Data Representation

- At its most basic level, all digital information must reduce to 0s and 1s, which can be discussed as binary, octal, or hex data.
- There's no practical limit on how it can be interpreted to represent more complex structured data

Type of Data	Representation
Integers	2's complement
Floating point numbers	IEEE 754
Text	ASCII, Unicode
Images	GIF, JPEG, etc.
Video	MPEG, etc.
Audio	MP4, etc.

Integers

Suppose we only have 4 bits to work with to represent a single integer. There are $2^4 = 16$ different bit sequences possible. Here are 4 different ways to interpret what number the possible bits represent (there are others, too).

Bits	Unsigned	2's Comp	Excess 7	Excess 8
0000	0	0	-7	-8
0001	1	1	-6	-7
0010	2	2	-5	-6
0011	3	3	-4	-5
0100	4	4	-3	-4
0101	5	5	-2	-3
0110	6	6	-1	-2
0111	7	7	0	-1
1000	8	-8	1	0
1001	9	-7	2	1
1010	10	-6	3	2
1011	11	-5	4	3
1100	12	-4	5	4
1101	13	-3	6	5
1110	14	-2	7	6
1111	15	-1	8	7

There are advantages and disadvantages for each scheme. For integer arithmetic in hardware 2's complement is the most commonly implemented. Unsigned integers are used in systems programming languages like C. Excess notation is used for exponents in the IEEE 754 floating point formats.

2's Complement Numbers

Some Facts About N-bit 2's Complement Numbers

- There are 2^N patterns or numbers possible in N bits.
  - We need to reserve half for positive, half for negative numbers.
  - Half of $2^N = 2^N/2 = 2^{N-1}$
  - We must reserve one bit pattern for zero, so the number of positive numbers is one less than the number of negative numbers.
- Most negative number: -2^{N-1}
- Most positive number: $+2^{N-1} - 1$
- Leftmost bit can be used as a sign bit
  - 0: positive
  - 1: negative
- Positive overflow and negative overflow occurs when a result is produced that is outside the legal range.
- Illegal value cannot be represented, so a "wrap-around" occurs to a different number.
  - Positive Integer Overflow: Most positive number + 1 = Most negative number
  - Negative Integer Overflow: Most negative number – 1 = Most positive number
  - In most programming environments, overflow is not checked as a runtime error. Think about that for a moment: your program will compile and run with no errors and yet still generate incorrect results.

Here's a simple C program that demonstrates positive overflow.

```
int x = 2147483647;
printf("%d\n",x);
x++;
printf("%d\n",x);
```

Output

```
2147483647
-2147483648
```

There is no easy way to detect overflow. Programmer must be cognizant of the quirks of finite-precision arithmetic.

Note that in Java, the statement:

```
int x = 2147483648;
```

would be caught as an error by the compiler. Your C compiler may or may not catch it.

IEEE 754

Floating point arithmetic has always been handled separately from integer arithmetic in computer chips. Each type of calculation has its own unique issues, they can't be lumped together. In early hardware implementations, differences in execution performance were severe, calculations used integer arithmetic if at all possible, and floating point only when necessary. In the meantime, great progress has been made in floating point algorithms and hardware, the performance penalty still exists but is much smaller.

The history of floating point computer arithmetic has many infamous cases where mistakes were discovered. See for example the story of an error discovered in 1994 for the FDIV (floating point divide) instruction in the Intel Pentium, Wikipedia article at https://en.wikipedia.org/wiki/Pentium_FDIV_bug. The Institute of Electrical and Electronic Engineers (IEEE) is a professional organization for engineers and computer scientists that host a large number of conferences and publish standards for many technologies. The documents are numbered sequentially and the document number becomes part of the name of the standard. Some common standards:

- IEEE 754: floating point number representation
- IEEE 802.3: wired Ethernet
- IEEE 802.11: Wi-Fi networks

The IEEE 754 standard for floating point was a major step forward in providing a more solid foundation for implementing floating point algorithms in hardware. Almost all CPU manufacturers now use it, and this representation trickles upward into the programming language. When performing floating point arithmetic in C, C++, Java, C#, almost any language today, it's a good assumption that the calculation will be performed according to the rules of IEEE 754.

