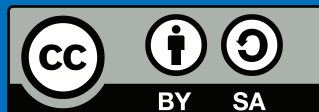
A Practical Introduction to Computer Architecture

Daniel Page ⟨dan@phoo.org⟩

git # 5ac601b @ 2018-07-13



MATHEMATICAL PRELIMINARIES

In Mathematics you don't understand things. You just get used to them.

– von Neumann

The goal of this Chapter is to provide a fairly comprehensive overview of theory that underpins the rest of the book. At first glance the content may seem a little dry, and is often excluded in other similar books. It seems clear, however, that without a solid understanding of said theory, using the constituent topics to solve practical problems will be much harder.

The topics covered all relate to the field of discrete Mathematics; they include propositional logic, sets and functions, Boolean algebra and number systems. These four topics combine to produce a basis for formal methods to describe, manipulate and implement digital systems. Readers with a background in Mathematics or Computer Science might skip this Chapter and use it simply for reference; those approaching it from some other background would be advised to read the material in more detail.

1 Propositional logic

Definition 0.1. *A* **proposition** *is a statement whose meaning, termed the* **truth value***, is either* **true** *or* **false** *(less formally, we say the statement is true if it has a truth value of* **true** *and false if it has a truth value of* **false***). A given proposition can involve one or more* **variables***; only when concrete values are* **assigned** *to the variables can the meaning of a proposition be* **evaluated***.*

In part because we use them naturally in language, it almost seems too formal to define what a proposition *is*. However, by doing so we can start to use them as a building block to describe what propositional logic is and how it works. This is best explained step-by-step by example:

Example 0.1. The statement

"the temperature is 90°C"

is a proposition since it is definitely either true or false. When we take a proposition and decide *whether* it is true or false, we say we have evaluated it. However, there are clearly a lot of statements that are *not* propositions because they do not state any proposal. For example,

"turn off the heat"

is a command or request of some kind, it does not evaluate to a truth value. Propositions must also be well defined in the sense that they are definitely either true or false, i.e., there are no "grey areas" in between. The statement

"90°C is too hot"

is *not* a proposition, because it could be true *or* false depending on the context: 90°C is probably too hot for body temperature, but not for a cup of coffee. Finally, some statements *seem* to be propositions but cannot be evaluated because they are paradoxical: a famous example is the so-called liar paradox, usually attributed to the Greek philosopher Eubulides, who stated it as

"a man says that he is lying, is what he says true or false?"

although a clearer version is the more commonly referenced

"this statement is false" .

If the man is telling the truth, everything he says must be true which means he is lying and hence everything he says is false. Conversely, if the man is lying everything he says is false so he cannot be lying (because he said that he was). In terms of the statement, we cannot be sure of the truth value so this cannot be classified as a proposition.

Example 0.2. When a proposition contains one or more variables, we can only evaluate it having first assigned each a concrete value. For example, consider

"$x°C$ equals $90°C$"

where x is a variable. By assigning x a value we get a proposition; setting $x = 10$, for example, gives

"$10°C$ equals $90°C$"

which clearly evaluates to false. Setting $x = 90°C$ gives

"$90°C$ equals $90°C$"

which evaluates to true.

Definition 0.2. *Informally, a* **propositional function** *is just a short-hand way of writing a proposition; we give the function a name and a list of free variables. So, for example, the function*

$$f(x, y) : x = y$$

is called f and has two variables named x and y. If we use the function as $f(10, 20)$, performing the binding $x = 10$ and $y = 20$, it has the same meaning as $10 = 20$.

Example 0.3. We might write

g : "the temperature is $90°C$"

and hence use g (the left-hand side) as a short-hand for the longer proposition (the right-hand side): it works the same way in the sense that if g tells us the truth value of said proposition. Here, g has no free variables; imagine we extend our example to write

$h(x)$: "$x°C$ equals $90°C$".

Now, h is representing a longer proposition. When we bind x to a value via $h(10)$, we find

$h(10)$ = "$10°C$ equals $90°C$"

which can be evaluated to false.

1.1 Connectives

Definition 0.3. *A* **connective** *binds together a number of propositional* **terms** *into a single, compound proposition called an* **expression***. For brevity, we use symbols to denote common connectives:*

- *"not x" is denoted $\neg x$, and often termed logical* **complement** *(or* **negation***).*
- *"x and y" is denoted $x \wedge y$, and often termed logical* **conjunction***.*
- *"x or y" is denoted $x \vee y$, and often called an inclusive-or, and termed logical (inclusive)* **disjunction***.*
- *"x or y but not x and y" is denoted $x \oplus y$, and often called an exclusive-or, and termed logical (exclusive)* **disjunction***.*
- *"x implies y" is denoted $x \Rightarrow y$, and sometimes written as "if x then y", and termed logical* **implication***, and finally*
- *"x is equivalent to y" is denoted $x \equiv y$, and sometimes written as "x if and only if y" or even "x iff. y". termed logical* **equivalence***.*

Note that we group statements using parentheses when there could be some confusion about the order they are applied. As such $(x \wedge y)$ *is the same as* $x \wedge y$*, and* $(x \wedge y) \vee z$ *simply means we apply the* $\wedge$ *connective to* x *and* y *first, then* $\vee$ *to the result and* z*.*

Definition 0.4. *Provided we include parentheses in a compound proposition, there will be no ambiguity wrt. the order connectives are applied. For instance, if we write*

$$(x \wedge y) \vee z$$

it is clear that we first resolve the conjunction of x *and* y*, then the disjunction of that result and* z*.*

If parentheses are not *included however, we rely on* **precedence** *rules to determine the order for us. In short, the following list*

1. $\neg$,
2. $\wedge$,
3. $\vee$,
4. $\Rightarrow$,
5. $\equiv$

assigns a precedence level to each connective. Using the same example as above, if we omit the parentheses and instead write

$$x \wedge y \vee z$$

we still get the same result: $\wedge$ *has a higher precedence level than* $\vee$ *(sometimes we say* $\wedge$ *"binds more tightly" to operands than* $\vee$*), so we resolve the former before the latter.*

Example 0.4. For example, the expression

"the temperature is less than 90°C $\wedge$ the temperature is greater than 10°C"

contains two terms that propose

"the temperature is less than 90°C"

and

"the temperature is greater than 10°C" .

These terms are joined together using the $\wedge$ connective so that the whole expression evaluates to true if both of the terms are true, otherwise it evaluates to false. In a similar way we might write a compound proposition

"the temperature is less than x°C $\wedge$ the temperature is greater than y°C"

which can only be evaluated when we assign values to the variables x and y.

Definition 0.5. *The meaning of connectives is usually describe in a tabular form which enumerates the possible values each term can take and what the resulting truth value is; we call this a* **truth table***.*

x	y	$\neg x$	$x \wedge y$	$x \vee y$	$x \oplus y$	$x \Rightarrow y$	$x \equiv y$
false	**false**	**true**	**false**	**false**	**false**	**true**	**true**
false	**true**	**true**	**false**	**true**	**true**	**true**	**false**
true	**false**	**false**	**false**	**true**	**true**	**false**	**false**
true	**true**	**false**	**true**	**true**	**false**	**true**	**true**

Example 0.5. The $\neg$ connective complements (or negates) the truth value of a given expression. Considering the expression

$$\neg(x > 10),$$

we find that the expression $\neg(x > 10)$ is true if the term $x > 10$ is false and the expression is false if $x > 10$ is true. If we assign $x = 9$, $x > 10$ is false and hence the expression $\neg(x > 10)$ is true. If we assign $x = 91$, $x > 10$ is true and hence the expression $\neg(x > 10)$ is false.

Example 0.6. The meaning of the $\wedge$ connective is also as one would expect; the expression

$$(x > 10) \wedge (x < 90)$$

is true if *both* the expressions $x > 10$ and $x < 90$ are true, otherwise it is false. So if $x = 20$, the expression is true. But if $x = 9$ or $x = 91$, then it is false: even though one or other of the terms is true, they are not both true.

Example 0.7. The inclusive-or and exclusive-or connectives are fairly similar. The expression

$$(x > 10) \lor (x < 90)$$

is true if *either* $x > 10$ *or* $x < 90$ is true or both of them are true. Here we find that all the assignments $x = 20$, $x = 9$ and $x = 91$ mean the expression is true; in fact it is hard to find an x for which it evaluates to false! Conversely, the expression

$$(x > 10) \oplus (x < 90)$$

is only true if *only one* of either $x > 10$ or $x < 90$ is true; if they are both true then the expression is false. We now find that setting $x = 20$ means the expression is false while both $x = 9$ and $x = 91$ mean it is true.

Example 0.8. Implication is more tricky. If we write $x \Rightarrow y$, we typically call x the **hypothesis** and y the **conclusion**. In order to justify the truth table for implication, consider the example

$$(x \text{ is prime }) \land (x \neq 2) \Rightarrow (x \equiv 1 \pmod 2)$$

i.e., if x is a prime other than 2, it follows that it is odd. Therefore, if x is prime then the expression is true if $x \equiv 1 \pmod 2$ and false otherwise (since the implication is invalid). If x is not prime, then the expression does not really say *anything* about the expected outcome: we only know what to expect if x *was* prime. Since it *could* still be that $x \equiv 1 \pmod 2$ even when x is not prime, based on what we know from the example, we assume it is true when this case occurs.

Put in a less formal way, the idea is that *anything* can follow from a false hypothesis. If the hypothesis is false, we cannot be sure whether or not the conclusion is false: we therefore assume it is possibly true, which is sort of an "optimistic default". Consider a less formal example to support this. The statement "if I am unhealthy then I will die" means x = "I am unhealthy" and y = "I will die", and that $r = x \Rightarrow y$ has four possible cases:

1. I am healthy and do not die, so x = **false**, y = **false** and r = **true**,
2. I am healthy and die, so x = **false**, y = **true** and r = **true**,
3. I am unhealthy and do not die, so x = **true**, y = **false** and r = **false**, and
4. I am unhealthy and die, so x = **true**, y = **true** and r = **true**.

The first two cases do not contradict the original statement (since in them I am healthy, so it doesn't apply): only the third case does, in that I do not die (maybe I had a good doctor for instance).

Example 0.9. In contrast, equivalence is fairly simple. The expression $x \equiv y$ is only true if x and y evaluate to the same value. This matches the concept of equality in other contexts, such as between numbers. As an example, consider

$$(x \text{ is odd }) \equiv (x \equiv 1 \pmod 2).$$

This expression is true since if the left side is true, the right side must also be true and vice versa. If we change it to

$$(x \text{ is odd }) \equiv (x \text{ is prime }),$$

then the expression is false. To see this, note that only some odd numbers are prime: just because a number is odd does not mean it is always prime although if it is prime it must be odd (apart from the corner case of $x = 2$). So the equivalence works in one direction but not the other and hence the expression is false.

Definition 0.6. *An expression which is equivalent to true, no matter what values are assigned to any variables, is called a* **tautology***; an expression which is equivalent to false is called a* **contradiction***.*

Definition 0.7. *We call two expressions logically* **equivalent** *if they are composed of the same variables and have the same truth value for every possible assignment to those variables. More formally, two expressions x and y are equivalent iff. $x \equiv y$ can be proved a tautology.*

Various subtleties emerge when trying to prove two expressions are logically equivalent, but for our purposes it suffices to adopt a brute-force approach by a) enumerating the values each variable can take, then b) checking whether or not the expressions produce identical truth values in all cases. Note that, in practise, this can clearly become difficult wrt. amount of work required: with n variables there will be 2^n possible assignments, which grows (very) quickly as n grows.

1.2 Quantifiers

Definition 0.8. *A* **free variable** *in a given expression is one which has not yet been assigned a value. Roughly speaking, a* **quantifier** *allows a free variable to take one of many values:*

- *the* **universal quantifier** *"for all x, y is true" is denoted $\forall\ x\ [y]$, while*
- *the* **existential quantifier** *"there exists an x such that y is true" is denoted $\exists\ x\ [y]$.*

We say that **binding** *a quantifier to a variable quantifies it; after it has been quantified we say it is* **bound** *(rather than free).*

As an aside, quantifiers can be roughly viewed as moving from propositional logic into predicate (or first-order) logic (with second-order logic then a further extension, e.g., allowing quantification of relations). Put more simply, however, when we encounter an expression such as

$$\exists\ x\ [y]$$

we are essentially assigning x all possible values; to make the expression true, *just one* of these values needs to make the expression y true. Likewise, when we encounter

$$\forall\ x\ [y]$$

we are again assigning x all possible values. This time however, to make the expression true, *all of them* need to make the expression y true.

Example 0.10. Consider the following

$$\text{"there exists an } x \text{ such that } x \equiv 0 \pmod 2\text{"}$$

which we can rewrite symbolically as

$$\exists\ x\ [x \equiv 0 \pmod 2].$$

In this case, x is bound by an $\exists$ quantifier; we are asserting that for some value of x, it is true that $x \equiv 0 \pmod 2$. Restating the same thing another way, if just *one* x means $x \equiv 0 \pmod 2$ is true then the whole (quantified) expression is true. Clearly $x = 2$ satisfies this condition, so the expression is true.

Example 0.11. Consider the following

$$\text{"for all } x, x \equiv 0 \pmod 2\text{"}$$

which we can rewrite symbolically as

$$\forall\ x\ [x \equiv 0 \pmod 2].$$

This is a more general assertion about x, demanding that for *all* x it is true that $x \equiv 0 \pmod 2$. Taking the opposite approach to the above, to conclude the whole (quantified) expression is false we need an x such that $x \not\equiv 0 \pmod 2$. This is easy, because any odd value of x is good enough, so the expression is false.

2 Collections: sequences

Definition 0.9. *A* **sequence** *is an ordered collection of* **elements**, *which can be of any (but normally homogeneous) type.*

The **size** *or* **length** *of a sequence, denoted $|X|$ (or, alternativly, $\#X$ elsewhere), is the number of elements it contains. The order of elements is important, with an* **index** *used to refer to each one: the i-th element of a sequence X is denoted X_i, st. $0 \leq i < |X|$ and $X_j = \perp$ for $j < 0$ or $j \geq |X|$.*

2.1 Basic definition

Example 0.12. Consider a sequence of elements

$$A = \langle 0, 3, 1, 2 \rangle$$

which one can think of as like a list, read from left-to-right. In this case, we conclude, for example, that $|A| = 4$, $A_0 = 0$, $A_1 = 3$, $A_2 = 1$, and $A_3 = 2$; $A_4 = \perp$, because that element does not exist (i.e., the index 4 is too large, and so deemed out-of-bounds).

Example 0.13. Each element in the sequence A is a *number*, but we might equally define a sequence of *characters* such as

$$B = \langle \text{'a'}, \text{'b'}, \text{'c'}, \text{'d'}, \text{'e'} \rangle.$$

However, since the order or elements is important if we define

$$C = \langle 2, 1, 3, 0 \rangle$$

then clearly $A \neq C$ because, for example, $A_0 \neq C_0$. Both A and B are sequences of homogeneous type: their elements are *all* numbers and characters respectively.

2.2 Operations

The **concatenate** operator can be used to join two together sequences. Although most often used on the *right*-hand side of an equality (or an assignment), it is also allowed on the *left*-hand side: in such a case, it performs "deconcatination" by splitting apart a sequence.

Example 0.14. Imagine we start with two 4-element sequences

$$\begin{array}{rcl} F & = & \langle 0, 1, 2, 3 \rangle \\ G & = & \langle 4, 5, 6, 7 \rangle \end{array}$$

Their concatenation is denoted

$$H = F \parallel G = \langle 0, 1, 2, 3 \rangle \parallel \langle 4, 5, 6, 7 \rangle = \langle 0, 1, 2, 3, 4, 5, 6, 7 \rangle$$

noting that the result H is an 8-element sequence whose first (resp. last) four elements match F (resp. G). Likewise, we might write

$$I \parallel J = H$$

where now the concatenation operator appears on the left-hand side: this works basically the same way but in reverse, meaning

$$I \parallel J = H = \langle 0, 1, 2, 3, 4, 5, 6, 7 \rangle = \langle 0, 1, 2, 3 \rangle \parallel \langle 4, 5, 6, 7 \rangle$$

and so $I = F = \langle 0, 1, 2, 3 \rangle$ and $J = G = \langle 4, 5, 6, 7 \rangle$. Note that this approach demands the left- and right-hand sides have the same length, so elements can be organised appropriately.

2.3 Advanced definition and short-hands

It can make sense to avoid enumerating a sequence completely, which is the approach used above: explicitly including each element can become laborious, error prone, or simply inconvenient. The examples below show various short-hands to address this problem:

1. Where there is no ambiguity, **ellipses** (or continuation dots) are allowed to replace one or more elements. For example, again consider the sequence

   $$B = \langle \text{'a'}, \text{'b'}, \text{'c'}, \text{'d'}, \text{'e'} \rangle$$

   which we *could* rewrite as

   $$B = \langle \text{'a'}, \text{'b'}, \ldots, \text{'e'} \rangle.$$

   with the ellipsis representing the sub-sequence $\langle \text{'c'}, \text{'d'} \rangle$. In fact, this approach is sometimes *required*. In B there was a well defined start and end to the sequence, but in

   $$E = \langle 1, 2, 3, 4, \ldots \rangle.$$

   the ellipsis represents elements we either do not know, or which do not matter: because there is no end to the sequence, we cannot necessarily fill in the ellipsis as before. Note that this also means $|E|$ might be infinite or simply unknown.

2. It can be convenient to apply similar reasoning to the indices used to specify elements. For example,

   $$\begin{array}{rcl} B_{0,1,\ldots,3} & = & B_{0,1,2,3} \\ & = & B_0 \parallel B_1 \parallel B_2 \parallel B_3 \\ & = & \langle \text{'a'}, \text{'b'}, \text{'c'}, \text{'d'} \rangle \end{array}$$

3. The so-called **comprehension** (or **builder**) notation allows *generation* of a sequence using a rule. Consider

   $$F = \langle x \mid 4 \leq x < 8 \rangle = \langle 4, 5, 6, 7 \rangle$$

   for example: the comprehension includes a) an output expression (i.e., x) and b) a rule (or predicate, i.e., $4 \leq x < 8$) that limits the instances of variables considered when forming the output. Informally, you might read this example as "all x such that x is between 4 and 7".

3 Collections: sets

Definition 0.10. *A* **set** *is an unordered collection of* **elements***; the elements may only occur once (otherwise we have a* **bag** *or* **multi-set***), and can normally be of any (but homogeneous) type.*

The **size** *or* **cardinality** *of a set, denoted $|X|$ (or, alternativly, $\#X$ elsewhere), is the number of elements it contains. If the element x is in (resp. not in) the set X, we say x is a* **member** *of X (resp. not a member) or write $x \in X$ (resp. $x \notin X$).*

As an aside, this suggests the elements can potentially be *other* sets. Russell's paradox, a discovery by Bertrand Russell in 1901, describes an issue with formal set theory that stems from this fact. In a sense, the paradox is a rephrasing of the liar paradox seen earlier. Consider A, the set of all sets which do not contain themselves: the question is, does A contain itself? If it does, it should not be in A by definition but it is; if it does not, it should be in the set A by definition but it is not.

3.1 Basic definition

Example 0.15. Consider the set of integers between two and eight (inclusive), which we can define as

$$A = \{2, 3, 4, 5, 6, 7, 8\}.$$

In this case, we conclude, for example, that $|A| = 7$, $2 \in A$, and $9 \notin A$ (i.e., 2 is a member, but 9 is *not* a member, of A). Notice that, unlike a sequence, because the order of elements is irrelevant, it makes no sense to refer to them via an index: A_i implies there is some specific i-th element, but, without a specific order, *which* element it refers to is unclear. However, also note the same fact means if we define

$$B = \{8, 7, 6, 5, 4, 3, 2\}$$

then we can conclude $A = B$.

