**Computer Science 52**

# Logic, Words, and Integers

## 1 Words and Data

The basic unit of information in a computer is the *bit;* it is simply a quantity that takes one of two values, 0 or 1. A sequence of $k$ bits is a *$k$-bit word.* Words have no intrinsic meaning beyond the value of the bits. We can assign meaning to words by interpreting them in various ways. Depending on the context we impose, a word can be viewed as an integer, as a boolean value, as a floating point number, as a string of letters, or as an instruction to the computer.

A single bit can be viewed as a boolean value with the possible values "false" and "true." In the next section, we look at the logical operations on truth values—or, equivalently, on single bits.

A *byte* is an 8-bit word. Memory in modern computers is arranged as a sequence of bytes, where adjacent bytes can be considered to be longer words. As we shall see shortly, bytes can be viewed as the binary representations of integers ranging either between 0 and $2^8 - 1$ or between $-2^7$ and $2^7 - 1$.

## 2 Propositional Logic

Propositional logic is a language to manipulate truth values. Expressions in propositional logic are constructed from

- constants, $\bot$ and $\top$;
- variables, $A$, $B$, ...;
- parentheses, ) and (; and
- connectives, $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\oplus$, and $\equiv$.

We interpret the expressions as statements about truth values or bits. The symbol $\bot$ is read "bottom" and represents either 0 or "false." The symbol $\top$ is read "top" and represents either 1 or "true." The connectives represent the following operations:

- $\neg$ means "not" or "(boolean) negation."

| $A$ | $(\neg A)$ |
|---|---|
| $\top$ | $\bot$ |
| $\bot$ | $\top$ |

| $A$ | $B$ | $(A \wedge B)$ | $(A \vee B)$ | $(A \Rightarrow B)$ | $(A \oplus B)$ | $(A \equiv B)$ |
|---|---|---|---|---|---|---|
| $\top$ | $\top$ | $\top$ | $\top$ | $\top$ | $\bot$ | $\top$ |
| $\top$ | $\bot$ | $\bot$ | $\top$ | $\bot$ | $\top$ | $\bot$ |
| $\bot$ | $\top$ | $\bot$ | $\top$ | $\top$ | $\top$ | $\bot$ |
| $\bot$ | $\bot$ | $\bot$ | $\bot$ | $\top$ | $\bot$ | $\top$ |

Table 1: The truth tables for the propositional connectives.

$$
\begin{aligned}
S &\to E \\
E &\to E \equiv I \mid I \\
I &\to D \Rightarrow I \mid D \\
D &\to D \vee X \mid X \\
X &\to X \oplus C \mid C \\
C &\to C \wedge N \mid N \\
N &\to \neg N \mid B \\
B &\to U \mid V \mid (E) \\
U &\to \bot \mid \top \\
V &\to A \mid B \mid \ldots
\end{aligned}
$$

Table 2: A context-free grammar that generates the expressions of propositional logic.

---

- $\wedge$ means “and.”
- $\vee$ means “(inclusive) or.”
- $\Rightarrow$ means “implies” or “not greater than.”
- $\oplus$ means “exclusive or” or “unequal.”
- $\equiv$ means “equivalent” or “equal”

If we assign truth values to the variables in an expression, we can evaluate the expression. We specify the actions of the connectives using *truth tables.* Table 1 shows the resulting truth tables for the connectives of propositional logic.

The logical expressions may be generated by a *context-free grammar,* which we discuss elsewhere in the course. For reference, the grammar appears in Table 2.

The truth tables for the connectives can be used to evaluate logical expressions. For example, we can verify one of the de Morgan laws,

$$\neg(A \wedge B) = \neg A \vee \neg B.$$

An equality like this one means that the left and right sides have the same truth value, regardless of the truth values of $A$ and $B$. Table 3 shows some common logical equivalences.

We adopt the convention that, in the absence of parentheses, the operation $\neg$ is done first, followed in order by $\wedge$, $\oplus$, $\vee$, $\Rightarrow$, and $\equiv$.

## 3 Gates

For each connective there is a corresponding component called a *gate.* A gate takes electrical signals encoding to one or two logical constants as input, and produces an output in the same format. The gates for the fundamental logical operations are listed in Table 4. When working with gates, we usually use 0 and 1 for the logical constants and call them *bits.* Combinations of logical operations correspond to electrical circuits constructed from gates, as we shall see toward the end of this document.

