Rounding, Logical Ops
Introduction to Computer Systems, Fall 2022

Instructor: Travis McGaha

TAs:

Ali Krema	Andrew Rigas	Anisha Bhatia
Audrey Yang	Craig Lee	Daniel Duan
David LuoZhang	Eddy Yang	Ernest Ng
Heyi Liu	Janavi Chadha	Jason Hom
Katherine Wang	Kyrie Dowling	Mohamed Abaker
Noam Elul	Patricia Agnes	Patrick Kehinde Jr.
Ria Sharma	Sarah Luthra	Sofia Mouchtaris

How are you feeling about binary representation?

Powered by Poll Everywhere

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

Logistics Part 1

- HW00 Binary Quiz: **This Friday** 9/16 @ 11:59 pm
  - Quiz On Canvas
  - Should have everything you need

- Recitations Starting this week!
  - Optional, but can be very useful
  - Increasingly useful as the semester goes on
  - More info on Ed

Logistics Part 2

- HW01 bits.c: to be released sometime this week
  - Will require VM setup (also to be released soon)
  - Has you “program” in C
  - Today’s lecture is very relevant for it
- Starting to count PollEverywhere
- More OH posted on the course website
  - (including mine)

Any questions on anything before I begin?

Lecture Outline

- **Floats Continued**
- Logical Operators
  - Shifting
- Boolean Algebra

Lecture 2 Take-aways

- We can represent Negative integers with 2C

- We can represent fractional numbers with Floats

- C/Java data types like int and float are limited by their number of bits
  - A data type of N bits has 2^N unique bit patterns
  - More on this later in lecture

Binary Scientific Notation

- Scientific notation in Binary: `-1.1001 * 2`^{4}
  - Sign: whether we are negative or positive
  - Ones place: Always starts with a non-zero 'bit'
    - (unless overall expression is 0)

A non-zero bit
must be **1**!
This 1 can be implicit

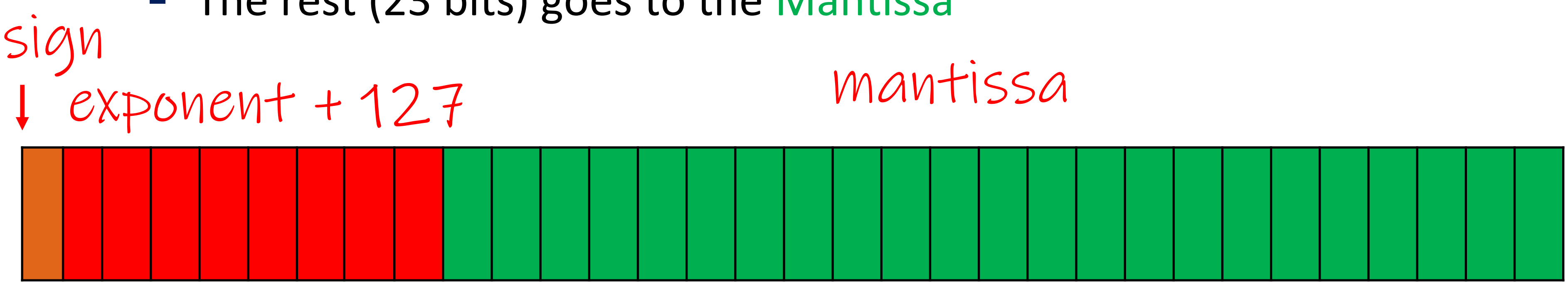
  - Mantissa: Everything after the binary point
  - Exponent: We are in base 2, so we raise 2 to this value

- We can represent a scientific notation binary number with only the Sign, Mantissa, and Exponent

IEEE Floating Point Notation

- We can represent a scientific notation binary number with only the Sign, Mantissa, and Exponent

- Allocate 32 bits, with
  - First bit goes to the Sign (1 for negative, 0 for non-negative)
  - The next 8 bits go to the Exponent + 127 (as an unsigned 8-bit int)
    - This means the exponent must fall between -127 and 128
  - The rest (23 bits) goes to the Mantissa



Special Numbers

- There are some special values to IEEE floating point representation

Value	Sign	Exponent	Mantissa
0	0	All 0's	All 0's
-0	1	All 0's	All 0's
NaN	1 or 0	All 1's	Not 0
∞	0	All 1's	All 0's
-∞	1	All 1's	All 0's

<- e.g. 0b00000...