3.2 Operations

Definition 0.11. *A* **sub-set***, say Y, of a set X is such that for every $y \in Y$ we have that $y \in X$. This is denoted $Y \subseteq X$. Conversely, we can say X is a* **super-set** *of Y and write $X \supseteq Y$.*

From this definition, it follows that every set is a valid sub-set and super-set of itself and, therefore, that $X = Y$ iff. $X \subseteq Y$ and $Y \subseteq X$. If $X \neq Y$ we use the terms **proper sub-set** *and* **proper super-set***, and so write $Y \subset X$ and $X \supset Y$ respectively.*

Definition 0.12. *For sets X and Y, we have that*

- *the* **union** *of X and Y is $X \cup Y = \{x \mid x \in X \lor x \in Y\}$,*
- *the* **intersection** *of X and Y is $X \cap Y = \{x \mid x \in X \land x \in Y\}$,*
- *the* **difference** *of X and Y is $X - Y = \{x \mid x \in X \land x \notin Y\}$, and*
- *the* **complement** *of X is $\overline{X} = \{x \mid x \in \mathcal{U} \land x \notin X\}$.*

We say X and Y are **disjoint** *(or* **mutually exclusive***) if $X \cap Y = \emptyset$. Note also that the complement operation can be rewritten $X - Y = X \cap \overline{Y}$.*

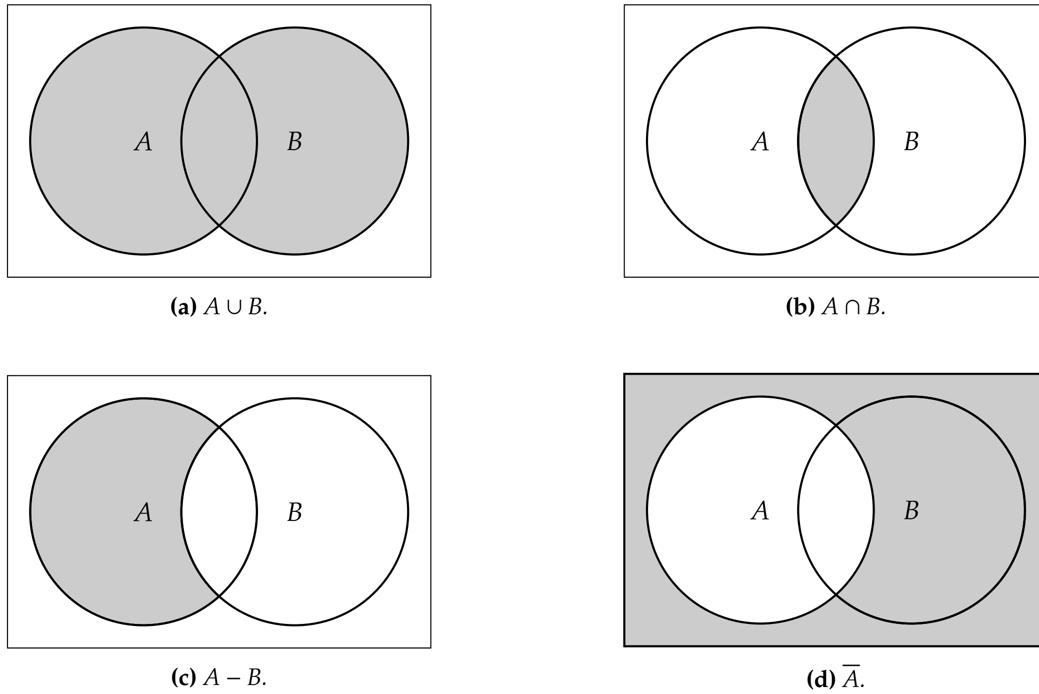
Definition 0.13. *The union and intersection operations preserve a law of cardinality called the* **principle of inclusion***, which states*

$$|A \cup B| = |A| + |B| - |A \cap B|.$$

This property has a simple intuition, in that elements in both A and B will be counted twice by $|A|$ and $|B|$; this is corrected via the last term (i.e., via $|A \cap B|$).

Definition 0.14. *The power set of a set X, denoted $\mathcal{P}(X)$, is the set of every possible sub-set of X. Note that $\emptyset$ is a member of all power sets.*

On first reading, these definitions can seem quite abstract. However, we have another tool at our disposal which describes what they mean in a more concrete, visual way. This tool is a **Venn diagram**, named after mathematician John Venn who invented the concept in 1881. The idea is that sets are represented by regions drawn inside a frame that implicitly represents the universal set $\mathcal{U}$. By placing the regions inside each other and overlapping their boundaries, we can describe most set-related concepts very easily.



(a) $A \cup B$.

(b) $A \cap B$.

(c) $A - B$.

(d) $\overline{A}$.

Figure 1: *A collection of Venn diagrams for standard set operations.*

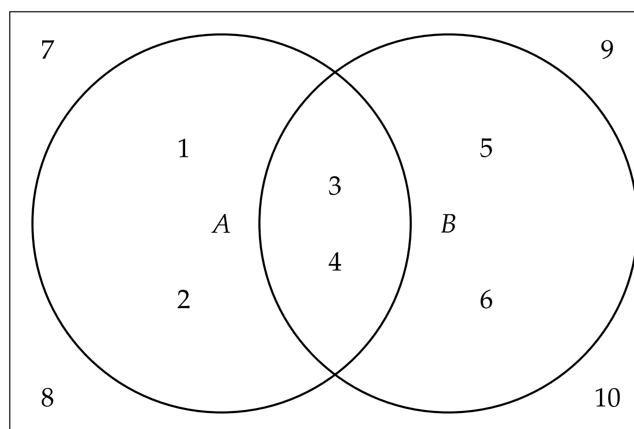


Figure 2: *An example Venn diagram showing membership of two sets.*

Example 0.16. Figure 1 includes four Venn diagrams which describe the union, intersection, difference, and complement operations: there is a shaded region representing members of each resulting set. For example, in the diagram for $A \cup B$ the shaded region covers all of the sets A and B: the result contains all elements in either A or B or both.

Example 0.17. Consider the sets

$$\begin{array}{rcl} A & = & \{1, 2, 3, 4\} \\ B & = & \{3, 4, 5, 6\} \end{array}$$

where the universal set is

$$\mathcal{U} = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}.$$

Recalling that elements within a given region are members of that set, Figure 2 describes several cases. Notice that

1. the union of A and B is $A \cup B = \{1, 2, 3, 4, 5, 6\}$, i.e., elements which are either members of A or B or both; note that the elements 3 and 4 do not appear twice because said result is a set,

2. the intersection of A and B is $A \cap B = \{3, 4\}$, i.e., elements that are members of both A and B,

3. the difference between A and B is $A - B = \{1, 2\}$, i.e., elements that are members of A but not also members of B, and

4. the complement of A is $\overline{A} = \{5, 6, 7, 8, 9, 10\}$, i.e., elements that are not members of A.

We can also use this example to verify that the principle of inclusion holds: given $|A| = 4$ and $|B| = 4$, checking the above shows $|A \cup B| = 6$ and $|A \cap B| = 2$ so by the principle of inclusion we have $6 = 4 + 4 - 2$.

3.3 Products

Definition 0.15. *The* **Cartesian product** *(or* **cross product***) of n sets, say* $X_0, X_1, \ldots, X_{n-1}$*, is defined as*

$$X_0 \times X_1 \times \cdots \times X_{n-1} = \{\langle x_0, x_1, \ldots, x_{n-1} \rangle \mid x_0 \in X_0 \wedge x_1 \in X_1 \wedge \cdots \wedge x_{n-1} \in X_{n-1}\}.$$

In the most simple case of $n = 2$, the Cartesian product $X_0 \times X_1$ is the set of all possible pairs where the first item in the pair is a member of X_0 and the second item is a member of X_1.

Definition 0.16. *The Cartesian product of a set X with itself n times is denoted X^n; for completeness, we define $X^0 = \emptyset$ and $X^1 = X$. A special-case of this notation is X^*, which applies the* **Kleene star** *operator: this captures the Cartesian product of X with itself a* finite *number of times (i.e., zero or more): a more precise definition is therefore*

$$X^* = \{\langle x_0, x_1, \ldots, x_{n-1} \rangle \mid n \geq 0, x_i \in X\},$$

which is sometimes extended to include a so-called **Kleen plus** *st.*

$$X^+ = \{\langle x_0, x_1, \ldots, x_{n-1} \rangle \mid n \geq 1, x_i \in X\}.$$

Example 0.18. Imagine we have the set $A = \{0, 1\}$. The Cartesian product of A with itself is

$$A \times A = A^2 = \{\langle 0, 0 \rangle, \langle 0, 1 \rangle, \langle 1, 0 \rangle, \langle 1, 1 \rangle\}.$$

That is, the pairs in $A \times A$ (or A^2, if you prefer) represent all possible sequences a) whose length is two, and b) whose elements are members of A.

3.4 Advanced definition and short-hands

Definition 0.17. *Some sets are hard (or impossible) to define using the notation used so far, and therefore need some special treatment:*

- *The set $\emptyset$, called the* **null set** *or* **empty set***, contains no elements: it is empty, meaning $|\emptyset| = 0$. Note that $\emptyset$ is a set not an element: one cannot write the empty set as $\{\emptyset\}$ since this is the set with one element, that element being the empty set itself.*

- *The contents of the set $\mathcal{U}$, called the* **universal set***, depends on the context. Roughly speaking, it contains every element from the problem being considered.*

It can make sense to avoid enumerating a set completely, which is the approach used above: explicitly including each element can become laborious, error prone, or simply inconvenient. The examples below show various short-hands to address this problem:

1. Where there is no ambiguity, **ellipses** (or continuation dots) are allowed to replace one or more elements. For example, we might rewrite the set A as

$$A = \{2, 3, \ldots, 7, 8\}$$

   with the ellipsis representing the sub-set $\{4, 5, 6\}$. In fact, this approach is sometimes *required*. Imagine we want to define a set of even integers which are greater than or equal to two: this set has an infinite size, so we need to defined it as

$$C = \{2, 4, 6, 8, \ldots\}.$$

2. The so-called **comprehension** (or **builder**) notation allows *generation* of a set using a rule. Consider

$$D = \{x \mid f(x)\}.$$

   for example: the comprehension includes a) an output expression (i.e., x) and b) a rule (or predicate, i.e., $f(x)$) that limits the instances of variables considered when forming the output. Informally, you might read this example as "all x such that $f(x) =$ **true**". Using the same idea, we could rewrite previous examples as

$$A = \{x \mid 2 \leq x \leq 8\},$$

   and

$$C = \{x \mid x > 0 \wedge x \equiv 0 \pmod{2}\}$$

   and so define the same sets we defined explicitly.

Definition 0.18. *Several useful sets that relate to numbers can be defined:*

- *The* **integers** *are whole numbers which can be positive or negative and also include zero; this set is denoted by*

$$\mathbb{Z} = \{\ldots, -3, -2, -1, 0, +1, +2, +3, \ldots\}$$

  or alternatively

$$\mathbb{Z} = \{0, \pm 1, \pm 2, \pm 3, \ldots\}.$$

- *The* **natural numbers** *are whole numbers which are positive; they are denoted by the set*

$$\mathbb{N} = \{0, 1, 2, 3, \ldots\}.$$

  and represent a sub-set of $\mathbb{Z}$.

- *The* **binary numbers** *are simply one and zero, i.e.,*

$$\mathbb{B} = \{0, 1\},$$

  and represent a sub-set of $\mathbb{N}$.

- *The* **rational numbers** *are those which can be expressed in the form* x/y*, where* x *and* y *are both integers and termed the* **numerator** *and* **denominator***. This set is denoted*

$$\mathbb{Q} = \{x/y \mid x \in \mathbb{Z} \wedge x \in \mathbb{Z} \wedge y \neq 0\}$$

  where we disallow a value of $y = 0$ *to avoid problems. Clearly the set of rational numbers is a super-set of* $\mathbb{Z}$, $\mathbb{N}$, *and* $\mathbb{B}$*, since, for example, we can write* $x/1$ *to convert any* $x \in \mathbb{Z}$ *as a member of* $\mathbb{Q}$*. However, not all numbers are rational: some are* **irrational** *in the sense that it is impossible to find a* x *and* y *such that they exactly represent the required result. Examples include the value of* π *which is approximated by, but not exactly equal to,* $22/7$.

4 Collections: some additional special-cases

4.1 Tuples

Definition 0.19. *It is common to use the term* **tuple** *as a synonym for sequence: a sequence of* n *elements is an* n**-tuple**, *or simply a* **tuple** *if the number of elements is irrelevant. Note that the special cases*

$$\begin{array}{ccccc} n = 2 & \rightsquigarrow & 2\text{-}tuple & \rightsquigarrow & \textbf{pair} \\ n = 3 & \rightsquigarrow & 3\text{-}tuple & \rightsquigarrow & \textbf{triple} \end{array}$$

have intuitive names.

Note that, from here on, we use the terms sequence and tuple as an informal way to distinguish between cases where elements are, respectively, a) of (potentially) homogeneous and heterogeneous type, *and/or* b) mutable (i.e., can be altered) and immutable (i.e., cannot be altered).

Example 0.19. Noting the bracketing style used to differentiate it from a sequence, we can define an example 2-tuple or pair as

$$A = (4, \text{`f'})$$

In this case, the elements $A_0 = 4$ and $A_1 = \text{`f'}$ clearly have different types: the first is a number and one which is a character.

4.2 Strings

Definition 0.20. *An* **alphabet** *is a non-empty set of* **symbols**.

Definition 0.21. *A* **string** X *wrt. some alphabet* Σ *is a sequence, of finite length, whose elements are members of* Σ, *i.e.,*

$$X = \langle X_0, X_1, \ldots, X_{n-1} \rangle$$

for some n *st.* $X_i \in \Sigma$ *for* $0 \leq i < n$; *if* n *is zero, we term* X *the* **empty string** *and denote it* ϵ. *It can be useful, and is common to write elements in in human-readable form termed a* **string literal**: *this basically just means writing them from right-to-left without any associated notation (e.g., brackets or commas).*

Definition 0.22. *A* **language** *is a set of strings.*

Example 0.20. If

$$\Sigma = \{0, 1\}$$

then the strings of length $n = 2$ (left), and the corresponding literal (right), are as follows:

$$\begin{array}{ccc} \langle 0, 0 \rangle & \equiv & 00 \\ \langle 1, 0 \rangle & \equiv & 01 \\ \langle 0, 1 \rangle & \equiv & 10 \\ \langle 1, 1 \rangle & \equiv & 11 \end{array}$$

Example 0.21. If

$$\Sigma = \{\text{`a'}, \text{`b'}, \ldots, \text{`z'}\}$$

then the strings of length $n = 2$ (left), and the corresponding literal (right), are as follows:

$$\begin{array}{ccc} \langle \text{`a'}, \text{`a'} \rangle & \equiv & aa \\ \langle \text{`b'}, \text{`a'} \rangle & \equiv & ab \\ & \vdots & \\ \langle \text{`a'}, \text{`b'} \rangle & \equiv & ba \\ \langle \text{`b'}, \text{`b'} \rangle & \equiv & bb \\ & \vdots & \\ \langle \text{`z'}, \text{`z'} \rangle & \equiv & zz \end{array}$$

5 Functions

Definition 0.23. *If X and Y are sets, a* **function** *f from X to Y is a process that maps each element of X to an element of Y. We write this as*

$$f : X \rightarrow Y$$

where X is termed the **domain** *of f and Y is the* **codomain** *of f. For an element $x \in X$, which we term the* **pre-image***, there is only one $y = f(x) \in Y$ which is termed the* **image** *of x. Finally, the set*

$$\{y \mid y = f(x) \wedge x \in X \wedge y \in Y\}$$

which is all possible results, is termed the **range** *of f and is always a sub-set of the* **codomain***.*

From this definition it might seem as though we can only have functions with one input and one output. However, we are perfectly entitled to use sets of sets; this means we can use a Cartesian product as the domain. For example, we can define a function

$$f : A \times A \rightarrow B$$

which takes elements from the Cartesian product $A \times A$ as input, and produces an element of B as output. So since the inputs are of the form $\langle x, y \rangle \in A \times A$, f takes two input values "packaged up" as a single pair.

Example 0.22. Consider a function Inv which takes an integer x as input, and produces the rational number $1/x$ as output:

$$\textsc{Inv} : \left\{ \begin{array}{lcl} \mathbb{Z} & \rightarrow & \mathbb{Q} \\ x & \mapsto & 1/x \end{array} \right.$$

Note that here we write the **function signature**, which defines the domain and codomain of Inv, inline with the definition of the **function behaviour**. This is simply a short-hand for writing the function signature

$$\textsc{Inv} : \mathbb{Z} \rightarrow \mathbb{Q}$$

and function behaviour

$$\textsc{Inv}(x) \mapsto 1/x$$

separately. In either case the domain of Inv is $\mathbb{Z}$, because it accepts an integer as input; the codomain is $\mathbb{Q}$, because it produces a rational number as output. If we take an integer and apply the function to get something like $\textsc{Inv}(2) = 1/2$, we have that $1/2$ is the image of 2 or conversely 2 is the pre-image of $1/2$ under Inv.

Example 0.23. Consider the function

$$\textsc{Max} : \left\{ \begin{array}{lcl} \mathbb{Z} \times \mathbb{Z} & \rightarrow & \mathbb{Z} \\ \langle x, y \rangle & \mapsto & \left\{ \begin{array}{ll} x & \text{if } x > y \\ y & \text{otherwise} \end{array} \right. \end{array} \right.$$

This is the maximum function on integers; it takes two integers as input and produces an integer, the maximum of the inputs, as output. So if we take the pair of integers $\langle 2, 4 \rangle$ say, and then apply the function, we get $\textsc{Max}(2, 4) = 4$. In this case, the domain of Max is $\mathbb{Z} \times \mathbb{Z}$ and the codomain is $\mathbb{Z}$; the integer 4 is the image of the pair $\langle 2, 4 \rangle$ under Max.

5.1 Composition

Definition 0.24. *Given two functions $f : X \rightarrow Y$ and $g : Y \rightarrow Z$, the* **composition** *of f and g is denoted*

$$g \circ f : X \rightarrow Z.$$

The notation $g \circ f$ should be read as "apply g to the result of applying f". That is, given some input $x \in X$, this composition is equivalent to applying $y = f(x)$ and then $z = g(y)$ to get the result $z \in Z$. More formally, we have

$$(g \circ f)(x) = g(f(x)).$$

5.2 Properties

Definition 0.25. *For a given function f, we say that f is*

- **surjective** *if the range equals the codomain, i.e., there are no elements in the codomain which do not have a pre-image in the domain,*
- **injective** *if no two elements in the domain have the same image in the range, and*
- **bijective** *if the function is both surjective and injective, i.e., every element in the domain is mapped to exactly one element in the codomain.*

Using the examples above, we clearly have that Inv is not surjective but Max is. This follows because we can construct a rational $2/3$ which does not have an integer pre-image under Inv so the function cannot be surjective. Equally, for any integer x in the range of Max there is always a pair $\langle x, y \rangle$ in the domain such that $x > y$ so Max is surjective, in fact there are lots of them since $\mathbb{Z}$ is infinite in size! In the same way, we have that Inv is injective but Max is not. Only one pre-image x maps to the value $1/x$ in the range under Inv but there are multiple pairs $\langle x, y \rangle$ which map to the same image under Max, for example 4 is the image of both $\langle 1, 4 \rangle$ and $\langle 2, 4 \rangle$ under Max.

Definition 0.26. *The* **identity function** $\mathcal{I}$ *on a set X is defined by*

$$\mathcal{I} : \begin{cases} X & \rightarrow & X \\ x & \mapsto & x \end{cases}$$

so that it maps all elements to themselves. Given two functions f and g defined by $f : X \rightarrow Y$ and $g : Y \rightarrow X$, if $g \circ f$ is the identity function on set X and $f \circ g$ is the identity on set Y, then f is the **inverse** *of g and g is the inverse of f. We denote this by $f = g^{-1}$ and $g = f^{-1}$. If a function f has an inverse, we hence have $f^{-1} \circ f = \mathcal{I}$.*

The inverse of a function maps elements from the codomain *back* into the domain, reversing the original function. It is easy to see that not *all* functions have an inverse. In particular, if a function is not injective there will be more than one potential pre-image for the inverse of any image; this suggests we cannot sensibly map from the codomain back into the domain. The Inv function is another, more concrete example: some value such as $1/x$ do have an inverse, namely $x/1$, yet others such as $2/3$ do not. For example, $3/2$ is not an integer, i.e., not a member of the domain $\mathbb{Z}$, so we cannot map it from the codomain back into the domain. Put another way, Inv, as we have defined it at least, has no inverse.

Example 0.24. Consider the successor function on integers

$$\textsc{Succ} : \begin{cases} \mathbb{Z} & \rightarrow & \mathbb{Z} \\ x & \mapsto & x + 1 \end{cases}$$

which takes an integer x as input and produces the successor (or next) integer $x + 1$ as output. This function is bijective, since the codomain and range are the same and no two integers have the same successor. As a result, the inverse is easy to describe as

$$\textsc{Pred} : \begin{cases} \mathbb{Z} & \rightarrow & \mathbb{Z} \\ x & \mapsto & x - 1 \end{cases}$$

which is the predecessor function: it takes an integer x as input and produces $x - 1$ as output. To see that $\textsc{Succ}^{-1} = \textsc{Pred}$ and $\textsc{Succ}^{-1} = \textsc{Pred}$ note that

$$(\textsc{Pred} \circ \textsc{Succ})(x) = (x + 1) - 1 = x$$

which is the identity function, and conversely that

$$(\textsc{Succ} \circ \textsc{Pred})(x) = (x - 1) + 1 = x$$

which is also the identity function.

