UNIVERSITY OF BRISTOL
DEPARTMENT OF COMPUTER SCIENCE
http://www.cs.bris.ac.uk

A Practical Introduction to Computer Architecture

Dan Page
⟨page@cs.bris.ac.uk⟩

MATHEMATICAL PRELIMINARIES

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

– J. 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***.*

Since we use them so naturally, 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. 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 since it could be true or false depending on the context, or your point of view: 90°C probably is too hot for body temperature but probably not for a cup of coffee. Finally, some statements look like propositions but cannot be evaluated because they are paradoxical. The most famous example of this situation is the 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 since he said he was! In terms of the statement, we cannot be sure of the truth value so this is not normally classed as a proposition.

As stated above, a proposition can contain variables: by **assigning** the variable a value, we can evaluate the corresponding truth 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. In fact, we can represent the *entire* proposition as a variable; this is essentially what a propositional function is:

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 $10 = 20$.

So we might write

$$g : \text{"the temperature is 90°C"}$$

as a short-hand: g is representing a longer proposition, but works the same way in the sense that if g tells us the truth value of said proposition. In this case g has no free variables, but we can extend our existing example to write

$$h(x) : \text{"}x\text{°C equals 90°C"}.$$

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

$$h(10) = \text{"10°C equals 90°C"}$$

which can be evaluated to false.

1.1 Connectives

Definition 0.3 *A* **connective** *binds together single, atomic propositions into a compound proposition. 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.

A proposition involving connectives is built from **terms**; the connectives join together terms into an **expression**. 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**

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.

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.

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

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

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.

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

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

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 we 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 that x = "I am unhealthy" and y = "I will die". As such, $x \Rightarrow$ 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).

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 *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.*

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***.*

Some subtleties emerge when trying to prove two expressions are logically equivalent. However, we will skirt around these. For our purposes it suffices to simply enumerate all possible values each variable can take, and check the two expressions produce identical truth values in all cases. In practise this can be hard since with n variables there will be 2^n possible assignments, an amount which grows quickly as n grows! More formally, two expressions x and y are only equivalent if $x \equiv y$ can be proved a tautology.

1.2 Quantifiers

Definition 0.7 *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, use of quantifiers can be roughly viewed as moving us from propositional logic into predicate (or first-order) logic (with second-order logic a further extension that allows quantification of relations as well as variables).

Put more simply, 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. Consider the following example:

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

which we can rewrite symbolically as

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

In this case, x is bound by the $\exists$ quantifier; we are asserting that for some value of x it is true that $x \equiv 0 \pmod 2$. To make the expression true just one of these values needs to make the term $x \equiv 0 \pmod 2$ true. The assignment $x = 2$ satisfies this so the expression is true. As another example, consider the expression

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

which we rewrite

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

Here we are making a more general assertion about x by saying that for all x, it is true that $x \equiv 0 \pmod 2$. To decide if this particular expression is false, we need simply to find an x such that $x \not\equiv 0 \pmod 2$. This is easy (since any odd value of x is good enough), so the expression is false.

2 Sequences

Definition 0.8 *A* **sequence** *is an ordered collection of* **elements** *which can be of any (but typically homogeneous) type, and where the order of elements is important. We refer to a specific element using an* **index***; the i-th element of a sequence X is denoted X_i. The* **size** *or* **length** *of a sequence, denoted $|X|$ (or sometimes $\#X$ elsewhere), is the number of elements it contains.*

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***, when the number of elements is irrelevant. Note that the case of $n = 0$, i.e., a 0-tuple, represents an empty sequence, and that we usually term the specific case of $n = 2$ a* **pair***. From here on, we use the terms sequence and tuple as an informal device to distinguish between cases where elements are (potentially) of homogeneous and heterogeneous type respectively.*

2.1 Definition

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 know for example that $|A| = 4$ and that $A_0 = 0$, $A_1 = 3$, $A_2 = 1$ and $A_3 = 2$. Each element in the sequence A is a number, but we might just as well 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 $A_0 \neq C_0$ and so on. The elements in A are all numbers while the elements in B are all characters: the sequences are both homogeneous in that the types of elements are the same (within each case). When we define a tuple, we can have elements of different types. For example,

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

has one element which is a number and one which is a character: the tuple is heterogeneous in that the types of element differ. Notice the slight difference in how the two are written in terms of the bracketing style.

It can make sense to avoid enumerating every element of a sequence or tuple within the definition. For example we might rewrite the sequence B as

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

where the continuation dots written as ... represent the elements 'c' and 'd'. Another example is the sequence E written as

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

When we used ... in the definition of B, there was a well defined start and end to the sequence. However, with E we use ... to represent elements either we do not know, or which do not matter: since there is no end to the sequence we cannot necessarily fill in the ... part appropriately as before. Note that this also means $|E|$ might be infinite or simply unknown.

2.2 Operations

It is possible to join together, or **concatenate**, two sequences For example, imagine we start with two 4-element sequences

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

and want to join them together. Their concatenation is denoted

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

Notice that the result H is an 8-element sequence, where $H_{0\ldots3}$ are those from F and $H_{4\ldots7}$ are those from G.

3 Sets

Definition 0.9 *A* **set** *is an unordered collection of* **elements***; the elements may only occur once (otherwise we have a* **bag** *or* **multi-set***), and are typically of any (but homogeneous) type.*

The **size** *or* **cardinality** *of a set is the number of elements it contains; for a set X, this is denoted $|X|$ (or sometimes $\#X$ elsewhere). 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 means the elements can potentially be other sets. Russell's paradox, discovered by mathematician Bertrand Russell in 1901, is a problem with formal set theory that results from this fact. The paradox is similar to the liar paradox seen earlier and is easily stated by considering 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 Definition

A set can be defined using one of several methods. The most straightforward method is to enumerate the elements themselves. For example, we might define the set of integers between two and eight (inclusive) as

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

In this case, we know for example that $|A| = 7$ and that $2 \in A$ but $9 \notin A$, i.e., 2 is a member of the set A, but 9 is not. Since the ordering of elements does not matter, only their membership or non-membership, we can define

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

and be safe in the knowledge that $A = B$.

It can sometimes makes sense to avoid enumerating every element of a set within the definition. For example we might rewrite the set A as

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

where the continuation dots written as ... represent the elements 4, 5 and 6: we assume that the ... part can be filled in appropriately. That is, it should always be clear and unambiguous what ... should mean. In this case, the definition of A uses ... simply as a short-hand; in other cases this approach is a necessity. Imagine we want to define the set of even integers greater than or equal to two: this set has an infinite size, so we are forced to defined it as

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

In essence, ... now represents the elements from 10 through to infinity and hence save quite a lot of space!

Definition 0.10 *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.*

We can also use an alternative way to specify the elements; this is sometimes called **set builder** notation. Basically, we specify the elements in the set using f, a propositional function:

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

One should read this as "all elements $x \in \mathcal{U}$ such that the proposition $f(x)$ is true". Using set builder notation we can define sets in a more programmatic manner, for example

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

and

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

define the same sets as we explicitly wrote down above.

Definition 0.11 *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 integer* $x \in \mathbb{Z}$ *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,* $22/7$.

3.2 Operations

Definition 0.12 *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* $X \supseteq Y$.