Not a Number ->

At least one of the 23 bits in mantissa must be a 1

All 1's mean
0b11111111 or 0xFF

- There are also "Subnormal" values, but we won't talk about that

Floating Point: Finite Size Issues

- Float's are only 32 bits, and computers are finite
  - there is a limit to representable numbers
- DEMO
  - 1.1 + 2.2 != 3.3 ? (float_add.c)
  - 240000001 != 240000001 ? (int_float.c)
- "Underflow" can also be an issue
  - When a result is too small in magnitude to be representable
  - (Common issue with Bayesian computations)

Takeaway: Finite Resources

- Computers are physical machines, and limited by being physical machines
  - Many numbers are stored as approximations
  - Overflow or underflow can occur

- These errors can be catastrophic:

Ariane flight V88

Boeing 787

Data Representation Work Arounds

- There are “Workarounds” to data types with limited bits:
  - Choose data types with more bits (C examples)
    - `int128_t` (128-bit integer)
    - `double` (64-bit floating-point number)
  - Use custom data types that are only bound by memory size
    - Python has `integer` and `decimal`
    - Java has `BigInteger` and `BigDecimal`
  - Rigorous testing of software ☺

Lecture Outline

- Floats Continued
- **Logical Operators**
  - **Shifting**
- Boolean Algebra

Logical Operations on `bool`

- Operations on Boolean (True/False) values
  - Likely familiar with most of these from Java
  - AND, OR, XOR, NOT

XOR == eXclusive OR

A	NOT A
False	True
True	False

A	B	A AND B	A OR B	A XOR B
False	False	False	False	False
False	True	False	True	True
True	False	False	True	True
True	True	True	True	False

Bits as "`bool`"

- A Boolean value can be represented by a single bit
  - 1 is true, 0 is false
  - We can represent our logical operations as operations on bits

A	NOT A
0	1
1	0

A	B	A AND B	A OR B	A XOR B
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Bitwise Operators

Useful for HW01

- An individual bit is not a datatype, data types are group of bits. Instead, these operations work on all bits in a type
  - Each operator acts on each bit position independently
  - Consider the following examples on an imaginary 2-bit type
  - (Parenthesis in table contain the C syntax)

A	NOT A (~A)
00	11
01	10
10	01
11	00

A	B	A AND B (A & B)	A OR B (A | B)	A XOR B (A ^ B)
00	00	00	00	00
01	10	00	11	11
10	01	00	11	11
11	10	10	11	01
….	…	…	…	…
11	11	11	11	00

Bitwise Operators in C

Useful for HW01

- These bitwise operators exist in C
- Table below contains descriptions and example of syntax
  - Assume **`A`** and **`B`** are of type **`int`**

Logical Name	C Syntax	Example
AND	`&`	`A & B`
OR	`|`	`A | B`
XOR	`^`	`A ^ B`
NOT	`~`	`~A`

Shifting Bits

Useful for HW01

- Two more bitwise operators, left shift and right shift

Description	C Syntax	Bit pattern
Original x	`--`	`0b01101011`
X left shift by 1	`x << 1`	`0b11010110`
X right shift by 1	`x >> 1`	`0b00110101`
X left shift by 2	`x << 2`	`0b10101100`

  - Still confined to the size of the data type, bits can be shifted off on the left or right side.
  - During left shift, always fill in with 0's from the right
  - During a right shift: (More on these in a second)
    - Either fill with 0's from left (Logical)
    - Duplicate the MSB (Arithmetic)

Arithmetic vs Logical Shift

Useful for HW01

Description	Bit pattern
Original x	`0b10111011`
X >> 1 - X right shift by 1 (logical)	`0b01011101`
X >> 1 - X right shift by 1 (arithmetic)	`0b11011101`

- In C
  - The syntax for both shifts is the same (**`x >> 1`**)
  - the shift type is automatically chosen based on the data type
    - Unsigned types like **`unsigned int`** for logical right shift
    - Signed types like **`int`** or **`signed int`** for arithmetic right shift

Shifts & Powers of 2

Assume ints are 4 bits for examples

- When dealing with binary , Powers of 2 are everywhere
- Note that shifting to the left by one is the same as multiplying by 2

Before	Operation	After
`int x = 2; (0b0010)`	`x = x << 1;`	`x == 4; (0b0100)`

  - This extends to **`x << n`** being the same as **`x * 2`**^{n}

- Similar applies to right shifts for division

Before	Operation	After
`int x = -4; (0b1100)`	`x = x >> 1; (arithmetic)`	`x == -2; (0b1110)`
`unsigned int x = 12; (0b1100)`	`x = x >> 1; (logical)`	`x == 6; (0b0110)`

  - This extends to **`x >> n`** <u>usually</u> being the same as **`x / 2`**^{n}

Getting & Clearing Bits

Useful for HW01

- Can use a combination of shifts, ANDs and ORs to manipulate bits

0 indexed from the right

- Say I wanted to set get the 5th bit from an 8-bit integer 'a'
  - Answer **`(a >> 5) & 0x01`**
  - Walkthrough:

```
a = 0bYYXYYYYY  // X = bit we want
                // Y = bit we don't want
(a >> 5) = 0b*****YYX  // * = bit padded from
                       //     shift
(a >> 5) & 0x01 = 0b*****YYX
                 &0b00000001
                = 0b0000000X
```

At a bit level:

X & 0 = 0

x & 1 = X

Poll Everywhere

pollev.com/tqm

- Which of the following sets the MSB of any unsigned 8-bit int 'a' to 0, and leaves the rest of the bits the same?

