COSC 511: Computer Architecture
Arithmetic for Computers

Week 3

Last Week

- Instruction Set Architectures (ISAs)
  - Examples of ISAs (x86, ARM)
  - Tradeoffs between different types of ISAs
  - Compatibility between ISAs
  - Real-time ISA translation use cases, benefits, and limitations
- ISA Design Principles
- ISAs optimize for the computer, not for the human.
  - Example: Twos compliment for handling negative integers.
- High-Level Language Optimization
  - Example: `gcc` allows programmers to control code optimization.
- Interesting side effects of ISA implementations
  - Illegal Opcodes

Arithmetic for Computers

- The obvious stuff:
  - Addition and Subtraction
  - Multiplication and Division
  - Following Order of Operations
- The not-so-obvious stuff:
  - Dealing with overflow
  - Representing fractional (floating point) values in binary

Arithmetic for Computers – Addition

- Remember:
  - `0 + 0 = 0`
  - `0 + 1 = 1`
  - `1 + 1 = 0, carry 1`
  - `1 + 1 + 1 = 1, carry 1`

```
   1  1111
   01010010   82
 + 01001110   78
 ___________
   10100000  160
```

Arithmetic for Computers – Subtraction

- Remember:
  - 0 - 0 = 0
  - 1 - 0 = 1
  - 1 - 1 = 0
  - 0 - 1 = 1, borrow 1

```
      1
    01
  01010010   82
- 00001110   14
  --------
  01000100   68
```

This works, but it's not how computers do it.

Arithmetic for Computers – Subtraction

- Remember: Make ISA design as simple as possible.
  - Is it easier to implement subtraction separately at the hardware level?
    - No, it is not.
  - It is much better to use a cool little trick to use addition to do subtraction.
    - Twos compliment!

Arithmetic for Computers – Subtraction

- Twos Compliment
  1. Apply bitwise NOT to the integer representation of the number.
  2. Add one to it.
  3. Add values together.

00001110

Arithmetic for Computers – Subtraction

- Twos Compliment
  1. Apply bitwise NOT to the integer representation of the number.
  2. Add one to it.
  3. Add values together.

00001110

[illegible]

Arithmetic for Computers – Subtraction

- Twos Compliment
  1. Apply bitwise NOT to the integer representation of the number.
  2. Add one to it.
  3. Add values together.

$$
\begin{array}{r}
\texttt{1} \\
\texttt{11110001} \\
+\quad\texttt{1} \\
\hline
\texttt{11110010}
\end{array}
$$

Arithmetic for Computers – Subtraction

- Remember:
  - 0 + 0 = 0
  - 0 + 1 = 1
  - 1 + 1 = 0, carry 1
  - 1 + 1 + 1 = 1, carry 1

```
 1 111   1
   01010010   82
 + 11110010  -14
 -----------
   01000100   68
```

Arithmetic for Computers – Multiplication

- Remember:
  - $0 \times 0 = 0$
  - $0 \times 1 = 0$
  - $1 \times 0 = 0$
  - $1 \times 1 = 1$

$$\begin{array}{r} 11101 \\ \times\ \underline{01001} \end{array}$$

Arithmetic for Computers – Multiplication

- Remember:
  - $0 \times 0 = 0$
  - $0 \times 1 = 0$
  - $1 \times 0 = 0$
  - $1 \times 1 = 1$

```
  11101
× 01001
-------
  11101
    000
```

Arithmetic for Computers – Multiplication

- Remember:
  - $0 \times 0 = 0$
  - $0 \times 1 = 0$
  - $1 \times 0 = 0$
  - $1 \times 1 = 1$

$$
\begin{array}{r}
11101 \\
\times\ \underline{01001} \\
11101 \\
+\ \underline{11101000}
\end{array}
$$

Arithmetic for Computers – Multiplication

- Remember:
  - 0 × 0 = 0
  - 0 × 1 = 0
  - 1 × 0 = 0
  - 1 × 1 = 1

```
   11101
+11101000
---------
100000101
```

How do computers make it easy to do this?

