Computer Systems

Week 1. Numbers

Computers are a collection of transistors. They can be either “on” or “off”, according to a particular electrical state (conducting or not).

This fact forces binary representation: “off” — 0, “on” — 1. Binary digits are called **bits**, all data must be represented in the sequences of ones and zeros.

For practical engineering reasons it is better to stick with basic components that only have two states — binary. In practice, though, it is possible to build other types of computers, and they have been built. (LOOK UP)

All the data is a pattern of bits. The interpretation put on it is down to the program.

Historically, memory and processing were very limited. It still is, but not to the same extent.

**Representing text**

**(DO LATER)**

Representing numbers is more subtle, as we need to worry about arithmetic.

**Representing numbers**

Different notations

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| System | Base | Symbols | Used by humans? | Used in computers? |
| Decimal | 10 | 0, 1, … 9 | Yes | No |
| Binary | 2 | 0, 1 | No | Yes |
| Octal | 8 | 0, 1, … 7 | No | No |
| Hexadecimal | 16 | 0, 1, … 9  A, B, … F | No | No |

Counting in different number systems

|  |  |  |  |
| --- | --- | --- | --- |
| Decimal | Binary | Octal | Hexadecimal |
| 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 |
| 2 | 10 | 2 | 2 |
| 3 | 11 | 3 | 3 |
| 4 | 100 | 4 | 4 |
| 5 | 101 | 5 | 5 |
| 6 | 110 | 6 | 6 |
| 7 | 111 | 7 | 7 |
| 8 | 1000 | 10 | 8 |
| 9 | 1001 | 11 | 9 |
| 10 | 1010 | 12 | A |
| 11 | 1011 | 13 | B |
| 12 | 1100 | 14 | C |
| 13 | 1101 | 15 | D |
| 14 | 1110 | 16 | E |
| 15 | 1111 | 17 | F |
| 16 | 10000 | 20 | 10 |
| 17 | 10001 | 21 | 11 |
| 18 | 10010 | 22 | 12 |
| 19 | 10011 | 23 | 13 |
| 20 | 10100 | 24 | 14 |
| 21 | 10101 | 25 | 15 |
| 22 | 10110 | 26 | 16 |
| 23 | 10111 | 27 | 17 |

Extra table

|  |  |
| --- | --- |
| Decimal | Hexadecimal |
| 24 | 18 |
| 25 | 19 |
| 26 | 1A |
| 27 | 1B |
| 28 | 1C |
| 29 | 1D |
| 30 | 1E |
| 31 | 1F |
| 32 | 20 |
| … |  |
| 161 | A1 |
| 162 | A2 |

Conversion among different notations

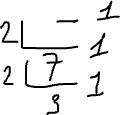
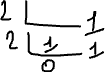
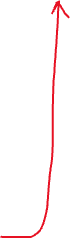
**Decimal to Binary and Binary to Decimal**

1. **Decimal to Binary**
   1. Divide by two, keep track of remainder
   2. Go from the last remainder to the first
   3. That will be a binary representation of a decimal number

Example: 12510  = ?2



12510 = 11111012



1. **Binary to Decimal** 
   1. Multiply each bit by 2n, where n is the “weight” (“positional value”) of the bit.
   2. The weight is the position of the bit, starting from 0 on the right.
   3. Add the results

Example: 1010112  = ?10

1010112 =>

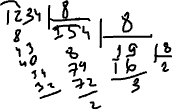
|  |  |
| --- | --- |
| 1 x 20 | 1 |
| 1 x 21 | 2 |
| 0 x 22 | 0 |
| 1 x 23 | 8 |
| 0 x 24 | 0 |
| 1 x 25 | 32 |

1 + 2 + 8 + 32 = 43

1010112 = 4310

**Decimal to Octal and Octal to Decimal**

1. Decimal to Octal
   1. Divide by 8



* 1. Keep track of the remainder
  2. Go from the last remainder to the first

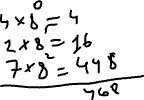


Example: 123410 = 23228

1. Octal to Decimal
   1. Multiply each digit by 8n, where n is the “weight” of the digit



* 1. The weight is the position of the digit, starting from 0 on the right



* 1. Add the results



Example: 7248 = 46810

**Decimal to Hexadecimal and Hexadecimal to Decimal**



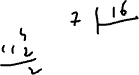
1. Decimal to Hexadecimal



* 1. Divide by 16



* 1. Keep track of the remainder



* 1. Go from the last remainder to the first



Example: 123410  = 4D216



1. Hexadecimal to Decimal



* 1. Multiply each digit by 16n, where n is the “weight” of the digit



* 1. The weight is the position of the digit, starting from 0 on the right
  2. Add the results