A. ((1 << 7) & a) ^ a

B. ~(1<<7) & a

C. ((a >> 7) & 0) << 7

D. I’m not sure

```
a =         0bXYYYYYYY
result =    0b0YYYYYYY
```

Poll Everywhere

pollev.com/tqm

- Which of the following sets the MSB of any unsigned 8-bit int 'a' to 0, and leaves the rest of the bits the same?

A. ((1 << 7) & a) ^ a

B. ~(1<<7) & a

C. ((a >> 7) & 0) << 7

D. I'm not sure

```
a =         0bXYYYYYYY
result =    0b0YYYYYYY

a & 0b01111111 = result

a & ~(0b10000000) = result

a & ~(1 << 7) = result
```

Lecture Outline

- Floats Continued
- Logical Operators
  - Shifting
- **Boolean Algebra**

Disclaimer

- We just talked about bit-wise logical operators, and I will be using bit-wise operator syntax for the next section
  - 1 is still equal to TRUE
  - 0 is still equal to FALSE

- It may be easier to think of this next section as applying specifically to Boolean data types
  - (Though this can also be applied to bit-wise operators)
    - Treat True as the "all 1" bit pattern
    - Treat False as the "all 0" bit pattern

Useful for HW01

Boolean rules

- Identity
  - A & 1 = A
  - A & 0 = 0
  - A | 1 = 1
  - A | 0 = A
  - ~~A = NOT NOT A = A
- Associative
  - A & (B & C) = (A & B) & C
  - A | (B | C) = (A | B) | C
- Distributive
  - A & (B | C) = (A & B) | (A & C)
  - A | (B & C) = (A | B) & (A | C)
- More Identity
  - A & A = A
  - A | A = A
  - A & ~A = 0
  - A | ~A = 1

More on De Morgan's later

- De Morgan's Law
  - ~(A & B) = ~A | ~B
  - ~(A | B) = ~A & ~B

Truth Tables

- A table you can write for an expression to represent all possible combinations of input and output for an expression

- Truth Table for (A & (A & ~B)):

A (input)	B (input)	Output
0	0	0
0	1	0
1	0	1
1	1	0

Boolean Simplification

- We can apply rules to simplify Boolean patterns

- Consider the previous example
  - (A & (A & ~B))
  - ((A & A) & ~B) // By associative property
  - (A & ~B) // By distributive Property

- Consider:
  - (A | B) & (A | ~B)

Boolean rules

- Identity
  - A & 1 = A
  - A & 0 = 0
  - A | 1 = 1
  - A | 0 = A
  - ~~A = NOT NOT A = A
- Associative
  - A & (B & C) = (A & B) & C
  - A | (B | C) = (A | B) | C
- Distributive
  - A & (B | C) = (A & B) | (A & C)
  - A | (B & C) = (A | B) & (A | C)
- More Identity
  - A & A = A
  - A | A = A
  - A & ~A = 0
  - A | ~A = 1

More on De Morgan's soon

- De Morgan's Law
  - ~(A & B) = ~A | ~B
  - ~(A | B) = ~A & ~B

Simplify:

(A | B) & (A | ~B)

Boolean Simplification

- We can apply rules to simplify Boolean patterns

- Consider the previous example
  - (A & (A & ~B))
  - ((A & A) & ~B) // By associative property
  - (A & ~B) // By distributive Property

Simplification can have
Multiple correct simplifications

- Consider:
  - (A | B) & (A | ~B)
  - A | (B & ~B) // by distributive property
  - A | 0 // by identity property
  - A // by identity property

De Morgan's Law

- De Morgan's Law
  - ~(A & B) = ~A | ~B
  - ~(A | B) = ~A & ~B
- Provides a way to convert between AND to OR
  - (with some help from NOT)
- Truth Tables for proof:

A	B	~(A | B)	~A & ~B	~(A & B)	~A | ~B
0	0	1	1	1	1
0	1	0	0	1	1
1	0	0	0	1	1
1	1	0	0	0	0

De Morgan’s Law: Demo

- Write a statement equivalent to OR, but without using OR
  - A | B
  - ~~(A | B) // identity property
  - ~(~A & ~B) // De Morgan’s Law

- This still works for multi-bit data and bitwise operations

Useful for HW01

Boolean rules

These apply to multi-bit operations as well!

- Identity
  - A & 1 = A
  - A & 0 = 0
  - A | 1 = 1
  - A | 0 = A
  - ~~A = NOT NOT A = A

Bit-wise operations just follow these N times for N bits

- Associative
  - A & (B & C) = (A & B) & C
  - A | (B | C) = (A | B) | C
- Distributive
  - A & (B | C) = (A & B) | (A & C)
  - A | (B & C) = (A | B) & (A | C)
- More Identity
  - A & A = A
  - A | A = A
  - A & ~A = 0
  - A | ~A = 1
- De Morgan's Law
  - ~(A & B) = ~A | ~B
  - ~(A | B) = ~A & ~B

Next Lecture

- Next Time: We start hardware!
  - Start with Transistors & circuits
  - Booleans & bits will still be necessary
    - Be sure to be familiar with C bitwise ops, Boolean logic & De Morgan's Law

- HW00 Due this Friday!!!!

- HW01 & VM Setup to come out soon