Answer: Shifting, then adding!

Arithmetic for Computers – Multiplication

- Remember:
  - 0 × 0 = 0
  - 0 × 1 = 0
  - 1 × 0 = 0
  - 1 × 1 = 1

$$
\begin{array}{r}
11101 \\
\times\ \underline{01001} \\
11101 \\
+\ \underline{11101000}
\end{array}
$$

Shift left 3 spaces.

Arithmetic for Computers – Division

- Division is hard for computers!
  - Programmers who work on building systems where tiny fractions of time matter try to avoid using division because it is slower than other operations.
  - There are two many division implementations: slow and fast division

- Slow Division
  - Keep subtracting until you get to a point at which you cannot subtract anymore.
  - Example: 7 ÷ 3
    1. 7 – 3 = 4
    2. 4 – 3 = 1
    3. $1 < 3$
    4. So, 7 ÷ 3 == 2 and 7 % 3 == 1

Arithmetic for Computers – Division

- Fast Division
  - There are many different fast division methods.
  - These fast division methods take less computation time but rely on estimating the final value.
  - Example: Goldschmidt Fast Division
    - AMD has used this approach in their CPUs since the launch of AMD Athlon in June 1999.

Arithmetic for Computers

- Order of Operations
  - Computers follow the same approach to order of operations that we use.
- How do computers handle order of operations?
  - They don't! That's your problem.
  - At least, it is if you're writing assembly.
- Compilers are responsible for generating Assembly code that obeys order of operations.

$$(5 + 3) / 45 * 3 \% 9 – 2 / 52$$

Arithmetic for Computers

Assume we are using a high-level language, and the datatype is byte.

$$\begin{array}{r} 11111111 \\ +00000001 \\ \hline 100000000 \end{array}$$

What's wrong?

We can't fit this in a byte. ☹

Arithmetic for Computers

- Year 2038 Problem
  - Many systems store time using the Unix epoch
  - Unix epoch: An integer value representing the number of seconds that have passed since January 1, 1970 at 00:00 UTC
    - At the start of today's class, the Unix epoch was 1694037600
  - Many systems use signed 32-bit integers for storing the Unix epoch
    - One integer is lost due to signing, so 31 bits are available for storing the timestamp.
  - On January 19, 2038 at 03:14:07 UTC, the timestamp will be:
    - `01111111 11111111 11111111 11111111`
  - On January 19, 2038 at 03:14:08 UTC, the timestamp will be:
    - `10000000 00000000 00000000 00000000`
  - Overflow!

Arithmetic for Computers

- Year 2038 Problem
  - Overflow!
  - Systems that use signed integers will travel back in time.
    - They will report time as December 13, 1901 at 20:45:52 UTC

Binary : 01111111 11111111 11111111 11110000

Decimal : 2147483632

Date : 2038-01-19 03:13:52 (UTC)

Date : 2038-01-19 03:13:52 (UTC)

  - Solutions
    - Use unsigned 32-bit integer
      - We'd be fine until February 7, 2106 at 06:28:15 UTC
    - Use a 64-bit integer
      - We'll be fine for approximately 292 billion years.

Arithmetic for Computers

- Back to our addition problem: How do we fix this?
  - Use a short instead of a byte? (2 bytes)
  - Use an int? (4 bytes)
  - Use a long? (8 bytes)
  - But what if that's still not enough?

- That's what Binary Coded Decimal (BCD) is for!
  - BCD stores each digit in a value separately.
  - Java's **`BitInteger`** is a BCD implementation.