Note that every set is a sub-set and super-set of itself and that $X = Y$ *only if* $X \subseteq Y$ *and* $Y \subseteq X$. *If* $X \neq Y$, *we use the terms* **proper sub-set** *and* **proper super-set** *and write* $Y \subset X$ *and* $X \supset Y$ *respectively.*

Definition 0.13 *For sets* X *and* Y, *we have that*

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

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

Figure 1 details four Venn diagrams which describe how the union, intersection, difference and complement operations work. The shaded areas of each Venn diagram represent the elements which are in the resulting set. For example, in the diagram for $A \cup B$ the shaded area covers all of the sets A and B: the result contains all elements in either A or B or both. As a simple concrete example, consider the sets

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

where the universal set is

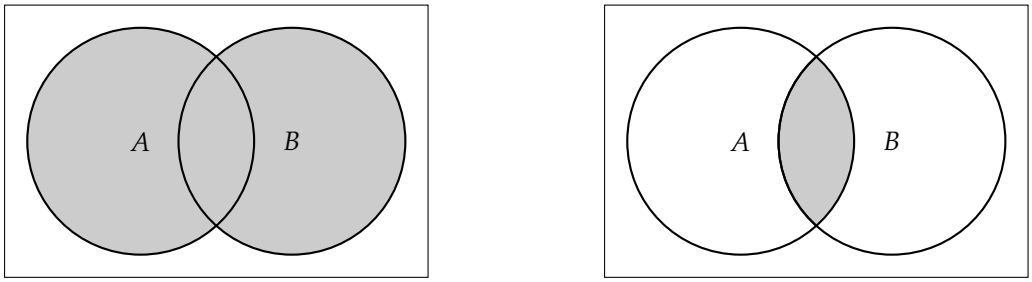
$$\mathcal{U} = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}.$$

Figure 2 shows membership in various settings; recall that those elements within a given region are members of that set. Firstly, we can take the union of A and B as $A \cup B = \{1, 2, 3, 4, 5, 6\}$ which contains all the elements which are either members of A or B or both. Note that elements 3 and 4 do not appear twice in the result. The intersection of A and B can be calculated as $A \cap B = \{3, 4\}$ since these are the elements that are members of both A and B. The difference between A and B, that is the elements in A that are not in B, is $A - B = \{1, 2\}$. Finally, the complement of A is all numbers which are not in A, that is $\overline{A} = \{5, 6, 7, 8, 9, 10\}$.

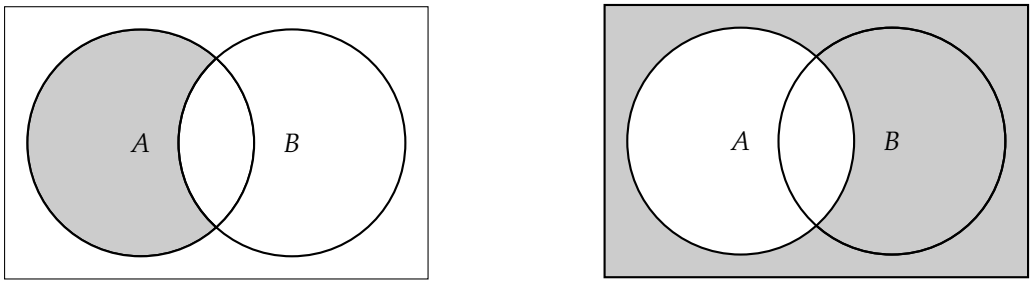
The union and intersection operations preserve a law of cardinality called the **principle of inclusion** in the sense that we can calculate the cardinality of the output from the cardinality of the inputs as

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

This property has a simple intuition in that elements in both A and B will be counted twice, and hence need subtraction via the last term. We can even check it: in our example above $|A| = 4$ and $|B| = 4$. Checking our results we have that $|A \cup B| = 6$ and $|A \cap B| = 2$ and so by the principle of inclusion we have $6 = 4 + 4 - 2$.



(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.*

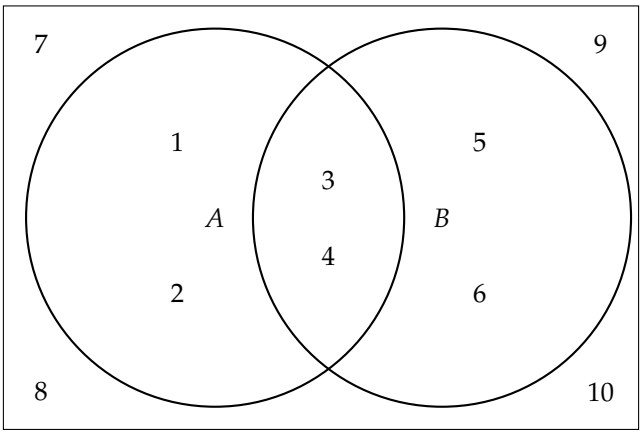


Figure 2: *An example Venn diagram showing membership of two sets.*

3.3 Products

Definition 0.15 *The* **Cartesian 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.

The Cartesian product of a set X with itself n times is denoted X^n; to be complete, we define $X^0 = \emptyset$ and $X^1 = X$. Finally, by writing X^* we mean the Cartesian product of X with itself a finite number of times.

For example, 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 whose length is two and whose elements are taken from A.

4 Functions

Definition 0.16 *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*

$$f : X \to Y$$

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**.

As a concrete example, consider a function Inv which takes an integer x as input and produces the rational number $1/x$ as output:

$$\textsc{Inv} : \begin{cases} \mathbb{Z} & \to & \mathbb{Q} \\ x & \mapsto & 1/x \end{cases}$$

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

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

and function behaviour

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

separately. In either case the domain of Inv is $\mathbb{Z}$ since it takes integers as input; the codomain is $\mathbb{Q}$ since it produces rational numbers 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.

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

$$f : A \times A \to B$$

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 can be said to two input values "packaged up" as a single pair. As a concrete example consider the function

$$\textsc{Max} : \begin{cases} \mathbb{Z} \times \mathbb{Z} & \to & \mathbb{Z} \\ \langle x, y\rangle & \mapsto & \begin{cases} x & \text{if } x > y \\ y & \text{otherwise} \end{cases} \end{cases}$$

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.

4.1 Composition

Definition 0.17 *Given two functions $f : X \to Y$ and $g : Y \to Z$, the* **composition** *of f and g is denoted*

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

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

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

4.2 Properties

Definition 0.18 *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 0.19 *The* **identity function** *$\mathcal{I}$ on a set X is defined by*

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

so that it maps all elements to themselves. Given two functions f and g defined by $f : X \to Y$ and $g : Y \to X$, 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, essentially reversing the original function. It is easy to see not all functions have an inverse. For example, if the function is not injective then there will be more than one potential pre-image for the inverse of any image.

At first glance, it might seem like our example functions both have inverses but they do not. For example, given some value $1/x$, we can certainly find x but we have already said that numbers like 2/3 also exist in the codomain so we cannot invert all the values we might come across. However, consider the example of a successor function on integers

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

which takes an integer x as input and produces $x + 1$ as output. The function is bijective since the codomain and range are the same and no two integers have the same successor. Thus we have an inverse and it is easy to describe as

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

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.

4.3 Relations

Definition 0.20 *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**.

The easiest way to think about this is to consider a concrete example such as the case where our set is $A = \{1,2,3,4\}$ such that the 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

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

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) in the sense that we can pick out the pairs (x, y)

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

where $\textsc{Equ}(x,y) = \textbf{true}$. For members of A, say $x, y, z \in A$, we have that $\textsc{Equ}(x,x) = \textbf{true}$ so the relation is reflexive. If $\textsc{Equ}(x,y) = \textbf{true}$, then $\textsc{Equ}(y,x) = \textbf{true}$ so the relation is also symmetric. Finally, if $\textsc{Equ}(x,y) = \textbf{true}$ and $\textsc{Equ}(y,z) = \textbf{true}$, then we must have that $\textsc{Equ}(x,z) = \textbf{true}$ so the relation is also transitive and hence an equivalence relation.

