiplication** / **Division**: `*` `/`
- **Bitwise operations**
  - Bitwise AND (`x & mask`): clear bits that are 0 in the mask
  - Bitwise OR (`x | mask`): set bits that are 1 in the mask
  - Bitwise XOR (`x ^ mask`): flip bits that are 1 in the mask
  - Bitwise NOT (~x): flip all bits

  Note the difference between ~x (bitwise NOT) and !x (logical NOT)
- **Shift operations**
  - Left shift (`x << n`): fill in zeros
  - Right shift (`x >> n`): fill in zeros (unsigned) or repeat MSB (signed)

Exercises

- Are the statements always true?
  - `x + ~x == -1`
  - `x + ~x + 1 == 0`
  - `-x == ~x + 1`

Exercises

- Are the statements always true?
  - `x + ~x == -1` Yes
  - `x + ~x + 1 == 0` Yes
  - `-x == ~x + 1` Yes

`a - b` ⇔ `a + ~b + 1`

Exercises

- Are the functions correct?

```
int odd(int x) {
    return x & 1 == 1;
}

int even(int x) {
    return x & 1 == 0;
}
```

Exercises

- Are the functions correct?

```
int odd(int x) {
    return (x & 1) == 1;
}

int even(int x) {
    return (x & 1) == 0;
}
```

C Tips:
Operator precedence!
‘==’ has higher precedence than ‘&’

Exercises

- Is the function correct?

```
int mul9(int x) {
    return x << 3 + x;
}
```

Exercises

- Is the function correct?

```
int mul9(int x) {
    return (x << 3) + x;
}
```

```
C Tips:
Operator precedence!
‘+’ has higher precedence than ‘<<’
```

C Operator Precedence:
https://en.cppreference.com/w/c/language/operator_precedence

Exercises

- Is the function correct?

```
int getSum(int n, int a[]) {
    int sum = 0;
    unsigned i;
    for (i = n - 1; i >= 0; i--)
        sum += a[i];
    return sum;
}
```

Exercises

- Is the function correct?

```
int getSum(int n, int a[]) {
    int sum = 0;
    unsigned i;
    for (i = n - 1; i >= 0; i--)
        sum += a[i];
    return sum;
}
```

- Always `i >= 0` !

Exercises

```
int x = foo();   /* x is arbitrary int */
int y = bar();   /* y is arbitrary int */
unsigned ux = x;
unsigned uy = y;
```

Do the following statements always hold?

- `ux >= 0`
- `ux > -1`
- `x * x >= 0`
- `ux >> 3 == ux / 8`
- `x >> 3 == x / 8`

Exercises

```
int x = foo();   /* x is arbitrary int */
int y = bar();   /* y is arbitrary int */
unsigned ux = x;
unsigned uy = y;
```

Do the following statements always hold?

- `ux >= 0` YES
- `ux > -1` NO, -1 => UMAX
- `x * x >= 0` NO, when overflow
- `ux >> 3 == ux / 8` YES
- `x >> 3 == x / 8` NO, when x < 0

Exercises

```
int x = foo();   /* x is arbitrary int */
int y = bar();   /* y is arbitrary int */
unsigned ux = x;
unsigned uy = y;
```

Do the following statements always hold?

- `if x < 0,` `then` `x * 2 < 0`
- `if x > y,` `then` `-x < -y`
- `if x > 0 && y > 0,` `then` `x + y > 0`
- `if x >= 0,` `then` `-x <= 0`
- `if x <= 0,` `then` `-x >= 0`

Exercises

```
int x = foo();   /* x is arbitrary int */
int y = bar();   /* y is arbitrary int */
unsigned ux = x;
unsigned uy = y;
```

Do the following statements always hold?