Arithmetic for Computers

- BCD of 123
  - Big-Endian Order – Leftmost digit comes first.
    - `00000001 00000010 00000011`
  - Little-Endian Order – Rightmost digit comes first.
    - `00000011 00000010 00000001`

- This is great, but it wastes space. Why?
  - We only ever use the first four bits of our byte!
  - This is called unpacked BCD.

```
0 = 0000
1 = 0001
2 = 0010
3 = 0011
4 = 0100
5 = 0101
6 = 0110
7 = 0111
8 = 1000
9 = 1001
```

Arithmetic for Computers

- Packed BCD
  - Packed BCD is more efficient.
  - Use one byte to store two digits.
- Packed BCD of 123
  - Big-Endian Order – Leftmost digit comes first.
    - `00000001 00100011`
  - Little-Endian Order – Rightmost digit comes first.
    - `00110010 00010000`

Arithmetic for Computers

- Pros of BCD
  - You can store really large values!
  - BCD is useful for dealing with rounding errors.
- Cons of BCD
  - Harder to do math operations with.
  - Depending on the value you are storing, more space is used.
    - Example: **`123`**
      - Without BCD: **`01111011`**
      - Big-Endian Packed BCD: **`00000001 00100011`**

Arithmetic for Computers

- Your data is what you make of it.
  - `00000001 00100011` has meaning as big-endian packed BCD.
  - `00000001 00100011` has meaning as a short.
  - This is why remote code execution vulnerabilities are a problem.
    - The same data can be interpreted in different ways.

```
public class ArrayProblem {
	public static void main(String[] args) {
		int ar[] = { 1, 2, 3, 4, 5 };
		for (int i = 0; i <= ar.length; i++)
			System.out.println(ar[i]);
	}
}
```

Arithmetic for Computers

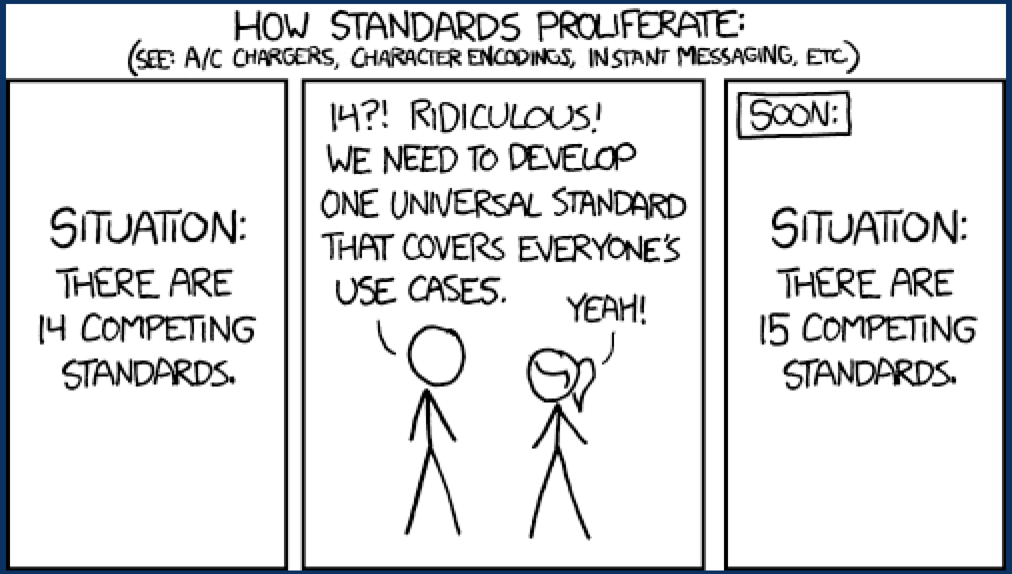
- Your data is what you make of it.
  - **00000001 00100011** has meaning as big-endian packed BCD.
  - **00000001 00100011** has meaning as a short.
  - This is why remote code execution vulnerabilities are a problem.
    - The same data can be interpreted in different ways.
  - …, other data stored in memory adjacent to the …ned integer value.
    - …he danger is in writing.
    - …n stores data in memory, I can deliberately exceed the …de stored in memory to be executed.
    - …am will malfunction and/or crash.
    - …re into an otherwise safe application.

```
#include <iostream>
using namespace std;

int main()
{
    int arr[] = {1,2,3,4,5};
    for (int x = 0; x <= 10; x++)
        cout << arr[x] << endl;

    return 0;
}
```

1	1
2	2
3	3
4	4
5	5
0	0
-2127166208	452279552
-137567531	31503784
1	1
0	0
-1557131888	-987005552

Arithmetic for Computers

- Storing floating point values in binary is very confusing.
  - Many different implementations for storing floating point values used to exist.