Now imagine we define another function

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

which tests whether one input is less than another, and consider the sub-set of A

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

of all pairs (x, y) with $x, y \in A$ such that $\textsc{Lth}(x,y) = \textbf{true}$. It cannot be that $\textsc{Lth}(x,x) = \textbf{true}$, so the relation is not reflexive; we say it is irreflexive. It also cannot be that if $\textsc{Lth}(x,y) = \textbf{true}$ then $\textsc{Lth}(y,x) = \textbf{true}$, so the relation is also not symmetric; it is anti-symmetric. However, if both $\textsc{Lth}(x,y) = \textbf{true}$ and $\textsc{Lth}(y,z) = \textbf{true}$, then $\textsc{Lth}(x,z) = \textbf{true}$ so the relation is transitive.

5 Boolean algebra

Most people are taught **elementary algebra** in school. One has

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

You might not know what the axioms are called, but you probably do know how they work; for example you probably know that for some $x, y, z \in \mathbb{Z}$ we can write $x + (y + z) = (x + y) + z$, i.e., say that addition is associative, or $x \cdot 1 = x$, i.e., the multiplicative identity of x is 1. But we can be much more general than this. When we talk about "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 concerned with what happens when we replace the set of values with things which are not strictly numbers.

Definition 0.21 *Closely following our discussion of operators, functions and relations in the context of logic, we can define an abstract algebra as including*

- *a set of values, say X,*
- *a set of* **binary operators**

$$\odot : X \times X \to X,$$

unary operators

$$\oslash : X \to X,$$

and **binary relations**

$$\ominus : X \times X \to \{\textbf{false}, \textbf{true}\},$$

and

- *a list 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) logic and set theory; the result forms what we now call **Boolean algebra** [1]. Put (rather too) simply, Boole saw that working with logic expressions is much the same as working with numeric expressions, 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 $\overline{x}$. We can see that the identity axiom applies in same way therefore:

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

Definition 0.22 *Putting everything together produces the following definition for Boolean algebra. Consider the set $\mathbb{B} = \{0, 1\}$ on which there are two binary operators*

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

and

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

and a unary operator

$$\neg : \begin{cases} \mathbb{B} & \to \quad \mathbb{B} \\ x & \mapsto \begin{cases} 1 & \text{if } x = 0 \\ 0 & \text{if } x = 1 \end{cases} \end{cases}$$

These operators are governed the following axioms

commutativity	$x \wedge y$	$\equiv$	$y \wedge x$
association	$(x \wedge y) \wedge z$	$\equiv$	$x \wedge (y \wedge z)$
distribution	$x \wedge (y \vee z)$	$\equiv$	$(x \wedge y) \vee (x \wedge z)$
identity	$x \wedge 1$	$\equiv$	x
null	$x \wedge 0$	$\equiv$	0
idempotency	$x \wedge x$	$\equiv$	x
inverse	$x \wedge \neg x$	$\equiv$	0
absorption	$x \wedge (x \vee y)$	$\equiv$	x
de Morgan	$\neg(x \wedge y)$	$\equiv$	$\neg x \vee \neg y$
commutativity	$x \vee y$	$\equiv$	$y \vee x$
association	$(x \vee y) \vee z$	$\equiv$	$x \vee (y \vee z)$
distribution	$x \vee (y \wedge z)$	$\equiv$	$(x \vee y) \wedge (x \vee z)$
identity	$x \vee 0$	$\equiv$	x
null	$x \vee 1$	$\equiv$	1
idempotency	$x \vee x$	$\equiv$	x
inverse	$x \vee \neg x$	$\equiv$	1
absorption	$x \vee (x \wedge y)$	$\equiv$	x
de Morgan	$\neg(x \vee y)$	$\equiv$	$\neg x \wedge \neg y$
equivalence	$x \equiv y$	$\equiv$	$(x \Rightarrow y) \wedge (y \Rightarrow x)$
implication	$x \Rightarrow y$	$\equiv$	$\neg x \vee y$
involution	$\neg\neg x$	$\equiv$	x

and are termed NOT, AND and OR respectively.

Notice that the $\wedge$ and $\vee$ operations in Boolean algebra behave in a similar way to $\cdot$ and $+$ in a elementary algebra. As such, $\wedge$ and $\vee$ are often termed the "product" and "sum" operations and sometimes written $\cdot$ and $+$ as a result.

Definition 0.23 *In line with what we already saw with propositional logic, it is common to add a third binary operator called XOR:*

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

This type of addition is often called a **derived operator**, *hinting at the fact it is simply a short-hand derived from operators we already have. Put another way, because*

$$x \oplus y \equiv (\neg x \wedge y) \vee (x \wedge \neg y),$$

XOR can be defined in terms of AND, OR and NOT. Other useful examples are

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

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

and

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

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

Definition 0.24 *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. For example*

$$x \wedge 0$$

is monotone because changing x does not change the result, which is always 0. Likewise

$$x \wedge 1$$

is monotone. Notice that if $x = 0$ then the result is 0 and if $x = 1$ then the result is 1; thus, changing x from 0 to 1 (resp. from 1 to 0) changes the result in the same way.

Definition 0.25 *The fact there are AND and OR forms of most axioms hints at a more general underlying* **principle of duality**. *Consider a Boolean expression e: 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* $e^D \equiv f^D$. *For example, consider the axoims 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 0.26 *The de Morgan axiom can be generalised into a* **principle of complements**. *Consider a Boolean expression e: 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.*

If we start with $e \equiv f$, *then* $\neg e \equiv \neg f$. *For 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

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

5.1 Manipulation

It might seem weird to see the equality

$$x \vee x = x$$

written down: if you think in terms of elementary algebra,

$$x + x = x$$

does not make a lot of sense! However, we can show that a similar equality does make sense when phrased in the context of Boolean algebra by **manipulating** it using our axioms:

$$\begin{array}{rcll} x \vee x & = & (x \vee x) \wedge 1 & \text{(identity)} \\ & = & (x \vee x) \wedge (x \vee \neg x) & \text{(inverse)} \\ & = & x \vee (x \wedge \neg x) & \text{(distribution)} \\ & = & x \vee 0 & \text{(inverse)} \\ & = & x & \text{(identity)} \end{array}$$

It is common to perform a similar process in order to simplify an expression in some way, e.g., so it contains less terms. A simplified expression might be more opaque, in the sense that it is not as clear what it means, but looking at things computationally it will generally be "cheaper" to evaluate. As a concrete example of simplification in action, consider the exclusive-or connective $x \oplus y$ which we can write as the more complicated expression

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

or even as

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

The first alternative above uses five connectives while the second uses only four; the second could be described as simpler therefore. One can prove that these are equivalent by constructing truth tables for them. The question is, how did we get from one to the other by manipulating the expressions?

To answer this, we simply start with one expression and apply our axiomatic laws to move toward the other. So starting with the first alternative, we try to apply the axioms until we get the second: think of the axioms like rules that allow one to rewrite the expression in a different way. To start with, we can manipulate each term which looks like $p \wedge \neg q$ as follows

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

Using this new identity, we can rewrite the whole expression as

$$\begin{array}{rcll} (y \wedge \neg x) \vee (x \wedge \neg y) & = & (x \wedge (\neg x \vee \neg y)) \vee (y \wedge (\neg x \vee \neg y)) & \\ & = & (x \vee y) \wedge (\neg x \vee \neg y) & \text{(distribution)} \\ & = & (x \vee y) \wedge \neg(x \wedge y) & \text{(de Morgan)} \end{array}$$

which gives us the second alternative we are looking for.

5.2 Functions

Definition 0.27 *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.,*

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

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 (i.e., sequences) over the set $\mathbb{B}$ rather than single elements of the set.*

This definition is more or less the same as the one we encountered earlier. For example, consider a function f that takes two inputs whose signature we can write as

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

so that 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 $r = f(x, y)$. We can specify the function itself in two ways:

1. As previously, we can 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

2. With a large number of inputs, the first approach becomes difficult to use. As a short-hand, we can 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 essentially tells us how f works, i.e., how to compute outputs rather than listing those outputs explicitly.

