1 Introduction

Boolean algebra and set theory were both developed in the 19^{th} century as approaches to formalizing mathematics. It was only one hundred years later that these foundations were used in computers and computing (hence the book called *Engines of Logic* (**book:engines-of-logic**)).

These notes discuss the basics of and Boolean algebra and set theory, including definitions and notation.

2 Set theory

A **set** is a well-defined collection of elements, with no order and no repetition, that can be be finite or infinite. The empty set ($\emptyset$) is a set with zero elements.

$$\{a, b, c\} = \{b, c, a\}$$
$$\{1, 47, 18181\} = \{\text{positive divisors of } 854,507\}$$
$$\emptyset = \{\}$$

There are three ways to enumerate the elements of a set: roster, descriptive, or set-builder notation. In the following examples, the roster (simply the list of the elements) is on the right and an example of descriptive followed by an example of set-builder notation is on the left of the equals sign.

$$\{\text{letters in 'mathematics'}\} = \{m, a, t, h, e, i, c, s\}$$
$$= \{a, c, e, h, i, m, s, t\}$$
$$\{x : x \text{ is a positive integer } < 5\} = \{1, 2, 3, 4\}$$

2.1 Vocabulary

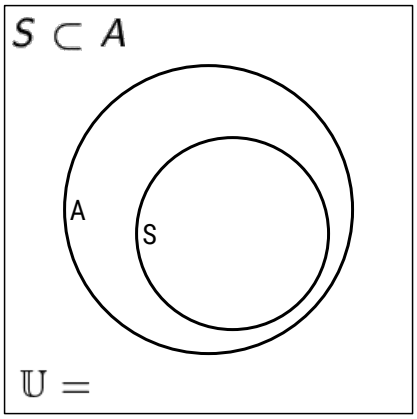
element The symbol '$\in$' means 'is an element of,' 'is a member of,' 'belongs to,' 'is in.'

equality Two sets are equal if they have exactly the same elements.

universal set The symbol '$\mathbb{U}$' represents the set that contains all the elements of all sets of interest.

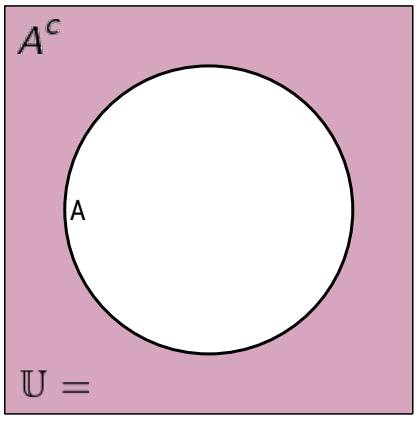
cardinality The number of elements in a set. The cardinality of A is denoted $|A|$. The cardinality of the set of all subsets of $A = 2^{|A|}$.

subsets The symbol '$\subset$' means 'is a subset of,' 'is contained in.' A *proper subset* has zero or more elements of the set, but not the entire set. That is, IF: $S \subset A$, THEN: $|S| < |A|$. An *improper subset*, denoted '$\subseteq$,' can include the entire set, so $A \subseteq A$ is true.

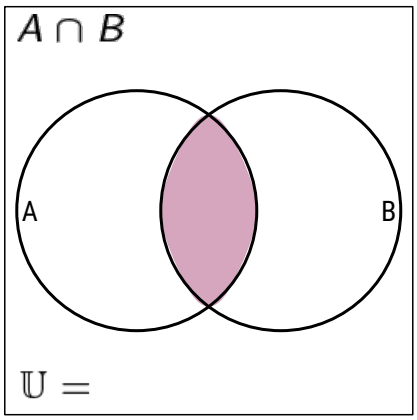


complement Denoted '$\overline{A}$,' 'A',' 'A^c,' '$a \notin A$' and is defined: 'When all sets under consideration are… subsets of a given set $\mathbb{U}$, the *absolute* complement of A is the set of elements in $\mathbb{U}$ but not in A. The *relative* complement of A with respect to a set B, also termed the difference of sets A and B, written $B \setminus A$, is the

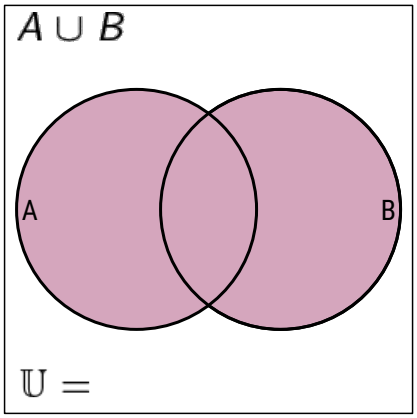
set of elements in B but not in A.' (**wiki:complement-set-theory**) (Or, $B \setminus A = \{x \in B \mid x \notin A\}$.)



intersection The *conjunction* or intersection of two sets A and B is the set of elements in both A and B and is denoted '$A \cap B$.'



union The *disjunction* or union of two sets A and B is the set of elements in at least one of A or B and is denoted '$A \cup B$.'



2.2 Paradox

So-called *naïve set theory* (**wiki:naive-set-theory**) (discussed here) can lead to paradoxes. The most common is *Russell's paradox* (**wiki:russells-paradox**), which states, 'Let R be the set of all sets that are not members of themselves. If R is not a member of itself, then its definition dictates that it must contain itself, and if it contains itself, then it contradicts its own definition as the set of all sets that are not members of themselves.'

Another way to think of it is, the *Barber paradox* (**wiki:barber-paradox**): The barber is the "...one who shaves all those, and only those, who do not shave themselves." The question is, does the barber shave himself?

3 Boolean algebra

'**Boolean algebra** (**wiki:boolean-algebra**) was introduced by George Boole in his first book *The Mathematical Analysis of Logic* (**book:mathematical-analysis-of-logic**)...' It is a system of algebra based on two values: *true* and *false* (usually 1 and 0 respectively) and three functions: logical *conjunction* (AND) and logical *disjunction* (OR), and logical *negation* (NOT).