5.3 Relations

Definition 0.27. *Informally, a* **binary relation** f *on a set* X *is like a propositional function which takes members of the set as input and "filters" them to produce an output. As a result, for a set* X *the relation* f *forms a sub-set of* $X \times X$*. For a given set* X *and a binary relation* f*, we say* f *is*

- **reflexive** *if* $f(x, x) = \textbf{true}$ *for all* $x \in X$*,*
- **symmetric** *if* $f(x, y) = \textbf{true}$ *implies* $f(y, x) = \textbf{true}$ *for all* $x, y \in X$*, and*
- **transitive** *if* $f(x, y) = \textbf{true}$ *and* $f(y, z) = \textbf{true}$ *implies* $f(x, z) = \textbf{true}$ *for all* $x, y, z \in X$*.*

If f *is reflexive, symmetric and transitive, then we call it an* **equivalence relation***.*

Example 0.25. Consider a set $A = \{1, 2, 3, 4\}$, whose Cartesian product is

$$A \times A = \left\{ \begin{array}{llll} \langle 1,1 \rangle, & \langle 1,2 \rangle, & \langle 1,3 \rangle, & \langle 1,4 \rangle, \\ \langle 2,1 \rangle, & \langle 2,2 \rangle, & \langle 2,3 \rangle, & \langle 2,4 \rangle, \\ \langle 3,1 \rangle, & \langle 3,2 \rangle, & \langle 3,3 \rangle, & \langle 3,4 \rangle, \\ \langle 4,1 \rangle, & \langle 4,2 \rangle, & \langle 4,3 \rangle, & \langle 4,4 \rangle \end{array} \right\}.$$

Imagine we define a function

$$\textsc{Equ} : \left\{ \begin{array}{lcl} \mathbb{Z} \times \mathbb{Z} & \rightarrow & \{\textbf{false}, \textbf{true}\} \\ \langle x, y \rangle & \mapsto & \left\{ \begin{array}{ll} \textbf{true} & \text{if } x = y \\ \textbf{false} & \text{otherwise} \end{array} \right. \end{array} \right.$$

which tests whether two inputs are equal. Using the function we can form a sub-set of $A \times A$ called $A_{\textsc{Equ}}$, for example, by "filtering out" the pairs (x, y) $\textsc{Equ}(x, y) = \textbf{true}$ to get

$$A_{\textsc{Equ}} = \{\langle 1,1 \rangle, \langle 2,2 \rangle, \langle 3,3 \rangle, \langle 4,4 \rangle\}.$$

For members of A, say $x, y, z \in A$,

1. $\textsc{Equ}(x, x) = \textbf{true}$, so the relation is reflexive,
2. if $\textsc{Equ}(x, y) = \textbf{true}$ then $\textsc{Equ}(y, x) = \textbf{true}$, so the relation is symmetric, and
3. if $\textsc{Equ}(x, y) = \textbf{true}$ and $\textsc{Equ}(y, z) = \textbf{true}$ then $\textsc{Equ}(x, z) = \textbf{true}$, so the relation is transitive

and hence an equivalence relation. Now imagine we define another function

$$\textsc{Lth} : \left\{ \begin{array}{lcl} \mathbb{Z} \times \mathbb{Z} & \rightarrow & \{\textbf{false}, \textbf{true}\} \\ \langle x, y \rangle & \mapsto & \left\{ \begin{array}{ll} \textbf{true} & \text{if } x < y \\ \textbf{false} & \text{otherwise} \end{array} \right. \end{array} \right.$$

which tests whether one input is less than another. Taking the same approach as above, we can form

$$A_{\textsc{Lth}} = \{\langle 1,2 \rangle, \langle 1,3 \rangle, \langle 1,4 \rangle, \langle 2,3 \rangle, \langle 2,4 \rangle, \langle 3,4 \rangle\}$$

of all pairs (x, y) with $x, y \in A$ st. $\textsc{Lth}(x, y) = \textbf{true}$. Now, for members of A, say $x, y, z \in A$,

1. $\textsc{Lth}(x, x) = \textbf{false}$, so the relation is not reflexive (it is irreflexive),
2. if $\textsc{Lth}(x, y) = \textbf{true}$ then $\textsc{Lth}(y, x) = \textbf{false}$, so the relation is not symmetric (it is anti-symmetric), but
3. if $\textsc{Lth}(x, y) = \textbf{true}$ and $\textsc{Lth}(y, z) = \textbf{true}$ then $\textsc{Lth}(x, z) = \textbf{true}$, so the relation is transitive.

6 Boolean algebra

Most people encounter **elementary algebra** fairly early on at school. Even if the name is unfamiliar, the basic idea should be: one has

- a set of **values**, e.g., $\mathbb{Z}$,
- a set of **operators**, e.g., $+$,
- a set of **relations**, e.g., $=$, and
- a set of **axioms** which dictate what the operators and relations mean and how they work.

Again, you may not know what these axioms are called, but you probably do know how they work. For example, given $x, y, z \in \mathbb{Z}$, you might know a) we can write $x + (y + z) = (x + y) + z$, i.e., say that addition is associative, or b) we can write $x \cdot 1 = x$, i.e., say that the multiplicative identity of x is 1. In reality, we can be much more general than this: when we discuss "an" algebra, all we really mean is a set of values for which there is a well defined set of operators, relations and axioms; **abstract algebra** is basically concerned with sets of values that are, potentially, *not* numbers.

Definition 0.28. *An* **abstract algebra** *includes*

- *a set of values, say X,*
- *a set of* **binary operators**
$$\odot : X \times X \rightarrow X,$$
- *a set of* **unary operators**
$$\oslash : X \rightarrow X,$$
- *a set of* **binary relations**
$$\ominus : X \times X \rightarrow \{\textbf{false}, \textbf{true}\},$$
and
- *a set of* **axioms** *which dictate what the operators and relations mean and how they work.*

In the early 1840s, mathematician George Boole put this generality to good use by combining (or, in fact, *unifying*) concepts in logic and set theory: the result forms **Boolean algebra** [1]. Put (rather too) simply, Boole saw that working with logic a expression is much the same as working with an arithmetic expression, and reasoned that the axioms of the latter should apply to the former as well. Based on what we already know, for example, 0 and **false** and $\emptyset$ are all sort of equivalent, as are 1 and **true** and $\mathcal{U}$; likewise, $x \wedge y$ and $x \cap y$ are sort of equivalent, as are $x \vee y$ and $x \cup y$ and $\neg x$ and $\overline{x}$. More formally, we can see that the identity axiom applies in same way:

$$\begin{array}{lcl@{\qquad}lcl} x \vee \textbf{false} & = & x & x \wedge \textbf{true} & = & x \\ x \cup \emptyset & = & x & x \cap \mathcal{U} & = & x \\ x + 0 & = & x & x \cdot 1 & = & x \end{array}$$

Ironically, this was viewed as somewhat obscure; Boole *himself* did not necessarily regard logic directly as a mathematical concept. It was not until 1937 that Claude Shannon, then a student of Electrical Engineering and Mathematics, saw the potential of using Boolean algebra to represent and manipulate digital information [7]. This insight is fundamentally important, essentially allowing a "link" between theory (i.e., Mathematics) and practice (i.e., physical circuits that we can build).

Definition 0.29. *Putting everything together produces the following definition for Boolean algebra. Consider the set* $\mathbb{B} = \{0, 1\}$ *on which there are two binary operators*

$$\wedge : \left\{ \begin{array}{lcl} \mathbb{B} \times \mathbb{B} & \rightarrow & \mathbb{B} \\ \langle x, y \rangle & \mapsto & \left\{ \begin{array}{ll} 0 & \text{if } x = 0 \text{ and } y = 0 \\ 0 & \text{if } x = 0 \text{ and } y = 1 \\ 0 & \text{if } x = 1 \text{ and } y = 0 \\ 1 & \text{if } x = 1 \text{ and } y = 1 \end{array} \right. \end{array} \right.$$

and

$$\lor : \left\{ \begin{array}{lcl} \mathbb{B} \times \mathbb{B} & \rightarrow & \mathbb{B} \\ \langle x, y \rangle & \mapsto & \left\{ \begin{array}{ll} 0 & \text{if } x = 0 \text{ and } y = 0 \\ 1 & \text{if } x = 0 \text{ and } y = 1 \\ 1 & \text{if } x = 1 \text{ and } y = 0 \\ 1 & \text{if } x = 1 \text{ and } y = 1 \end{array} \right. \end{array} \right.$$

and a unary operator

$$\neg : \left\{ \begin{array}{lcl} \mathbb{B} & \rightarrow & \mathbb{B} \\ x & \mapsto & \left\{ \begin{array}{ll} 1 & \text{if } x = 0 \\ 0 & \text{if } x = 1 \end{array} \right. \end{array} \right.$$

AND, OR and NOT respectively; they are governed the following axioms

$$\begin{array}{llcl}
\textit{commutativity} & x \land y & \equiv & y \land x \\
\textit{association} & (x \land y) \land z & \equiv & x \land (y \land z) \\
\textit{distribution} & x \land (y \lor z) & \equiv & (x \land y) \lor (x \land z) \\
\textit{identity} & x \land 1 & \equiv & x \\
\textit{null} & x \land 0 & \equiv & 0 \\
\textit{idempotency} & x \land x & \equiv & x \\
\textit{inverse} & x \land \neg x & \equiv & 0 \\
\textit{absorption} & x \land (x \lor y) & \equiv & x \\
\textit{de Morgan} & \neg(x \land y) & \equiv & \neg x \lor \neg y \\
\\
\textit{commutativity} & x \lor y & \equiv & y \lor x \\
\textit{association} & (x \lor y) \lor z & \equiv & x \lor (y \lor z) \\
\textit{distribution} & x \lor (y \land z) & \equiv & (x \lor y) \land (x \lor z) \\
\textit{identity} & x \lor 0 & \equiv & x \\
\textit{null} & x \lor 1 & \equiv & 1 \\
\textit{idempotency} & x \lor x & \equiv & x \\
\textit{inverse} & x \lor \neg x & \equiv & 1 \\
\textit{absorption} & x \lor (x \land y) & \equiv & x \\
\textit{de Morgan} & \neg(x \lor y) & \equiv & \neg x \land \neg y \\
\\
\textit{equivalence} & x \equiv y & \equiv & (x \Rightarrow y) \land (y \Rightarrow x) \\
\textit{implication} & x \Rightarrow y & \equiv & \neg x \lor y \\
\textit{involution} & \neg\neg x & \equiv & x
\end{array}$$

Note that the $\land$ and $\lor$ operations in Boolean algebra behave in a similar way to $\cdot$ and $+$ in a elementary algebra: as such, they are sometimes referred to as "product" and "sum" operations (and denoted $\cdot$ and $+$ as a result).

Definition 0.30. *In line with propositional logic, it is common to add a third binary operator called XOR:*

$$\oplus : \left\{ \begin{array}{lcl} \mathbb{B} \times \mathbb{B} & \rightarrow & \mathbb{B} \\ \langle x, y \rangle & \mapsto & \left\{ \begin{array}{ll} 0 & \text{if } x = 0 \text{ and } y = 0 \\ 1 & \text{if } x = 0 \text{ and } y = 1 \\ 1 & \text{if } x = 1 \text{ and } y = 0 \\ 0 & \text{if } x = 1 \text{ and } y = 1 \end{array} \right. \end{array} \right.$$

More generally, XOR is an example of a **derived operator**, *a name which hints at the fact it is a short-hand derived from operators we already have. Put another way, because*

$$x \oplus y \equiv (\neg x \land y) \lor (x \land \neg y),$$

XOR can be defined in terms of AND, OR and NOT. Two other examples, which will be useful later, are

- *"NOT-AND" or* **NAND**, *which is denoted and defined as*

  $$x \overline{\land} y \equiv \neg(x \land y),$$

  and

- *"NOT-OR" or* **NOR**, *which is denoted and defined as*

$$x \overline{\vee} y \equiv \neg(x \vee y).$$

Definition 0.31. *A* **functionally complete** *(or* **universal***) set of Boolean operators is st. every possible truth table can be described by combining the constituent members into a Boolean expression. For example, the sets* $\{\neg, \wedge\}$ *and* $\{\neg, \vee\}$ *are functionally complete.*

In 1921, Emil Post developed [5] a set of necessary and sufficient conditions for such a description to be valid (i.e., a method to prove whether a given set is or is not functionally complete); where such a set is singleton, i.e., contains one operator only, that operator is termed a **Sheffer function** [8] (after Henry Sheffer, who, during 1912, independently *re*discovered work of 1880 by Charles Sanders Peirce). For example, the singleton sets $\{\overline{\wedge}\}$ and $\{\overline{\vee}\}$ are functionally complete, meaning NAND and NOR can both be described as Sheffer functions.

Definition 0.32. *Certain operators (and hence axioms) are termed* **monotone***: this means changing an operand either leaves the result unchanged, or that it always changes the same way as the operand. Conversely, other operators are termed* **non-monotone** *when these conditions do not hold.*

Example 0.26. We can describe

$$x \wedge 0$$

as monotone, because changing x does not change the result (which is always 0); the same argument applies to

$$x \wedge 1.$$

In this case, notice that if $x = 0$ then the result is 0 whereas if $x = 1$ then the result is 1: this suggests changing x from 0 to 1 (resp. from 1 to 0) changes the result in the same way.

Definition 0.33. *The fact there are AND and OR forms of most axioms hints at a more general underlying principle. Consider a Boolean expression* e*: the* **principle of duality** *states that the* **dual expression** e^D *is formed by*

1. *leaving each variable as is,*
2. *swapping each* $\wedge$ *with* $\vee$ *and vice versa, and*
3. *swapping each* 0 *with* 1 *and vice versa.*

Of course e *and* e^D *are different expressions, and clearly not equivalent; if we start with some* $e \equiv f$ *however, then we do still get* $e^D \equiv f^D$.

As an example, consider axioms for

1. *distribution, e.g., if*

$$e = x \wedge (y \vee z) \equiv (x \wedge y) \vee (x \wedge z)$$

then

$$e^D = x \vee (y \wedge z) \equiv (x \vee y) \wedge (x \vee z)$$

and

2. *identity, e.g., if*

$$e = x \wedge 1 \equiv x$$

then

$$e^D = x \vee 0 \equiv x.$$

Definition 0.34. *The de Morgan axiom can be turned into a more general principle. Consider a Boolean expression* e*: the* **principle of complements** *states that the* **complement expression** $\neg e$ *is formed by*

1. *swapping each variable* x *with the complement* $\neg x$*,*
2. *swapping each* $\wedge$ *with* $\vee$ *and vice versa, and*
3. *swapping each* 0 *with* 1 *and vice versa.*

As an example, consider that if

$$e = x \wedge y \wedge z,$$

then by the above we should find

$$f = \neg e = (\neg x) \vee (\neg y) \vee (\neg z).$$

Proof:

x	y	z	$\neg x$	$\neg y$	$\neg z$	e	f
0	0	0	1	1	1	0	1
0	0	1	1	1	0	0	1
0	1	0	1	0	1	0	1
0	1	1	1	0	0	0	1
1	0	0	0	1	1	0	1
1	0	1	0	1	0	0	1
1	1	0	0	0	1	0	1
1	1	1	0	0	0	1	0

6.1 Manipulation

Saying we have manipulated an expression just means we have transformed it from one form to another; when done correctly, this should imply the original and alternative, transformed forms are equivalent. Often this is presented as a **derivation**, or sequence of steps which relate to an axiom or assumption (so is assumed valid by definition).

Example 0.27. Consider the (supposed) equality

$$(a \wedge b \wedge c) \vee (\neg a \wedge b) \vee (a \wedge b \wedge \neg c) = b,$$

for example, which we can prove is valid via the derivation

$$\begin{array}{clllll}
 & (a \wedge b \wedge c) \vee (\neg a \wedge b) & & \vee\ (a \wedge b \wedge \neg c) & & \\
= & (a \wedge b \wedge c) \vee (a \wedge b \wedge \neg c) & & \vee\ (\neg a \wedge b) & & \text{(commutativity)} \\
= & (a \wedge b) & \wedge\ (c \vee \neg c) & \vee\ (\neg a \wedge b) & & \text{(distribution)} \\
= & (a \wedge b) & \wedge\ 1 & \vee\ (\neg a \wedge b) & & \text{(inverse)} \\
= & (a \wedge b) & & \vee\ (\neg a \wedge b) & & \text{(identity)} \\
= & b & \wedge\ (a \vee \neg a) & & & \text{(distribution)} \\
= & b & \wedge\ 1 & & & \text{(inverse)} \\
= & b & & & & \text{(identity)}
\end{array}$$

Of course we *might* employ a brute-force approach instead. If we write a truth table for the left- and right-hand sides, this allows us to compare them: if the outputs match in all rows, we can conclude the left- and right-hand sides are equivalent. For example,

a	b	c	$t_0 = a \wedge b \wedge c$	$t_1 = \neg a \wedge b$	$t_2 = a \wedge b \wedge \neg c$	$t_0 \vee t_1 \vee t_2$
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	1	0	1
0	1	1	0	1	0	1
1	0	0	0	0	0	0
1	0	1	0	0	0	0
1	1	0	0	0	1	1
1	1	1	1	0	0	1

shows the left- and right-hand sides *are* equivalent, as expected. Of course if there were more variables, we would need to enumerate all possible values of each one. Our truth table would grow, and, at some point, the derivation-type approach starts to become more attractive: we achieve the same outcome, but *without* brute-force enumeration.

Example 0.28. Another motivation for manipulating a given expression is to produce an alternative with some goal or metric in mind; a common metric to use is the number of operators each expression uses, i.e., how simple they are st. with the task then termed **simplification**, which is one way to judge their evaluation cost. Consider the exclusive-or operator, i.e., an expression $x \oplus y$, which we can write as the more complicated expression

$$(y \wedge \neg x) \vee (x \wedge \neg y)$$

An aside: how *many* n-input Boolean functions are there?

To be more concrete, imagine we are interested in the function

$$f : \mathbb{B}^n \rightarrow \mathbb{B}.$$

Note that each of the n inputs can obviously be assigned one of two values, namely 0 or 1, so there are 2^n possible assignments to n inputs. For example, if f were to have $n = 1$ input, say x, there would be $2^1 = 2$ possible assignments because x can either be 0 or 1. In the same way, for $n = 2$ inputs, say x and y, there are $2^2 = 4$ possible assignments: we can have

$$\begin{array}{ll} x = 0 & y = 0 \\ x = 0 & y = 1 \\ x = 1 & y = 0 \\ x = 1 & y = 1 \end{array}$$

This is why a truth table for n inputs will have 2^n rows: each row details one assignment to the inputs, and the associated output.

So, how many *functions* are there? A function with n-inputs is specified by a truth table with 2^n rows; each row includes an output that is assigned 0 or 1, depending on exactly *which* function the truth table describes. So to count how many functions there are, we can just count how many possible assignments there are to the 2^n outputs. The correct answer is 2^{2^n}.

or the less complicated (i.e., simpler) expression

$$(x \vee y) \wedge \neg(x \wedge y).$$

One can prove these are equivalent by writing truth tables for them, as we did above. To do so, however, we need the expressions in the first place: how did we get the alternative from the original one?

The answer is we start with one expression, and (somehow intelligently) apply axioms to move step-by-step toward the other. For example, to do so more easily, notice that we can manipulate each term in the first expression whose form is $p \wedge \neg q$ as follows:

$$\begin{array}{lll} & (p \wedge \neg q) & \\ = & (p \wedge \neg q) \vee 0 & \text{(identity)} \\ = & (p \wedge \neg q) \vee (p \wedge \neg p) & \text{(inverse)} \\ = & p \wedge (\neg p \vee \neg q) & \text{(distribution)} \end{array}$$

This introduces a *new* rule that we can make use of; since it was derived from axioms we assume are valid, we can assume it is valid as well. Using it, we can rewrite the original expression as

$$\begin{array}{lll} & (y \wedge \neg x) \vee (x \wedge \neg y) & \\ = & (x \wedge (\neg x \vee \neg y)) \vee (y \wedge (\neg x \vee \neg y)) & (p \wedge \neg q \text{ rule above}) \\ = & (x \vee y) \wedge (\neg x \vee \neg y) & \text{(distribution)} \\ = & (x \vee y) \wedge \neg(x \wedge y) & \text{(de Morgan)} \end{array}$$

which gives us the alternative we are looking for, noting it requires 4 operators rather than 5.

6.2 Functions

Definition 0.35. *Given the definition of Boolean algebra, it is perhaps not surprising that a generic n-input, 1-output* **Boolean function** *f can be described as*

$$f : \mathbb{B}^n \rightarrow \mathbb{B}.$$

It is possible to extend this definition so it caters for m-outputs; we write the function signature as

$$g : \mathbb{B}^n \rightarrow \mathbb{B}^m.$$

This can be thought of as m separate n-input, 1-output Boolean functions, i.e.,

$$\begin{array}{lcl} g_0 & : & \mathbb{B}^n \rightarrow \mathbb{B} \\ g_1 & : & \mathbb{B}^n \rightarrow \mathbb{B} \\ & \vdots & \\ g_{m-1} & : & \mathbb{B}^n \rightarrow \mathbb{B} \end{array}$$

An aside: an enumeration of 2-input Boolean functions.

We know there are 2^{2^n} Boolean functions with n inputs; this represents a *lot* of functions as n grows. However, for a small number of inputs, say $n = 2$, $2^{2^n} = 2^{2^2} = 2^4 = 16$ functions is fairly manageable. In fact, we can easily write them *all* down: if f_i denotes the i-th such function, we find

x	y	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}	f_{15}
0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	0	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	1	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

This hints that one way to see *why* 2^{2^n} is correct is to view each column for f_i as filled by i represented in binary. How many (unsigned) integers can be represented in m bits? The answer is 2^m, suggesting 2^{2^n} (unsigned) integers can represented in 2^n bits and hence there are 2^{2^n} functions.