Example: ABC16 = 274810



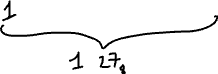
**Binary to Octal and Octal to Binary**



1. Binary to Octal
   1. Group bits in threes, starting from right
   2. Convert to octal digits



Example: 10110101112 = ?8



1. Octal to Binary
   1. Convert each digit to a 3-bit equivalent binary representation
   2. Every digit will be converted to 3 bits, even if all bits are zeros

Example: 7058 = ?2

7 -> 111

0 -> 000

5 -> 101

7058 = 1110001012

**Binary to Hexadecimal and Hexadecimal to Binary**

1. Binary to Hexadecimal
   1. Group to bits in fours, starting on right
   2. Convert to hexadecimal digits

Example: 10101110112 = ?16



|  |  |
| --- | --- |
| 1011 | B |
| 1011 | B |
| 0010 | 2 |

10101110112 = 2BB16

1. Hexadecimal to Binary
   1. Convert each digit to a 4-bit equivalent binary representation
   2. Every digit will be converted to 4 bits, even if all bits are zeros

Example: 10AF16 = ?2

|  |  |
| --- | --- |
| 1 | 0001 |
| 0 | 0000 |
| A | 1010 |
| F | 1111 |

10АF10 = 00010000101011112

**Octal to Hexidecimal and Hexidecimal to Octal**

1. Octal to Hexidecimal
   1. Use a binary notation as an intermediate representation

10768 = ?16



10768 = 23E16

1. Hexadecimal to Octal
   1. Use binary notation as an intermediate representation

Example: 1F0C16 = ?8



1F0C16 = 174148

**Bitwise Boolean Operations (can add diagrams and other info)**

0 = False

1 = True

1. OR
   1. 0 OR 0 => 0
   2. 1 OR 0 => 1
   3. 0 OR 1 => 1
   4. 1 OR 1 => 1
2. AND
   1. 0 AND 0 => 0
   2. 1 AND 0 => 0
   3. 0 AND 1 => 0
   4. 1 AND 1 => 1
3. XOR (eXclusive OR)
   1. 0 XOR 0 => 0
   2. 0 XOR 1 => 1
   3. 1 XOR 0 => 1
   4. 1 XOR 1 => 0
4. NOT
   1. NOT 1 => 0
   2. NOT 0 => 1
5. NAND (inverted AND)
   1. 0 NAND 0 => 1
   2. 0 NAND 1 => 1
   3. 1 NAND 0 => 1
   4. 1 NAND 1 => 0
6. NOR (inverted OR)
   1. 0 NOR 0 => 1
   2. 0 NOR 1 => 0
   3. 1 NOR 0 => 0
   4. 1 NOR 1 => 0

Can do these bit-wise on binary values.

Example for XOR:

0011101111

0001001001

0010100110

Bits and Numbers

|  |  |  |  |
| --- | --- | --- | --- |
| No of bits | Min (Binary) | Max (Binary) | Max (Decimal) |
| 1 | 0 | 1 | 1 |
| 2 | 00 | 11 | 3 |
| 3 | 000 | 111 | 7 |
| 4 | 0000 | 1111 | 15 |
| 5 | 00000 | 11111 | 31 |
| 6 | 000000 | 111111 | 63 |
| 7 | 0000000 | 1111111 | 127 |
| 8 | 00000000 | 11111111 | 255 |
| **n** | 0…0 (n bits) | 1…1 (n bits) | **2n - 1** |

Approximating Big Powers of 2

|  |  |  |  |
| --- | --- | --- | --- |
| Binary Value | Value (Exact) | Value (Approx) | Equivalent (Decimal) |
| 210 | 1024 | 1000 | 103 (kilo k) |
| 220 | 1048576 | 1000000 (million) | 106 (Mega M) |
| 230 | 1073741824 | 1000000000 (billion) | 109 (Giga G) |
| 240 | 1099511628000 | 1000000000000 (trillion) | 1012 (Tera T) |

In computing, particularly w.r.t. memory, base-2 interpretation generally applies (Column 2)

For example, currently my disk is using 114,264,887,296 bytes of memory. In order to convert that to gigabytes, I need to divide this figure by 230, which equals to 106.4 Gb.