The three logic functions (AND, OR, NOT) are represented by different symbols in Venn diagrams (**wiki:venn-diagram**), Boolean algebra, logic, and the programming languages Python and Java.

Name	Venn	Boolean	Logic	Python	Java
AND	$\cap$	$\bullet$	$\wedge$	`and`	`&&`
OR	$\cup$	$+$	$\vee$	`or`	`||`
NOT	^{c}	$'$	$\neg$	`not`	`!`

3.1 Functions

The logic functions AND and OR are binary infix operations (of two operands where the operator is placed between the operands). The logic function NOT is a unary prefix operation (of one operand where the operator is placed before the operand).

Logic functions can be fully specified by their truth tables as follows:

AND

A	B	$A \bullet B$
0	0	0
0	1	0
1	0	0
1	1	1

'Only `true` if both are `true`'

OR

A	B	$A + B$
0	0	0
0	1	1
1	0	1
1	1	1

'Only `false` if both are `false`'

NOT

A	$A' = \neg A = \overline{A}$
0	1
1	0

[1]To convert Boolean expressions using De Morgan's laws, "change AND to OR / OR to AND, complement the inputs, complement the outputs."

3.2 Identities

In Boolean algebra the algebraic properties of equality (reflexive, symmetric, transitive, and substitution) hold. Boolean algebra also has identities, just as Euclidean algebra. Every Boolean algebra identity has a *dual* — one in the AND form and one in the OR form.

Property	**AND Form**	**OR Form**
Identity	$1 \bullet A = A$	$0 + A = A$
Null	$0 \bullet A = 0$	$1 + A = 1$
Idempotent	$A \bullet A = A$	$A + A = A$
Complement	$A \bullet \overline{A} = 0$	$A + \overline{A} = 1$
Commutative	$A \bullet B = B \bullet A$	$A + B = B + A$
Associative	$(A \bullet B) \bullet C = A \bullet (B \bullet C)$	$(A + B) + C = A + (B + C)$
Distributive	$A + (B \bullet C) = (A + B) \bullet (A + C)$	$A \bullet (B + C) = (A \bullet B) + (A \bullet C)$
Absorption	$A \bullet (A + B) = A$	$A + (A \bullet B) = A$
De Morgan's[1]	$\overline{(A \bullet B)} = \overline{A} + \overline{B}$ $A \bullet B = \overline{(\overline{A} + \overline{B})}$	$\overline{(A + B)} = \overline{A} \bullet \overline{B}$ $A + B = \overline{(\overline{A} \bullet \overline{B})}$
Double Complement	$\overline{\overline{A}} = A$	

3.3 Proofs

Boolean algebra proofs are easier than other types of proofs, because they involve using truth tables. There are a finite number of possible input combinations and, if the truth table for one expression matches the truth table for the other expression, then the assertions are equal!

Prove the *Associative Law for* AND of Boolean algebra:

$$(A \bullet B) \bullet C = A \bullet (B \bullet C)$$

ABC	$A \bullet B$	$(A \bullet B) \bullet C$	$B \bullet C$	$A \bullet (B \bullet C)$
000	0	0	0	0
001	0	0	0	0
010	0	0	0	0
011	0	0	1	0
100	0	0	0	0
101	0	0	0	0
110	1	0	0	0
111	1	1	1	1

Since every output of the $(A \bullet B) \bullet C$ column is equal to every output of the $A \bullet (B \bullet C)$ column for every *possible* combination of inputs, the two statements are equivalent, so the *Associative Law for* AND of Boolean algebra is proven.

Now *you* prove the *Associative Law for* OR of Boolean algebra.

3.3.1 Proof of De Morgan's laws

AB	$A \bullet B$	$\overline{(A \bullet B)}$	$\overline{A}$	$\overline{B}$	$\overline{A} + \overline{B}$
00	0	1	1	1	1
01	0	1	1	0	1
10	0	1	0	1	1
11	1	0	0	0	0

De Morgan's Law — AND Form

AB	$A + B$	$\overline{(A + B)}$	$\overline{A}$	$\overline{B}$	$\overline{A} \bullet \overline{B}$
00	0	1	1	1	1
01	1	0	1	0	0
10	1	0	0	1	0
11	1	0	0	0	0

De Morgan's Law — OR Form

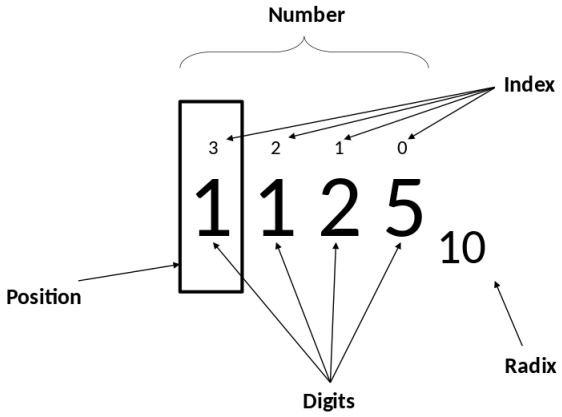
4 Arithmetic

Arithmetic can be carried out using only logic. That is the foundation of calculation in a computer.

The basic unit of information capable of distinguishing two values (*e.g. true* and *false*) is the **bit** (**wiki:bit**) (or *binary digit*) — the answer to one yes-no question.

4.1 Positional number systems

A positional number system (**wiki:positional-notation**) uses a fixed set of symbols (numerals) to represent numbers and their position within the written form to indicate their order of magnitude in the radix or base. In computing there are (typically) three radixes in use: decimal (base 10), binary (base 2), and hexadecimal (base 16).