Some of the functions should look familiar, and, either way, we can try to describe them in English language terms vs. their truth table. Note, for example, that

- f_0 is the constant 0 function (i.e., $f_0(x, y) = 0$, ignoring x and y),
- f_1 is disjunction composed with complement (i.e., $f_1(x, y) = \neg(x \vee y)$),
- f_2 is inhibition (i.e., $f_2(x, y) = y \wedge \neg x$, which is like $x < y$),
- f_3 is complement (i.e., $f_3(x, y) = \neg x$, ignoring y),
- f_4 is inhibition (i.e., $f_4(x, y) = x \wedge \neg y$, which is like $y < x$),
- f_5 is complement (i.e., $f_5(x, y) = \neg y$, ignoring x),
- f_6 is non-equivalence (i.e., $f_5(x, y) = x \oplus y$, which is like $x \neq y$),
- f_7 is conjunction composed with complement (i.e., $f_7(x, y) = \neg(x \wedge y)$),
- f_8 is conjunction (i.e., $f_8(x, y) = x \wedge y$),
- f_9 is equivalence (i.e., $f_5(x, y) = \neg(x \oplus y)$, which is like $x = y$),
- f_{10} is identity (i.e., $f_{10}(x, y) = y$, ignoring x),
- f_{11} is implication (i.e., $f_{11}(x, y) = y \implies x$),
- f_{12} is identity (i.e., $f_{12}(x, y) = x$, ignoring y),
- f_{13} is implication (i.e., $f_{13}(x, y) = x \implies y$),
- f_{14} is disjunction (i.e., $f_{14}(x, y) = x \vee y$), and
- f_{15} is the constant 1 function (i.e., $f_{15}(x, y) = 1$, ignoring x and y).

where the output of g is described by

$$g(x) \mapsto g_0(x) \parallel g_1(x) \parallel \ldots \parallel g_{m-1}(x).$$

That is, the output of g is just the m individual 1-bit outputs $g_i(x)$ concatenated together. This is often termed a **vectorial Boolean function***: the inputs and outputs are vectors (or sequences) over the set $\mathbb{B}$ rather than single elements of it.*

Definition 0.36. *A* **Boolean-valued function** *(or* **predicate function***)*

$$f : X \to \{0, 1\}$$

is a function whose output is a Boolean value: note the contrast with a Boolean function, in so far as it places no restriction on what the input (i.e., the set X) must be.

Example 0.29. Consider a 2-input, 1-output Boolean function, whose signature we can write as

$$f : \mathbb{B}^2 \to \mathbb{B}$$

st. for $r, x, y \in \mathbb{B}$, the input is a pair $\langle x, y\rangle$ and the output for a given x and y is written $r = f(x, y)$. The function itself can be specified in two ways. First, as previously, we could enumerate all possible input combinations, and specify corresponding outputs. This can be written equivalently in the form of an inline function behaviour, or as a truth table:

$$f(x,y) \mapsto \begin{cases} 0 & \text{if } x = 0 \text{ and } y = 0 \\ 1 & \text{if } x = 0 \text{ and } y = 1 \\ 1 & \text{if } x = 1 \text{ and } y = 0 \\ 0 & \text{if } x = 1 \text{ and } y = 1 \end{cases} \equiv$$

x	y	$f(x,y)$
0	0	0
0	1	1
1	0	1
1	1	0

However, with a large number of inputs, this becomes difficult. As a short-hand, we can therefore specify f as a Boolean expression instead, e.g.,

$$f : \langle x, y\rangle \mapsto (\neg x \wedge y) \vee (x \wedge \neg y).$$

This basically tells us how to compute outputs, rather than listing those outputs explicitly.

Example 0.30. Consider a 2-input, 2-output Boolean function

$$h : \begin{cases} \mathbb{B}^2 & \to & \mathbb{B}^2 \\ \langle x, y\rangle & \mapsto & \begin{cases} \langle 0,0\rangle & \text{if } x = 0 \text{ and } y = 0 \\ \langle 1,0\rangle & \text{if } x = 0 \text{ and } y = 1 \\ \langle 1,0\rangle & \text{if } x = 1 \text{ and } y = 0 \\ \langle 0,1\rangle & \text{if } x = 1 \text{ and } y = 1 \end{cases} \end{cases}$$

which we might write more compactly as the truth table

x	y	$h(x,y)$
0	0	$\langle 0,0\rangle$
0	1	$\langle 1,0\rangle$
1	0	$\langle 1,0\rangle$
1	1	$\langle 0,1\rangle$

Clearly we can decompose h into

$$h_0 : \begin{cases} \mathbb{B}^2 & \to & \mathbb{B} \\ \langle x, y\rangle & \mapsto & \begin{cases} 0 & \text{if } x = 0 \text{ and } y = 0 \\ 1 & \text{if } x = 0 \text{ and } y = 1 \\ 1 & \text{if } x = 1 \text{ and } y = 0 \\ 0 & \text{if } x = 1 \text{ and } y = 1 \end{cases} \end{cases}$$

and

$$h_1 : \begin{cases} \mathbb{B}^2 & \to & \mathbb{B} \\ \langle x, y\rangle & \mapsto & \begin{cases} 0 & \text{if } x = 0 \text{ and } y = 0 \\ 0 & \text{if } x = 0 \text{ and } y = 1 \\ 0 & \text{if } x = 1 \text{ and } y = 0 \\ 1 & \text{if } x = 1 \text{ and } y = 1 \end{cases} \end{cases}$$

meaning that

$$h(x, y) \equiv h_0(x, y) \parallel h_1(x, y).$$

6.3 Normal (or standard) forms

Definition 0.37. *Consider a Boolean expression:*

1. *When the expression is written as a sum (i.e., OR) of terms which each comprise the product (i.e., AND) of variables, e.g.,*

$$\underbrace{(a \wedge b \wedge c)}_{minterm} \vee (d \wedge e \wedge f),$$

it is said to be in **disjunctive normal form** *or* **Sum of Products (SoP)** *form; the terms are called the* **minterms**. *Note that each variable can exist as-is* or *complemented using NOT, meaning*

$$\underbrace{(\neg a \wedge b \wedge c)}_{minterm} \vee (d \wedge \neg e \wedge f),$$

is also a valid SoP expression.

2. *When the expression is written as a product (i.e., AND) of terms which each comprise the sum (i.e., OR) of variables, e.g.,*

$$\underbrace{(a \vee b \vee c)}_{maxterm} \wedge (d \vee e \vee f),$$

it is said to be in **conjunctive normal form** *or* **Product of Sums (PoS)** *form; the terms are called the* **maxterms**. *As above each variable can exist as-is* or *complemented using NOT.*

Example 0.31. Consider a 1-input, 1-output Boolean function

$$g : \left\{ \begin{array}{lcl} \mathbb{B} & \rightarrow & \mathbb{B} \\ x & \mapsto & \left\{ \begin{array}{ll} 0 & \text{if } x = 0 \text{ and } y = 0 \\ 1 & \text{if } x = 0 \text{ and } y = 1 \\ 1 & \text{if } x = 1 \text{ and } y = 0 \\ 0 & \text{if } x = 1 \text{ and } y = 1 \end{array} \right. \end{array} \right.$$

Writing this as a truth table, i.e.,

x	y	$g(x, y)$
0	0	0
0	1	1
1	0	1
1	1	0

the minterms are the second and third rows, while the maxterms are the first and fourth lines. An expression for g in SoP form is

$$g_{SoP}(x, y) = (\neg x \wedge y) \vee (x \wedge \neg y).$$

where terms $\neg x \wedge y$ and $x \wedge \neg y$ represent minterms of g: when the term is 1 or 0 the corresponding output is 1 or 0. It is usually crucial that all the variables appear in all the minterms so that the function is exactly described. To see why this is so, consider writing an *incorrect* SoP expression by removing the reference to y from the first minterm so as to get

$$(\neg x) \vee (x \wedge \neg y).$$

Now $\neg x$ is 1 for the first and second rows, rather than the second (as was the case with $\neg x \wedge y$), so we have described *another* function $h \neq g$ described by

x	y	$h(x, y)$
0	0	1
0	1	1
1	0	1
1	1	0

In a similar way, we can construct a PoS expression for g as

$$g_{PoS}(x, y) = (x \vee y) \wedge (\neg x \vee \neg y).$$

where $x \vee y$ and $\neg x \vee \neg y$ are the maxterms of g. By manipulating the expressions, we can prove that g_{SoP} and g_{PoS} are just two different ways to write the same function, i.e., g. Recall that for p and q

$$\begin{array}{lcll} (p \wedge \neg q) & = & (p \wedge \neg q) \vee 0 & \text{(identity)} \\ & = & (p \wedge \neg q) \vee (p \wedge \neg p) & \text{(inverse)} \\ & = & p \wedge (\neg p \vee \neg q) & \text{(distribution)} \end{array}$$

Using this rule, we can show

$$\begin{array}{lcll} g_{SoP}(x, y) & = & (\neg x \wedge y) \vee (x \wedge \neg y) & \\ & = & (y \wedge \neg x) \vee (x \wedge \neg y) & \text{(commutativity)} \\ & = & (x \wedge (\neg x \vee \neg y)) \vee (y \wedge (\neg x \vee \neg y)) & (p \wedge \neg q \text{ rule above}) \\ & = & (x \vee y) \wedge (\neg x \vee \neg y) & \text{(distribution)} \\ & = & g_{PoS} & \end{array}$$

7 Signals

Definition 0.38. *In general, a* **signal** *can be described as a descriptive function (abstractly), or a physical quantity (concretely), that varies in time or space so as to represent and/or communicate (i.e., convey) information. We say that*

- *a* **discrete-time** *signal is valid for a discrete (so finite) range of time indices, e.g.,* $t \in \mathbb{Z}$*,*
- *a* **continuous-time** *signal is valid for a continuous (so infinite) range of time indices, e.g.,* $t \in \mathbb{R}$*,*
- *a* **discrete-value** *signal has a value from a discrete (so finite) range, e.g.,* $f(t) \in \mathbb{Z}$*, and*
- *a* **continuous-value** *signal has a value from a continuous (so infinite) range, e.g.,* $f(t) \in \mathbb{R}$*.*

Definition 0.39. *The term* **analogue signal** *is a synonym of continuous-value signal: a physical quantity that varies in time is typically used to represent (that is, it is* analagous *to) some abstract variable.*

Definition 0.40. *Strictly speaking,* **digital signal** *is a synonym of discrete-value signal: it will have a digital (i.e., discrete or exact) value. This terminology is often overloaded, however, and taken to mean a signal whose value is either* 0 *or* 1 *(cf.* **logic signal***).*

The **transition** *of a digital signal from* 0 *to* 1 *(resp.* 1 *to* 0*) is called a positive (resp. negative)* **edge***; we often say it has* **toggled** *from* 0 *to* 1 *(resp.* 1 *to* 0*). During any time the signal has a value of* 1 *(resp.* 0*), we say it is at a positive (resp. negative)* **level** *(and use the term* **pulse** *as a synonym for positive level, i.e., the period between a positive and negative edge).*

Definition 0.41. *It is common to describe a signal by plotting it as a* **waveform***: the y-axis represents the value of the signal as time varies over time as represented by the x-axis.*

Note that it is common, though incorrect, to describe discrete-time signals by using a *continuous* plot; *connecting* discrete points implies a formally incorrect description (i.e., it gives the impression of a continuous-time signal). Doing so typically stems from either a) the fact said discrete-time signal is derived from an associated continuous-time signal (e.g., the latter has been quantised wrt. time, by sampling it at discrete time indices), or b) aesthetics, in the sense it is easier to see when printed.

8 Representations

God made the integers; all the rest is the work of man.

– Kronecker

8.1 Bits, bytes and words

Definition 0.42. *Used as a term by Claude Shannon in 1948 [6] (but attributed to John Tukey), a* **bit** *is a* **binary digit***. As a result, a given bit is a member of the set* $=\{0, 1\}$*; it can be used to represent a truth value, i.e.,* **false** *or* **true***, and hence a Boolean value within the context of Boolean algebra.*

Definition 0.43. *An n-bit* **bit-sequence** *(or* **binary sequence***) is a member of the set* $\mathbb{B}^n$*, i.e., it is an n-tuple of bits. Much like other sequences, we use* X_i *to denote the i-th bit of a binary sequence X and* $|X| = n$ *to denote the number of bits in X.*

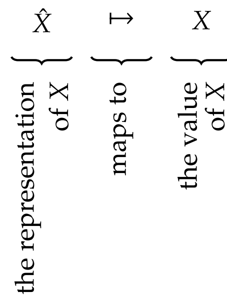
Definition 0.44. *Instead of writing out $X \in \mathbb{B}^n$ symbolically, i.e., writing $\langle X_0, X_1, \ldots, X_{n-1} \rangle$, we sometimes prefer to list the bits within a* **bit-literal** *(or* **bit-string**, *wrt. an implicit alphabet $\Sigma = \{0, 1\}$). For example, consider the following bit-sequence*

$$X = \langle 1, 1, 0, 1, 1, 1, 1 \rangle$$

st. $|X| = 7$, which can be written as the bit-literal

$$X = 1111011.$$

The question is however, what does a bit-sequence *mean*: what does it represent, other than just an (unstructured) sequence of bits? The answer is they can represent anything *we* decide they do; there is just one key concept, namely



That is, all we need is a) a representation and mapping specified concretely (i.e., written down, vs. reasoned about abstractly), and b) a mapping that means the right thing wrt. values, plus is ideally consistent in both directions (e.g., does not change based on the context, and is injective st. a single representation cannot be interpreted ambiguously). Notice the (subtle) annotation on the left-hand side of the mapping: $\hat{X}$ is intended to highlight this is a representation of some X, whose value therefore depends on the mapping used. Put another way, this suggests *different* mappings may legitimately map the same $\hat{X}$ to different values by interpreting the bit-sequence differently. This is, essentially, what means we can represent such a rich set of data (e.g., the pixels in an image) using only a bit (or sequence thereof) as a starting point.

8.1.1 Properties

Definition 0.45. *Following the idea of vectorial Boolean function, given an n-element bit-sequence X, and an m-element bit-sequence Y we can clearly*

1. *overload $\oslash \in \{\neg\}$, i.e., write*

   $$R = \oslash X,$$

   to mean

   $$R_i = \oslash X_i$$

   for $0 \leq i < n$,

2. *overload $\ominus \in \{\wedge, \vee, \oplus\}$, i.e., write*

   $$R = X \ominus Y,$$

   to mean

   $$R_i = X_i \ominus Y_i$$

   for $0 \leq i < n = m$, where if $n \neq m$, we pad either X or Y with 0 until the $n = m$.

Definition 0.46. *Given two n-bit sequences X and Y, we can define some important properties named after Richard Hamming, a researcher at Bell Labs:*

- *The* **Hamming weight** *of X is the number of bits in X that are equal to 1, i.e., the number of times $X_i = 1$. This can be expressed as*

  $$\mathcal{H}(X) = \sum_{i=0}^{n-1} X_i.$$

- *The* **Hamming distance** *between X and Y is the number of bits in X that differ from the corresponding bit in Y, i.e., the number of times $X_i \neq Y_i$. This can be expressed as*

  $$\mathcal{D}(X, Y) = \sum_{i=0}^{n-1} X_i \oplus Y_i.$$

An aside: the origins and impact of endianness.

The term endianness stems from a technical article [2], written in the 1980s by Danny Cohen, using Gulliver's Travels as an inspiration/analogy: an argument over whether cracking the big- or small-end of a soft-boiled egg is proper in the former, inspired terminology wrt. arguments over byte ordering in the latter. It does a brilliant job of surveying the significant impact of what is, at face value, a fairly trivial choice.

Note that both quantities naturally generalise to non-binary sequences.

Example 0.32. For example, given $A = \langle 1, 0, 0, 1 \rangle$ and $B = \langle 0, 1, 1, 1 \rangle$ we find that

$$\mathcal{H}(A) = \sum_{i=0}^{n-1} A_i = 1 + 0 + 0 + 1 = 2$$

and

$$\mathcal{D}(A, B) = \sum_{i=0}^{n-1} A_i \oplus B_i = (1 \oplus 0) + (0 \oplus 1) + (0 \oplus 1) + (1 \oplus 1) = 1 + 1 + 1 + 0 = 3$$

st. two bits in A equal 1, and three bits differ between A and B.

8.1.2 Ordering

There is, by design, no "structure" to a bit-literal. This can be problematic if, for example, we need a way to make sure the order of bits in the bit-literal is clear wrt. the corresponding bit-sequence. The same issues appear *whenever* describing a large(r) quantity in terms of small(er) parts, but, focusing on bits, we can describe **endianness** as follows:

Definition 0.47. *A given literal, say*

$$X = 1111011,$$

can be interpreted in two *ways:*

1. *A* **little-endian** *ordering is where we read bits in a literal from right-to-left, i.e.,*

   $$X_{LE} = \langle X_0, X_1, X_2, X_3, X_4, X_5, X_6 \rangle = \langle 1, 1, 0, 1, 1, 1, 1 \rangle,$$

   where

   - *the Least-Significant Bit (LSB) is the right-most in the literal (i.e., X_0), and*
   - *the Most-Significant Bit (MSB) is the left-most in the literal (i.e., $X_{n-1} = X_6$).*

2. *A* **big-endian** *ordering is where we read bits in a literal from left-to-right, i.e.,*

   $$X_{BE} = \langle X_6, X_5, X_4, X_3, X_2, X_1, X_0 \rangle = \langle 1, 1, 1, 1, 0, 1, 1 \rangle,$$

   where

   - *the Least-Significant Bit (LSB) is the left-most in the literal (i.e., $X_{n-1} = X_6$), and*
   - *the Most-Significant Bit (MSB) is the right-most in the literal (i.e., X_0).*

Unless specified, from here on it is (fairly) safe to assume that a little-endian convention is used. Keep in mind that having selected an endianness convention, which acts as a rule for conversion, there is no real distinction between a bit-sequence and a bit-literal: we can convert between them in either little-endian or bit-endian cases.

8.1.3 Grouping

Definition 0.48. *Some bit-sequences are given special names depending on their length. Given a* **word size** w *(e.g., the natural size as dictated by a given processor), we can defined*

$$
\begin{array}{rcr}
\textit{bit} & \equiv & 1\textit{-bit} \\
\textit{nybble} & \equiv & 4\textit{-bit} \\
\textit{byte} & \equiv & 8\textit{-bit} \\
 & & \\
\textit{half-word} & \equiv & (w/2)\textit{-bit} \\
\textit{word} & \equiv & w\textit{-bit} \\
\textit{double-word} & \equiv & (w \cdot 2)\textit{-bit} \\
\textit{quad-word} & \equiv & (w \cdot 4)\textit{-bit}
\end{array}
$$

but note that standards in particular often use the term octet as a synonym for byte (st. an octet string is therefore a byte-sequence): although less natural, we follow this terminology where it seems of value to match associated literature.

Example 0.33. Given a bit-sequence

$$B = \langle 1,1,0,0,0,0,0,0,1,0,0,0,1,0,1,0 \rangle$$

it can be attractive to group the bits into short(er) sub-sequences. For example, we could rewrite the sequence as either

$$
\begin{array}{rcl}
C & = & \langle \langle 1,1,0,0 \rangle, \langle 0,0,0,0 \rangle, \langle 1,0,0,0 \rangle, \langle 1,0,1,0 \rangle \rangle \\
 & = & \langle 1,1,0,0 \rangle \parallel \langle 0,0,0,0 \rangle \parallel \langle 1,0,0,0 \rangle \parallel \langle 1,0,1,0 \rangle \\
 & = & \langle 1,1,0,0,0,0,0,0,1,0,0,0,1,0,1,0 \rangle \\
 & = & B \\
 & & \\
D & = & \langle \langle 1,1,0,0,0,0,0,0 \rangle, \langle 1,0,0,0,1,0,1,0 \rangle \rangle \\
 & = & \langle 1,1,0,0,0,0,0,0 \rangle \parallel \langle 1,0,0,0,1,0,1,0 \rangle \\
 & = & \langle 1,1,0,0,0,0,0,0,1,0,0,0,1,0,1,0 \rangle \\
 & = & B
\end{array}
$$

st. C has four elements (each of which is a sub-sequence of four bits from B), while D has two elements (each of which is a sub-sequence of eight bits from B). It is important to see that we have not altered the bits themselves, just how they are grouped together: we can easily "flatten out" the sub-sequences and reconstruct the original sequence B.

Example 0.34. Consider four nibbles in C, i.e., the four 4-bit sub-sequences

$$
\begin{array}{rcl}
C_0 & = & \langle 1,1,0,0 \rangle \\
C_2 & = & \langle 0,0,0,0 \rangle \\
C_3 & = & \langle 1,0,0,0 \rangle \\
C_4 & = & \langle 1,0,1,0 \rangle
\end{array}
$$

If we want to reconstruct C itself, we need to know which order to put the sub-sequences in: via a little-endian convention we get

$$C_{LE} = \langle C_0, C_1, C_2, C_3 \rangle = \langle \langle 1,1,0,0 \rangle, \langle 0,0,0,0 \rangle, \langle 1,0,0,0 \rangle, \langle 1,0,1,0 \rangle \rangle$$

whereas via a big-endian convention we get

$$C_{BE} = \langle C_3, C_2, C_1, C_0 \rangle = \langle \langle 1,0,1,0 \rangle, \langle 1,0,0,0 \rangle, \langle 0,0,0,0 \rangle, \langle 1,1,0,0 \rangle \rangle.$$

8.1.4 Units

There is a standard notation for measuring multiplicities of bits and bytes: a suffix specifies the quantity ('b' or "bit" for bits, 'B' for bytes), and a prefix specifies a multiplier. Although the notation remains consistent, some ambiguities about how to interpret prefixes complicate matters.