Data Rep

Single precision

- Sign: bit 31 (0 = positive, 1 = negative)
- Exponent: 8 bits (bits 30-23) base 2 exponent in excess 127 notation (-
- Mantissa: 23 bits (bits 22-0) normalized base 2 fraction

In addition, special representations defined for

- $+\infty$
- $-\infty$
- NaN (not a number)
- +0
- –0
- Denormalized numbers (for delaying the occurrence of underflow)

Normalized Mantissa

A normalized mantissa means that the first bit is assumed to be 1. In base 2, positive numbers are represented on the range:

$1.00000\ldots_2 \times 2^{-126}$ through $1.11111\ldots_2 \times 2^{+127}$

Similarly on the negative half of the number line:

$-1.0000\ldots_2 \times 2^{-126}$ through $-1.1111\ldots_2 \times 2^{+127}$

Since the mantissa always starts with 1, it does not need to be explicitly stored. But you have to remember to add it in when calculating what a floating point format value really represents. Also note that even though the exponent range is -127 to +128, the values -127 and +128 are reserved for special values. So for normalized values, the exponent range is -126 to +127.

Special Values

Special values lead to greatly improved handling of computations that would otherwise result in inconsistent values.

Denormalized Numbers
Denormalized numbers allow the range of very small values to be slightly extended before reaching underflow. For example, imagine taking the smallest positive normalized number and dividing it by two:

$1.00000 \text{ X } 2^{-127} / 2 = ?$ (-127 is the most negative exponent, answer cannot be $1.0 \text{ X } 2^{-128}$)

Without denormalized numbers, the result would be set to 0, an example of positive underflow. But denormalized numbers allow the calculation to be continued for another 23 bits:

$1.00000 \text{ X } 2^{-127} / 2 = 0.10000 \text{ X } 2^{-127}$

This is a denormalized number because the first digit of the mantissa is no longer 1, but 0. A special exponent value is used to distinguish between normalized and denormalized values. Since there are 23 bits in the mantissa, even smaller denormalized numbers can be represented, all the way down to

$0.00000000000000000000001 \text{ X } 2^{-127}$

If you divide the smallest positive denormalized number by 2, the operation will generate positive underflow and the result will be set to positive 0.

Conversion Example
Let's do some simple conversion operations that don't involve special values like infinity, denormalized numbers, or NaN. Let's decode the IEEE 754 single precision value 42E00000.

- Expand to binary: `0100 0010 1110 0000 0000 0000 0000 0000`
  - Sign: `0`
  - Exponent: `100 0010 1`
  - Mantissa: `110 0000 0000 0000 0000 0000`
- Decode
  - Sign: 0 is positive
  - Exponent: $1000\ 0101 = 85_{16} = 133_{10}$; $133 - 127 = 6$
  - Mantissa: $1100\ldots. = 1 + 2^{-1} + 2^{-2} = 1 + 0.5 + 0.25 = 1.75$
  - Value: $1.75 \text{ X } 2^6 = 1.75 \text{ X } 64 = 112$
  - Use technique in C described in C notes #4 to confirm

Precision Limitation
- Only some numbers can be represented exactly in a finite number of bits.
- Other numbers will have to be approximated with some round off error.

Overflow
- Number becomes too positive or too negative to be correctly represented.
- IEEE 754 notation includes representations for positive and negative infinity.

Underflow
- Number becomes too close to zero to be distinguishable from zero, and is therefore set to zero.
- Can happen from both the positive or negative side of zero.
- Denormalized numbers are used to avoid underflow in a specific narrow range.

Some Exact Normalized Fractions

Although any floating point number can be approximated in IEEE 754, some numbers have a simpler representation than others. If you consider that the mantissa represents the summation of a series of negative powers of 2, then numbers which can be represented by a mantissa with only a few 1s in the leftmost bit positions (2^{-1}, 2^{-2}, 2^{-3}, etc) will be easier to calculate than numbers whose mantissa requires larger negative powers of 2 (2^{-20}, 2^{-21}, 2^{-22}, etc).

Mantissa	Sum	Value
10000 ….	1 + 0.5	1.5
01000 ….	1 + 0.25	1.25
00100 ….	1 + 0.125	1.125
00010 ….	1 + 0.0625	1.0625
11000 ….	1 + 0.5 + 0.25	1.75
10100 ….	1 + 0.5 + 0.125	1.625
11100 ….	1 + 0.5 + 0.25 + 0.125	1.875

Combining a mantissa with an exponent that represents a power of 2 gives rise to all possible values that can be represented exactly.