For example, the decimal number 1125:

$$1125_{10} = 1 \times 10^3 + 1 \times 10^2 + 2 \times 10^1 + 5 \times 10^0$$

The *numeral* 1 in the 3rd indexed position is the same as the *numeral* 1 in the 2nd indexed position, but they represent different orders of magnitude — 10^3 and 10^2, respectively.

The number 71_{10} is represented in the three number bases (10, 2, 16) as:

$$71_{10} = 7 \times 10^1 + 1 \times 10^0$$
$$0100\,0111_2 = 1 \times 2^6 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$$
$$47_{16} = 4 \times 16^1 + 7 \times 16^0$$

It is important to realize that all three numeral forms (decimal, binary, hexadecimal) represent *the same number*: the number we refer to in every-day language as 'seventy-one.'

Expressing numbers in different bases requires:

- In each radix there are *radix* numerals (symbols).
  - Binary (base 2) has 2: 0, 1

- Decimal (base 10) has 10: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
- Hexadecimal (base 16) has 16: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

- Zero and one in *any* radix are equal to themselves.
- In radix R, $10_R = R$. (10 is the radix.)

Examples of representing numbers in different radixes (with their positional orders of magnitude) are:

10^2	10^1	10^0		2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0		16^1	16^0
		8_{10}	=	0	0	0	0	1	0	0	0_2	=		8_{16}
	1	3_{10}	=	0	0	0	0	1	1	0	1_2	=		D_{16}
	7	1_{10}	=	0	1	0	0	0	1	1	1_2	=	4	7_{16}
1	9	2_{10}	=	1	1	0	0	0	0	0	0_2	=	C	0_{16}
2	5	5_{10}	=	1	1	1	1	1	1	1	1_2	=	F	F_{16}

Expressed in binary, 2^N consists of a single 1 in the N^{th} index position, while $2^N - 1$ consists of N 1s in a row.

4.1.1 Conversion

To convert from hexadecimal to decimal, multiply each numeral of the number H by it's appropriate power of 16. For example, for $H = 4D2_{16}$:

$$4D2_{16} = \quad 4 \times 16^2 \; + 13 \times 16^1 \; + 2 \times 16^0 \quad = 1234_{10}$$

To convert from decimal to hexadecimal, divide the number D by the largest power of $16 < D$ — accumulating the floor of the quotient as the hexadecimal representation — continuing with the remainder until dividing by 16^0. For example:

$$1234_{10} \div 16^2 = \boxed{4}\, R210$$
$$210_{10} \div 16^1 = \boxed{13}\, R2$$
$$2_{10} \div 16^0 = \boxed{2}\, R0$$
$$\therefore 1234_{10} = 4D2_{16}$$

4.1.2 Binary *is* hexadecimal

Hexadecimal (base 16) is another form of binary (base 2), because $16 = 2^4$ — all of which are powers of two — so one hex digit is the equivalent of four binary digits (bits).

To convert from binary ⟷ hexadecimal simply break a binary number into nybbles (**wiki:nybble**) (4-bits at a time) and write them as hexadecimal digits ⟷ write the hexadecimal digits as binary nybbles following this pattern:

Decimal	Binary	Hex	Decimal	Binary	Hex
0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	10	1010	*A*
3	0011	3	11	1011	*B*
4	0100	4	12	1100	*C*
5	0101	5	13	1101	*D*
6	0110	6	14	1110	*E*
7	0111	7	15	1111	*F*

4.2 Addition

Let's add $92 + 33 = 125$ and, for the fun of it, perform the operations in all three radixes.

	1												
	9	2_{10}	=	0	1	0	1	1	1	0	0_2	= 5	C_{16}
+	3	3_{10}	=	0	0	1	0	0	0	0	1_2	= 2	1_{16}
1	2	5_{10}	=	0	1	1	1	1	1	0	1_2	= 7	D_{16}

Is there a *logical* operation that could be used to combine each bit? Bitwise OR works in this example.

Now let's add $92 + 57 = 149$ in all three radixes.

	1			1	1	1	1					1	
	9	2_{10}	=	0	1	0	1	1	1	0	0_2	= 5	C_{16}
⊕	5	7_{10}	=	0	0	1	1	1	0	0	1_2	= 3	9_{16}
1	4	9_{10}	=	1	0	0	1	0	1	0	1_2	= 9	5_{16}

[2] To insure that it doesn't matter which order the carry is added, it is necessary to verify that $(A \oplus B) \oplus C = A \oplus (B \oplus C)$, the *Associative Law for* XOR of Boolean algebra.

Is there a logical operation that could be used to combine each bit? Yes! Bitwise three-input XOR works in this example[2].

Here is the truth table for the Σ and C_{out} functions of a (one-bit) full adder (**wiki:adder-electronics**):

ABC_{in}	$A \oplus B \oplus C_{in}$	$A \oplus B$	$(A \oplus B) \bullet C_{in}$	$A \bullet B$	C_{out}
000	0	0	0	0	0
010	1	1	0	0	0
110	0	0	0	1	1
100	1	1	0	0	0
001	1	0	0	0	0
011	0	1	1	0	1
111	1	0	0	1	1
101	0	1	1	0	1

Here are Σ and C_{out} as functions of A, B, and C_{in}:

$$\begin{aligned} \Sigma &= A \oplus B \oplus C_{in} \\ C_{out} &= (A \bullet B) + (A \bullet C_{in}) + (B \bullet C_{in}) \\ &= (A \bullet B) + ((A \oplus B) \bullet C_{in}) \end{aligned}$$

4.2.1 Bitwise operations

For multi-bit cells (called bit strings or binary numerals), logical operations are defined between corresponding bits of the cells. A detailed discussion of bitwise logical operations is available on Wikipedia (**wiki:bitwise-operation**).