As a concrete example, consider a 2-input, 2-output Boolean function

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

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

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

and

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

meaning that

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

5.3 Normal (or standard) forms

Definition 0.28 *Consider a Boolean function f with n inputs. When the expression for f is written as a sum (i.e., OR) of terms which each comprise the product (i.e., AND) of a number of inputs, it is said to be in* **disjunctive normal form** *or* **Sum of Products (SoP)** *form; the terms in this expression are called the* **minterms**. *For example,*

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

is in SoP form. 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.

Conversely, when the expression for f is written as a product (i.e., AND) of terms which each comprise the sum (i.e., OR) of a number of inputs, it is said to be in **conjunctive normal form** *or* **Product of Sums (PoS)** *form; the terms in this expression are called the* **maxterms**. *For example,*

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

is in PoS form. As above, each variable can exist as-is or *complemented using NOT.*

This is a formal definition for something very simple to describe by example. Consider a function g which is described by

$$g : \begin{cases} \mathbb{B} & \to & \mathbb{B} \\ x & \mapsto & \begin{cases} 0 & \text{if } x = 0 \text{ and } y = 0 \\ 1 & \text{if } x = 0 \text{ and } y = 1 \\ 1 & \text{if } x = 1 \text{ and } y = 0 \\ 0 & \text{if } x = 1 \text{ and } y = 1 \end{cases} \end{cases}$$

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

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

Notice how the terms $\neg x \wedge y$ and $x \wedge \neg y$ are the 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

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

Since $\neg x$ is 1 for the first and second rows, rather than just the second as was the case with $\neg x \wedge y$, 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 to above, we can construct a PoS expression for g as

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

In this case the terms $x \vee y$ and $\neg x \vee \neg y$ are the maxterms.

We have already covered the manipulation of expressions, which will allow us to show that g_{SoP} and g_{PoS} are just two different ways to write the same thing. Recall that for p and q

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

Using this new identity, we can show

$$\begin{array}{rcl} g_{SoP}(x, y) & = & (\neg x \wedge y) \vee (x \wedge \neg y) \\ & = & (y \wedge \neg x) \vee (x \wedge \neg y) \\ & = & (x \wedge (\neg x \vee \neg y)) \vee (y \wedge (\neg x \vee \neg y)) \\ & = & (x \vee y) \wedge (\neg x \vee \neg y) \\ & = & g_{PoS} \end{array}$$

Although this might not seem particularly interesting at first glance, it gives us quite a powerful technique: given a truth table for an arbitrary function, we can construct an expression for the function simply by extracting either the minterms or maxterms and constructing standard SoP or PoS forms.

6 Representations

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

– L. Kronecker

6.1 Bits, bytes and words

Definition 0.29 *Used as a term by Claude Shannon in 1948 [?] (but attributed to John Tukey), a* **bit** *is a single* **binary digit**. *Since a given bit is a member of the set* $\mathbb{B} = \{0, 1\}$, *it can be used to represent a truth value, i.e.,* **false** *or* **true**, *and hence a value within the context of Boolean algebra.*

Definition 0.30 *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. Like other sequences, we use* X_i *to denote the i-th bit of the binary sequence* X *and* $|X| = n$ *to denote the number of bits in* X.

Instead of writing out $X \in \mathbb{B}^n$ *in full as* $\langle X_0, X_1, \ldots, X_{n-1} \rangle$, *we sometimes prefer to list the bits within a* **bit-literal**. *For example, consider the following bit-sequence*

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

which has seven bits in it, i.e., $|X| = 7$, *and can be less formally written as the bit-literal*

$$X = 1111011.$$

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

$$\underbrace{\hat{X}}_{\text{the representation of } X} \quad \underbrace{\mapsto}_{\text{maps to}} \quad \underbrace{X}_{\text{the value of } X}$$

That is, all we need is a concrete representation that we can write down, and a mapping that means the right thing wrt. values, *plus* is consistent in both directions.

6.1.1 Properties

Definition 0.31 *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 0.32 *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 quantity 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 quantity can be expressed as*

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

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

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

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

6.1.2 Ordering

There is, by design, no "structure" to a bit-literal. This can be problematic; for example, we need a way to make sure the order of bits in the bit-literal is clear relative to the corresponding bit-sequence. More formally, this relates to the concept of **endianness**. We might, as above, use a **little-endian** convention by reading the bits from right-to-left. This means, as above, that given the bit-literal

$$A = 1111011$$

we are clear that the corresponding bit-sequence is given by

$$A_{LE} = \langle A_0, A_1, A_2, A_3, A_4, A_5, A_6 \rangle = \langle 1, 1, 0, 1, 1, 1, 1 \rangle.$$

In this case the right-most bit of the bit-literal is the **least-significant** and the left-most is the **most-significant**; these correspond to the 0-th and $(n-1)$-th elements in the corresponding bit-sequence.

There is no reason why little-endian this is the only option however: it might be attractive to select a **big-endian** convention which reverses the bits. Now we read them left-to-right, meaning the left-most bit of the bit-literal is now the **least-significant** and the right-most is the **most-significant**. If the same bit-literal is interpreted in big-endian notation, the corresponding bit-sequence is given by

$$A_{BE} = \langle A_6, A_5, A_4, A_3, A_2, A_1, A_0 \rangle = \langle 1, 1, 1, 1, 0, 1, 1 \rangle.$$

Note that the **Least-Significant Bit (LSB)** and **Most-Significant Bit (MSB)** of the bit-sequence now appear at the left-most and right-most ends of the corresponding bit-literal: we have reversed the ordering in some sense.

Unless otherwise specified, it is (fairly) safe to assume that a little-endian convention has been used. However, it is important to notice that once given an endianness convention, which essentially 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.

6.1.3 Grouping

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 shorter sub-sequences. For example, we could rewrite the sequence as

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

or

$$D = \langle\langle 1,1,0,0,0,0,0,0\rangle,\langle 1,0,0,0,1,0,1,0\rangle\rangle.$$

So C has four elements (each of which is a sub-sequence of four bits from the original sequence), while D has two elements (each of which is a sub-sequence of eight bits from the original sequence). 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.

Definition 0.33 *Certain bit-sequences are given special names depending on their length. Specifically, a sequence of four bits is termed a* **nibble** *while a sequence of eight bits is termed a* **byte** *(or, in some contexts, an* **octet***).*

A given context will, in some sense, have a "natural" value for n, the length of bit-sequences. In such a context, an n-bit sequence is commonly termed a **word***; for example, when you hear a computer processor described as being 32-bit or 64-bit this is, roughly speaking, specifying that the word size is 32 or 64 bits. One may also see* **half-word** *and* **double-word** *used to describe $(n/2)$-bit and $2n$-bit sequences.*

Previously we discussed the problem of endianness within bit-sequences; the same issue is important here since we are still constructing larger sequences from smaller ones. For example, consider the 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: using 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 using a big-endian convention we get

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

Clearly we need to know which order to place the nibbles in, otherwise we get the wrong result: using a little-endian convention where C_0 is read as the least-significant nibble and C_3 as the most-significant we get the right result, but using a big-endian convention the result is backwards.