- `if x < 0,` `then` `x * 2 < 0` NO, overflow
- `if x > y,` `then` `-x < -y` NO, TMIN
- `if x > 0 && y > 0,` `then` `x + y > 0` NO, overflow
- `if x >= 0,` `then` `-x <= 0` YES
- `if x <= 0,` `then` `-x >= 0` NO, TMIN

DataLab: What to implement (1)

- **Integer Problems:** Only 1-byte constants (`0xFA`), no loops (`for`, `while`), no conditionals (`if`), no macros (`INT_MAX`), no comparisons (`x==y`, `x>y`), no `unsigned int`, no operators `- && ||`, only the operators `! ~ & | ^ + << >>`
- `int tmin(void)`: return minimum two's complement integer
- `int bitOr(int x, int y)`: return x | y using only ~ and &
- `int negate(int x)`: return −x
- `int isNotEqual(int x, int y)`: return 0 if x == y, otherwise 1
- `int isGreater(int x, int y)`: return 1 if x > y, otherwise 0
- `int subtractionOK(int x, int y)`: determine if can compute x - y w/o overflow
- `int conditional(int x, int y, int z)`: same as x ? y : z
- `int satMul2(int x)`:multiplies by 2, saturating to Tmin or Tmax if overflow
- `int byteSwap(int x, int n, int m)`: swaps the nth byte and the mth byte

Exercise: Build large constants

Write a function `int abcd()` that returns the constant **`0xABCD0000`**.
Use: bitwise/shift ops (`&|^~ << >>`), negation (`!`), **1-byte** const (`0x00` to `0xFF`).

Exercise: Build large constants

Write a function `int abcd()` that returns the constant **`0xABCD0000`**.
Use: bitwise/shift ops (`&|^~ << >>`), negation (`!`), **1-byte** const (`0x00` to `0xFF`).

```
#include <stdio.h>
static int abcd() {
    return ((0xAB << 8) | 0xCD) << 16;
}

/* 0x000000AB    0xAB
   0x0000AB00    0xAB << 8
   0x0000ABCD   (0xAB << 8) | 0xCD
   0xABCD0000 ((0xAB << 8) | 0xCD) << 16 */
```

Exercise: Check if variable is zero

Write a function `int isZero(int x)` that returns 1 if `x==0` and 0 otherwise. Use only `!`

Exercise: Check if variable is zero

Write a function `int isZero(int x)` that returns 1 if `x==0` and 0 otherwise. Use only `!`

```
#include <stdio.h>
static int isZero(int x) {
    return !x;
}
```

`!x is 1, if x is 0`

`!x is 0, if x is non-zero (e.g. 1, 152, 0xFF),`

Exercise: Check if variable is non-zero

Write a function `int isNonZero(int x)` that returns 1 if `x!=0`, 0 otherwise. Use only `!`

Exercise: Check if variable is non-zero

Write a function `int isNonZero(int x)` that returns 1 if `x!=0`, 0 otherwise. Use only `!`

```
#include <stdio.h>
static int isNonZero(int x) {
    return !!x;
}
```

Exercise: Extract the last byte

Write a function `int leastSignificantByte(int x)` that returns the least significant byte of the input `x`.

Use: bitwise/shift ops (`&|^~ << >>`), negation (`!`), 1-byte const (`0x00` to `0xFF`).

```
x: 01010101 10101010 01010101 10101010
```

Exercise: Extract the last byte

Write a function `int leastSignificantByte(int x)` that returns the least significant byte of the input `x`.

Use: bitwise/shift ops (`&|^~ << >>`), negation (`!`), 1-byte const (`0x00` to `0xFF`).

```
#include <stdio.h>
static int leastSignificantByte(int x) {
    return x & 0xFF;
}
```

```
      x: 01010101 10101010 01010101 10101010
   0xFF: 00000000 00000000 00000000 11111111
x & 0xFF: 00000000 00000000 00000000 10101010
```

Exercise: Extract the last three bits

Write a function `int lastThreeBits(int x)` that returns the last three bits of the input `x`.

Use: bitwise/shift ops (`&|^~ << >>`), negation (`!`), 1-byte const (`0x00` to `0xFF`).

```
x: 10101010 01010101 10101010 01010101
```

Exercise: Extract the last three bits

Write a function `int lastThreeBits(int x)` that returns the last three bits of the input `x`.