4.3 Fixed-point arithmetic

Computers often represent numbers using fixed-point arithmetic (**wiki:fixed-point-arithmetic**), *i.e.* as integers. These number types differ from floating-point[3] (**wiki:floating-point-arithmetic**) numbers in that:

- fixed-point numbers are exact (while floating-point numbers may be approximations of rational or irrational numbers);
- fixed-point numbers have 'signedness' (**wiki:signedness**) and can represent signed or unsigned numbers;
- fixed point numbers represent a finite number of different integers (with n fixed digits in radix R representing R^n different integers).

In Java (**wiki:java-primitive-types**), there are four fixed-point data types: `byte` (8 bits), `short` (16 bits), `int` (32 bits), `long` (64 bits). The examples in this document limit themselves to 8-bit numbers, though the results are the same for any fixed-point data type.

4.3.1 Overflow

In fixed-point arithmetic there will always be overflow (**wiki:integer-overflow**), as in the (8-bit) example of $60_{16} + C0_{16}$.

$$\begin{array}{rrrrrrrrrrrrrrrrr} & & & \boxed{1} & 1 & & & & & & & & & \boxed{1} & & \\ & & 9 & 6_{10} & = & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0_2 & = & 6 & 0_{16} \\ + & 1 & 9 & 2_{10} & = & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0_2 & = & C & 0_{16} \\ \hline & & 3 & 2_{10} & = & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0_2 & = & 2 & 0_{16} \end{array}$$

Clearly, $96_{10} + 192_{10} = 288_{10} \neq 32_{10}$, but 288_{10} cannot be represented in 8 bits, because 8 bits can only represent 2^8 different numbers. The carry (shown in red in the ninth bit) has nowhere to go. That carry is known as an 'overflow.'

Every fixed-point arithmetic operation must consider overflow and restrict itself to operations whose results are within the representation of the integer.

4.4 Subtraction

Making the connection between logic and arithmetic, addition is implemented by the XOR Boolean logic function. Consider the result and overflow implications of the (8-bit) subtraction example $60_{16} - 20_{16}$.

$$\begin{array}{rrrrrrrrrrrrrrr} & 9 & 6_{10} & = & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0_2 & = & 6 & 0_{16} \\ - & 3 & 2_{10} & = & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0_2 & = & 2 & 0_{16} \\ \hline & 6 & 4_{10} & = & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0_2 & = & 4 & 0_{16} \end{array}$$

The result is correct, beacuse $96_{10} \geq 32_{10}$. Now consider the (8-bit) example of $20_{16} - 60_{16}$.

[3] For further information, see Appendix A.

$$\begin{array}{rrcccccccccccc}
 & & & & 1 & & & & & & & & & \\
 & & & \not{1} & \not{1}\not{0} & 10 & & & & & & \not{1} & 12 \\
 & 3 & 2_{10} = & & \not{0} & \not{0} & 1 & 0 & 0 & 0 & 0 & 0_2 = & & \not{2}\ 0_{16} \\
- & 9 & 6_{10} = & & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0_2 = & & 6\ 0_{16} \\
\hline
 & 1 & 9\ 2_{10} = & & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0_2 = & & C\ 0_{16}
\end{array}$$

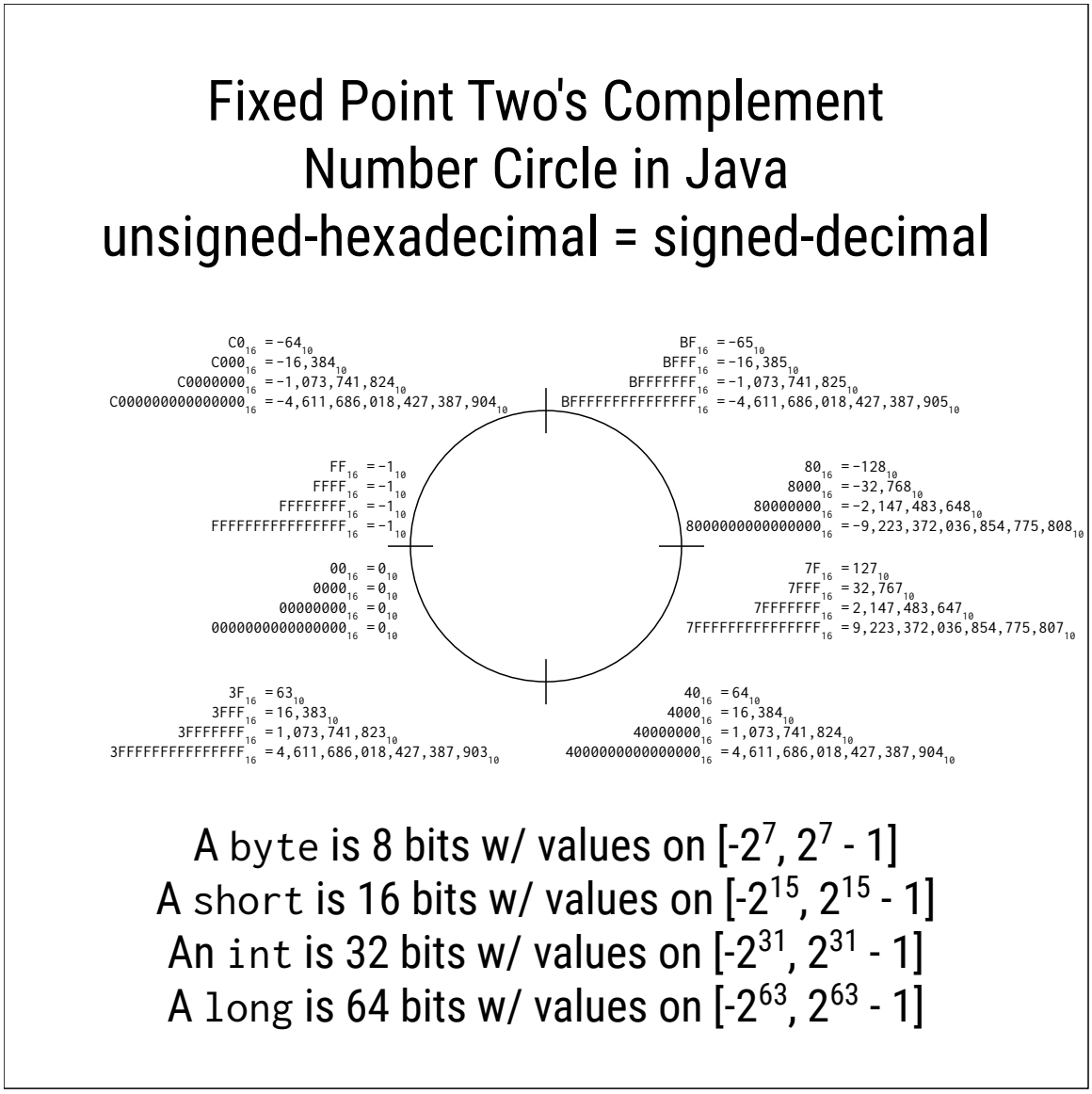
Clearly, $32_{10} - 96_{10} = -64_{10} \neq 192_{10}$, but -64_{10} cannot be represented by non-negative numbers. The borrow (shown in red in the ninth bit) — sometimes called an integer 'underflow' — is related to integer overflow in addition. It is a bit that isn't there, but the result is incorrect without it (because $120_{16} - 60_{16}$ is, in fact, $C0_{16}$).