The International System of Units (SI) works with decimal, base-10 prefixes so, for example, a kilobit means $10^3 = 1000$ bits. As a result, we find that

An aside: the shift-and-mask paradigm, part #1.

Given some w-bit word, the shift-and-mask paradigm allows us to extract (or isolate) individual or contiguous sequences of bits. Understanding this is crucial in many areas, and often used in lower-level C programs; this, and related techniques, it is often termed “bit twiddling” or “bit bashing”.

- Imagine we want to set the i-th bit of some x, i.e., x_i, to 1. This can be achieved by computing

  $$x \vee (1 \ll i)$$

  For example, if $x = 0011_{(2)}$ and $i = 2$ then we compute

  $$\begin{array}{lcccccc} x & \vee & (& 0001_{(2)} & \ll & i &) \\ 0011_{(2)} & \vee & (& 0001_{(2)} & \ll & 2 &) \\ 0011_{(2)} & \vee & & 0100_{(2)} & & & \\ 0111_{(2)} & & & & & & \end{array}$$

  meaning initially $x_2 = 0$, then we changed it so $x_2 = 1$.
- Imagine we want to set the i-th bit of some x, i.e., x_i, to 0. This can be achieved by computing

  $$x \wedge \neg(1 \ll i)$$

  For example, if $x = 0111_{(2)}$ and $m = 2$ then we compute

  $$\begin{array}{lccccccc} x & \wedge & \neg & (& 0001_{(2)} & \ll & i &) \\ 0111_{(2)} & \wedge & \neg & (& 0001_{(2)} & \ll & 2 &) \\ 0111_{(2)} & \wedge & \neg & (& 0100_{(2)} & & &) \\ 0111_{(2)} & \wedge & & & 1011_{(2)} & & & \\ 0011_{(2)} & & & & & & & \end{array}$$

  meaning initially $x_2 = 1$, then we changed it so $x_2 = 0$.

In both cases, the idea is to first create an appropriate **mask** then combine it with x to get x'; in both cases we do no actual arithmetic, only Boolean-style operations.

An aside: the shift-and-mask paradigm, part #2.

Imagine we want to extract an m-bit sub-word (i.e., m contiguous bits) starting at the i-th bit of some x. This can be achieved by computing

$$(x \gg i) \wedge ((1 \ll m) - 1)$$

The computation is a little more complicated, but basically the same principles apply: first we create an appropriate mask (the right-hand term) and combine it with x (the left-hand term). For example, if $x = 1011_{(2)}$ and $m = 2$:

- If $i = 0$ then we want to extract the sub-word $\langle x_1, x_0 \rangle$

$$\begin{array}{ccccccccccccccc}
(& x & \gg & i &) & \wedge & (& (& 1 & \ll & m &) & - & 1 &) \\
(& 1011_{(2)} & \gg & 0 &) & \wedge & (& (& & \ll & 2 &) & - & 1 &) \\
(& 1011_{(2)} & & &) & \wedge & (& (& 0100_{(2)} & & &) & - & 1 &) \\
(& 1011_{(2)} & & &) & \wedge & (& & 0011_{(2)} & & & & & &) \\
 & 0011_{(2)} & & & & & & & & & & & & &
\end{array}$$

  meaning $\langle x_1, x_0 \rangle = \langle 1, 1 \rangle$ as expected.

- If $i = 1$ then we want to extract the sub-word $\langle x_1, x_2 \rangle$

$$\begin{array}{ccccccccccccccc}
(& x & \gg & i &) & \wedge & (& (& 1 & \ll & m &) & - & 1 &) \\
(& 1011_{(2)} & \gg & 1 &) & \wedge & (& (& 1 & \ll & 2 &) & - & 1 &) \\
(& 0101_{(2)} & & &) & \wedge & (& (& 0100_{(2)} & & &) & - & 1 &) \\
(& 0101_{(2)} & & &) & \wedge & (& & 0011_{(2)} & & & & & &) \\
 & 0001_{(2)} & & & & & & & & & & & & &
\end{array}$$

  meaning $\langle x_1, x_2 \rangle = \langle 1, 0 \rangle$ as expected.

- If $i = 2$ then we want to extract the sub-word $\langle x_2, x_3 \rangle$

$$\begin{array}{ccccccccccccccc}
(& x & \gg & i &) & \wedge & (& (& 1 & \ll & m &) & - & 1 &) \\
(& 1011_{(2)} & \gg & 2 &) & \wedge & (& (& 1 & \ll & 2 &) & - & 1 &) \\
(& 0010_{(2)} & & &) & \wedge & (& (& 0100_{(2)} & & &) & - & 1 &) \\
(& 0010_{(2)} & & &) & \wedge & (& & 0011_{(2)} & & & & & &) \\
 & 0010_{(2)} & & & & & & & & & & & & &
\end{array}$$

  meaning $\langle x_2, x_3 \rangle = \langle 0, 1 \rangle$ as expected.

Notice that the $(0001_{(2)} \ll m) - 1$ term is basically giving us a way to create a value y where $y_{m-1\ldots0} = 1$, i.e., whose 0-th through to $(m-1)$-th bits are 1. If we know m ahead of time, we can clearly simplify this by providing y directly rather than computing it.

An aside: the shift-and-mask paradigm, part #3.

As a special case of extracting an m-element sub-sequence, when we set $m = 1$ we extract the i-th bit of x alone. This is a useful and common operation: following the above, it is achieved by computing

$$(x \gg i) \wedge 1,$$

i.e., replacing the general-purpose mask with the special-purpose constant $(1 \ll 1) - 1 = 2 - 1 = 1$. For example:

- If $x = 0011_{(2)}$ and $i = 2$ then we compute

$$\begin{array}{cccccccc} (& x & \gg & i &) & \wedge & 1 \\ (& 0011_{(2)} & \gg & 2 &) & \wedge & 1 \\ (& 0000_{(2)} & & &) & \wedge & 1 \\ & 0000_{(2)} & & & & & \end{array}$$

  meaning $x_2 = 0$.

- If $x = 0011_{(2)}$ and $i = 0$ then we compute

$$\begin{array}{cccccccc} (& x & \gg & i &) & \wedge & 1 \\ (& 0011_{(2)} & \gg & 0 &) & \wedge & 1 \\ (& 0011_{(2)} & & &) & \wedge & 1 \\ & 0001_{(2)} & & & & & \end{array}$$

  meaning $x_0 = 1$.

$$\begin{array}{llllllll} 1 & \text{kilobit} & \text{(kbit)} & = & 10^3 & \text{bits} & = & 1000 & \text{bits} \\ 1 & \text{megabit} & \text{(Mbit)} & = & 10^6 & \text{bits} & = & 1\,000\,000 & \text{bits} \\ 1 & \text{gigabit} & \text{(Gbit)} & = & 10^9 & \text{bits} & = & 1\,000\,000\,000 & \text{bits} \\ 1 & \text{terabit} & \text{(Tbit)} & = & 10^{12} & \text{bits} & = & 1\,000\,000\,000\,000 & \text{bits} \\ \\ 1 & \text{kilobyte} & \text{(kB)} & = & 10^3 & \text{bytes} & = & 1000 & \text{bytes} \\ 1 & \text{megabyte} & \text{(MB)} & = & 10^6 & \text{bytes} & = & 1\,000\,000 & \text{bytes} \\ 1 & \text{gigabyte} & \text{(GB)} & = & 10^9 & \text{bytes} & = & 1\,000\,000\,000 & \text{bytes} \\ 1 & \text{terabyte} & \text{(TB)} & = & 10^{12} & \text{bytes} & = & 1\,000\,000\,000\,000 & \text{bytes} \end{array}$$

However, in the context of Computer Science the same English prefixes are commonly (ab)used to specify a binary, base-2 multiplier. For example, kilo will be read to means $2^{10} = 1024 \simeq 1000$: RAM or hard disk capacity is commonly measured in this way, for example. To eliminate resulting ambiguity, the International Electrotechnical Commission (IEC) added some *more* SI prefixes; the result is that

$$\begin{array}{llllllll} 1 & \text{kibibit} & \text{(Kibit)} & = & 2^{10} & \text{bits} & = & 1024 & \text{bits} \\ 1 & \text{mebibit} & \text{(Mibit)} & = & 2^{20} & \text{bits} & = & 1\,048\,576 & \text{bits} \\ 1 & \text{gigibit} & \text{(Gibit)} & = & 2^{30} & \text{bits} & = & 1\,073\,741\,824 & \text{bits} \\ 1 & \text{tebibit} & \text{(Tibit)} & = & 2^{40} & \text{bits} & = & 1\,099\,511\,627\,776 & \text{bits} \\ \\ 1 & \text{kibibyte} & \text{(KiB)} & = & 2^{10} & \text{bytes} & = & 1024 & \text{bytes} \\ 1 & \text{mebibyte} & \text{(MiB)} & = & 2^{20} & \text{bytes} & = & 1\,048\,576 & \text{bytes} \\ 1 & \text{gigibyte} & \text{(GiB)} & = & 2^{30} & \text{bytes} & = & 1\,073\,741\,824 & \text{bytes} \\ 1 & \text{tebibyte} & \text{(TiB)} & = & 2^{40} & \text{bytes} & = & 1\,099\,511\,627\,776 & \text{bytes} \end{array}$$

The question is, which should we use? Does it *really* matter? Clearly, yes: if we buy a hard disk which says it holds 1 terabyte of data, we *hope* they are talking, in traditional terms, about a tebibyte, i.e., $1,099,511,627,776$ bytes rather than $1,000,000,000,000$ bytes, because then we get more storage capacity! In the same way, imagine we are comparing two hard disks: we need to make sure their quoted storage capacity use the same units, or the comparison will be unfair.

From here on, we try to make consistent use of the new SI prefixes: when we say kilobyte or kB we mean 10^3 bytes, and when we say kibibyte or KiB we mean 2^{10} bytes. On one hand, this might not be popular from a historical point of view; on the other hand, it should mean we clear and consistent.

8.2 Positional number systems

As humans, and because (mostly) we have ten fingers and toes, we are used to working with numbers written down using digits from the set $\{0, 1, \ldots, 9\}$. Imagine we write down such a number, say 123. It may not be as common, but hopefully you can believe this is roughly the same as writing the sequence

$$\begin{array}{rcl} \hat{A} & = & \langle A_0, A_1, A_2 \rangle \\ & = & \langle 3, 2, 1 \rangle \end{array}$$

given that 3 is the first digit of 123, 2 is the second digit and so on; we are reading digits in the sequence from left-to-right vs. right-to-left in the literal, but otherwise they capture the same meaning.

But how do we know what either 123 or $\hat{A}$ means? Informally at least, writing 123 intuitively means the value "one hundred and twenty three" which might be rephrased as "one hundred, two tens and three units". The latter case suggests how to add formalism to this intuition: we are just weighting each digit 1, 2 and 3 by some amount then adding everything up. For example, per the above we are computing the value via

$$\hat{A} \mapsto 123 = 1 \cdot 100 + 2 \cdot 10 + 3 \cdot 1.$$

We could also write the same thing as

$$\hat{A} \mapsto 123 = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0$$

given $10^0 = 1$ and $10^1 = 10$, or more formally still as

$$\hat{A} \mapsto 123 = \sum_{i=0}^{|A|-1} A_i \cdot 10^i.$$

meaning we add up the terms

$$\begin{array}{rcrcrcr} A_0 \cdot 10^0 & = & 3 \cdot 10^0 & = & 3 \cdot 1 & = & 3 \\ A_1 \cdot 10^1 & = & 2 \cdot 10^1 & = & 2 \cdot 10 & = & 20 \\ A_2 \cdot 10^2 & = & 1 \cdot 10^2 & = & 1 \cdot 100 & = & 100 \end{array}$$

to make a total of 123 as expected. Put another way, the sequence A **represents** the value "one hundred and twenty three". Two two facts start to emerge, namely

1. each digit is being weighted by a power of some **base** (or **radix**), which in this case is 10, and

2. the exponent in said weight is related to the position of the corresponding digit: the i-th digit is weighted by 10^i.

A neat outcome of identifying the base as some sort of parameter is that we can consider choices other than $b = 10$. Generalising the example somewhat provides the following definition:

Definition 0.49. *A base-b (or radix-b)* **positional number system** *uses digits from a* **digit set** $X = \{0, 1, \ldots, b-1\}$. *A number x is represented using n digits in total, m of which form the fractional part, i.e.,*

$$\begin{array}{rcl} \hat{x} & = & \langle x_0, x_1, \ldots, x_{n-1} \rangle \\ & \mapsto & \pm \sum_{i=-m}^{n-m-1} x_i \cdot b^i \end{array}$$

where $x_i \in X$; we term $\hat{x}$ the base-b **expansion** *of X.*

Definition 0.50. *The following common choices correspond to*

$$\begin{array}{rcl} b = 2 & \rightsquigarrow & \textbf{binary} \\ b = 8 & \rightsquigarrow & \textbf{octal} \\ b = 10 & \rightsquigarrow & \textbf{decimal} \\ b = 16 & \rightsquigarrow & \textbf{hexadecimal} \end{array}$$

numbers.

Example 0.35. Reconsider the example above: imagine we select $b = 2$, then make a claim that "one hundred and twenty three" is represented by

$$\begin{array}{rcl} \hat{B} & = & \langle B_0, B_1, B_2, B_3, B_4, B_5, B_6, B_7 \rangle \\ & = & \langle 1, 1, 0, 1, 1, 1, 1, 0 \rangle \end{array}$$

where, per the definition, now the digit set used is st. $B_i \in \{0, 1\}$ for $0 \leq i < 8$ (and thus implicitly setting $n = 8$ and $m = 0$). The value represented by $\hat{B}$ is given using exactly the same approach, i.e.,

$$\sum_{i=0}^{|B|-1} B_i \cdot 2^i,$$

noting that where we previously had 10 we now have 2, st. we add up the terms

$$\begin{array}{rcrcrcr} B_0 \cdot 2^0 & = & 1 \cdot 2^0 & = & 1 \cdot 1 & = & 1 \\ B_1 \cdot 2^1 & = & 1 \cdot 2^1 & = & 1 \cdot 2 & = & 2 \\ B_2 \cdot 2^2 & = & 0 \cdot 2^2 & = & 0 \cdot 4 & = & 0 \\ B_3 \cdot 2^3 & = & 1 \cdot 2^3 & = & 1 \cdot 8 & = & 8 \\ B_4 \cdot 2^4 & = & 1 \cdot 2^4 & = & 1 \cdot 16 & = & 16 \\ B_5 \cdot 2^5 & = & 1 \cdot 2^5 & = & 1 \cdot 32 & = & 32 \\ B_6 \cdot 2^6 & = & 1 \cdot 2^6 & = & 1 \cdot 64 & = & 64 \\ B_7 \cdot 2^7 & = & 0 \cdot 2^7 & = & 0 \cdot 128 & = & 0 \end{array}$$

to obtain a total of 123 as before.

8.2.1 Digits

Describing elements in the digit set $\{0, 1, \ldots, b - 1\}$, for whatever b, using a *single* digit can be fairly important; using multiple digits, for example, can start to introduce some ambiguity wrt. how we interpret a literal. In particular, once we select a $b > 10$ we hit a problem: we run out of single Roman-style digits that we can write down.

Example 0.36. Consider the same example as above where we have the literal 123: we know that if $b = 10$ and $\hat{A} = \langle 3, 2, 1 \rangle$ then

$$\hat{A} \mapsto 123 = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0.$$

However, if $b = 16$, although we know

$$123 = 7 \cdot 16^1 + 11 \cdot 16^0$$

we have no single-digit way to write 11. To solve this problem, we use the symbols (or in fact characters) $A \ldots F$ to represent $10 \ldots 15$. Otherwise everything works the same way, meaning for example, that if $\hat{B} = \langle B, 7 \rangle$ then

$$\begin{array}{rcrcrcr} \hat{B} & \mapsto & 123 & = & 7 \cdot 16^1 & + & B \cdot 16^0 \\ & & & = & 7 \cdot 16^1 & + & 11 \cdot 16^0 \end{array}$$

8.2.2 Notation

Amazingly there are not *many* jokes about Computer Science, but here are two (bad, comically speaking) examples:

1. There are only 10 types of people in the world: those who understand binary, and those who do not.
2. Why did the Computer Scientist always confuse Halloween and Christmas? Because 31 Oct equals 25 Dec.

Whether or not you laughed at them, both jokes stem from ambiguity in the representation of numbers: there is an ambiguity between "ten" written in decimal and binary in the former, and "twenty five" written in octal and decimal in the latter.

Look at the first joke: it is basically saying that the literal 10 can be interpreted as binary *or* decimal, i.e., as $1 \cdot 2 + 0 \cdot 1 = 2$ in binary and $1 \cdot 10 + 0 \cdot 1 = 10$ in decimal. So the two types of people are those who understand that 2 can be represented by 10, and those that do not. Now look at the second joke: this is a play on words in that "Oct" can mean "October" but also "octal" or base-8 and "Dec" can mean "December" but also "decimal" or base-10. With this in mind, we see that

$$3 \cdot 8 + 1 \cdot 1 \;=\; 25 \;=\; 2 \cdot 10 + 5 \cdot 1.$$

An aside: octal and hexadecimal as a short-hand for binary.

It is useful to remember is that octal and hexadecimal can be viewed as just a short-hand for binary: each octal or hexadecimal digit represents exactly three or four binary digits respectively. This can make it *much* easier to write and remember long sequences of binary digits. As an example, consider hexadecimal. Each hexadecimal digit $x_i \in \{0, 1, \ldots, 15\}$ can be represented using four bits (since there are $2^4 = 16$ possible combinations), so can be viewed instead as those four *binary* digits.

Using a concrete example, the following translation steps

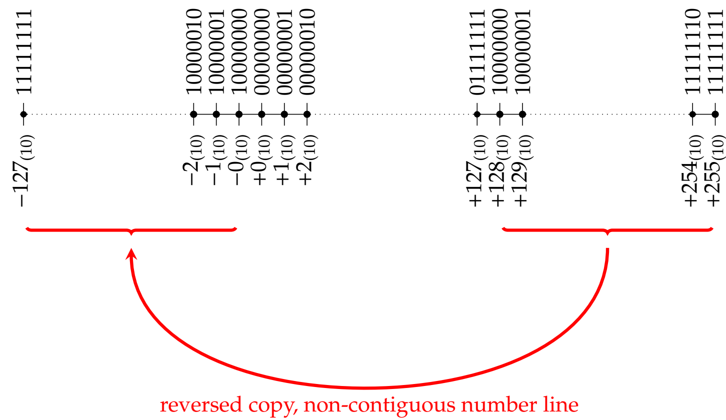
$$
\begin{array}{rcllllllll}
2223 & = & 1 \cdot 1 & + 1 \cdot 2 & + 1 \cdot 4 & + 1 \cdot 8 & + 0 \cdot 16 & + 1 \cdot 32 & + \\
 & & 0 \cdot 64 & + 1 \cdot 128 & + 0 \cdot 256 & + 0 \cdot 512 & + 0 \cdot 1024 & + 1 \cdot 2048 \\
 & = & 1 \cdot 2^0 & + 1 \cdot 2^1 & + 1 \cdot 2^2 & + 1 \cdot 2^3 & + 0 \cdot 2^4 & + 1 \cdot 2^5 & + \\
 & & 0 \cdot 2^6 & + 1 \cdot 2^7 & + 0 \cdot 2^8 & + 0 \cdot 2^9 & + 0 \cdot 2^{10} & + 1 \cdot 2^{11} \\
 & = & \multicolumn{7}{l}{\langle 1, 1, 1, 1, 0, 1, 0, 1, 0, 0, 0, 1 \rangle_{(2)}} \\
 & = & \multicolumn{7}{l}{\langle \langle 1, 1, 1, 1 \rangle_{(2)}, \langle 0, 1, 0, 1 \rangle_{(2)}, \langle 0, 0, 0, 1 \rangle_{(2)} \rangle_{(16)}} \\
 & = & \multicolumn{7}{l}{\langle 15_{(10)}, 10_{(10)}, 8_{(10)} \rangle_{(16)}} \\
 & = & \multicolumn{7}{l}{\langle F_{(16)}, A_{(16)}, 8_{(16)} \rangle_{(16)}} \\
 & = & \multicolumn{7}{l}{\langle F, A, 8 \rangle_{(16)}} \\
 & = & 15 \cdot 16^0 & + 10 \cdot 16^1 & + 8 \cdot 16^2 \\
 & = & 15 \cdot 1 & + 10 \cdot 16 & + 8 \cdot 256 \\
 & = & 2223
\end{array}
$$

are clearly valid.