Use: bitwise/shift ops (`&|^~ << >>`), negation (`!`), 1-byte const (`0x00` to `0xFF`).

```c
#include <stdio.h>
static int lastThreeBits(int x) {
    return x & 7;
}
```

```
    x: 10101010 01010101 10101010 01010101
    7: 00000000 00000000 00000000 00000111
x & 7: 00000000 00000000 00000000 00000101
```

Exercise: Extract the first bit (sign bit)

Write a function `int getFirstBit(int x)` that returns the MSB of `x`.
Use: bitwise/shift ops (`&|^~ << >>`), negation (`!`), 1-byte const (`0x00` to `0xFF`).

```
x: 10101001 00100111 11101001 11010101
```

Exercise: Extract the first bit (sign bit)

Write a function `int getFirstBit(int x)` that returns the MSB of `x`.
Use: bitwise/shift ops (`&|^~ << >>`), negation (`!`), 1-byte const (`0x00` to `0xFF`).

```c
#include <stdio.h>
static int getFirstBit(int x) {
    return (x >> 31) & 1;
}
```

```
           x: 10101001 00100111 11101001 11010101
      x >> 31: 11111111 11111111 11111111 11111111
(x >> 31) & 1: 00000000 00000000 00000000 00000001
```

Exercise: Check if numbers have same sign

Write a function `int sameSign(int x, int y)` that returns 1 if `x` and `y` have the same sign.
Use: bitwise/shift ops (`&|^~ << >>`), negation (`!`), 1-byte const (`0x00` to `0xFF`).

Exercise: Check if numbers have same sign

Write a function `int sameSign(int x, int y)` that returns 1 if x and y have the same sign.
Use: bitwise/shift ops (`&|^~ << >>`), negation (`!`), 1-byte const (`0x00` to `0xFF`).

```c
#include <stdio.h>
static int sameSign(int x, int y) {
    return !(  ((x >> 31) & 1) ^ ((y >> 31) & 1)  );
}
```

```
0 xor 0 == 0
1 xor 1 == 0
0 xor 1 == 1
1 xor 0 == 1
```

Variation

- Can we reduce the number of operations?
- The solution

```
!(  ((x >> 31) & 1) ^ ((y >> 31) & 1)  )
```

is equivalent to

```
!(  ((x ^ y) >> 31) & 1  )
```

Swap without extra memory

```
int x, y;
... ...

// swap x and y
x = x ^ y;
y = x ^ y;
x = x ^ y;
```

```
int *x, *y;
... ...

// swap *x and *y
if (x != NULL && y != NULL) {
    if (x != y) {
        *x = *x ^ *y;
        *y = *x ^ *y;
        *x = *x ^ *y;
    }
}
```

Exercise: Extract the byte after the bit sign

Write a function `int getBits23to30(int x)` that returns the byte starting after the first bit of `x`.

Use: bitwise/shift ops (`&|^~ << >>`), negation (`!`), 1-byte const (`0x00` to `0xFF`).

```
x: 10101110 10101010 10101010 10101010
```

Exercise: Extract the byte after the bit sign

Write a function `int getBits23to30(int x)` that returns the byte starting after the first bit of `x`.

Use: bitwise/shift ops (`&|^~ << >>`), negation (`!`), 1-byte const (`0x00` to `0xFF`).

```c
#include <stdio.h>
static int getBits23to30(int x) {
    return (x >> 23) & 0xFF;
}
```

```
              x: 10101110 10101010 10101010 10101010
      x >> 23: 11111111 11111111 11111111 01011101
         0xFF: 00000000 00000000 00000000 11111111
(x >> 23) & 0xFF: 00000000 00000000 00000000 01011101
```

Exercise: Conditionals without `if`

Write a function `int negOrElse(int x, int y)` that returns

- `x` if `(x < 0)`
- `y` if `(x >= 0)`

Use only `>>` `~` `&` `|`

Exercise: Conditionals without `if`

Write a function `int negOrElse(int x, int y)` that returns

- `x` if `(x < 0)`
- `y` if `(x >= 0)`

Use only `>>` `~` `&` `|`

```
#include <stdio.h>
static int negOrElse(int x, int y) {
    int isNeg = x >> 31;  /* 0xFFFFFFFF or 0x00000000 */
    return  (isNeg & x) | (~isNeg & y);
}
```

```
if  x < 0,   isNeg == 11111111 11111111 11111111 11111111
        (isNeg & x) == x,   (~isNeg & y) == 0

if x >= 0,   isNeg == 00000000 00000000 00000000 00000000
        (isNeg & x) == 0,   (~isNeg & y) == y
```

Exercise: Multiply using shifts

Write a function `void mult(int x)` that multiplies `x`

- by 6, using 2 shifts and 1 add/sub;
- by 31, using 1 shifts and 1 add/sub;
- by -6, using 2 shifts and 1 add/sub;
- by 55, using 2 shifts and 2 add/sub.