  - IEEE Standard for Floating-Point Arithmetic (IEEE 754), established in 1985, has become the standard implementation.

Arithmetic for Computers

- IEEE 754 defines two sizes for storing floating point values
  - Single precision numbers
    - Use 32 bits
    - We usually call these "floats"
  - Double precision numbers
    - Use 64 bits
    - We usually call these "doubles"
- IEEE 754 Floating Point Components
  1. Sign bit (0 for positive, 1 for negative)
  2. Exponent
     - 8 bits used for single, 11 bits for double
  3. Mantissa
     - 23 bits used for single, 52 bits for double

Arithmetic for Computers

- In most cases today, doubles are used for better precision.
- Converting to floating point values to binary:
    1. Convert the whole number portion to binary as you normally would.

$$85.125$$

$$85 == 1010101_2$$

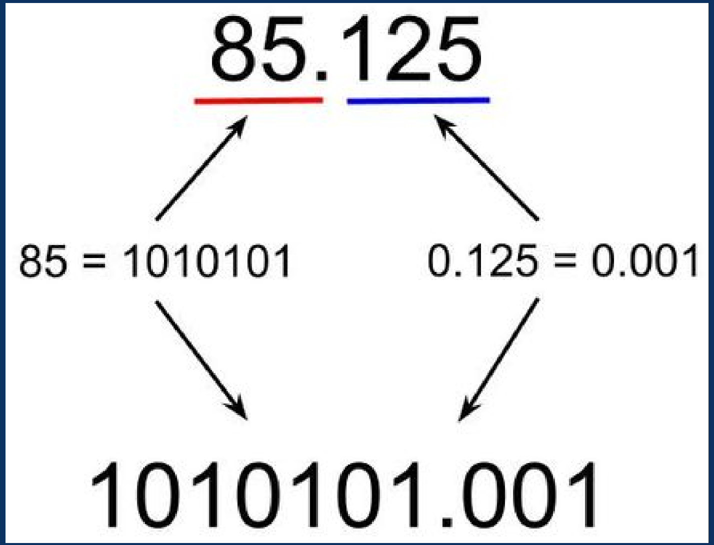
    2. Convert the decimal portion to binary.

Decimal Number Multiplication	Result	Number in front of decimal
0.125 x 2	0.25	0
0.25 x 2	0.5	0
0.5 x 2	1.0	1
0.0 x 2	0.0	0

0.125 = 001

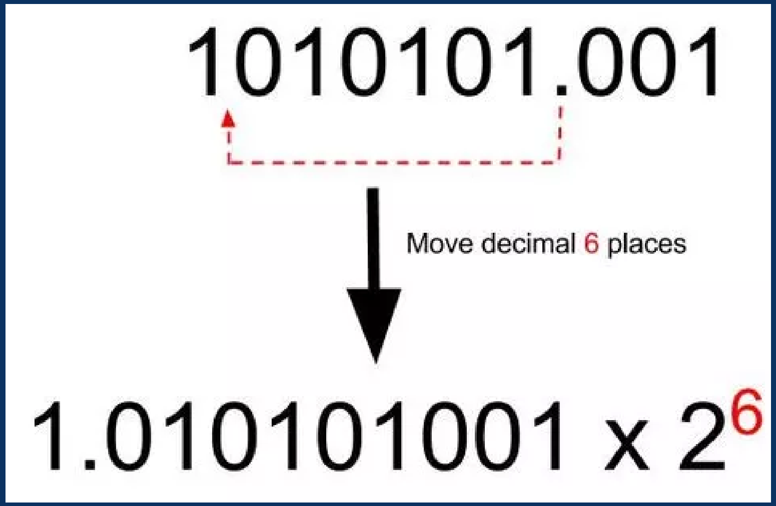
Arithmetic for Computers

- Converting to floating point values to binary:
  3. For organizational purposes, concatenate both binary components together.