6.1.4 Units

If there is one subject guaranteed to drive a Computer Scientist into a rage, it is the choice of standard units for measuring multiplicities of bits and bytes. The short version is as follows: the International System of Units (SI) works with decimal orders of magnitude: in SI terms, the prefix kilo always means $10^3 = 1000$. However, since $2^{10} = 1024 \simeq 1000$, the term kilobyte has traditionally been used to mean 1024 bytes; a similar mismatch occurs with the terms mega, giga, tera and so on. So, applying SI units directly mean that

1	kilobit	(kbit)	=	10^3	bits	=	1,000	bits
1	megabit	(Mbit)	=	10^6	bits	=	1,000,000	bits
1	gigabit	(Gbit)	=	10^9	bits	=	1,000,000,000	bits
1	terabit	(Tbit)	=	10^{12}	bits	=	1,000,000,000,000	bits
1	kilobyte	(kB)	=	10^3	bytes	=	1,000	bytes
1	megabyte	(MB)	=	10^6	bytes	=	1,000,000	bytes
1	gigabyte	(GB)	=	10^9	bytes	=	1,000,000,000	bytes
1	terabyte	(TB)	=	10^{12}	bytes	=	1,000,000,000,000	bytes

In 1998, the International Electrotechnical Commission (IEC) added some more prefixes to this list in order to eliminate any ambiguity; the result is 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

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

  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

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

  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 $\langle x_1, x_0 \rangle$

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

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

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

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

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

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

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

  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 want to extract the i-th bit of x alone. This is a useful and common operation: following the above, it can be 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

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

  meaning $x_2 = 0$.

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

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

  meaning $x_0 = 1$.

1	kibibit	(kbit)	=	2^{10}	bits	=	1,024 bits
1	mebibit	(Mbit)	=	2^{20}	bits	=	1,048,576 bits
1	gigibit	(Gbit)	=	2^{30}	bits	=	1,073,741,824 bits
1	tebibit	(Tbit)	=	2^{40}	bits	=	1,099,511,627,776 bits
1	kibibyte	(kB)	=	2^{10}	bytes	=	1,024 bytes
1	mebibyte	(MB)	=	2^{20}	bytes	=	1,048,576 bytes
1	gigibyte	(GB)	=	2^{30}	bytes	=	1,073,741,824 bytes
1	tebibyte	(TB)	=	2^{40}	bytes	=	1,099,511,627,776 bytes

The question is, which should we use? Does it *really* matter? Clearly, yes: if we buy a disk drive which says it holds 1 terabyte of data, we hope they are simply talking traditionally about tebibyte, i.e., 1,099,511,627,776 bytes rather than 1,000,000,000,000 bytes, because then we get more storage capacity for our money. In the same way, imagine we are trying to compare two disks: we need to make sure their quoted storage capacity use the same units, or the comparison will not be fair.

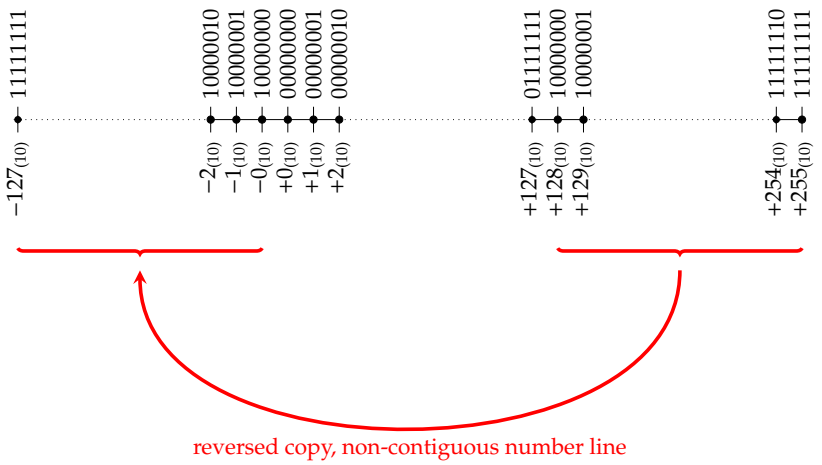
We try to make consistent use of the new SI prefixes: when we say kilobyte or kB we mean 10^3 bytes, and when we say kibibyte or KiB we mean 2^{10} bytes. This might not be popular from a historical point of view, but it does mean we are at least clear what is meant.

6.2 Positional number systems

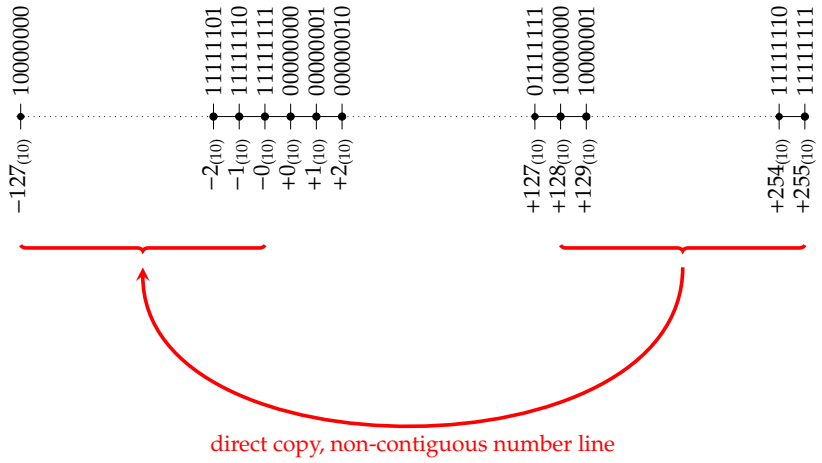
As humans, and because (mostly) we have ten fingers and toes, we are used to working with numbers written down using digits from the set $\{0, 1, \ldots, 9\}$. Imagine we write down the number 123. Hopefully you can believe this is sort of the same as writing the sequence

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

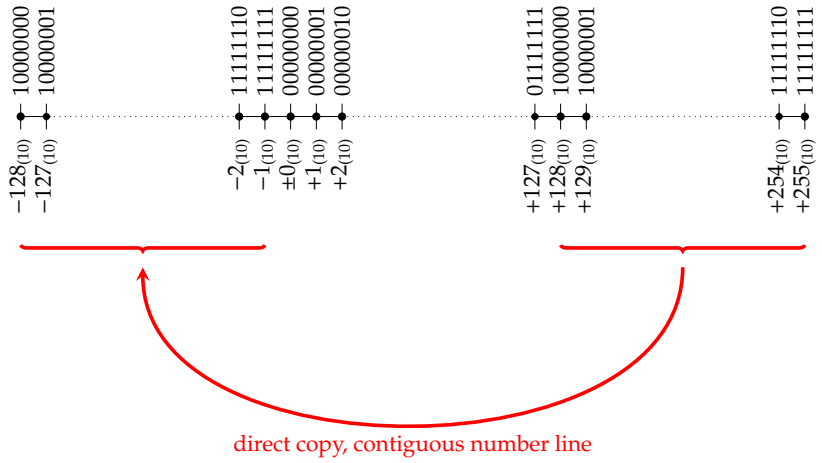
given that 3 is the first digit of 123, 2 is the second digit and so on; we are just reading the digits from left-to-right rather than from right-to-left. How do we know what 123 or A *means*? What is their value? In



(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.*

simple terms, we just weight each of the digits 1, 2 and 3 by a different amount and then add everything up. We can see for example that

$$A = 123 = 1 \cdot 100 + 2 \cdot 10 + 3 \cdot 1$$

which we might say out loud as "one lot of hundred, two lots of ten and three units" or "one hundred and twenty three". We could also write the same thing as

$$A = 123 = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0$$

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

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

All this means is that we add up

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

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

1. each digit is weighting by a power of a **base** (or **radix**) which in our case is 10, and
2. the power used relates to the position of the corresponding digit; basically the i-th digit is weighted by 10^i.

The neat outcome is that there are *many* other ways of representing the same value. For example, suppose we use a different base; all this means is our weights and digit set from above change. We could now express the same thing as

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

where now the digit set used is $\{0, 1\}$. The value of 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. This means we add up the terms

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

to obtain a total of 123 as before.