## 4 Words

As we mentioned, a sequence of $k$ bits is a *$k$-bit word.* The operations of propositional logic may be applied to words, in which case they operate on the bits in parallel. For example, the negation of the three-bit word 011 is 100, and the result of an and-operation on 011 and 100 is 000. The study of these operations is called "digital logic."

Words often represent integers. There are two commonly-used ways to interpret words as integers. Unsigned integers represent non-negative values, while signed integers include negative values.

### 4.1 Unsigned Integers

A $k$-bit word can be interpreted as an *unsigned integer* by viewing the bits as the "digits" in a binary expansion. Normally we take the right-most bit as the least significant, or the "ones place." If the word is $b_{k-1}b_k \ldots b_1 b_0$, then the value of the unsigned word is

$$2^{k-1}b_{k-1} + 2^{k-2}b_{k-2} + \ldots_+ 2b^1 + b_0 = \sum_{i=0}^{k-1} 2^i b_i.$$

| | |
|---|---|
| $A \vee \top = \top$ | $A \wedge \top = A$ |
| $A \vee \bot = A$ | $A \wedge \bot = \bot$ |
| $A \oplus \top = \neg A$ | $\top \Rightarrow A = A$ |
| $A \oplus \bot = A$ | $\bot \Rightarrow A = \top$ |
| | $A \Rightarrow \top = \top$ |
| | $A \Rightarrow \bot = \neg A$ |
| $A = \neg\neg A$ | $(\neg A \Rightarrow B) \wedge \neg B = A$ |
| $A = A \wedge A$ | $A = A \vee A$ |
| $A \vee \neg A = \top$ | $A \wedge \neg A = \bot$ |
| $A \Rightarrow A = \top$ | $A \equiv A = \top$ |
| $A \oplus \neg A = \top$ | $A \oplus A = \bot$ |
| $A \vee B = B \vee A$ | $A \wedge B = B \wedge A$ |
| $A \equiv B = B \equiv A$ | $A \Rightarrow B = \neg B \Rightarrow \neg A$ |
| $A \oplus B = B \oplus A$ | |
| $A \vee (B \vee C) = (A \vee B) \vee C$ | $A \wedge (B \wedge C) = (A \wedge B) \wedge C$ |
| $A \equiv (B \equiv C) = (A \equiv B) \equiv C$ | $A \oplus (B \oplus C) = (A \oplus B) \oplus C$ |
| $A \vee (B \wedge C) = (A \vee B) \wedge (A \vee C)$ | $A \wedge (B \vee C) = (A \wedge B) \vee (A \wedge C)$ |
| $A \vee (A \wedge B) = A$ | $A \wedge (A \vee B) = A$ |
| $A \equiv B = (A \Rightarrow B) \wedge (B \Rightarrow A)$ | $A \oplus B = \neg(A \equiv B)$ |
| $A \Rightarrow B = \neg A \vee B$ | $A \Rightarrow B = \neg(A \wedge \neg B)$ |
| $A \vee B = \neg(\neg A \wedge \neg B)$ | $A \wedge B = \neg(\neg A \vee \neg B)$ |
| $A \vee B = \neg A \Rightarrow B$ | $A \wedge B = \neg(A \Rightarrow \neg B)$ |

Table 3: Some common propositional identities.

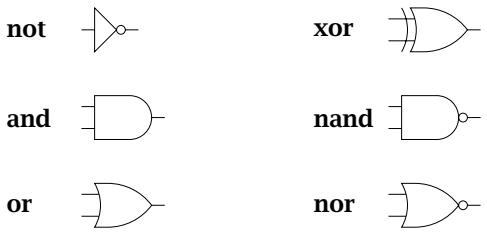


Table 4: The gates corresponding to the logical connectives. The input lines are on the left, and the output is on the right.

---

Unsigned $k$-bit integers range between 0 and $2^k - 1$, inclusive. The operations on unsigned integers are the arithmetic ones: addition, subtraction, multiplication, division, remainder, etc. There are also the usual comparisons: greater, less, greater or equal, and so forth.

Often, people make no distinction between a word and the unsigned integer it represents. One use of unsigned integers is to specify locations, or addresses, in a computer's memory.