There *is* a way to interpret these results so that they make sense.

4.4.1 Two's complement

One way to implement signed arithmetic is with *two's complement* (**wiki:twos-complement**). In n-bit fixed-point arithmetic the integer number line behaves as if it were a number circle, mod 2^n. That is, all integers are non-negative and the next number after $2^n - 1$ is *zero*.

The two's complement system breaks the number circle at a different point — between $2^{n-1} - 1$ and 2^{n-1}, treating half the integers on the number circle as negative and the other half as non-negative. Any n-bit integers on the *unsigned* interval $[0, 2^n - 1] \geq 2^{n-1}$ are negative and the others on $[0, 2^{n-1} - 1]$ are non-negative, as shown below.



Java two's complement number circle

The two's complement of an n-bit binary number m, defined as $2^n - m$, can be calculated as one more than the ones' complement (**wiki:ones-complement**) (the bit-wise complement of the binary representation) of m, ignoring overflow. Calculating that $+1$ in the binary representation of m by hand can be tricky, as there can be up to m carries. Consequently, the preferred algorithm for calculating the two's-complement of an n-bit binary number m is:

> . . . start at the least significant bit (LSB), and copy all the zeros, working from LSB toward the most significant bit (MSB) until the first 1 is reached; then copy that 1, and [complement] all the remaining bits (Leave the MSB as a 1 if the initial number was in sign-and-magnitude representation). This shortcut allows a person to convert a number to its two's complement without first forming its ones' complement. — Wikipedia

By interpreting n-bit binary numbers as their two's-complement representation, subtraction is implemented as unsigned addition of the two's-complement of the subtrahend to the minuend, ignoring overflow. Revisiting the examples from section 4.4 and interpreting the numbers as two's complement, the operations between double lines are exactly that form of subtraction.

Here is the example of $(+96_{10}) - (+32_{10}) = (+60_{16}) - (+20_{16})$.

$$
\begin{array}{rrcr}
(+96_{10}) = (& +0\,1\,1\,0\,0\,0\,0\,0_2 &) = & (+60_{16}) \\
-(+32_{10}) = (& +0\,0\,1\,0\,0\,0\,0\,0_2 &) = & (+20_{16}) \\
\hline
 & 1\,1\,1 & & \\
 & 0\,1\,1\,0\,0\,0\,0\,0_2 & = & 60_{16} \\
+ & 1\,1\,1\,0\,0\,0\,0\,0_2 & = & E0_{16} \\
\hline
 & 0\,1\,0\,0\,0\,0\,0\,0_2 & = & 40_{16} \\
\hline
(+64_{10}) = (& +0\,1\,0\,0\,0\,0\,0\,0_2 &) = & (+40_{16})
\end{array}
$$

Here is the example of $(+32_{10}) - (+96_{10}) = (+20_{16}) - (+60_{16})$.

$$
\begin{array}{rrcr}
(+32_{10}) = (& +0\,0\,1\,0\,0\,0\,0\,0_2 &) = & (+20_{16}) \\
-(+96_{10}) = (& +0\,1\,1\,0\,0\,0\,0\,0_2 &) = & (+60_{16}) \\
\hline
 & 1 & & \\
 & 0\,0\,1\,0\,0\,0\,0\,0_2 & = & 20_{16} \\
+ & 1\,0\,1\,0\,0\,0\,0\,0_2 & = & A0_{16} \\
\hline
 & 1\,1\,0\,0\,0\,0\,0\,0_2 & = & C0_{16} \\
\hline
(-64_{10}) = (& -0\,1\,0\,0\,0\,0\,0\,0_2 &) = & (-40_{16})
\end{array}
$$

4.5 Multiplication

Multiplication is repeated addition — as indicated by the English reading of 3×4 as, 'three times four;' that is add four to itself three times. The standard multiplication algorithm is a shortcut to repeated addition where, to obtain a product, '. . . multiply the multiplicand by each digit of the multiplier and then add up all the properly shifted results.' (**wiki:multiplication-algorithm**) Binary multiplication of $m \times n$ is easy, as there are only two values to multiply m by in each digit of n: $0 \times m = 0$ and $1 \times m = m$. For example:

$7_{10} \times 6_{10} = 42_{10}$

$$\begin{array}{rcccccccc}
 & & & & & 0 & 1 & 1 & 1_2 \\
\times & & & & & 0 & 1 & 1 & 0_2 \\
\hline
 & & & & & 0 & 0 & 0 & 0 \\
 & & & & 0 & 1 & 1 & 1 & \\
 & & & 0 & 1 & 1 & 1 & & \\
+ & & 0 & 0 & 0 & 0 & & & \\
\hline
 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0_2
\end{array}$$

$47_{10} \times 17_{10} = 799_{10}$

$$\begin{array}{rcccccccccc}
 & & & & & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1_2 \\
\times & & & & & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1_2 \\
\hline
 & & & & & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \\
 & & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \\
 & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & \\
 & & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & & & \\
 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & & \\
+ & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1_2
\end{array}$$

$127_{10} \times 127_{10} = 16129_{10}$

$$\begin{array}{rcccccccccccccccc}
 & & & & & & & & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1_2 \\
\times & & & & & & & & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1_2 \\
\hline
 & & & & & & & & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
 & & & & & & & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \\
 & & & & & & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & & \\
 & & & & & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & & & \\
 & & & & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & & & & \\
 & & & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & & & & & \\
 & & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & & & & & & \\
 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & & & & & & & \\
+ & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & & & & & \\
\hline
0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1_2
\end{array}$$

- These examples exemplify products of two non-negative two's complement numbers. It is (of course) possible to multiply negative numbers, but it is more involved.[4]
- The product of two n-bit numbers will, in general, be a $2n$-bit number.
- If this algorithm is limited to non-negative n-bit two's complement numbers, the maximum possible product is $2^{2(n-1)} - (2^n - 1)$, though the maximum (positive) value of a $2n$-bit two's complement number is $2^{2n} - 1$.

4.6 Division

Division is repeated subtraction, as described in the Euclidean algorithm in *Euclid's Elements* Book VII Proposition I (**book:elements-one-volume**).

The long division algorithm is a shortcut to repeated subtraction where, to obtain a quotient and remain-

[4] To multiply negative numbers, either add additional shifted values of the multiplicand due to sign extension of the multiplier, ignoring overflow, or multiply absolute values and adjust the sign of the result.

NOTE

Include example...

der, shift '…gradually from the left to the right end of the dividend, subtracting the largest possible multiple of the divisor at each stage; the multiples become the digits of the quotient, and the final difference is the remainder.' (**wiki:division-algorithm**) Binary long division can be implemented as in the following examples (the underlined bits correspond in each row):

$7_{10} \div 3_{10} = 2_{10}\text{R}1_{10}$ (Note: $-3_{10} = 1101_2$)

N	R	Q	K/R
$\underline{0}\ 1\ 1\ 1$	$0\ 0\ 0\ \underline{0}$		
	$+\ 1\ 1\ 0\ 1$		
	$\underline{1}\ 1\ 0\ 1$	$\underline{0}$	K
$0\ \underline{1}\ 1\ 1$	$0\ 0\ 0\ \underline{1}$		
	$+\ 1\ 1\ 0\ 1$		
	$\underline{1}\ 1\ 1\ 0$	$0\ \underline{0}$	K
$0\ 1\ \underline{1}\ 1$	$0\ 0\ 1\ \underline{1}$		
	$+\ 1\ 1\ 0\ 1$		
	$\underline{0}\ 0\ 0\ 0$	$0\ 0\ \underline{1}$	R
$0\ 1\ 1\ \underline{1}$	$0\ 0\ 0\ \underline{1}$		
	$+\ 1\ 1\ 0\ 1$		
	$\underline{1}\ 1\ 1\ 0$	$0\ 0\ 1\ \underline{0}$	K
	$0\ 0\ 0\ 1$	$0\ 0\ 1\ 0$	

$6_{10} \div 2_{10} = 3_{10}\text{R}0_{10}$ (Note: $-2_{10} = 1110_2$)

N	R	Q	K/R
$\underline{0}\ 1\ 1\ 0$	$0\ 0\ 0\ \underline{0}$		
	$+\ 1\ 1\ 1\ 0$		
	$\underline{1}\ 1\ 1\ 0$	$\underline{0}$	K
$0\ \underline{1}\ 1\ 0$	$0\ 0\ 0\ \underline{1}$		
	$+\ 1\ 1\ 1\ 0$		
	$\underline{1}\ 1\ 1\ 1$	$0\ \underline{0}$	K
$0\ 1\ \underline{1}\ 0$	$0\ 0\ 1\ \underline{1}$		
	$+\ 1\ 1\ 1\ 0$		
	$\underline{0}\ 0\ 0\ 1$	$0\ 0\ \underline{1}$	R
$0\ 1\ 1\ \underline{0}$	$0\ 0\ 1\ \underline{0}$		
	$+\ 1\ 1\ 1\ 0$		
	$\underline{0}\ 0\ 0\ 0$	$0\ 0\ 1\ \underline{1}$	R
	$0\ 0\ 0\ 0$	$0\ 0\ 1\ 1$	

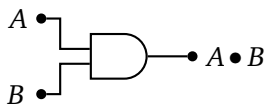
$127_{10} \div 10_{10} = 12_{10}\text{R}7_{10}$ (Note: $-10_{10} = 11110110_2$)