**Overflow**

For example, we have 8 bits. The highest value we can store is 28 – 1, which equals to 255. What if arithmetic goes outside of the storable range?

255 + 1 should be equal 256.

|  |  |
| --- | --- |
| 1111 1111 | 255 |
| 0000 0001 | 1 |
| 1 0000 0000 | 0 |

1 in this answer is an **overflow bit**, as it is not represented within the range of 8 bits, so the answer is confused with zero.

|  |
| --- |
| Overflow is the term for an operation whose results exceed the space allocated for a number. |

In a given type of computer, the size of integers is a fixed number of bits.

32 or 64 bits are popular choices these days. It is possible that addition of two n bit numbers yields a result requiring n+1 bits.

Going back to the example with 255 + 1 in 8-bit arithmetic, it seems that 255 is behaving like -1. In fact, 11111111 is two’s complement representation of -1.

If we want to show a negative number X in n number of bits, we can use the formula **2n – X**

**For example,** I would like to represent the number -15 in 8-bits: 28 – 15 = 256 – 15 = 241

24110 = 11110001

-15 = 11110001

**2’s complement — General method**

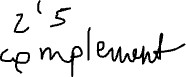
1. Flip all the bits
2. Add 1

-15 in 2’s complement

|  |  |
| --- | --- |
| 15 in 8-bits | 0000 1111 |
| Flip the bits | 1111 0000 |
| Add 1 | 1111 0001 |

This method works in any number of bits

**8 bit signed integers**



|  |  |
| --- | --- |
| 127 | **0**1111111 |
| … | … |
| 1 | **0**0000001 |
| 0 | **0**0000000 |
| -1 | **1**1111111 |
| -2 | **1**1111110 |
| … |  |
| -128 | **1**0000000 |

Most significant bit (MSB) shows sign:

**0** for non-negative (zero or positive)

**1** for negative numbers

**Range for n-bit signed integers**

For signed integers: **-2n-1 … 2n-1 – 1**

|  |  |  |
| --- | --- | --- |
| **Bits** | **Minimum** | **Maximum** |
| 8 | -128 | 127 |
| 16 | -32768 | 32767 |
| 32 | -2147483648 | 2147483647 |

For un-signed integers: **0 … 2n – 1**

|  |  |  |
| --- | --- | --- |
| **Bits** | **Minimum** | **Maximum** |
| 8 | 0 | 255 |
| 16 | 0 | 65535 |
| 32 | 0 | 4294967295 |

**How should we treat binary numbers?**

For much of the arithmetic (+, -, \*) it does not matter.

For comparisons we need to know, whether the numbers are signed or unsigned:

For example:

* *Signed -1 < 0* **TRUE**
* *Unsigned 255 < 0* **FALSE**

**Integer Numbers in Java**

For integer (whole number) arithmetic:

* Values are treated as signed
* Bits beyond storable range are lost
* Overflow ignored — no errors flagged!

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Data Types** | byte | short | int | long |
| Bytes | 1 | 2 | 4 | 8 |
| Bits | 8 | 16 | 32 | 64 |

A Boolean value (true or false) is 1 bit (1 or 0)

**Integer Numbers — Sign Extension**

Sometimes you extend a number by giving it more space

For unsigned arithmetic: just provide extra zeros

Eg. Extending 8 to 16:

|  |  |
| --- | --- |
|  | 11111111 |
| 00000000 | 11111111 |

For signed arithmetic: use sign extension, meaning, repeat the MSB.

**1**1111111 -> **11111111 1**1111111

Java uses sign extension when covering byte to short (short to int, int to long).

|  |
| --- |
| It is important to make sure that the data type is big enough for the numbers you want to compute with! |

Example for choosing the data type for banking software. Let’s take int.