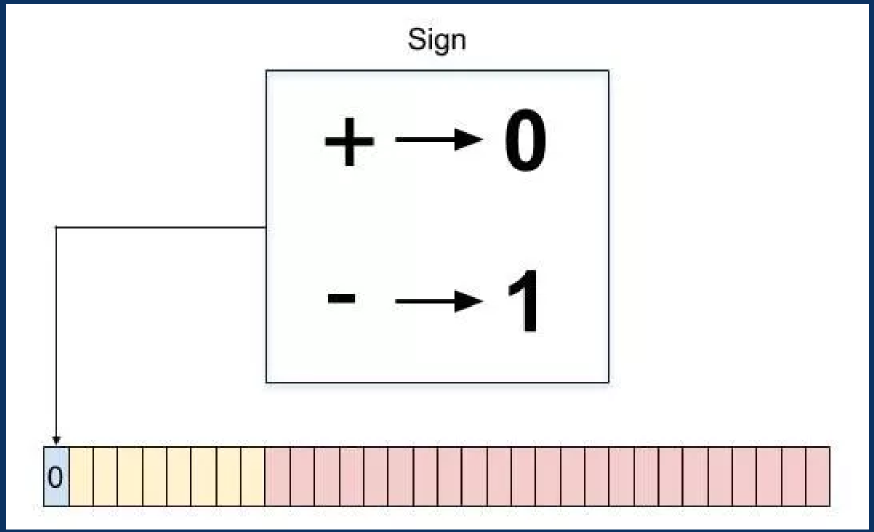
Arithmetic for Computers

- Converting to floating point values to binary:
  4. Convert the binary value to base 2 scientific notation.



Arithmetic for Computers

- Converting to floating point values to binary:
  5. Based on the sign of the original number, take note of the most significant bit.



Arithmetic for Computers

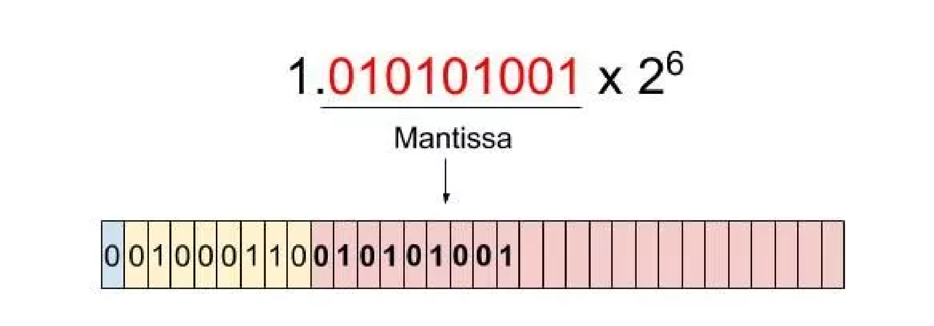
- Converting to floating point values to binary:
   6. Determine the exponent to use by applying the “exponent bias”
      - The purpose of exponent bias is to prevent issues with floating point signing
      - If you are using single precision, you calculate bias by adding 127 to the exponent from step 4.
      - If you are using double precision, you calculate bias by adding 1023 to the exponent from step 4.

$$1.010101001 \times 2^{6}$$

$$127 + 6 = 133$$

Arithmetic for Computers

- Converting to floating point values to binary:
  7. Determine the mantissa
     - Mantissa – Refers to the binary component after the decimal point in your scientific notation representation.



Arithmetic for Computers

- Converting to floating point values to binary:
  8. Assemble the components of your calculations together!