N	R	Q	K/R
$\underline{0}1111111$	$0000000\underline{0}$		
	$+11110110$		
	$\underline{1}1111010$	$\underline{0}$	K
$0\underline{1}111111$	$0000000\underline{1}$		
	$+11110110$		
	$\underline{1}1111011$	$0\underline{0}$	K
$01\underline{1}11111$	$0000001\underline{1}$		
	$+11110110$		
	$\underline{1}1111001$	$00\underline{0}$	K
$011\underline{1}1111$	$0000011\underline{1}$		
	$+11110110$		
	$\underline{1}1111101$	$000\underline{0}$	K
$0111\underline{1}111$	$0000111\underline{1}$		
	$+11110110$		
	$\underline{0}0000101$	$0000\underline{1}$	R
$01111\underline{1}11$	$0000101\underline{1}$		
	$+11110110$		
	$\underline{0}0000001$	$0000\underline{1}1$	R
$011111\underline{1}1$	$0000001\underline{1}$		
	$+11110110$		
	$\underline{1}1111001$	$0000\underline{1}10$	K
$0111111\underline{1}$	$0000011\underline{1}$		
	$+11110110$		
	$\underline{1}1111101$	$0000\underline{1}100$	K
	00000111	00001100	

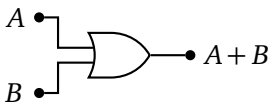
- In each step, shift bits of the dividend N one at a time into the least-significant bit of remainder R and subtract the divisor D (shown here as adding its opposite).
- If the result of the subtraction is negative ($R < D$), the corresponding bit of the quotient Q is zero and the remainder R is kept for the next bit — as signified by K in the rightmost column.
- If the result of the subtraction is non-negative ($R \geq D$), the corresponding bit of the quotient Q is one and the remainder R is replaced by $R - D$ (already calculated as a result of the test) for the next bit — as signified by R in the rightmost column.

5 Circuits

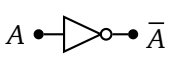
To implement logic functions in actual *hardware*, designers use integrated circuits called *gates* (**wiki:logic-gate**) and two voltage levels to represent '0' and '1.' The basics of these implement binary (two-input) logic functions (*e.g.* AND, OR, XOR) and the unary (one-input) logic function NOT. Symbols used on schematic diagrams of logic circuits (where lines represent electrical connections — wires) are:



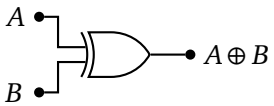
AND Gate



OR Gate



NOT Gate

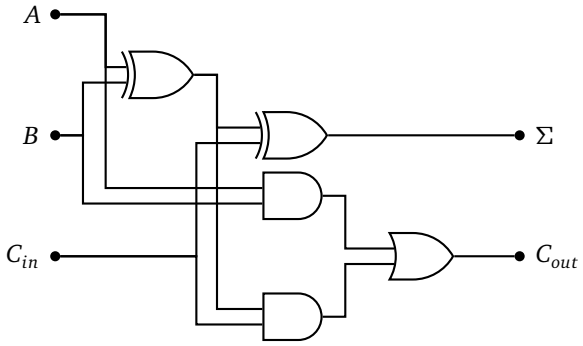


XOR Gate

The equations for a (one-bit) full adder (sum Σ and carry C_{out} as functions of two addend bits A and B and the carry from the next rightmost bit C_{in}) are:

$$\Sigma = \mathrm{A} \oplus \mathrm{B} \oplus \mathrm{C_{in}}$$
$$\mathrm{C_{out}} = (\mathrm{A} \bullet \mathrm{B}) + ((\mathrm{A} \oplus \mathrm{B}) \bullet \mathrm{C_{in}})$$

The corresponding circuit for a (one-bit) full adder is:



Full Adder

5.0.1 Functional completeness

Boolean logic is functionally complete using AND, OR, and NOT, *or* NAND by itself, *or* NOR by itself. Proof of functional completeness requires showing that all 16 two-input, one-output truth tables can be implemented using the specified logic functions.

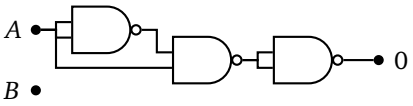
The proof of functional completeness for AND, OR, and NOT logic is:

$$0 \leftrightarrow A \bullet \overline{A}$$
$$\overline{A+B} \leftrightarrow \overline{A+B}$$
$$A < B \leftrightarrow \overline{A} \bullet B$$
$$\overline{A} \leftrightarrow \overline{A}$$
$$A > B \leftrightarrow A \bullet \overline{B}$$
$$\overline{B} \leftrightarrow \overline{B}$$
$$A \neq B \leftrightarrow \left(A \bullet \overline{B}\right) + \left(\overline{A} \bullet B\right)$$
$$\overline{A \bullet B} \leftrightarrow \overline{A \bullet B}$$
$$A \bullet B \leftrightarrow A \bullet B$$
$$A = B \leftrightarrow \overline{\left(A \bullet \overline{B}\right) + \left(\overline{A} \bullet B\right)}$$
$$B \leftrightarrow B$$
$$A \leq B \leftrightarrow \overline{A \bullet \overline{B}}$$
$$A \leftrightarrow A$$
$$A \geq B \leftrightarrow \overline{\overline{A} \bullet B}$$
$$A + B \leftrightarrow A + B$$
$$1 \leftrightarrow A + \overline{A}$$

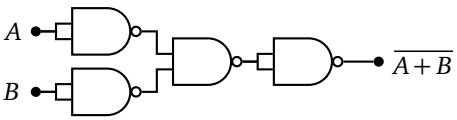
5.0.2 Functional completeness redux

The proof of functional completeness for NAND logic is below (the proof for NOR logic is the dual of this proof):

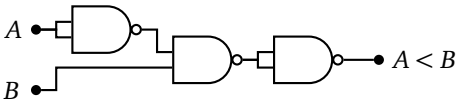
AB	0
00	0
01	0
10	0
11	0



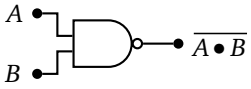
AB	$\overline{A+B}$
00	1
01	0
10	0
11	0



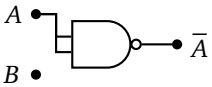
AB	$A < B$
00	0
01	1
10	0
11	0



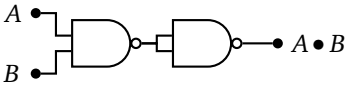
AB	$\overline{A \bullet B}$
00	1
01	1
10	1
11	0