Exercise: Multiply using shifts

Write a function `void mult(int x)` that multiplies x

- by 6, using 2 shifts and 1 add/sub;
- by 31, using 1 shifts and 1 add/sub;
- by -6, using 2 shifts and 1 add/sub;
- by 55, using 2 shifts and 2 add/sub.

```
#include <stdio.h>
static void mult(int x) {  printf("\nx = %d\n", x);
    printf(" 6 * x =    (8-2) * x = %d\n", (x << 3) - (x << 1));
    printf("31 * x =   (32-1) * x = %d\n", (x << 5) - x);
    printf("-6 * x =    (2-8) * x = %d\n", (x << 1) - (x << 3));
    printf("55 * x = (64-8-1) * x = %d\n", (x << 6)-(x << 3)-x);
}
int main() {
    mult(0); mult(1); mult(-1); mult(10); mult(-100); mult(7);
}
```

Dividing Two's-Complement by Powers of 2

- **x / 2 k** when x >= 0: `x >> k`
- **x / 2 k** when x < 0: `(x + (1 << k) - 1) >> k`
  - Consider `(-3)/2` with `signed char` (1 byte)
  - `0xFD >> 1` gives `0xFE` which is -2 (instead, **-3/2 gives -1 in C**)
  - `x >> k` rounds toward -∞ for negative x, not toward 0 (unlike x/y in C)
  - In other words, it computes ⌊x / 2 k⌋ instead of ⌈x / 2 k⌉ for x < 0
  - But, it is always true that `⌊(x + (y-1)) / y⌋ = ⌈x / y⌉`
  - **Biasing**: add **2 k - 1** before the shift when x < 0

k	Bias	−12,340 + Bias (Binary)	>> k (Binary)	Decimal	−12340/2^k
0	0	1100111111001100	1100111111001100	−12340	−12340.0
1	1	1100111111001101	1110011111100110	−6170	−6170.0
4	15	1100111111011011	1111110011111101	−771	−771.25
8	255	1101000011001011	1111111111010000	−48	−48.203125

Figure 2.29 **Dividing two's-complement numbers by powers of 2.** By adding a bias before the right shift, the result is rounded toward zero.

Fixed Point vs Floating Point

Fixed-point format: a fixed number of bits is reserved for the fractional part.

- Example: use unsigned chars (1 byte) and reserve 2 bits for fractional part.

8				7			
1	0	0	0	0	**1**	**1**	**1**
32	16	8	4	2	1	0.5	0.25

0x87 represents **33.75**

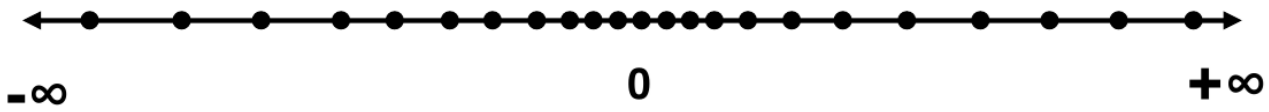
The range for unsigned chars was 0 to 255.

By reserving 2 bits for the fractions part:

- The range is now [0, 63.75] (0x00 to 0xFF)
- We can represent fractional values with increments of 0.25

Floating-point format: the position of the binary point can change.

- Flexible trade-off between **range** and **precision**



IEEE 754 Standard: 32-bit

Binary32 Format (`float`)

sign	exponent	fraction
1 bit	8 bits	23 bits

- **Exponent** encodes values [-126, 127] as unsigned integers with bias
- Exponent of all 0's reserved for:
  - Zeros: `0x00000000` (0.0), `0x80000000` (-0.0)
  - Denormalized values: $(-1)^{\text{sign}} \times 0.(\text{fraction}) \times 2^{1-127}$ (nonzero fraction)
- Exponent of all 1's reserved for:
  - Infinity: `0x7F800000` (∞), `0xFF800000` (-∞)
  - NaN: with any nonzero fraction
- **Decimal value** (Normalized): $(-1)^{\text{sign}} \times 1.(\text{fraction}) \times 2^{\text{exponent - 127}}$
- **Decimal range**: (7 significant decimal digits) $\times 10^{\pm 38}$