## 4.2 Signed Integers

A $k$-bit word can also be interpreted as a *signed integer* in the range $-2^{k-1}$ through $2^{k-1}-1$. The words that have a most significant, or leftmost, bit equal to 1 represent the negative values. The leftmost bit is called the *sign bit.* The signed value of the word $b_{k-1}b_k \ldots b_1 b_0$ is

$$-2^{k-1}b_{k-1} + 2^{k-2}b_{k-2} + \ldots_+ 2b^1 + b_0 = -2^{k-1}b_{k-1} + \sum_{i=0}^{k-2} 2^i b_i.$$

The difference between the unsigned value and the signed value is the minus sign on the largest power of two. Notice that the word "signed" does not mean "negative." The signed interpretation gives us positive *and* negative values.

This representation of signed integers is called *two's complement.* Other representations of signed integers exist, but they are rarely used. The programming languages Java and SML use two's complement signed integers exclusively. The programming languages C and C++ have both signed and unsigned integer types.

Table 5 shows the signed and unsigned values of three-bit words. Most current processors use 32-bit words, although 64-bits is becoming more common. Here, we use fewer bits to make our examples easier, but the ideas are the same.

Interestingly, addition and subtraction can be carried out on unsigned and signed interpretations using the same algorithms. We simply use the binary analogs of the

| integer | unsigned | signed |
|---|---|---|
| 7 | 111 | |
| 6 | 110 | |
| 5 | 101 | |
| 4 | 100 | |
| 3 | 011 | 011 |
| 2 | 010 | 010 |
| 1 | 001 | 001 |
| 0 | 000 | 000 |
| −1 | | 111 |
| −2 | | 110 |
| −3 | | 101 |
| −4 | | 100 |

Table 5: Three bit words, as unsigned and two's complement signed values. Observe that the bit string 101 can mean 5 or −3, depending on the interpretation.

usual methods for adding and subtracting decimal numbers. Other operations, like comparisons and multiplication, require different methods—depending on whether we are thinking of the values as unsigned or signed.

### 4.3 Negation and Sign Extension

Negation of a two's complement signed value is accomplished by bitwise complementing the value and then adding one (and ignoring any carry into the $k+1$-st place). To convince yourself that this process is correct, let $x$ be a $k$-bit word and $\overline{x}$ be its bitwise complement. The sum $x + \overline{x}$ is a word with all 1's, whose value as a signed integer is $-2^{k-1} + \sum_{i=0}^{k-2} 2^i = -1$. Therefore, $x + \overline{x} = -1$, so that $\overline{x} + 1 = -x$.

We sometimes want to convert a $k$-bit word into a $j$-bit word, with $j$ greater than $k$. If we are thinking of the words as unsigned integers, we just add zeroes on the left. If we are thinking of them as signed integers, then we copy the sign bit on the left as many times as necessary to create the longer integer. The latter process is called *sign extension,* and it yields a longer (more bits) word representing the same signed integer. For example, −3 is represented in three bits by 101 and in six bits by 111101.

### 4.4 Words in Modern Computer Systems

Computers and programming languages use different size words as their fundamental quantity, and unfortunately, there is considerable variation from one system or

language to another. A byte is always 8 bits.

In Java, a `short` is a 16-bit integer quantity, an `int` has 32 bits, and a `long` has 64. All of these are interpreted as signed.

In C, the number of bits in a `short`, `int`, and `long` can vary across different computers, but nowadays, C usually agrees with Java. Any of these can be given signed or unsigned interpretation.

## 4.5 Addition

As mentioned earlier, the process of addition is the same for unsigned and signed quantities. The addition

$$\begin{array}{r} 010 \\ +\,001 \\ \hline 011 \end{array}$$

is correct whether we are thinking of unsigned or signed numbers. However, sometimes a result is "out of range," the conditions for which differ depending on how we interpret the numbers. The addition

$$\begin{array}{r} 011 \\ +\,110 \\ \hline 001 \end{array}$$

is a correct instance of binary addition. The "carry out" from the leftmost bit is discarded. With *unsigned* integers, the result is erroneous; the result of the sum $3+6$ is out of the range of three-bit integers. On the other hand, with *signed* integers, the addition is $3+(-2)$ and the result is correct. Contrastingly, the result of the addition

$$\begin{array}{r} 011 \\ +\,010 \\ \hline 101 \end{array}$$

is correct for the unsigned interpretation, but in the signed case, it adds two positive values and obtains a negative one. Colloquially, such problematic situations are termed "overflow." Below, we discuss ways to detect it.