Definition 0.34 *A base-b (or radix-b)* **positional number system** *(or place-value number system) uses digits from the* **digit set** $X = \{0, 1, \ldots, b-1\}$. *The representation of a given number x in this system consists of n digits, m of which are in the fractional part, i.e.,*

$$\hat{x} \mapsto \pm \sum_{i=-m}^{n-m-1} x_i \cdot b^i.$$

where $x_i \in X$.

Choices of $b = 2$, $b = 8$, $b = 10$ and $b = 16$ correspond to **binary**, **octal**, **decimal** *and* **hexadecimal** *numbers. Usefully, each octal or hexadecimal digit can represent exactly three or four binary digits respectively; using this short-hand, one can more easily write and remember long sequences of binary digits.*

Of course, following the above we simply have $m = 0$ since there was no fractional part for 123. But since $10^{-1} = 1/10 = 0.1$ and $10^{-2} = 1/100 = 0.01$ for example, we can start to write down numbers that 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 in total, $m = 4$ of which are in the fractional part. Of course since the definition is the same, we can do the same thing using a different base, e.g.,

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

The **decimal point** in the first case, which we are used to seeing, works the same way when translated into a **binary point** in the second case; more generally we call this a **fractional point** where the base is irrelevant.

We previously mentioned that some numbers are irrational: our definition said that we could not find an x and y such that x/y was exactly what we want. Since we can now change the base a number is written in, we find that some numbers are irrational depending on that base. Informally, we already know that when we write $1/3$ as a decimal number we have $0.3333\ldots$ with the continuation dots meaning that the sequence recurs infinitely. $1/10$, however, can be written as 0.1 in decimal but is irrational when written in binary; the closest approximation is $0.000110011\ldots$.

6.2.1 Digits

Although the definition still works, once we select a base $b > 10$ we hit a problem: we run out of Roman-style digits that we can write down. Use of $b = 16$ is attractive for example, but we have no single-digit way to write 15. Having a single-digit for 15, and indeed any of $\{10, 11, \ldots b-1\}$ is important because if we use two digits then there could be confusion about how to apply the weighting.

As a result, we use the characters $A \ldots F$ to represent $10 \ldots 15$. Otherwise everything works the same way, meaning for example that

$$\begin{aligned} 7B_{(16)} &= 7 \cdot 16^1 + B \cdot 16^0 \\ &= 7 \cdot 16^1 + 11 \cdot 16^0 \\ &= 123_{(10)} \quad . \end{aligned}$$

6.2.2 Notation

Amazingly there are not many jokes about Computer Science, but here are two:

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" relate to what we have been discussing: in the first case there is an ambiguity between the number ten written in decimal and binary, and in the second between the number twenty five written in octal and decimal.

Look at the first joke: it is saying that the literal 10 can be interpreted as binary as well as decimal, i.e., as $1 \cdot 2 + 0 \cdot 1 = 2$ in binary and $1 \cdot 10 + 0 \cdot 1 = 10$. 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. Likewise "Dec" can mean "December" but also "decimal". With this in mind, we see that

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

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 A and the decimal number 123 are basically the same if we interpret A in the right way. But there is a problem of ambiguity: if we follow the same reasoning, we would also say that the binary sequence B and the number 01111011 are the same. But how do we know what base 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 $b = 10$!

To clear up this ambiguity where necessary, we write literal numbers with the base appended to them. For example $123_{(10)}$ is the number 123 written in base-10 whereas $01111011_{(2)}$ is the number 01111011 in base-2. We can now be clear, for example, that $123_{(10)} = 01111011_{(2)}$. If we write a sequence, we can do the same thing: $\langle 3, 2, 1 \rangle_{(10)}$ makes it clear we are still basically talking about the number 123. So our two "jokes" in this notation become $10_{(2)} = 2_{(10)}$ and $31_{(8)} = 25_{(10)}$.

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

The number systems above afford flexible ways to write down numbers. However, the set of integers $\mathbb{Z}$, for example, is infinite in size: the next challenge is to develop a concrete **representation** based on said number systems, but also cater for the fact that we have finite resource available to us. Most obviously, these finite resources place a bound on the number of digits we can have in a given base-b expansion.

The goal is to represent members of $\mathbb{Z}$ by encoding them as a bit-sequence of fixed length n. For unsigned integers, this is fairly simple since we can directly use what we know already.

Definition 0.35 *An unsigned integer can be represented in n bits simply via the natural binary expansion*

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

Note that n clearly puts a limit on the range of x. Rather than being a member of $\mathbb{Z}$ per se, x is limited to

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

As an aside, this is the first point where a subtle issue starts to become more important. If we write down the literal

$$x = 00000110,$$

what does it mean, or what is the associated value? The answer is whatever we want: it simply depends on our interpretation. A representation allows said interpretation, acting as a sort of template. If we interpret x as an unsigned integer using our representation above, then the value is

$$\begin{aligned} \hat{x} &= \langle 0, 1, 1, 0, 0, 0, 0, 0 \rangle \\ &\mapsto \textstyle\sum_{i=0}^{n-1} x_i \cdot 2^i \\ &\mapsto = 0 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 \\ &\mapsto = 6_{(10)} \end{aligned}$$

but of course there is no reason why the same literal might have a different value if we interpret it under a different representation.

We can start to show the impact of this fact by looking at representations for signed integer; this is more tricky in the sense that we need some way to represent the sign of the number. There are several ways to achieve this, but the goal is still the same: we want to represent signed integers using an n-bit sequence.

6.3.1 Sign-magnitude

Definition 0.36 *The* **sign-magnitude** *method represents a signed integer in n bits by allocating one bit to store the sign, typically the most-significant, and $n-1$ to store the magnitude. If the* **sign bit** *is set to one for a negative integer and zero for a positive integer, then*

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

where

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

Note that $-1^0 = +1$ and $-1^1 = -1$, the initial term above thereby controlling the overall sign, and that there are two representations of zero, i.e., $+0$ and -0 as a result of this.

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

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

6.3.2 One's-complement

Definition 0.37 *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*

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

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.

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

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

6.3.3 Two's-complement

Definition 0.38 *The* **two's-complement** *method represents a signed integer in n bits by using a large negative weight for the most-significant, $(n-1)$-bit. That is, the need for a dedicated sign bit is removed by weighting the $(n-1)$-th bit by -2^{n-1} rather than $+2^{n-1}$. Thus,*

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

where

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

If $n = 8$ for example, we can represent values in the range $-128 \ldots +127$ with the difference versus sign-magnitude coming as a result of there only being one representation of zero. Selected cases are as follows:

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

Since two's-complement is perhaps the most widely encountered signed integer representation (for example data-types such as `int` in C use it behind the scenes), it perhaps warrants some further explanation. One additional way to look at what is going on is to consider the number line in Figure 3c. The diagram shows that for $n = 8$, unsigned integers 0 through to 255 can use their natural binary representation. Sometimes you see this number line wrapped into a circle to emphasise the fact that the representation wraps-around: when we reach 11111111, the next representation available is 00000000. By using two's-complement, we represent negative integers by "borrowing" the range 128 through to 255 and using the associated representations to mean -128 through to -1; in a sense, we "shift" that range from the right-hand (larger positive) end of the number line to the left-hand (negative) end.

Another way to look at this is via an appeal to intuition: if we have x and add $-x$, i.e., compute $x + (-x)$ then we expect to get zero as a result. The two's-complement representation satisfies this. For example, we can see from the above that

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

meaning that if we ignore the carry-out (which cannot be retained because we have too few bits), we get the result expected. This hints at a useful fact:

Definition 0.39 *The term two's-complement can be used to describe the representation, or an operation: "taking the two's-complement of x" means negating it, i.e., computing the representation of $-x$. To do so, we simple compute*

$$-x \mapsto \neg x + 1.$$

Why does this work? First note that if we add x to $\neg x$, for any x, then we get -1 as a result. For example:

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

This should make sense, in that each bit of $\neg x$ is the opposite of that in x so either one is 0 and the other is 1 or vice versa: either way, their sum will always be 1. Our result is therefore off-by-one, meaning that if we instead compute

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

then we are back to the same example as above: since the result is zero, it must be the case that $-x \mapsto \neg x + 1$.