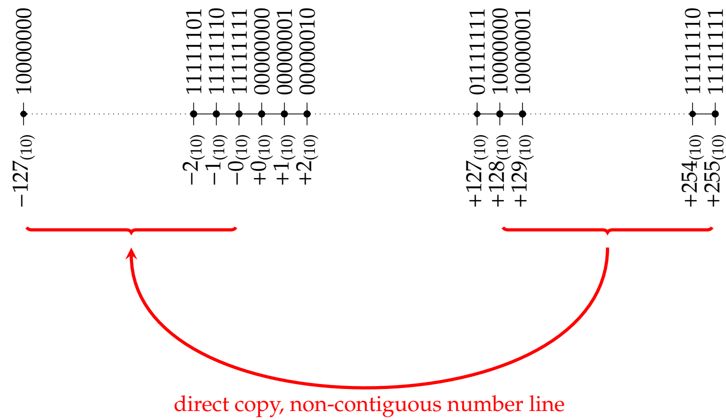
In C you can in fact write decimal literals (which is the default), and hexadecimal literals using the prefix `0x`. However, beware of literals starting with `0`: this will be interpreted as octal! For example, `012` has the same value as `10` because

$$
\begin{array}{rcl}
\texttt{012} & \mapsto & 1 \cdot 8^1 + 2 \cdot 8^0 \\
 & = & 10_{(10)} \\
 & = & 1 \cdot 10^1 + 0 \cdot 10^0 \\
 & \mapsto & \texttt{10}
\end{array}
$$

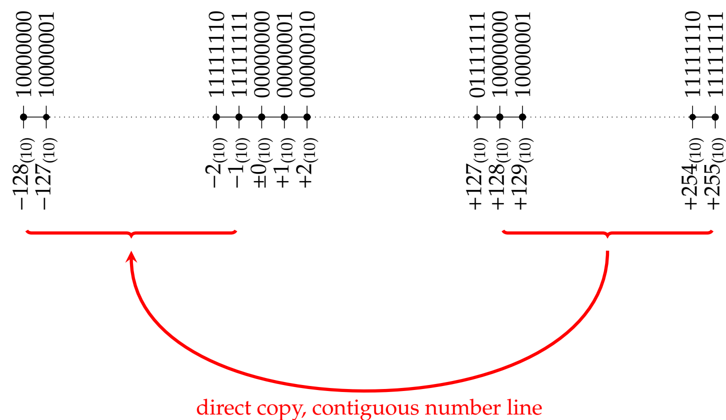
You cannot directly express binary literals in C, although doing so is possible in other languages (e.g., Python, via a `0b` prefix).



(a) *A number line for sign-magnitude representation.*



(b) *A number line for one's-complement representation.*



(c) *A number line for two's-complement representation.*

Figure 3: *Number lines illustrating the mapping of 8-bit sequences to integer values using three different representations.*

i.e., 31 Oct equals 25 Dec in the sense that 31 in base-8 equals 25 in base-10.

Put in context, we saw above that the decimal sequence $\hat{A}$ and decimal number 123 are basically the same *iff.* we interpret A in the right way. The *if* in that statement is a problem, in the sense there is ambiguity: if we follow the same reasoning as in the jokes, how do we *know* what base the literal 01111011 is written down in? It could mean the decimal number 123 (i.e., "one hundred and twenty three") if we interpret it using $b = 2$, or the decimal number 01111011 (i.e., "one million, one hundred and eleven thousand and eleven") if we interpret it using $b = 10$; clearly that is quite a difference!

To clear up this ambiguity, where necessary we write literal numbers and representations with the base appended to them. For example, we write $123_{(10)}$ to show that 123 should be interpreted in base-10, or $01111011_{(2)}$ to shows that 01111011 should be interpreted in base-2. We can now be clear, for example, that $123_{(10)} = 01111011_{(2)}$; using this notation, the two jokes become even less amusing when written simply as $10_{(2)} = 2_{(10)}$ and $31_{(8)} = 25_{(10)}$.

Example 0.37. Consider a case where $m \neq 0$, which allows negative values of i and therefore negative powers of the base: whereas $m = 0$ implies no fraction part to the resulting value, because $10^{-1} = 1/10 = 0.1$ and $10^{-2} = 1/100 = 0.01$, for example, when $m \neq 0$ we can write down numbers which *do* have fractional parts. Consider that

$$123.3125_{(10)} = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0 + 3 \cdot 10^{-1} + 1 \cdot 10^{-2} + 2 \cdot 10^{-3} + 5 \cdot 10^{-4}$$

given we have $n = 7$ digits, $m = 4$ of which capture the fractional part. Of course since the definition is the same, we can do the same thing using a different base, e.g.,

$$\begin{array}{rcl} 123.3125_{(10)} & = & 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 0 \cdot 2^{-3} + 1 \cdot 2^{-4} \\ & = & 1111011.0101_{(2)}. \end{array}$$

The **decimal point** in the former has the same meaning (i.e., as a separator between fractional and non-fractional parts) when translated into a **binary point** in the latter; more generally we call this a **fractional point** where the base is irrelevant.

Example 0.38. We mentioned previously that certain numbers are *irrational*: the definition of $\mathbb{Q}$ suggested that, in such cases, we could not find an x and y such that x/y provided the result required.

In fact, the base such numbers are represented in has some impact on their (ir)rationality. Informally, we already know that when we write 1/3 as a decimal number we have $0.3333\ldots$; the ellipsis mean the sequence recurs infinitely. 1/10, however, is rational when written as 0.1 in decimal, but irrational when written in binary; the closest approximation is $0.000110011\ldots$.

8.3 Representing integer numbers, i.e., members of $\mathbb{Z}$

The positional number systems explored above afford a flexible way to represent numbers in theory, e.g., if we just want to write down some examples. However, some (implicit) challenges exist when using them in practice. Considering the set of integers $\mathbb{Z}$, for example, we have no way to cater for a) the infinite size of this set, given the finite concrete resources available to us (which bound the number of digits we can have in a given base-b expansion), and b) the fact members of the set can be positive or negative. In even more concrete terms, consider some integer data types available in C:

$$\begin{array}{lcll} \texttt{unsigned char} & \mapsto & \mathbb{Z}_{\texttt{unsigned char}} & = \{ \quad 0, \ldots, +2^8 - 1 \} \\ \texttt{unsigned int} & \mapsto & \mathbb{Z}_{\texttt{unsigned int}} & = \{ \quad 0, \ldots, +2^{32} - 1 \} \\ \texttt{char} & \mapsto & \mathbb{Z}_{\texttt{char}} & = \{ -2^7, \ldots, 0, \ldots, +2^7 - 1 \} \\ \texttt{int} & \mapsto & \mathbb{Z}_{\texttt{int}} & = \{ -2^{31}, \ldots, 0, \ldots, +2^{31} - 1 \} \end{array}$$

This is *meant* to illustrate, for example, that the `int` data type, which one might *say* as "an integer", is in fact an *approximation* of the integers (i.e., of $\mathbb{Z}$): the range of values is finite. That said, however, why use this *particular* approximation?

We can answer this question by investigating concrete representations used in C, basing our discussion on positional number systems via use of bit-sequences (of fixed length n) to encode members of $\mathbb{Z}$. Note that where appropriate, we use colour to highlight parts of each representation that determine the sign and magnitude (or size) of the associated value; since we are representing integers, we implicitly set $m = 0$ within the general definition of a positional number system (since there is, by definition, no fractional part in an integer).

An aside: the *actual* range of C integer data types.

In describing the C data type `int` as implying an associated set (or range)

$$\mathbb{Z}_{\texttt{int}} = \{-2^{31}, \ldots, 0, \ldots + 2^{31} - 1\}$$

of values, we simplified what is, in reality, a somewhat complicated issue. In short, the C *language* specifies the above much more abstractly; the C *compiler* and *platform* (i.e., processor) make the details concrete, allowing us to reason as we did above.

It is worth looking at this issue in more detail: on one hand it is not often covered elsewhere, but, on the other hand, will help avoid making assumptions that may be (subtly, and infrequently) incorrect. Other descriptions exist, but we follow that in [9] due to the clarity of presentation. Considering integer data types only, i.e., for each type

$$T \in \{\texttt{char}, \texttt{short}, \texttt{int}, \texttt{long}, \texttt{long long}\},$$

the C *language* defines two abstract properties:

1. the signed'ness of a type T is denoted

$$S(T) = \begin{cases} 0 & \text{if } T \text{ is unsigned} \\ 1 & \text{if } T \text{ is signed} \end{cases}$$

   and allows us to destinguish between `unsigned int` and `int`, for example, and

2. the rank of a type T, denoted $R(T)$, is an abstract measure of size (and hence range); rather than a numerical value, types are simply ordered st.

$$R(\texttt{char}) < R(\texttt{short}) < R(\texttt{int}) < R(\texttt{long}) < R(\texttt{long long}).$$

The *platform* provides concrete detail, in particular assigning a width (or size) of

$$W(T) \in \{1, 2, 4, 8\}$$

bytes to each type; this is termed the **data model**. Based on use of two's-complement, we can derive the range of each type as

$$I(T) = \begin{cases} \{0, \ldots, +2^{8 \cdot W(T)} - 1\} & \text{if } S(T) = 0, \text{ st. } T \text{ is unsigned} \\ \{-2^{8 \cdot W(T) - 1}, \ldots, 0, \ldots, +2^{8 \cdot W(T) - 1} - 1\} & \text{if } S(T) = 1, \text{ st. } T \text{ is signed} \end{cases}$$

which matches our own definitions. Although the platform can select $W(T)$ for each T, a crucial restriction applies: for any types T_1 and T_2 where $R(T_1) < R(T_2)$, the property $W(T_1) \leq W(T_2)$ must hold. Put another way, we can be sure the width of `int` is less than or equal to that of `long`, *even if* those widths are not known; it cannot be the other way around, for example, st. `long` is wider than `int`.

So we *assumed* $W(\texttt{int}) = 4$ in our description, but this is *not* the only possibilty. [9, Table 1] surveys various data models, noting, for example, that

	LP32	ILP32
$W(\texttt{char})$	1	1
$W(\texttt{short})$	2	2
$W(\texttt{int})$	2	4
$W(\texttt{long})$	4	4
$W(\texttt{long long})$	8	8

are valid posibilities: if we *assume* $W(\texttt{int}) = 4$ in a program compiled and executed on a platform associated with the left-hand data model problems may well occur, whereas the right-hand data model matches.

8.3.1 Unsigned integers

Natural binary expansion

Definition 0.51. *An unsigned integer can be represented in n bits by using the natural binary expansion. That is, we have*

$$\begin{array}{rcl} \hat{x} & = & \langle x_0, x_1, \ldots, x_{n-1} \rangle \\ & \mapsto & \sum_{i=0}^{n-1} x_i \cdot 2^i \end{array}$$

for $x_i \in \{0,1\}$, *and*

$$0 \le x \le 2^n - 1.$$

Example 0.39. If $n = 8$ for example, we can represent values in the range $+0 \ldots +255$; selected cases are as follows:

$$\begin{array}{lclcr}
11111111 & \mapsto & 1 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 & = & +255_{(10)} \\
\vdots & & & \vdots & \\
10000101 & \mapsto & 1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 & = & +133_{(10)} \\
\vdots & & & \vdots & \\
10000000 & \mapsto & 1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 & = & +128_{(10)} \\
01111111 & \mapsto & 0 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 & = & +127_{(10)} \\
\vdots & & & \vdots & \\
01111011 & \mapsto & 0 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 & = & +123_{(10)} \\
\vdots & & & \vdots & \\
00000001 & \mapsto & 0 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 & = & +1_{(10)} \\
00000000 & \mapsto & 0 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 & = & +0_{(10)}
\end{array}$$

Binary Coded Decimal (BCD) BCD is an alternative method of representing unsigned integers: rather than representing the number *itself* as a bit-sequence, the idea is to write it in decimal and encode each decimal digit independently. The overall representation is the concatenation of bit-sequences which result from encoding the decimal digits:

Definition 0.52. *Consider the function*

$$f : \left\{ \begin{array}{lcl} \{0,1,\ldots,9\} & \to & \mathbb{B}^4 \\ d & \mapsto & \left\{ \begin{array}{ll} \langle 0,0,0,0 \rangle & \text{if } d = 0 \\ \langle 1,0,0,0 \rangle & \text{if } d = 1 \\ \langle 0,1,0,0 \rangle & \text{if } d = 2 \\ \langle 1,1,0,0 \rangle & \text{if } d = 3 \\ \langle 0,0,1,0 \rangle & \text{if } d = 4 \\ \langle 1,0,1,0 \rangle & \text{if } d = 5 \\ \langle 0,1,1,0 \rangle & \text{if } d = 6 \\ \langle 1,1,1,0 \rangle & \text{if } d = 7 \\ \langle 0,0,0,1 \rangle & \text{if } d = 8 \\ \langle 1,0,0,1 \rangle & \text{if } d = 9 \end{array} \right. \end{array} \right.$$

which encodes a decimal digit d into a corresponding 4-bit sequence; this function corresponds to the Simple Binary Coded Decimal (SBCD), or BCD 8421, standard. Given the decimal number

$$x = \langle x_0, x_1, \ldots, x_{n-1} \rangle_{(10)},$$

the BCD representation is

$$\hat{x} = \langle f(x_0), f(x_1), \ldots, f(x_{n-1}) \rangle.$$

Example 0.40. If $n = 8$ for example, we can represent values in the range $+0 \ldots +99999999$; selected cases are

as follows:

$$
\begin{array}{rcl}
10011001100110011001100110011001 & \mapsto & \left\langle \begin{array}{llll} \langle 1,0,0,1 \rangle, & \langle 1,0,0,1 \rangle, & \langle 1,0,0,1 \rangle, & \langle 1,0,0,1 \rangle, \\ \langle 1,0,0,1 \rangle, & \langle 1,0,0,1 \rangle, & \langle 1,0,0,1 \rangle, & \langle 1,0,0,1 \rangle \end{array} \right\rangle \\
 & \mapsto & \langle 9,9,9,9,9,9,9,9 \rangle_{(10)} \\
 & = & +99999999_{(10)} \\
 & \vdots & \\
00000000000000000000000100100011 & \mapsto & \left\langle \begin{array}{llll} \langle 1,1,0,0 \rangle, & \langle 0,1,0,0 \rangle, & \langle 1,0,0,0 \rangle, & \langle 0,0,0,0 \rangle, \\ \langle 0,0,0,0 \rangle, & \langle 0,0,0,0 \rangle, & \langle 0,0,0,0 \rangle, & \langle 0,0,0,0 \rangle \end{array} \right\rangle \\
 & \mapsto & \langle 3,2,1,0,0,0,0,0 \rangle_{(10)} \\
 & = & +123_{(10)} \\
 & \vdots & \\
00000000000000000000000000000001 & \mapsto & \left\langle \begin{array}{llll} \langle 1,0,0,0 \rangle, & \langle 0,0,0,0 \rangle, & \langle 0,0,0,0 \rangle, & \langle 0,0,0,0 \rangle, \\ \langle 0,0,0,0 \rangle, & \langle 0,0,0,0 \rangle, & \langle 0,0,0,0 \rangle, & \langle 0,0,0,0 \rangle \end{array} \right\rangle \\
 & \mapsto & \langle 1,0,0,0,0,0,0,0 \rangle_{(10)} \\
 & = & +1_{(10)} \\
00000000000000000000000000000000 & \mapsto & \left\langle \begin{array}{llll} \langle 0,0,0,0 \rangle, & \langle 0,0,0,0 \rangle, & \langle 0,0,0,0 \rangle, & \langle 0,0,0,0 \rangle, \\ \langle 0,0,0,0 \rangle, & \langle 0,0,0,0 \rangle, & \langle 0,0,0,0 \rangle, & \langle 0,0,0,0 \rangle \end{array} \right\rangle \\
 & \mapsto & \langle 0,0,0,0,0,0,0,0 \rangle_{(10)} \\
 & = & +0_{(10)}
\end{array}
$$

8.3.2 Signed integers

Sign-magnitude

Definition 0.53. *A signed integer can be represented in n bits by using the* **sign-magnitude** *approach; 1 bit is reserved for the sign (0 means positive, 1 means negative) and $n-1$ for the magnitude. That is, we have*

$$
\begin{array}{rcl}
\hat{x} & = & \langle x_0, x_1, \ldots, x_{n-1} \rangle \\
 & \mapsto & -1^{x_{n-1}} \cdot \sum_{i=0}^{n-2} x_i \cdot 2^i
\end{array}
$$

for $x_i \in \{0,1\}$, and

$$-2^{n-1}+1 \leq x \leq +2^{n-1}-1.$$

Note that there are two representations of zero (i.e., $+0$ and -0).

Example 0.41. If $n=8$, for example, we can represent values in the range $-127\ldots+127$; selected cases are as follows:

$$
\begin{array}{lclcccccccccl}
01111111 & \mapsto & -1^0 & \cdot & (& 1 \cdot 2^6 & +1 \cdot 2^5 & +1 \cdot 2^4 & +1 \cdot 2^3 & +1 \cdot 2^2 & +1 \cdot 2^1 & +1 \cdot 2^0 &) & = & +127_{(10)} \\
\vdots & & & & & & & & & & & & & & \vdots \\
01111011 & \mapsto & -1^0 & \cdot & (& 1 \cdot 2^6 & +1 \cdot 2^5 & +1 \cdot 2^4 & +1 \cdot 2^3 & +0 \cdot 2^2 & +1 \cdot 2^1 & +1 \cdot 2^0 &) & = & +123_{(10)} \\
\vdots & & & & & & & & & & & & & & \vdots \\
00000001 & \mapsto & -1^0 & \cdot & (& 0 \cdot 2^6 & +0 \cdot 2^5 & +0 \cdot 2^4 & +0 \cdot 2^3 & +0 \cdot 2^2 & +0 \cdot 2^1 & +1 \cdot 2^0 &) & = & +1_{(10)} \\
00000000 & \mapsto & -1^0 & \cdot & (& 0 \cdot 2^6 & +0 \cdot 2^5 & +0 \cdot 2^4 & +0 \cdot 2^3 & +0 \cdot 2^2 & +0 \cdot 2^1 & +0 \cdot 2^0 &) & = & +0_{(10)} \\
10000000 & \mapsto & -1^1 & \cdot & (& 0 \cdot 2^6 & +0 \cdot 2^5 & +0 \cdot 2^4 & +0 \cdot 2^3 & +0 \cdot 2^2 & +0 \cdot 2^1 & +0 \cdot 2^0 &) & = & -0_{(10)} \\
10000001 & \mapsto & -1^1 & \cdot & (& 0 \cdot 2^6 & +0 \cdot 2^5 & +0 \cdot 2^4 & +0 \cdot 2^3 & +0 \cdot 2^2 & +0 \cdot 2^1 & +1 \cdot 2^0 &) & = & -1_{(10)} \\
\vdots & & & & & & & & & & & & & & \vdots \\
11111011 & \mapsto & -1^1 & \cdot & (& 1 \cdot 2^6 & +1 \cdot 2^5 & +1 \cdot 2^4 & +1 \cdot 2^3 & +0 \cdot 2^2 & +1 \cdot 2^1 & +1 \cdot 2^0 &) & = & -123_{(10)} \\
\vdots & & & & & & & & & & & & & & \vdots \\
11111111 & \mapsto & -1^1 & \cdot & (& 1 \cdot 2^6 & +1 \cdot 2^5 & +1 \cdot 2^4 & +1 \cdot 2^3 & +1 \cdot 2^2 & +1 \cdot 2^1 & +1 \cdot 2^0 &) & = & -127_{(10)}
\end{array}
$$

One's-complement

Definition 0.54. *The* **one's-complement** *method represents a signed integer in n bits by assigning the complement of x (i.e., $\neg x$) the value $-x$. That is, given*

$$
\begin{array}{rcl}
\hat{x} & = & \langle x_0, x_1, \ldots, x_{n-1} \rangle \\
 & \mapsto & \sum_{i=0}^{n-2} x_i \cdot 2^i
\end{array}
$$

for $x_i \in \{0,1\}$, then the encoding of $\neg x$ is assumed to represent $-x$. This means we have

$$-2^{n-1} - 1 \leq x \leq +2^{n-1} - 1.$$

Note that there are two representations of zero (i.e., $+0$ and -0).