## 4.6 Subtraction

Subtraction is simply negation followed by addition. One complements the bits in the number to be subtracted and then adds with an extra "carry" into the low order bit. Thus

$$\begin{array}{r} 011 \\ -\,010 \\ \hline 001 \end{array} \quad \text{becomes} \quad \begin{array}{r} {\scriptstyle 1} \\ 011 \\ +\,101 \\ \hline 001 \end{array}$$

The "carry out" from the leftmost column is discarded. This method for subtraction is correct in both the unsigned and signed cases. As in the case of addition, there is the opportunity for erroneous results due to "overflow."

### 4.7 Shifts

In addition to the bitwise logical operations and the arithmetic operations, we may also *shift* the bits in a word to the left or to the right. Most programming languages provide shift operators.

A *left shift* shifts the bits to the left. The most significant bits are lost, and zeroes are shifted in on the right. A left shift is a handy way to multiply a small number by a power of two. Java and C++ use the notation `x<<k` to shift the bits in `x` to the left by `k` places.

An *arithmetic right shift* shifts bits to the right. The least significant bits are lost, and the sign bit is replicated on the left. A left shift preserves the sign of the original word, and it is a handy way to divide by a power of two. Java uses the notation `x>>k` to shift the bits in `x` to the right by `k` places.

A *logical right shift* also shifts the bits to the right, but the bits on the left are replaced by zeroes. Java uses the notation `x>>>k` for the logical right shift.

The programming language C has only one right shift operator, `>>`. Most compilers choose the arithmetic right shift when the first operand is a signed integer and the logical right shift when the first operand is unsigned.

## 5 Circuits for Addition and Subtraction

Since words are composed of bits, we can use the gates described in Table 4 to create circuits to carry out arithmetic operations. If we are adding two bits $A$ and $B$, the sum bit is simply the $A \oplus B$ and the carry bit is $A \wedge B$. The circuit on the left in Table 6 shows a *half adder* that computes both the sum and the carry.

The circuit on the right in Table 6 shows how two half adders can be combined to form a *full adder* which corresponds to one column of an addition operation. The inputs are the bits $A$ and $B$ and a carry-in bit from the previous column. The results is the sum bit and a a bit to carry out to the next column.

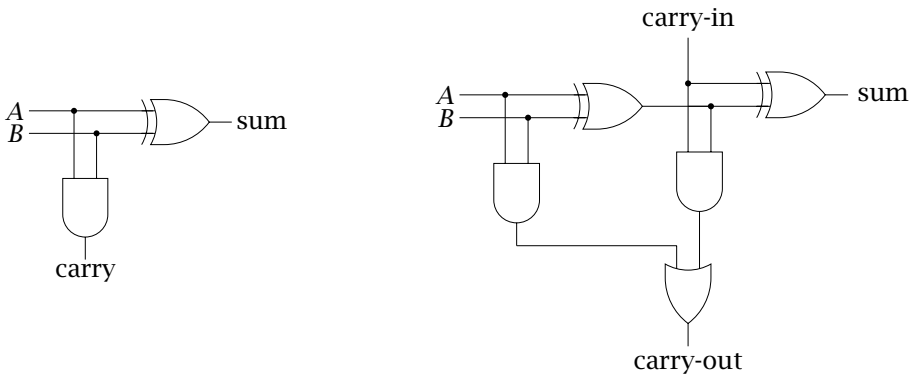


Table 6: A binary half adder (left) and a full adder (right).

If we want to add $n$-bit words, we can do it with a circuit constructed from $n$ full adders, as shown in Table 7. Each adder corresponds to one of the $n$ columns in the addition problem. The carry-in to the low order column is 0, and the carry-out from the high order column is discarded. The circuit is called a *ripple-carry adder.*

If we want to subtract $n$-bit words, we use a slight modification of the ripple-carry adder. Recall that subtraction is negation followed by addition and that negation is complementation followed by adding one. The bit $D$ in the circuit on the right in Table 7 controls whether the operation is addition or subtraction. If $D$ is zero, the circuit behaves just like the ripple-carry adder on the left. If $D$ is one, the circuit computes the sum of the $A$ bits, the complement of the $B$ bits, and 1. That operation amounts to subtraction.

## 6 Overflow and comparisons

When addition is performed on many computers, four one-bit quantities (besides the usual result) are produced:

$C$ is the the carry-out bit out of the leftmost column

$Z$ is 1 if result is zero and 0 otherwise,

$N$ is the sign bit of result, and

$V$ is 1 if there is "signed overflow" and 0 otherwise.