This *also* hints at another attractive fact, or rather a feature of two's-complement. In short, we can legitimately use the same approach to arithmetic using integers represented using two's-complement as with the simpler case of unsigned integers. Why? Our representations still map into a contiguous number

line (with one representation of zero), plus we have arguments like above wrt. the sanity of operations such as addition and subtraction: they still move one way or the other along the number line (modulo the end points). This is *not* the case, for instance, with sign-magnitude where we need to have a special rule to work out the sign of a result.

6.3.4 Binary Coded Decimal (BCD)

BCD is a method of representing an integer as a bit-sequence. Instead of representing the number itself as a bit-sequence (which is what the previous methods do), the idea is to first view the integer as decimal, then encode each decimal digit as a bit-sequence; the overall representation is then the concatenation of these sub-sequences.

Definition 0.40 *Consider the function*

$$f : \begin{cases} \{0,1,\ldots,9\} & \rightarrow \quad \mathbb{B}^4 \\ d & \mapsto \begin{cases} \langle 0,0,0,0\rangle & \text{if } d = 0 \\ \langle 1,0,0,0\rangle & \text{if } d = 1 \\ \langle 0,1,0,0\rangle & \text{if } d = 2 \\ \langle 1,1,0,0\rangle & \text{if } d = 3 \\ \langle 0,0,1,0\rangle & \text{if } d = 4 \\ \langle 1,0,1,0\rangle & \text{if } d = 5 \\ \langle 0,1,1,0\rangle & \text{if } d = 6 \\ \langle 1,1,1,0\rangle & \text{if } d = 7 \\ \langle 0,0,0,1\rangle & \text{if } d = 8 \\ \langle 1,0,0,1\rangle & \text{if } d = 9 \end{cases} \end{cases}$$

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

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

For example, given

$$x = 123_{(10)}$$

we find

$$\hat{x} = \langle\langle 1,1,0,0\rangle, \langle 0,1,0,0\rangle, \langle 1,0,0,0\rangle\rangle$$

which we can flatten out and write as the bit-literal

$$x \mapsto 000100100011.$$

Clearly it can be combined, for example, with a sign-magnitude approach to accommodate signed integers.

BCD has some disadvantages. For example, it is not very compact in that six possible encodings (i.e., $\langle 0,1,0,1\rangle \ldots \langle 1,1,1,1\rangle$ corresponding to $10 \ldots 15$) are unused, and we need extra information at the start of the encoding (basically a sign bit as per sign-magnitude) to tell us whether the number is positive or negative. Even so, wherever decimal arithmetic is de rigueur, financial quantities for example, it can also have advantages.

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

As above, the idea of representing real numbers is to lean on the generality afforded by positional number systems; again the goal is to represent members of $\mathbb{R}$ by encoding them as a bit-sequence of fixed length n.

Imagine we would like to represent some real number x. There is basically just one underlying trick: we approximate x by taking a base-b integer m (signed or otherwise) and **scaling** it, i.e., having

$$x \simeq m \cdot b^e$$

for some e. Our representation, which fills in the detail of how this works in practice, falls into one of two categories: if e is fixed (i.e., constant) we have a **fixed-point** representation, but if e can vary we have a **floating-point** representation.

6.4.1 Fixed-point

Definition 0.41 *The goal of a* **fixed-point** *representation is to allow expression of real numbers whose form is*

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

or equivalently in the form

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

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

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

This might seem quite confusing: for some integer x we are just shifting the fractional point around by a fixed amount to determine the value, rather than interpreting it as an integer. Imagine we set $b = 10$, $n = 7$, $q = 4$ and write the literal

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

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 are again altering the weights associated with each digit: taken as an integer, the i-th digit is weighted by b^i but interpreting the same digit in our fixed-point representation means weighting it by b^{i-q}.

Definition 0.42 *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.*

The definition above is general enough to accommodate different choices of b. It might not be surprising that $b = 2$ is attractive: this allows us to use what we already know about representing integers using bit-sequences, and apply it to representing real numbers using the idea of 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

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

  or rather $0 \leq x \leq 31.875$ 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 an 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

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

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

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

An example As an aside, there is a neat way to visualise the intuitive effect of adding more precision (i.e., increasing the number of fractional digits) in a fixed-point representation. Figure 4 shows several renderings of the famous Mandelbrot fractal, named after mathematician Benoît Mandelbrot. Each rendering uses a 32-bit integer, i.e., $n = 32$, to specify a fixed-point representation with different values of q, i.e., different numbers of fractional digits. Quite clearly, as we increase q there is more detail. Without going into it too far, 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 only sample points (x, y) with $x, y \in \{-2, -1, 0, +1, +2\}$; by adding more fractional digits we can sample intermediate points, e.g., $(0.5, 0.5)$ and so on, meaning more detail in the rendering.