Example: $1.875 \times 2^5 = 1.875 \times 32 = 60$

But 61 or 63 may not be represented exactly.

Exercise: what is the binary representation of 60?

Why Excess 127 Notation for Exponent?

The reason that the exponent is represented in excess 127 rather than regular two's complement is to make it possible to do a quick comparison of two's complement numbers without having to denormalize them.

In excess notation, a negative exponent looks like a smaller number than a positive number. This is not true with two's complement because of the sign bit: negative numbers have a 1 in the most significant bit, positive numbers have a 0. With the exponent to the left of the mantissa and using excess 127 notation, two floating point numbers can be compared as if they were unsigned integers to get a quick answer to what their positions are relative to one another. Note that the comparison is necessary only if the two numbers have the same sign bit – both are negative or both are positive. If one number is negative and the other is positive, then the negative number is trivially known to be less than the positive number.

IEEE 754 Double Precision
64 bit format

Circuits for Arithmetic
The ALU (arithmetic logic unit) of the CPU implements basic arithmetic operations in hardware.
Implementation must be based on a specific data representation
Usually it's designed to operate on integers in 2's complement format.

Signed Addition
If we assume 2's complement as the representation for integers, and we accept the possibility of overflow, then we can build a circuit to perform addition by starting with the half adder (HA) and full adder (FA). The half adder adds two bits and generates a sum and a carry. The full adder extends this by adding three bits – 2 data bits and carry-in – and produces the same two outputs as the half adder – sum and carry-out.

X	Y	Z	C_{out}
0	0	**0**	**0**
0	1	**1**	**0**
1	0	**1**	**0**
1	1	**0**	**1**

If we're adding two numbers A and B, we compute the sum one column at a time. If we zoom in on one column, this reduces to defining the logic for:

$$X + Y = Z + C_{out}$$

An example circuit that implements the required logic is:

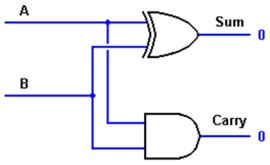

Figure from: http://www.circuitstoday.com/half-adder-and-full-adder

The full adder (FA) generalizes the half adder by accounting for the carry-in input. The possibilities:

C_{in}	X	Y	Z	C_{out}
0	0	0	**0**	**0**
0	0	1	**1**	**0**
0	1	0	**1**	**0**
0	1	1	**0**	**1**
1	0	0	**1**	**0**
1	0	1	**0**	**1**
1	1	0	**0**	**1**
1	1	1	**1**	**1**

- Z is a parity bit for C_{in}, X, and Y.
- C_{out} checks that 2 or more of the inputs are 1.

The calculation is

$$X + Y + C_{in} = Z + C_{out}$$

An example circuit that implements this logic is:

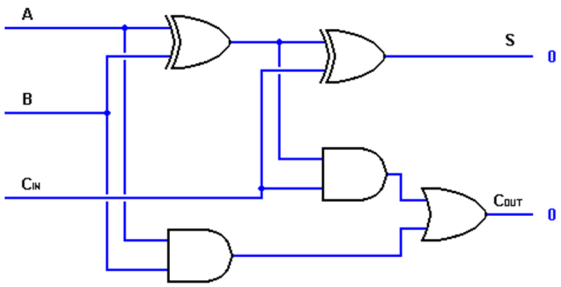

Figure taken from: http://www.circuitstoday.com/half-adder-and-full-adder

Keep in mind that the HA and FA circuit compute only 1 bit of a sum. To compute the sum of two 32-bit two's complement numbers, a circuit is required with 32 FA's hooked together in series, called a ripple carry or carry propagate adder. There are more complex circuits called carry lookahead adders that have better performance than ripple carry.