Example 0.42. If $n = 8$ for example, we can represent values in the range $-127 \ldots +127$; selected cases are as follows:

$$
\begin{array}{lclcr}
01111111 & \mapsto & 0 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 & = & +127_{(10)} \\
\vdots & & & \vdots & \\
01111011 & \mapsto & 0 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 & = & +123_{(10)} \\
\vdots & & & \vdots & \\
00000001 & \mapsto & 0 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 & = & +1_{(10)} \\
00000000 & \mapsto & 0 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 & = & +0_{(10)} \\
11111111 & \mapsto & & & -0_{(10)} \\
11111110 & \mapsto & & & -1_{(10)} \\
\vdots & & & & \\
10000100 & \mapsto & & & -123_{(10)} \\
\vdots & & & & \\
10000000 & \mapsto & & & -127_{(10)}
\end{array}
$$

Two's-complement

Definition 0.55. *A signed integer can be represented in n bits by using the* **two's-complement** *approach. The basic idea is to weight the $(n-1)$-th bit using -2^{n-1} rather than $+2^{n-1}$, and all other bits as normal. That is, we have*

$$
\begin{array}{rcl}
\hat{x} & = & \langle x_0, x_1, \ldots, x_{n-1} \rangle \\
 & \mapsto & x_{n-1} \cdot -2^{n-1} + \sum_{i=0}^{n-2} x_i \cdot 2^i
\end{array}
$$

for $x_i \in \{0,1\}$, and

$$-2^{n-1} \leq x \leq +2^{n-1} - 1.$$

Example 0.43. If $n = 8$ for example, we can represent values in the range $-128 \ldots +127$; selected cases are as follows:

$$
\begin{array}{lclcr}
01111111 & \mapsto & 0 \cdot -2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 & = & +127_{(10)} \\
\vdots & & & \vdots & \\
01111011 & \mapsto & 0 \cdot -2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 & = & +123_{(10)} \\
\vdots & & & \vdots & \\
00000001 & \mapsto & 0 \cdot -2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 & = & +1_{(10)} \\
00000000 & \mapsto & 0 \cdot -2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 & = & +0_{(10)} \\
11111111 & \mapsto & 1 \cdot -2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 & = & -1_{(10)} \\
\vdots & & & \vdots & \\
10000101 & \mapsto & 1 \cdot -2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 & = & -123_{(10)} \\
\vdots & & & \vdots & \\
10000000 & \mapsto & 1 \cdot -2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 & = & -128_{(10)}
\end{array}
$$

Given that two's-complement is the de facto choice for signed integer representation, it warrants some further explanation: it is important to grasp how the representation works.

One approach is to consider Figure 3c, which is a a number line of values in two's-complement representation. Offset a little to the left, it shows that 0 (bottom) is represented by the literal 00000000 (which is, of course, equivalent to a bit-sequence $\langle 0,0,0,0,0,0,0,0 \rangle$); reading from the point toward the right, shows *unsigned* integers up to 255 could be represented use their natural binary representation. Sometimes you see a number line wrapped into a circle, to emphasise the fact that the values it captures will wraps-around: when we reach 255 (or 11111111), and give we have $n = 8$ bits here, the next value is 0 (or 00000000) because the representation wraps-around. Toward the left of 0, it starts to be clear that two's-complement is basically "moving" the upper or right-hand range of what would be 128 to 255: by using a large, negative weight for the $(n-1)$-th bit it moves the (positive) range 128 to 255 into the (negative) range -128 to -1. This movement is direct, in the sense the order of the range is preserved; this contrasts with sign-magnitude, for example, which, per the same idea in Figure 3a, *reverses* the range as it is moved. This difference stems from the fact that two's-complement fits the concept of a positional number system naturally, whereas the same cannot be said of sign-magnitude where the sign bit is sort of a special case (i.e.,, weighted abnormally). However, subtle this point is, it *is* important. More specifically, the fact that

1. there is one representation of the value zero, and
2. as we step left or right through representations, they remain in-order wrt. the values they represent

means we can apply the *same* approach to arithmetic using signed integers represented using two's-complement as with the simpler case of unsigned integers; this is not true of sign-magnitude, for example, arguably making it less attractive as a result.

Another approach is via an appeal to intuition: if we have x and add $-x$, i.e., compute $x + (-x)$, then we intuitively expect to produce 0 as a result. The two's-complement representation satisfies this: we can see from the above that

$$
\begin{array}{rcrcccccccccc}
x & = & 2_{(10)} & \mapsto & & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
y & = & -2_{(10)} & \mapsto & & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & + \\
\hline
c & = & & & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
r & = & 0_{(10)} & \mapsto & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
\end{array}
$$

meaning that if we ignore the carry-out (which cannot be captured: we have too few bits), we get the result expected. As a by-product, this yields a useful fact:

Definition 0.56. *The term two's-complement can be used as a noun (i.e., to describe the* representation*) or a verb (i.e., to describe an* operation*): the latter case defines "taking the two's-complement of x" to mean negating x and thus computing the representation of* $-x$*. To do so, we compute* $-x \mapsto \neg x + 1$.

To see *why* this is true, first note that for an x represented in two's-complement, adding x to $\neg x$ produces -1 as a result. For example:

$$
\begin{array}{rcrcccccccccc}
x & = & 2_{(10)} & \mapsto & & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
y & = & \neg 2_{(10)} & \mapsto & & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & + \\
\hline
c & = & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
r & = & -1_{(10)} & \mapsto & & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\hline
\end{array}
$$

This should make sense, in that each corresponding i-th bit in x and $\neg x$ will be the opposite of each other: either one will be 0 and the other is 1 or vice versa, st. their sum will always be 1. The result is off-by-one, however, in the sense we produce -1 rather than the expected 0. So, if we compute

$$
\begin{array}{rcrcccccccccc}
x & = & 2_{(10)} & \mapsto & & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
y & = & \neg 2_{(10)} + 1 & \mapsto & & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & + \\
\hline
c & = & & & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\
r & = & 0_{(10)} & \mapsto & & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
\end{array}
$$

instead then we are back to the same example as above: the result is 0, so it $-x \mapsto \neg x + 1$.

8.4 Representing real numbers, i.e., members of $\mathbb{R}$

In the above, we considered concrete representations for elements in $\mathbb{Z}$. Each case used positional number systems as an underlying idea, but in different ways (so with different properties); each one coped with the infinite size of $\mathbb{Z}$ by approximating it with a bit-sequence of fixed length (i.e., using n bits). Using the same motivation, namely the observation that C yields the approximations

$$
\begin{array}{lclcl}
\texttt{float} & \mapsto & \mathbb{R}_{\texttt{float}} & \simeq & \{ -3.40 \cdot 10^{38}, \ldots, +3.40 \cdot 10^{38} \} \cup \{\pm\infty, \mathrm{NaN}\} \\
\texttt{double} & \mapsto & \mathbb{R}_{\texttt{double}} & \simeq & \{ -1.79 \cdot 10^{308}, \ldots, +1.79 \cdot 10^{308} \} \cup \{\pm\infty, \mathrm{NaN}\}
\end{array}
$$

we can apply roughly the same approach to represent $\mathbb{R}$, the set of real numbers. Since we *know* a positional number system can accommodate numbers with a fractional part (via an $m > 0$), the fact we can consider representations for $\mathbb{R}$ should not be surprising. However, the approach we use doe differs somewhat: it make senses to ignore the previous notation etc. and start afresh with *another* underlying idea. That is, we will approximate some x by taking a base-b integer m (signed or otherwise) and **scaling** it, i.e., have

$$\hat{x} \mapsto m \cdot b^e \simeq x$$

for some e. Two more concrete representations based on this idea can be described as follows:

1. if e is fixed (i.e., does not vary between different x and hence m) we have a **fixed-point** representation, whereas
2. if e is not fixed (i.e., can vary between different x and hence m) we have a **floating-point** representation.

8.4.1 Fixed-point

Definition 0.57. *The goal of a* **fixed-point** *representation is to allow expression of real numbers whose form is*

$$x = m \cdot b^{-q}$$

or, equivalently,

$$x = m \cdot \frac{1}{b^q},$$

where

- $m \in \mathbb{Z}$ *is the* **mantissa**, *and*
- $q \in \mathbb{N}$ *is the* **exponent**.

Informally, the magnitude of such a number is given by applying a scaling factor to the mantissa. Since the exponent is fixed, this essentially means interpreting m, and hence x, as two components, i.e.,

1. *a q-digit fractional component taken from the least-significant digits, and*
2. *a p-digit integral component taken from the most-significant digits*

where $n = p + q$; we use the notation $\mathbb{Q}_{p,q}$ to denote this. Abusing notation a little, we have that

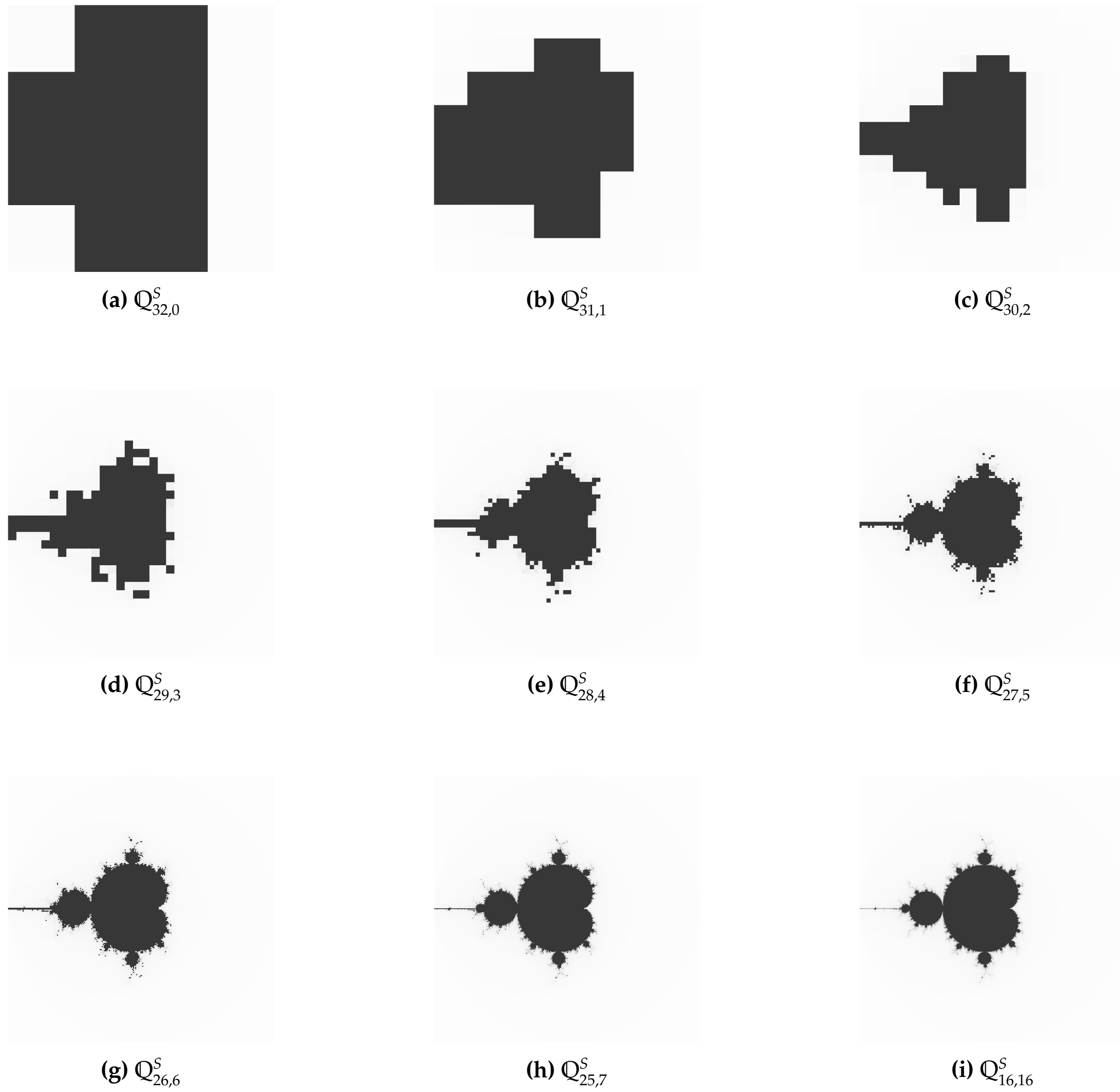
$$\begin{array}{rcl} \hat{x} & = & \langle x_0, x_1, \ldots, x_{n-1} \rangle \\ & \mapsto & \langle \underbrace{m_0, \ldots, m_{q-2}, m_{q-1}}_{q \text{ digits}}, \underbrace{m_{n-p}, \ldots, m_{n-2}, m_{n-1}}_{p \text{ digits}} \rangle_{\mathbb{Q}_{p,q}} \\ & \mapsto & m \cdot \frac{1}{b^q} \\ & \mapsto & \sum_{i=0}^{n-1} m_i \cdot b^i \cdot \frac{1}{b^q} \\ & \mapsto & \sum_{i=0}^{n-1} m_i \cdot b^{i-q} \end{array}$$

Definition 0.58. *There are some important quantities relating to a fixed-point representation $\mathbb{Q}_{p,q}$:*

- *The* **resolution** *is the smallest difference between representable values, i.e., the value $\frac{1}{b^q}$.*
- *The* **precision** *is essentially n, the number of digits in the representation; in a sense this (in combination with the resolution) governs the range of values that can be represented.*

Example 0.44. This might seem confusing, but the basic idea is as above. That is, given an integer x, we just shift the fractional point by a fixed amount to determine the associated value. Imagine we set $b = 10$, $n = 7$, $q = 4$ and write the literal

$$\hat{x} = 1233125.$$



(a) $\mathbb{Q}^{S}_{32,0}$ **(b)** $\mathbb{Q}^{S}_{31,1}$ **(c)** $\mathbb{Q}^{S}_{30,2}$

(d) $\mathbb{Q}^{S}_{29,3}$ **(e)** $\mathbb{Q}^{S}_{28,4}$ **(f)** $\mathbb{Q}^{S}_{27,5}$

(g) $\mathbb{Q}^{S}_{26,6}$ **(h)** $\mathbb{Q}^{S}_{25,7}$ **(i)** $\mathbb{Q}^{S}_{16,16}$

Figure 4: *A visualisation of the impact of increasing q, the number of fractional digits, in a fixed-point representation; the result is increased detail within the rendering of a Mandelbrot fractal.*

Interpreting x in the fixed-point representation specified by n and q means there are $q = 4$ fractional digits, i.e., 3125, and $p = n - q = 7 - 4 = 3$ integral digits, i.e., 123. Therefore

$$\hat{x} \mapsto x \cdot \frac{1}{b^q} = 1233125 \cdot \frac{1}{10^4} = 123.3125_{(10)},$$

meaning we have simply taken x and shifted the fractional point by $q = 4$ digits. Put yet *another* way, we again alter the weight associated with each digit: taken as an integer the i-th digit will be weighted by b^i, but interpreting the same digit as above means weighting it by b^{i-q}.

Example 0.45. There is a neat way to visualise the intuitive effect of adding more precision to (i.e., increasing the number of fractional digits in) a fixed-point representation. Figure 4 includes different renderings of the **Mandelbrot fractal**, named after mathematician Benoît Mandelbrot. Each rendering uses a 32-bit integer, i.e., $n = 32$, to specify a fixed-point representation but with different values of q, i.e., different numbers of fractional digits. Quite clearly, as we increase q there is more detail. Without expanding on the detail, the fractal is rendered by sampling points on a circle of radius 2 centred at the point $(0, 0)$. With no fractional digits, we can sample points (x, y) with $x, y \in \mathbb{Z}$ st. $x, y \in \{-2, -1, 0, +1, +2\}$; this is restrictive it the sense it allows only a few points. However, by adding more fractional digits we can sample many more intermediate points, e.g., $(0.5, 0.5)$ and so on, meaning more detail in the rendering.

Example 0.46. Although the definition is general enough to accommodate any choice of b, it may not be surprising that $b = 2$ is attractive: this allows us to reuse what we know about representing integers using bit-sequences, and apply it to representing real numbers using a fixed-point representation.

- We can describe an unsigned fixed-point representation based on an unsigned integer; imagine we select $n = 8$ with $p = 5$ and $q = 3$, denoted $\mathbb{Q}^U_{5,3}$. This means

  $$\begin{array}{rcl} \hat{x} &=& \langle x_0, x_1, \ldots, x_7 \rangle \\ &\mapsto& (\sum_{i=0}^{p+q-1} x_i \cdot 2^i) \cdot \frac{1}{2^q} \end{array}$$

  which produces a value in the range

  $$0 \le x \le 2^p - \frac{1}{2^q}$$

  or rather $0 \le x \le 31.875$ with a resolution of 0.125. For example

  $$\begin{array}{rcl} \hat{x} &=& 15_{(10)} \\ &=& 00001111_{(2)} \\ &\mapsto& 00001111_{(\mathbb{Q}^U_{5,3})} \\ &\mapsto& 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3} \\ &\mapsto& 1.875_{(10)} \end{array}$$

- We can describe an signed fixed-point representation based on a two's-complement signed integer; imagine we select $n = 8$ with $p = 5$ and $q = 3$, denoted $\mathbb{Q}^S_{5,3}$. This means

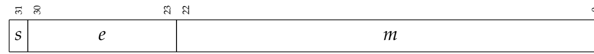
  $$\begin{array}{rcl} \hat{x} &=& \langle x_0, x_1, \ldots, x_7 \rangle \\ &\mapsto& (-x_{p+q-1} \cdot 2^{p+q-1} + \sum_{i=0}^{p+q-2} x_i \cdot 2^i) \cdot \frac{1}{2^q} \end{array}$$

  which produces a value in the range

  $$-2^{p-1} \le x \le 2^{p-1} - \frac{1}{2^q}$$

  or rather $-16 \le x \le 15.875$ with a resolution of 0.125. For example

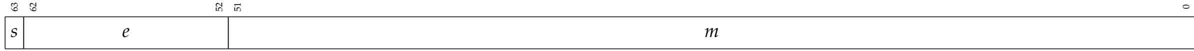
  $$\begin{array}{rcl} \hat{x} &=& 142_{(10)} \\ &=& 10001111_{(2)} \\ &\mapsto& 10001111_{(\mathbb{Q}^S_{5,3})} \\ &\mapsto& 1 \cdot -2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3} \\ &\mapsto& -14.125_{(10)} \end{array}$$



(a) 32-*bit, single-precision format as a bit-sequence.*

```
typedef struct __ieee32_t {
  uint32_t m : 23, // mantissa
           e :  8, // exponent
           s :  1; // sign
} ieee32_t;
```

(b) 32-*bit, single-precision format as a C structure.*



(c) 64-*bit, double-precision format as a bit-sequence.*

```
typedef struct __ieee64_t {
  uint64_t m : 52, // mantissa
           e : 11, // exponent
           s :  1; // sign
} ieee64_t;
```

(d) 64-*bit, double-precision format as a C structure.*

Figure 5: *Single- and double- precision IEEE-754 floating-point formats described graphically as bit-sequences and concretely as C structures.*

8.4.2 Floating-point

Definition 0.59. *The goal of a* **floating-point** *representation is to allow expression of real numbers whose form is*

$$x = -1^s \cdot m \cdot b^e$$

where

- $s \in \{0, 1\}$ *is the* **sign bit**,
- $m \in \mathbb{N}$ *is the* **mantissa**, *and*
- $e \in \mathbb{Z}$ *is the* **exponent**.

Informally, the magnitude of such a number is given by applying a scaling factor to the mantissa; since the exponent can vary, it acts to "float" the fractional point, denoted $\circ$*, into the correct position.*

We say that a number of the form

$$x = -1^s \cdot (\underbrace{m_{n-1} \circ m_{n-2} \ldots m_1\ m_0}_{n \text{ digits}}) \cdot b^e,$$

is **normalised**: *the fractional point is initially (i.e., before it is moved via the scaling factor) assumed to be after the first non-zero digit of the mantissa. Note that n determines the* **precision**.

Definition 0.60. *IEEE-754 specifies two floating-point representations (or formats); each format represents a floating-point number as a bit-sequence by concatenating together three components, i.e., the mantissa, the exponent and the sign bit. There are two features to keep in mind:*

- *Imagine x is normalised as above, since $b = 2$ we know $m_{n-1} = 1$ since the leading digit of the mantissa must be non-zero. This means we do not need to include m_{n-1} explicitly in the representation of x, the now implicit value being termed a (or the)* **hidden digit**.
- *The exponent needs a signed integer representation; one might imagine that two's-complement is suitable, but instead an approach called* **biasing** *is used. Essentially this means that the representation of x adds a constant β to the real value of e so it is always positive, i.e.,*

$$\hat{e} \mapsto e - \beta.$$

The formats are as follows:

- *The single-precision, 32-bit floating-point format allocates the least-significant 23 bits to the mantissa, the next-significant 8 bits to the exponent and the most-significant bit to the sign:*

$$\begin{array}{rcl} \hat{x} & = & \langle x_0, x_1, \ldots, x_{31} \rangle \\ & = & \langle \underbrace{m_0, m_1, \ldots, m_{22}}_{23 \textit{ bits}}, \underbrace{e_0, e_1, \ldots, e_7}_{8 \textit{ bits}}, s \rangle_{\mathbb{Q}_{IEEE-32-bit}} \\ & \mapsto & -1^s \cdot m \cdot 2^{e-127} \end{array}$$

  Note that here, $\beta = 127$.

- *The double-precision, 64-bit floating-point format allocates the least-significant 52 bits to the mantissa, the next-significant 11 bits to the exponent and the most-significant bit to the sign:*

$$\begin{array}{rcl} \hat{x} & = & \langle x_0, x_1, \ldots, x_{63} \rangle \\ & = & \langle \underbrace{m_0, m_1, \ldots, m_{51}}_{52 \textit{ bits}}, \underbrace{e_0, e_1, \ldots, e_{10}}_{11 \textit{ bits}}, s \rangle_{\mathbb{Q}_{IEEE-64-bit}} \\ & \mapsto & -1^s \cdot m \cdot 2^{e-1023} \end{array}$$

  Note that here, $\beta = 1023$.