Int is signed, so max int = 231 – 1 = 2 \* (210)3 – 1 ≈ 2 \* 109 pence = £2 \* 107 = 20 million pounds

So this type might not be useful, we can use long or BigInteger

**Representing Real Numbers**

Two of the common ways of representing real numbers:

* **Fixed Point:** binary point is fixed e.g. 1101101.0001001
* **Floating Point:** binary point floats to the right of the most significant 1 and an exponent is used e.g. 1.1011010001001 x 26

**Fixed Point Decimal-to-Binary**

* Integer part convert as before (repeated division by 2)
* Non-integer part follows the opposite process
* Repeated multiplication by 2, keeping the integer part:

Example 0.53110 = ?2

0.531 х 2 = **1**.074

0.074 х 2 = **0**.148

0.148 х 2 = **0**.296

0.296 х 2 = **0**.592

0.592 х 2 = **1**.184

0.184 x 2 = **0**.368

0.53710 = 0.1000102

**Fixed Point Binary-to-Decimal**

* Allocate subset bits to integer part, and the remainder to the non-integer part

For example,

4+4 bits: 1101.01012 = ?10

1 х 23 + 1 х 22 + 0 х 21 + 1 х 20 + 0 х 2-1 + 1 х 2-2 + 0 х 2-3 + 1 х 2-4 = 8 + 4 + 0 + 1 + 0.0 + 0.25 + 0.0 + 0.0625 = 13 + 0.3125 = 13.3125

**Fixed Point Arithmetic**

Everything is the same as for whole numbers

Example: 01001.010 – 00010.100

Take 2C and add:

**Decimal Fractions (Base 10)**

There are different ways to show the fractions in base 10.

**Infinite Decimal Fractions**

If only fixed number of decimal places allowed, the rest is lost.

Eg. Only four places 0.3333 the missing rest is rounding error! (If we multiply this figure by 1 \* 10^9, we will see the rounding error.

**Binary Fractions (Base 2)**

Similar e.g.

**Floating Point Representation**

General principle is like a “scientific notation”, but in binary

Numbers are represented as:

|  |  |  |
| --- | --- | --- |
| Sign (+/-) | Mantissa (m) | Exponent (e) |
| 1 bit  0 for +  1 for - | Actual significant digits | 2s complement signed binary. Shows where binary point goes. |

Floating Point Representation in Java (based on IEEE 754)

|  |  |  |
| --- | --- | --- |
| **S** | **Offset e** | **Mantissa m** |
| Sign 0, 1 for +,- | Exponent 2’s complement signed binary | Leading mantissa bit (1.) left out |

* take bits of m
* put “**1.**” at start, so normalized
* move binary point right **e** places
  + or left for negative **e**
* note – e is stored with an “offset” added to it (127)

**Java types for floating point**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **No. of bits** | | | | |  |
| **Type** | **Sign** | **Mantissa** | **Exponent** | **Total** | **Bytes** |
| float | 1 | 23 | 8 | 32 | 4 |
| double | 1 | 52 | 11 | 64 | 8 |

52 bit mantissa:  **2 to the power of 53, not 52 is because we use 1 extra for hidden bit.**

We get 15 significant decimal digits in double data type.

|  |  |  |
| --- | --- | --- |
| Sign | Exponent with an offset | Mantissa |
| 0 | 1000 0100 | 010 1010 1000 0000 0000 0000 |
| 0 for + | 5 + offset 127 = 132 | Mantissa without the leading 1 |

The offset is calculated:

Why offset is calculated like this? For easier comparison.

**Money as a Floating point? Not a good practice!**

Pence need infinite binary fractions 10p is 0.000110011001001100…, we will get rounding errors. Always use int or long (or BigInteger) for money.

We can calculate factorial with double up to 170, after that we will get Infinity as a result.

But let’s take 170 and look at the factorial calculated.

|  |
| --- |
| 170! = 7.257415615307994E306 |
| = 725741561530799**4**E291 this digit is wrong, and there are 291 more digits after it |
| The digit should be **8** |

* Floating point arithmetic loses accuracy in least significant digits.
* Most significant and overall size, are OK

**Why is 171! Too big?**

Needs binary exponent ≈ 1030 > 1024 = 2^10. Exponent is too big to fit 11 bits (signed) for Java double

**Floating Point Overflow**

In Java:

If result is too big for data type,

It’s a special value **POSITIVE\_INFINITY (Float.POSITIVE\_INFINITY, Double.POSITIVE\_INFINITY).**

Other special values:

If too big but negative: **NEGATIVE\_INFINITY**

Indistinguishable from 0 but known to be negative: -0.0

Impossible number (e.g. Sqrt(-1)) **NaN (“Not a number”)**

If you divide a negative number by zero, you will get **-infinity,** as numerator is negative, the **+infinity** will appear if the numerator is positive.

These special values allow you to check for overflow in a program, unlike the case for integer arithmetic.

Floating point unit – a dedicated circuit for such operations.