Ripple Carry or Carry Propagate Adder

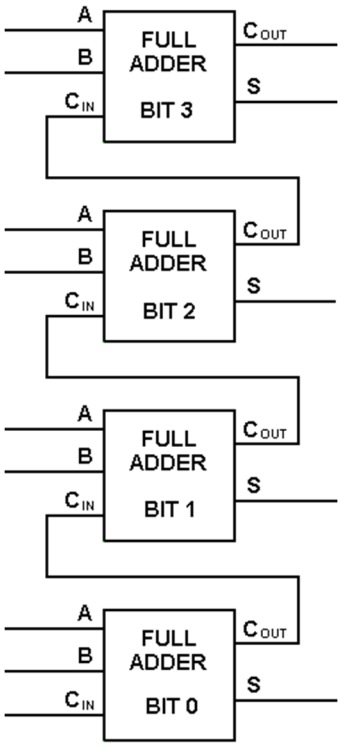

Figure from: http://www.circuitstoday.com/half-adder-and-full-adder

Carry Lookahead Adder

The problem with the carry propagate adder is that a given bit in the sum cannot be completed until the carry from the previous bit of the sum is available. In other words, the sums cannot be computed in parallel, one column must wait for the availability of the carry from the previous column. The greater the number of bits in the sum, the greater the propagation delay, hence overall slow performance.

One fix is the carry lookahead adder which uses additional complex logic to compute carry bits in parallel. See the Wikipedia article for more info at: https://en.wikipedia.org/wiki/Carry-lookahead_adder

Multiplication

Addition is most likely implemented with a carry lookahead adder, itself based on the full adder concept plus additional circuits to calculate carry bits in parallel. To multiply two numbers, we could do repeated addition, but a faster method is Booth's Algorithm. But before we study Booth's, let's first look at some obvious ways to implement multiplication that are easier to understand. Note the maximum size of the numbers involved when doing binary arithmetic. Addition of two n-bit unsigned integers will at most produce a sum that is an n+1 bit integer.

Example:
1111 (15) + 1111 (15) = 11110 (30)

But if we multiply two n-bit unsigned integers, the product will contain up to 2n bits.

Example:
1111 (15) X 1100 (12) = 10110100 (180)

The sum-of-partial-products approach that most people use when multiplying with pencil and paper works in binary just the same way that it does in base 10.

```
      1111
X     1100
_____________
      0000        0 X 1111 (partial product)
     0000         0 X 1111 (partial product)
    1111          1 X 1111 (partial product)
   1111           1 X 1111 (partial product)
_____________
  10110100        sum of partial products = 128+32+16+4 = 180
```

This approach will work, although it may have uninspiring performance. A bigger problem is how to adapt it to 2's complement numbers with both positive and negative inputs. One approach would be to first convert both inputs to positive numbers, multiply them, then convert to a negative result if needed:

1 Let Ax be the absolute value of A
2 Let Bx be the absolute value of B
3 Use the well-known sum-of-products multiplication algorithm that you might use with pencil and paper
4 If A and B are the same sign, you're done.
5 If A and B are different signs, take the two's complement of the product.

Example:

Multiply -15 X 12 = -180

To represent -15 as a tc integer, we need 5 bits.
The product of two 5-bit ints will be up to 10 bits.

As a positive number in 5 bits, 15 = 01111
In 5 bits 12 = 01100

```
     01111
X    01100
___________
     00000        0 X 1111 (partial product)
    00000         0 X 1111 (partial product)
   01111          1 X 1111 (partial product)
  01111           1 X 1111 (partial product)

_____________
0010110100        sum of partial products = 128+32+16+4 = 180
```

When performing arithmetic on tc integers, it's important to get the leftmost bit in the result correct: 0 if the result is positive, 1 if the result is negative. For this version of the algorithm, we convert all arguments to positive numbers to get a positive product. But then we have to adjust the answer if the result requires a negative number. In this case one input is positive, one input is negative, so the result must be negative. To negate the result, we convert the 10 bit result to its two's complement:

```
  0010110100      original

  1101001011      1's complement
+ 0000000001
_______________
  1101001100      2's complement = -180

Check:

  0010110100      +180
+ 1101001100      -180

_____________
  0000000000      0
```

When adding two tc ints, any carry out of the left most column is ignored.

Booth's Algorithm
This is a more realistic algorithm for implementing 2's comp integer multiplication efficiently in hardware. To help understand how it works, keep in mind the following facts about 2's comp arithmetic:

- Multiplying by 2 is the same as shifting left one position
- Dividing by 2 (integer divide) is the same as shifting right one position. But if the number is tc, then the new digit shifted in on the left must match the sign of the original number. If the leftmost bit was a 0, then the new bit should also be a 0. If the leftmost bit was a 1, then the new bit shifted in should also be a 1. This operation is also known as sign extension.
- Long sequences of 1's can be represented by a subtraction of a power of 2 on the right end and an addition of a power of two on the left end.
- Adding two n-bit 2's comp ints produces a sum that can occupy up to n+1 bits. Multiplying two n-bit ints produces a product that can occupy up to 2n bits. In other words, the output can be up to twice as long as the two inputs.