Definition 0.61. *The IEEE floating-point representations reserve some values in order to represent special quantities. For example, reserved values are used to represent $+\infty$, $-\infty$ and NaN, or* **not-a-number***: $+\infty$ and $-\infty$ can occur when a result overflows beyond the limits of what can be represented, NaN occurs, for example, as a result of division by zero. For the single-precision, 32-bit format these special values are*

$$\begin{array}{lcl} 00000000000000000000000000000000 & \mapsto & +0 \\ 10000000000000000000000000000000 & \mapsto & -0 \\ 01111111100000000000000000000000 & \mapsto & +\infty \\ 11111111100000000000000000000000 & \mapsto & -\infty \\ 01111111100000100000000000000000 & \mapsto & NaN \\ 11111111100100010001001010101010 & \mapsto & NaN \end{array}$$

with similar forms for the double-precision, 64-bit format.

Example 0.47. Imagine we want to represent $x = 123.3125_{(10)}$ in the single-precision, 32-bit IEEE floating-point format. First we write the number in binary

$$x = 1111011.0101_{(2)}$$

before normalising it, meaning we shift it so that there is only one digit to the left of the binary point, to get

$$x = 1.1110110101_{(2)} \cdot 2^6.$$

Recalling that we do not store the implicit hidden digit (i.e., the digit to the left of the binary point), our mantissa, exponent and sign become

$$\begin{array}{rcr} m & = & 11101101010000000000000_{(2)} \\ e & = & 00000110_{(2)} \\ s & = & 0_{(2)} \end{array}$$

noting we pad both with less-significant zeros to ensure each is of the correct length. Finally, we can convert each component into a literal using their associated representations, i.e.,

$$\begin{array}{rcr} \hat{m} & = & 11101101010000000000000 \\ \hat{e} & = & 10000101 \\ \hat{s} & = & 0 \end{array}$$

noting that we bias e (i.e., add 127 to $e = 6$) to get the result, and concatenate the components into the single literal

$$\hat{x} = 01000010111101101010000000000000.$$

Definition 0.62. *Consider a case where the result of some arithmetic operation (or conversion) requires more digits of precision than are available. That is, it cannot be represented exactly within the n digits of mantissa available. To combat this problem, we can use the concept of* **rounding**. *For example, you probably already know that if we only have two digits of precision available then*

- $1.24_{(10)}$ *is rounded to* $1.2_{(10)}$ *because the last digit is less than five, while*
- $1.27_{(10)}$ *is rounded to* $1.3_{(10)}$ *because the last digit is greater than or equal to five.*

Such a **rounding mode** *is essentially a rule that takes the ideal result, i.e., the result if one could use infinite precision, to the most suitable representable result.*

The IEEE-754 specification mandates the availability of four rounding modes. In each case, the idea is to imagine the ideal result x is written using an $l > n$ digit mantissa m, i.e.,

$$x = -1^s \cdot (\underbrace{m_{l-1} \circ m_{l-2} \ldots m_1 \, m_0}_{l \text{ digits}}) \cdot b^e.$$

To round x, we copy the most-significant n digits of m to get

$$x' = -1^s \cdot (\underbrace{m'_{n-1} \circ m'_{n-2} \ldots m'_1 \, m'_0}_{n \text{ digits}}) \cdot b^e$$

where $m'_i = m_{i+l-n}$, then "patch" $m'_0 = m_{l-n}$ according to rules given by the rounding mode. Within the following, we offer some decimal examples for clarity (minor alterations apply in binary), rounding for $n = 2$ digits of precision in each example. Note that the the C standard library offers access to these features, using constant values `FE_TONEAREST`, `FE_UPWARD`, `FE_DOWNWARD`, and `FE_TOWARDZERO` respectively to refer to the rounding modes themselves. For example, the `rint` function rounds a floating-point value using the currently selected IEEE-754 rounding mode; this can be inspected and set using the `fegetround` and `fesetround` functions.

Definition 0.63. *Sometimes termed* **Banker's Rounding**, *the* **round to nearest** *mode alters basic rounding to provide more specific treatment when the ideal result is exactly half way between representable results, i.e., when* $m'_0 = 5$. *It can be described via the following rules:*

- *If* $m_{l-n-1} \leq 4$, *then do not alter* m'_0.
- *If* $m_{l-n-1} \geq 6$, *then alter* m'_0 *by adding one.*
- *If* $m_{l-n-1} = 5$ *and at least one of the trailing digits from* m_{l-n-2} *onward is non-zero, the alter* m'_0 *by adding one.*
- *If* $m_{l-n} = 5$ *and all of the trailing digits from* m_{l-n-1} *onward are zero, then alter* m'_0 *to the nearest even digit. That is:*
  - *if* $m'_0 \equiv 0 \pmod 2$ *then do not alter it, but*
  - *if* $m'_0 \equiv 1 \pmod 2$ *then alter it by adding one.*

Example 0.48. Using the round to nearest mode, we find that

- $1.24_{(10)}$ rounds to $1.2_{(10)}$,
- $1.27_{(10)}$ rounds to $1.3_{(10)}$,
- $1.251_{(10)}$ rounds to $1.3_{(10)}$,
- $1.250_{(10)}$ rounds to $1.2_{(10)}$, and
- $1.350_{(10)}$ rounds to $1.4_{(10)}$.

Definition 0.64. *Sometimes termed* **ceiling**, *the* **round toward** $+\infty$ *mode can be described via the following rules:*

- *If x is positive (i.e.,* $s = 0$*), if* m_{l-n-1} *is non-zero then alter* m'_0 *by adding one.*
- *If x is negative (i.e.,* $s = 1$*), the trailing digits from* m_{l-n-1} *onward are discarded.*

Example 0.49. Under the round toward $+\infty$ mode, we find that

- $1.24_{(10)}$ rounds to $1.3_{(10)}$,
- $1.27_{(10)}$ rounds to $1.3_{(10)}$,
- $1.20_{(10)}$ rounds to $1.2_{(10)}$,
- $-1.24_{(10)}$ rounds to $-1.2_{(10)}$,
- $-1.27_{(10)}$ rounds to $-1.2_{(10)}$, and
- $-1.20_{(10)}$ rounds to $-1.2_{(10)}$.

Definition 0.65. *Sometimes termed* **floor**, *the* **round toward** $-\infty$ *mode can be described via the following rules:*

- *If x is positive (i.e., $s = 0$), the trailing digits from m_{l-n-1} onward are discarded.*
- *If x is negative (i.e., $s = 1$), if m_{l-n-1} is non-zero then alter m'_0 by adding one.*

Example 0.50. Under the round toward $-\infty$ mode, we find that

- $-1.24_{(10)}$ rounds to $-1.3_{(10)}$,
- $-1.27_{(10)}$ rounds to $-1.3_{(10)}$,
- $-1.20_{(10)}$ rounds to $-1.2_{(10)}$,
- $1.24_{(10)}$ rounds to $1.2_{(10)}$,
- $1.27_{(10)}$ rounds to $1.2_{(10)}$, and
- $1.20_{(10)}$ rounds to $1.2_{(10)}$.

Definition 0.66. *The* **round toward zero** *mode operates as round toward $-\infty$ for positive numbers and as round toward $+\infty$ for negative numbers.*

Example 0.51. Under the round toward zero mode, we find that

- $1.27_{(10)}$ rounds to $1.2_{(10)}$,
- $1.24_{(10)}$ rounds to $1.2_{(10)}$,
- $1.20_{(10)}$ rounds to $1.2_{(10)}$,
- $-1.27_{(10)}$ rounds to $-1.2_{(10)}$,
- $-1.24_{(10)}$ rounds to $-1.2_{(10)}$, and
- $-1.20_{(10)}$ rounds to $-1.2_{(10)}$.

Example 0.52. The (slightly cryptic) C program in Figure 6 offers a practical demonstration that floating-point works as expected. The idea is to "overlap" a single-precision, 32-bit floating-point value called `x` with an instance of the `ieee32_t` structure called `y`; `main` creates an instance of this union, calling it `t`. Since we can access individual fields within `t.y` (e.g., the sign bit `t.y.s`, or the mantissa `t.y.m`), we can observe the effect altering them has on the value of `t.x`. Compiling and executing the program gives the following output:

```
+2.800000 0 80 333333
-2.800000 1 80 333333
-5.600000 1 81 333333
     +nan 0 FF 400000
     +inf 0 FF 000000
```

The question is, what on earth does this mean? We can answer this by looking at the each part of the program (each concluding with a call to `printf` that produces the lines of output):

- `t.x` is set to $2.8_{(10)}$, and then `t.x` and each component of `t.y` is printed. The output shows that

  $$\begin{array}{lclcr} \texttt{t.y.s} & = & 0_{(16)} & \mapsto & 0 \\ \texttt{t.y.e} & = & 80_{(16)} & \mapsto & 10000000 \\ \texttt{t.y.m} & = & 333333_{(16)} & \mapsto & 01100110011001100110011 \end{array}$$

  Accounting for the bias and including the hidden bit, this of course represents the value

  $$-1^0 \cdot 1.01100110011001100110011_{(2)} \cdot 2^1$$

  or $2.8_{(10)}$ as expected.

```
typedef union __view32_t {
  float    x;
  ieee32_t y;
} view32_t;

typedef union __view64_t {
  double   x;
  ieee32_t y;
} view64_t;
```

(a) *Two unions which "overlap" the representations of an actual floating-point field* `x` *with an instance* `y` *of the structure(s) defined in Figure 5.*

```
int main( int argc, char* argv[] ) {
  view32_t t;

  t.x   = 2.8;
  printf( "%+9f %01X %02X %06X\n", t.x, t.y.s, t.y.e, t.y.m );

  t.y.s = 0x01;
  printf( "%+9f %01X %02X %06X\n", t.x, t.y.s, t.y.e, t.y.m );

  t.y.e = 0x81;
  printf( "%+9f %01X %02X %06X\n", t.x, t.y.s, t.y.e, t.y.m );

  t.y.s = 0x00;
  t.y.e = 0xFF;
  t.y.m = 0x400000;
  printf( "%+9f %01X %02X %06X\n", t.x, t.y.s, t.y.e, t.y.m );

  t.y.s = 0x00;
  t.y.e = 0xFF;
  t.y.m = 0x000000;
  printf( "%+9f %01X %02X %06X\n", t.x, t.y.s, t.y.e, t.y.m );

  return 0;
}
```

(b) *A driver function* `main` *that uses an instance* `x` *of* `view32_t` *to demonstrate how manipulating fields in* `t.y` *impacts on the value of* `t.x`*.*

Figure 6: *A short C program that performs direct manipulation of IEEE floating-point numbers.*

- `t.y.s` is set to $01_{(10)} = 1_{(10)}$, and then `t.x` and each component of `t.y` is printed. We expect that setting the sign bit to 1 rather than 0 will change `t.x` from being positive to negative; this is confirmed by the output, which shows `t.x` is equal to $-2.8_{(10)}$ as expected.
- `t.y.e` is set to $81_{(10)} = 129_{(10)}$, and then `t.x` and each component of `t.y` is printed. We expect that setting the exponent to 129 rather than 128 will double `t.x` (the unbiased value of the exponent is now $129 - 127 = 2$ st. the mantissa is scaled by $2^2 = 4$ rather than $2^1 = 2$); this is confirmed by the output, which shows `t.x` is equal to $-5.6_{(10)}$ as expected.
- `t.y.s`, `t.y.e` and `t.y.m` are set to reserved values corresponding to NaN and $+\infty$.

8.5 Representing characters

So far we have examined techniques to represent numbers, but clearly we might want to work with other types of data; a computer can process all manner of data such as emails, images, music and so on. The approach used to represent **characters** (or letters) is a good example: basically we just need a way translate from what we want into a numerical representation (which we already know how to deal with) and back again. More specifically, we need two functions: Ord(x) which takes a character x and gives us back the corresponding numerical representation, and Chr(y) which takes a numerical representation y and gives back the corresponding character. But how should the functions work? Fortunately, people have thought about this problem for us and provided standards we can use. One of the oldest and most simple is the **American Standard Code for Information Interchange (ASCII)**, pronounced "ass key".

ASCII has a rich history, but was developed to permit communication between early teleprinter devices. These were like a combination of a typewriter and a telephone, and were able to communicate text to each other before innovations such as the fax machine. Later, but long before monitors and graphics cards existed, similar devices allowed users to send input to early computers and receive output from them. Figure 8 shows the 128-entry ASCII table which tells us how characters are represented as numbers. Of the entries, 95 are printable characters we can instantly recognise (including *SPC* which is short for "space"). There are also 33

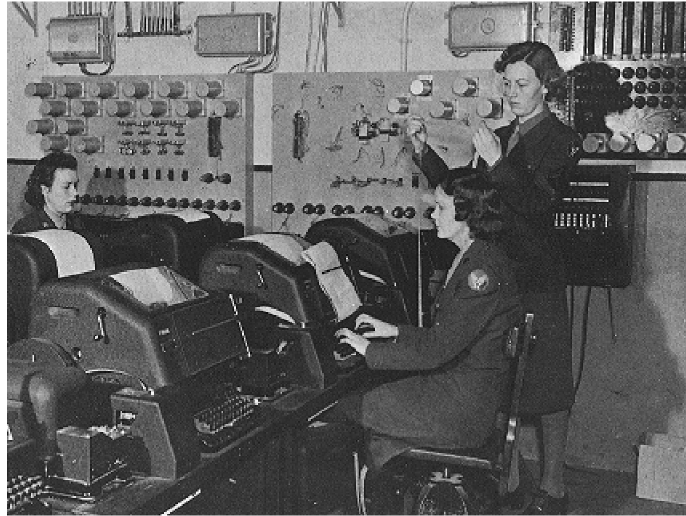


Figure 7: *A teletype machine being used by UK-based Royal Air Force (RAF) operators during WW2 (public domain image, source: http://en.wikipedia.org/wiki/File:WACsOperateTeletype.jpg).*

y Ord(x)	Chr(y) x	y Ord(x)	Chr(y) x	y Ord(x)	Chr(y) x	y Ord(x)	Chr(y) x
0	*NUL*	1	*SOH*	2	*STX*	3	*ETX*
4	*EOT*	5	*ENQ*	6	*ACK*	7	*BEL*
8	*BS*	9	*HT*	10	*LF*	11	*VT*
12	*FF*	13	*CR*	14	*SO*	15	*SI*
16	*DLE*	17	*DC1*	18	*DC2*	19	*DC3*
20	*DC4*	21	*NAK*	22	*SYN*	23	*ETB*
24	*CAN*	25	*EM*	26	*SUB*	27	*ESC*
28	*FS*	29	*GS*	30	*RS*	31	*US*
32	*SPC*	33	!	34	"	35	#
36	$	37	%	38	&	39	'
40	(	41	)	42	*	43	+
44	,	45	-	46	.	47	/
48	0	49	1	50	2	51	3
52	4	53	5	54	6	55	7
56	8	57	9	58	:	59	;
60	<	61	=	62	>	63	?
64	@	65	A	66	B	67	C
68	D	69	E	70	F	71	G
72	H	73	I	74	J	75	K
76	L	77	M	78	N	79	O
80	P	81	Q	82	R	83	S
84	T	85	U	86	V	87	W
88	X	89	Y	90	Z	91	[
92	\	93	]	94	^	95	_
96	`	97	a	98	b	99	c
100	d	101	e	102	f	103	g
104	h	105	i	106	j	107	k
108	l	109	m	110	n	111	o
112	p	113	q	114	r	115	s
116	t	117	u	118	v	119	w
120	x	121	y	122	z	123	{
124	|	125	}	126	~	127	*DEL*

Figure 8: *A table describing the printable ASCII character set.*

others which represent non-printable control characters: originally, these would have been used to control the teleprinter rather than to have it print something. For example, the *CR* and *LF* characters (short for "carriage return" and "line feed") would combine to move the print head onto the next line; we still use these characters to mark the end of lines in text files. Other control characters also play a role in modern computers. For example, the *BEL* (short for "bell") characters play a "ding" sound when printed to most UNIX terminals, we have keyboards with keys that relate to *DEL* and *ESC* (short for "delete" and "escape") and so on.

Since there are 128 entries in the table, ASCII characters can be and are represented by 8-bit bytes. However, notice that $2^7 = 128$ and $2^8 = 256$ so in fact we *could* represent 256 characters: essentially one of the bits is not used by the ASCII encoding. Specific computer systems sometimes use the unused bit to permit use of an "extended" ASCII table with 256 entries; the extra characters in the this table can be used for special purposes. For example, foreign language characters are often defined in this range (e.g., é or ø), and "block" characters are included for use by artists who form text-based pictures. However, the original use of the unused bit was as an error detection mechanism.

Given the table, we can see that for example that $\textsc{Chr}(104) = \text{`h'}$, i.e., if we see the number 104 then this represents the character 'h'. Conversely we have that $\textsc{Ord}(\text{`h'}) = 104$. Although in a sense any consistent translation between characters and numbers like this would do, ASCII has some useful properties. Look specifically at the alphabetic characters:

- Imagine we want to test if one character x is alphabetically before some other y. The way the ASCII translation is specified, we can simply compare their numeric representation. If we find $\textsc{Ord}(x) < \textsc{Ord}(y)$ then the character x is before the character y in the alphabet. For example 'a' is before 'c' because

$$\textsc{Ord}(\text{`a'}) = 97 < 99 = \textsc{Ord}(\text{`c'}).$$

- Imagine we want to convert a character x from lower-case into upper-case. The lower-case characters are represented numerically as the contiguous range $97 \ldots 122$; the upper-case characters as the contiguous range $65 \ldots 90$. So we can covert from lower-case into upper-case simply by subtracting 32. For example

$$\textsc{Chr}(\textsc{Ord}(\text{`a'}) - 32) = \text{`A'}.$$

9 A conclusion: steps toward a digital logic

If we were to summarise all the pieces of theory accumulated above, the list would be roughly as follows:

1. We know that we can define Boolean algebra, which gives us a) a set of values (i.e., $\mathbb{B} = \{0, 1\}$), b) a set of (unary and binary) operators (i.e., NOT, AND, and OR), and c) a set of axioms. This means we can construct Boolean expressions and manipulate them while preserving their meaning.

2. We can describe Boolean functions of the form

$$\mathbb{B}^n \rightarrow \mathbb{B}$$

and hence *also* construct more general functions of the form

$$\mathbb{B}^n \rightarrow \mathbb{B}^m$$

using m separate functions whose outputs are concatenated together. It therefore makes sense that NOT, AND, and OR are well-defined for the $\mathbb{B}^n$ as well as $\mathbb{B}$: we **overload** AND, for example, and write $r = x \wedge y$ as a short-hand for $r_i = x_i \wedge y_i$ where $0 \leq i < n$.

3. We can represent various objects, such as numbers using sequences of bits. Since we can describe Boolean functions of the form

$$\mathbb{B}^n \rightarrow \mathbb{B}^m,$$

we can construct such functions that perform arithmetic with numbers we are representing. Imagine, for example, we have two integers x and y represented by the n-bit sequences $\hat{x}$ and $\hat{y}$. To compute the (integer) sum of x and y, all we need is a Boolean function

$$f : \mathbb{B}^n \times \mathbb{B}^n \rightarrow \mathbb{B}^n$$

st.

$$\hat{r} = f(\hat{x}, \hat{y}) \quad \mapsto \quad x + y,$$

i.e., whose output $\hat{r}$ represents $r = x + y$.

What we end up with is the ability to perform meaningful computation; fairly simple computation, granted, but computation none the less. Fundamentally, this is what computers are: they are just devices that perform computation. So if you follow through all the theory, we have developed as "blueprint" for how to build a real computer. That is, we have a link (however tenuous) between a theoretical model of computation based on Mathematics and the first steps toward a practical realisation of that model.

References

[1] G. Boole. *An investigation of the laws of thought*. Walton & Maberly, 1854 (see pp. 27, 17).

[2] D. Cohen. "On Holy Wars and a Plea for Peace". In: *IEEE Computer* 14.10 (1981), pp. 48–54 (see pp. 37, 27).

[3] D. Goldberg. "What Every Computer Scientist Should Know About Floating-Point Arithmetic". In: *ACM Computing Surveys* 23.1 (1991), pp. 5–48.

[4] C. Petzold. *Code: Hidden Language of Computer Hardware and Software*. Microsoft Press, 2000.

[5] E.L. Post. "Introduction to a General Theory of Elementary Propositions". In: *American Journal of Mathematics* 43.3 (1921), pp. 163–185 (see pp. 29, 19).

[6] C.E. Shannon. "A mathematical theory of communication". In: *Bell System Technical Journal* 27.3 (1948), pp. 379–423 (see pp. 35, 25).

[7] C.E. Shannon. "A Symbolic Analysis of Relay and Switching Circuits". In: *Transactions of the American Institute of Electrical Engineers (AIEE)* 57.12 (1938), pp. 713–723 (see pp. 27, 17).

[8] H.M. Sheffer. "A set of five independent postulates for Boolean algebras, with applications to logical constants". In: *Transactions of the American Mathematical Society* 14.4 (1913), pp. 481–488 (see pp. 29, 19).

[9] C. Wressnegger et al. "Twice the Bits, Twice the Trouble: Vulnerabilities Induced by Migrating to 64-Bit Platforms". In: *Computer and Communications Security (CCS)*. 2016, pp. 541–552 (see pp. 47, 37).