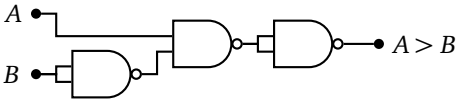
AB	$\overline{A}$
00	1
01	1
10	0
11	0



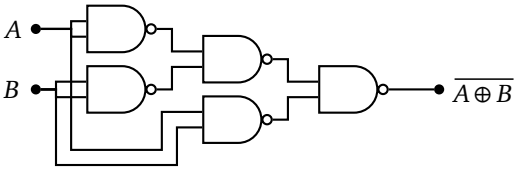
AB	$A \bullet B$
00	0
01	0
10	0
11	1



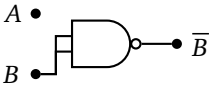
AB	$A > B$
00	0
01	0
10	1
11	0



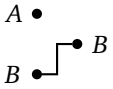
AB	$A = B$ $\overline{A \oplus B}$
00	1
01	0
10	0
11	1



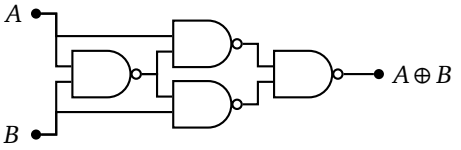
AB	$\overline{B}$
00	1
01	0
10	1
11	0



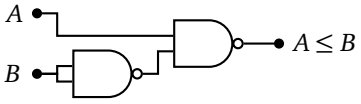
AB	B
00	0
01	1
10	0
11	1



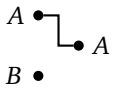
AB	$A \neq B$ $A \oplus B$
00	0
01	1
10	1
11	0



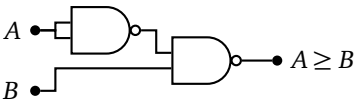
AB	$A \leq B$
00	1
01	1
10	0
11	1



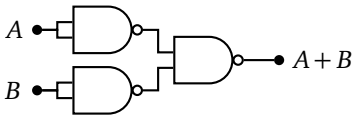
AB	A
00	0
01	0
10	1
11	1



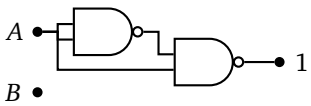
AB	$A \geq B$
00	1
01	0
10	1
11	1



AB	$A+B$
00	0
01	1
10	1
11	1



AB	1
00	1
01	1
10	1
11	1



5.0.3 NAND logic half adder

As we have already seen, addition (using a half adder for each bit) is implement with the Boolean logic function XOR. Implementing XOR $\left(\left(A \cdot \overline{B}\right) + \left(\overline{A} \cdot B\right)\right)$ using only NAND gates has a curious aspect. Each of the inputs A and B are NANDed with $\overline{(A \cdot B)}$, rather than ANDing each with the complement of the other. This approach actually *saves* gates (with one fewer inverter) while implementing the XOR logic as follows:

$$\begin{aligned}
A \oplus B &= \overline{\left(\overline{\left(\overline{(A \cdot B)} \cdot A\right)} \cdot \overline{\left(\overline{(A \cdot B)} \cdot B\right)}\right)} \\
&= \overline{\left(\overline{\left(\overline{\left(\overline{A} + \overline{B}\right) \cdot A}\right)} \cdot \overline{\left(\overline{\left(\overline{A} + \overline{B}\right) \cdot B}\right)}\right)} \\
&= \overline{\left(\overline{\left(\left(A \cdot \overline{A}\right) + \left(A \cdot \overline{B}\right)\right)} \cdot \overline{\left(\left(\overline{A} \cdot B\right) + \left(B \cdot \overline{B}\right)\right)}\right)} \\
&= \overline{\left(\overline{\left(0 + \left(A \cdot \overline{B}\right)\right)} \cdot \overline{\left(\left(\overline{A} \cdot B\right) + 0\right)}\right)} \\
&= \overline{\left(\left(A \cdot \overline{B}\right) \cdot \left(\overline{A} \cdot B\right)\right)} \\
&= \left(A \cdot \overline{B}\right) + \left(\overline{A} \cdot B\right)
\end{aligned}$$

Appendices

A Floating Point

NOTE

More tk on floating point arithmetic.

The main standard for floating-point arithmetic is IEEE 754 (**wiki:ieee-754**). This group of standards for various bit widths (*e.g.* 16, 32, 64, 80, 128, 256-bit, *etc.*) represent numbers in scientific notation with a sign, significand, and exponent.

- hidden '1'

- special numbers

- rounding https://en.wikipedia.org/wiki/Floating-point_arithmetic#Accuracy_problems

- operations

B Links

- http://en.wikipedia.org/wiki/Naive_set_theory Naïve set theory
- https://www.andrews.edu/~calkins/math/

webtexts/numball.pdf Numbers... chapter 1 *All about Sets*

- https://www.cs.odu.edu/~toida/nerzic/content/set/basics.html Basics of sets
- http://mathforum.org/library/topics/set_theory/ The Math Forum @ Drexel: set theory
- http://jeff560.tripod.com/set.html Earliest uses of symbols of set theory and logic
- http://www.combinatorics.org/Surveys/ds5/VennEJC.html Venn diagrams
- https://www.geeksforgeeks.org/digital-logic-functionality-completeness/ GeeksforGeeks: functional completeness
- https://mathcs.clarku.edu/~djoyce/java/elements/elements.html *Euclid's Elements*
- https://en.wikibooks.org/wiki/Java_Programming/Primitive_Types Java primitive types
- https://www.h-schmidt.net/FloatConverter/IEEE754.html IEEE 754 Converter