If you look at what you did to multiply using the sum of products, whenever there was a 1 in the second argument, you added a shifted version of the first argument, and whenever there was a 0 in the second argument, you did nothing. Booth slightly modifies this approach. As you read the 2nd argument from right to left:

- whenever you see a transition from 0 to 1, you **subtract** a shifted version of the 1st argument
- whenever you see a transition from 1 to 0, you **add** a shifted version of the 1st argument.

So Booth's algorithm does no work when the 2nd argument is in the middle of a long string of 0s **or** a long string of 1s. Work needs to be done only when transitioning from 0 to 1 or 1 to 0. However, it does require you to do both addition and subtraction of partial products.

Worked Example
First, let's use Booth's algorithm to show how it works in general using a mixture of base 2 and base 10. Then we'll do an example that has been optimized for hardware implementation.

Multiply 13 X 460 = 5980

460 has been chosen because its value in binary is 0111001100 with several sequences of multiple 0s and multiple 1s. Let's convert this string back to base 10 using the shortcut discussed earlier.

0111001100
- 0 ➔ 1 transition at position 2
- 1 ➔ 0 transition at position 4
- 0 ➔ 1 transition at position 6
- 1 ➔ 0 transition at position 9

So value = $2^9 - 2^6 + 2^4 - 2^2 = 512 - 64 + 16 - 4 = 460$

Now imagine using this value as the multiplier in binary

13
X 0111001100

- We initialize the result to 0
- The algorithm proceeds by stepping along the multiplier one bit at a time from right to left.
- Each step we shift the multiplicand one bit to the left (multiply it by two).
- We keep track of the bit transitions in the multiplier and do the following at each step
  0 ➔ 1: subtract the multiplicand from the result
  1 ➔ 0: add the multiplicand to the result
  0 ➔ 0: do nothing
  1 ➔ 1: do nothing

Bit	Multiplicand	Transition	Action	Result
0	13 X 2^0	00	--	0
1	13 X 2^1	00	--	0
2	13 X 2^2	10	Sub Mult	0 – 13 X 2^2 = –52
3	13 X 2^3	11	--	–52
4	13 X 2^4	01	Add Mult	–52 + 13 X 2^4 = –52 + 208 = 156
5	13 X 2^5	00	--	156
6	13 X 2^6	10	Sub Mult	156 – 13 X 2^6 = 676
7	13 X 2^7	11	--	–676
8	13 X 2^8	11	--	–676
9	13 X 2^9	01	Add Mult	–676 + 13 X 2^9 = 5980 = 13 X (0 – 2^2 + 2^4 – 2^6 + 2^9) = 13 X 460

Booth In Binary
Now we can show the algorithm in more detail in binary. Since we will need both the multiplier and its two's complement, we'll maintain both of these values as we shift.

Multiplicand m = … 01101
Multiplicand –m = … 10011

In 2's complement addition, you must sign extend arguments correctly on the left. When evaluating the result, you may have to discard the final carry based on the number of binary places.

In each step, the + and – versions of the multiplicand are multiplied by 2 (shifted left).

Bit	Multiplicand	Tran	Action	Result
0	01101 10011	00	--	0
1	011010 100110	00	--	0
2	0110100 1001100	10	Sub Mult	0000000 (result) 1001100 (-m) ---------- 1001100 (-52)
3	01101000 10011000	11	--	1001100 (-52)
4	011010000 100110000	01	Add Mult	111001100 (result) 011010000 (+m) ---------- 010011100 (156)
5	0110100000 1001100000	00	--	010011100 (156)
6	01101000000 10011000000	10	Sub Mult	00010011100 (result) 10011000000 (-m) ---------- 10101011100 (-676)
7	011010000000 100110000000	11	--	10101011100 (-676)
8	0110100000000 1001100000000	11	--	10101011100 (-676)
9	01101000000000 10011000000000	01	Add Mult	11110101011100 (result) 01101000000000 (+m) ---------- 01011101011100 (5980)