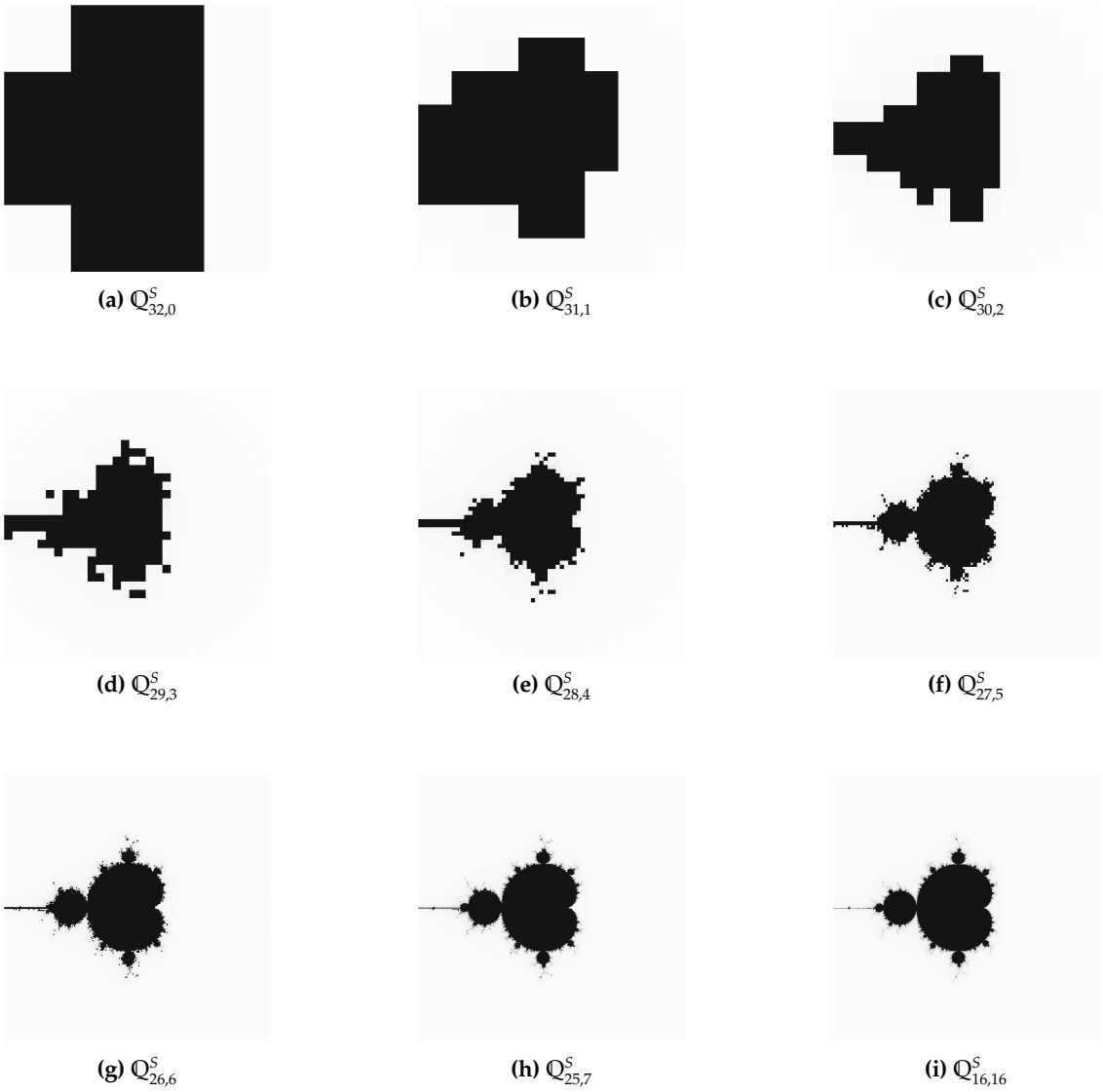
6.4.2 Floating-point

Definition 0.43 *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**, *noting that* $-1^0 = +1$ *and* $-1^1 = -1$,
- $m \in \mathbb{N}$ *is the* **mantissa**, *and*
- $e \in \mathbb{Z}$ *is the* **exponent**.

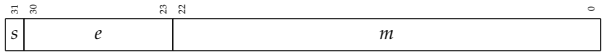


(a) $\mathbb{Q}^S_{32,0}$ **(b)** $\mathbb{Q}^S_{31,1}$ **(c)** $\mathbb{Q}^S_{30,2}$

(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)** $\mathbb{Q}^S_{16,16}$

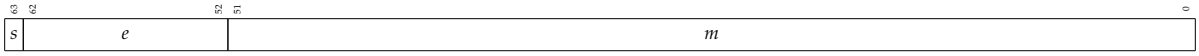
Figure 4: *A visualisation of the impact of increasing q, the number of fractional digits, in a fixed-point representation; the result is increased detail within the rendering of a Mandelbrot fractal.*



(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.*

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 0.44 *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, described graphically in Figure 5, 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:*

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

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:*

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

  Note that here, $\beta = 1023$.

Imagine we want to represent $x = 123.3125_{(10)}$ in the single-precision, 32-bit IEEE 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 need to store the implicit hidden digit (i.e., the digit to the left of the binary point), our mantissa, exponent and sign are

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

noting that we pad both with less-significant zeros to make them the right length. Finally, we can convert each component into a literal using their associated representations, i.e.,

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

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

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

Special values

Definition 0.45 *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*

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

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

Rounding modes 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 such modes. 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. Throughout the following description we use decimal examples for clarity (minor alterations apply in binary), rounding for $n = 2$ digits of precision in each example. Note that the C standard library offers access to these features. 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.

Round to nearest Sometimes termed **Banker's Rounding**, this mode alters basic rounding to provide more specific treatment when the ideal result is exactly half way between two representable results, i.e., when $m'_0 = 5$:

- 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.

For example:

- $1.24_{(10)}$ 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 $1.4_{(10)}$.

Round toward $+\infty$ This scheme is sometimes termed **ceiling**:

- 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.

For example:

- $1.24_{(10)}$ 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 $-1.2_{(10)}$.

Round toward $-\infty$ Sometimes termed **floor**:

- 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,

For example:

- $-1.24_{(10)}$ 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 $1.2_{(10)}$.

Round toward zero This scheme operates as round toward $-\infty$ for positive numbers and as round toward $+\infty$ for negative numbers. For example:

- $1.27_{(10)}$ 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 $-1.2_{(10)}$.

The C standard library uses the constant values `FE_TONEAREST`, `FE_UPWARD`, `FE_DOWNWARD` and `FE_TOWARDZERO` respectively to specify these rounding modes.

An example 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 called to `printf` that produces the lines of output):

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

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

  Accounting for the bias and including the hidden but, this represents the value

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

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

- `t.y.s` is set to $01_{(10)} = 1_{(10)}$, and then the value of `t.x` and the components of `t.y` are 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 now equal to $-2.8_{(10)}$ as expected.

- `t.y.e` is set to $81_{(10)} = 129_{(10)}$, and then the value of `t.x` and the components of `t.y` are printed. We expect that setting the exponent to 129 rather than 128 will double `t.x` (the unbiased value of the exponent is now $129 - 127 = 2$ meaning the mantissa is scaled by $2^2 = 4$ rather than $2^1 = 2$); this is confirmed by the output which shows `t.x` is now 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 $+\infty$, which the output confirms.

```
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.*

6.5 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 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

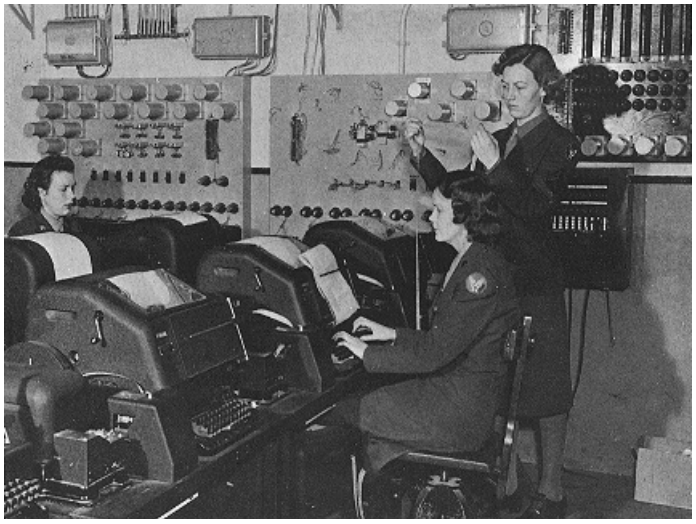


Figure 7: *A teletype machine being used by UK-based Royal Air Force (RAF) operators during WW2 (public domain image, source: http://en.wikipedia.org/wiki/File:WACsOperateTeletype.jpg).*

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.*

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

7 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 set of values (i.e., $\mathbb{B} = \{0, 1\}$), a set of operators (i.e., AND, OR and NOT which also give us XOR), and a list 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
$$\mathbb{B}^n \to \mathbb{B}$$
and hence also construct functions of the form
$$\mathbb{B}^n \to \mathbb{B}^m$$
simply by having m functions and grouping their outputs together. It therefore makes sense that AND, OR, XOR and NOT are well-defined for the set $\mathbb{B}^n$ as well as $\mathbb{B}$: we **overload** AND 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 know how to represent all sorts of different things using sequences of bits; an n-bit sequence is simply a member of $\mathbb{B}^n$, and we know how to deal with such things using Boolean algebra.

   In particular, we can represent numbers using bit-sequences; since we can build arbitrary Boolean functions of the form
$$\mathbb{B}^n \to \mathbb{B}^m,$$
   it should not be surprising that we can write functions that perform arithmetic with the numbers we are representing. That is, they have just the right effect on the bit-sequences so that the numbers they represents "behave" correctly in terms of operations such as addition and subtraction.

   Imagine we want to add two integers together. That is, say we have three integers $a, b, c \in \mathbb{Z}$ represented by three n-bit sequences $r, x, y \in \mathbb{B}^n$. To calculate the integer addition $b + c$, all we need to do is define n similar Boolean functions, say f_i, such that $r_i = f_i(x, y)$: each function takes x and y, the representations of b and c, as input and produces a single bit of r, the representation of c, as output.

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.

BIBLIOGRAPHY (AND FURTHER READING)

[1] G. Boole. *An investigation of the laws of thought*. Walton & Maberly, 1854.

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