Special Numbers (32-bit)

Description	exp (8 bits)	frac (23 bits)	Lower 31 bits (hex)	Decimal value
Zero	00…00	00…00	`0x00000000`	0.0
Smallest Pos Denormalized	00…00	00…01	`0x00000001`	$2^{-23} \times 2^{-126}$
Largest Denormalized	00…00	11…11	`0x007FFFFF`	$(1.0-\varepsilon) \times 2^{-126}$
Smallest Pos Normalized	00…01	00…00	`0x00800000`	1.0×2^{-126}
One	01…11	00…00	`0x3F800000`	1.0
Largest Normalized	11…10	11…11	`0x7F7FFFFF`	$(2.0-\varepsilon) \times 2^{127}$
Infinity	11…11	00…00	**`0x7F800000`**	Infinity
NaN	11…11	Nonzero	**`> 0x7F800000`**	NaN

IEEE 754 Standard: 64-bit

Binary64 Format (`double`)

sign	exponent	fraction
1 bit	11 bits	52 bits

- **Exponent** encodes values [-1022, 1023] as unsigned integers with bias
- Exponent of all 0's reserved for:
  - Zeros: `0x0000000000000000` (0.0), `0x8000000000000000` (-0.0)
  - Denormalized values: $(-1)^{\text{sign}} \times 0.(\text{fraction}) \times 2^{1-1023}$ (nonzero fraction)
- Exponent of all 1's reserved for:
  - Infinity: `0x7FF0000000000000` (∞), `0xFFF0000000000000` (-∞)
  - NaN: any nonzero fraction
- **Decimal value** (Normalized): $(-1)^{\text{sign}} \times 1.(\text{fraction}) \times 2^{\text{exponent - 1023}}$
- **Decimal range**: (≃ 16 significant decimal digits) $\times 10^{\pm 308}$

Other formats, same patterns

1 sign bit, **k** bits for exponent, **m** bits for fraction

Bias = $2^{k-1}-1$

Normalized: $(-1)^{\text{sign}} \times 1.(\text{fraction}) \times 2^{\text{exponent - Bias}}$

Denormalized: $(-1)^{\text{sign}} \times 0.(\text{fraction}) \times 2^{1-\text{Bias}}$

To **negate**, just flip the sign bit (except NaN)

Rounding and Casting in C

The IEEE 754 standard defines four **rounding modes**:

- **Round to nearest, ties to even**: default rounding in C for `float/double` ops
- **Round towards zero** (truncation): used to cast `float/double` to `int`
- **Round up** (ceiling): go towards +∞ (gives an upper bound)
- **Round down** (floor): go towards -∞ (gives a lower bound)

Floating point operations

- Addition and subtraction are **not associative**
  - Add small-magnitude numbers before large-magnitude ones
- Multiplication and division are **not associative** (**nor distributive**)
  - Control magnitude with divisions (if possible)
    `(big1 * big2) / (big3 * big4)` overflows on first multiplication
    `1/big3 * 1/big4 * big1 * big2` underflows on first multiplication
    `(big1 / big3) * (big2 / big4)` is likely better
- Comparison should use `fabs(x-y) < epsilon` instead of `x==y`
- **Instead:** 2's complement is associative (even after overflow), can use `x==y`

DataLab: What to implement (2)

Floating-point Problems: 4-byte constants (`0x12345678`), loops (`for`, `while`), conditionals (`if`), comparisons (`x==y`, `x>y`), operators `- && ||`,
but no macros (`INT_MAX`), no `float` types or operations.

The `unsigned` input and `int` output are the **bit-level equivalent** of 32-bit `floats`

- `int floatNegate(unsigned uf)`
- `int floatIsEqual(unsigned uf, unsigned ug)`
- `int floatFloat2Int(unsigned uf)`

Exercise: Floating-point Sign

Write a function `int sign(unsigned int x)` that returns the sign of `x` as 1/-1

Exercise: Floating-point Sign

Write a function `int sign(unsigned int x)` that returns the sign of `x` as 1/-1

```
int sign(unsigned int x) {
    return (x & 0x80000000) ? -1 : 1;
}
```

```
         x: 10101010 01010101 10101010 01010101
0x80000000: 10000000 00000000 00000000 00000000

        -1: 10000000 00000000 00000000 00000000
         1: 00000000 00000000 00000000 00000000
```

Exercise: Extract Exponent

Write a function `int exponent(unsigned int x)` that returns the exponent of `x` (as is, including the bias).

```
x:    00111111 10000000 00000000 00000000
       exponent
```

Exercise: Extract Exponent

Write a function `int exponent(unsigned int x)` that returns the exponent of `x` (as is, including the bias).

```
int exponent(unsigned int x) {
    return (x >> 23) & 0xFF;
}
```

```
x:   00111111 10000000 00000000 00000000
       exponent
```

Exercise: Extract Fraction

Write a function `int fraction(unsigned int x)` returning the fraction of `x`, including the implicit leading bit equal to 1 (ignore denormalized numbers).

```
x:    00111111 01101001 00000000 00000000
                fraction (without leading bit)

                11101001 00000000 00000000
                fraction (with leading bit 1)
```

Exercise: Extract Fraction

Write a function `int fraction(unsigned int x)` returning the fraction of x, including the implicit leading bit equal to 1 (ignore denormalized numbers).

```
int fraction(unsigned int x) {
    return (x & 0x007FFFFF) | 0x00800000;
}
```

```
 x:   00111111 01101001 00000000 00000000
                fraction (without leading bit)

                 11101001 00000000 00000000
                fraction (with leading bit 1)
```

Exercise: Detect Floating-point Zero

Write a function `int is_zero(unsigned int x)` returning 1 if x is 0.0 or -0.0, and 0 otherwise. (Trivial solution under relaxed assignment rules!)

Exercise: Detect Floating-point Zero

Write a function `int is_zero(unsigned int x)` returning 1 if x is 0.0 or -0.0, and 0 otherwise. (Trivial solution under relaxed assignment rules!)

```
int is_zero(unsigned int x) {
    return (x == 0x00000000 || x == 0x80000000) ? 1 : 0;
}
```

```
+0:  00000000 00000000 00000000 00000000
-0:  10000000 00000000 00000000 00000000
```

Exercise: Detect Denormalized Numbers

Write a function `int denorm(unsigned int x)` that returns 1 if `x` is denormalized, and 0 otherwise.

Exercise: Detect Denormalized Numbers

Write a function `int denorm(unsigned int x)` that returns 1 if `x` is denormalized, and 0 otherwise.

Solution 1 (5 Operators)

```
int denorm(unsigned int x) {
    return !((x >> 23) & 0xFF) && (x & 0x007FFFFF);
}
```

Exercise: Detect Denormalized Numbers

Write a function `int denorm(unsigned int x)` that returns 1 if x is denormalized, and 0 otherwise.

Solution 1 (5 Operators)

```
int denorm(unsigned int x) {
    return !((x >> 23) & 0xFF) && (x & 0x007FFFFF);
}
```

Solution 2 (3 Operators)

```
int denorm(unsigned int x) {
    if (x < 0x800000 && x > 0)
        return 1;
    else
        return 0;
}
```

Special Numbers (32-bit)

Ascending order	Ascending	Ascending	Ascending	Ascending
Description	**exp (8 bits)**	**frac (23 bits)**	**Lower 31 bits (hex)**	**Decimal value**
Zero	00…00	00…00	0x00000000	0.0
Smallest Pos Denormalized	00…00	00…01	0x00000001	$2^{-23} \times 2^{-126}$
Largest Denormalized	00…00	11…11	0x007FFFFF	$(1.0-\varepsilon) \times 2^{-126}$
Smallest Pos Normalized	00…01	00…00	0x00800000	1.0×2^{-126}
One	01…11	00…00	0x3F800000	1.0
Largest Normalized	11…10	11…11	0x7F7FFFFF	$(2.0-\varepsilon) \times 2^{127}$
Infinity	11…11	00…00	**0x7F800000**	Infinity
NaN	11…11	Nonzero	**> 0x7F800000**	NaN