"Signed overflow" means that two quantities with the same sign produce a sum with the opposite sign.

For subtraction, the bits are set similarly, except that $C$ is the "borrow bit," which is set if the subtraction requires a borrow into the leftmost column. (This convention is reflected in the right hand circuit in Table 7, where the carry-out bit is passed

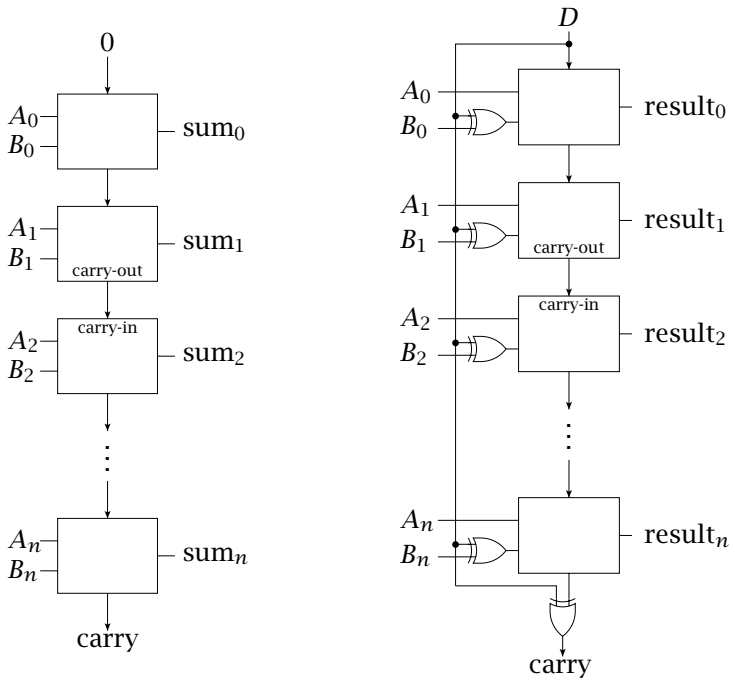


Table 7: A ripple-carry adder (left) and a variation that both adds and subtracts (right). Each box is a full adder.

---

through a xor-gate. The use of $C$ as a borrow bit is common but not universal. Some processors use the carry-out from the last adder without modification.) Signed overflow for the subtraction $x - y$ means that $x$ and $y$ have different signs, and the sign of $x - y$ is different from $x$.

It is not hard to imagine how to modify the ripple-carry adders in Table 7 to produce all four bits.

The flags allow us to compare integer values. When two quantities are subtracted, the flags will tell us whether the first was less than, equal to, or greater than the second. Any decision about the comparison can be based entirely on the flags; the actual result of the subtraction is not required.

To compare two unsigned integers, one subtracts the quantities, discards the result, and observes the flags. For unsigned interpretations, $x < y$ when there is a "borrow" into the leftmost column upon computing $x - y$. That corresponds to the condition $C = 1$. We see that

$x < y$ if $C = 1$,
$x \leq y$ if $C = 1$ or $Z = 1$,
$x = y$ if $Z = 1$,
$x \neq y$ if $Z = 0$,
$x \geq y$ if $C = 0$, and
$x > y$ if $C = 0$ and $Z = 0$.

For signed integers, we compute $x - y$ in a similar fashion and use a different interpretation of the values of the flags:

$x < y$ if $N \oplus V = 1$,
$x \leq y$ if $Z = 1$ or $N \oplus V = 1$,
$x = y$ if $Z = 1$,
$x \neq y$ if $Z = 0$,
$x \geq y$ if $N \oplus V = 0$, and
$x > y$ if $Z = 0$ or $N \oplus V = 0$.

To see why these conditions are correct, consider the example of $x < y$. The result $N \oplus V = 1$ means $N \neq V$, which is to say that the result $x - y$ is negative and correct, or it is non-negative and incorrect. Either way, $x < y$.

## 7 A Memory Circuit

Because a computer performs a sequence of operations, there must be a way of storing the result of one step for use in a later step. The circuit in Table 8 is a one-bit memory, called a *clocked D-latch*. Although the operation is easy to state, the mechanism is a little difficult to understand.

There is a line labeled clock, which is normally low. While it is low, the output line $V$ maintains a value that does not change. It can be used as input to other circuits, possibly for a long period of time. The value of $V$ is changed by raising the clock signal momentarily. When that happens, $V$ takes on whatever value is on the input line $D$. As with an adder, it is easy to imagine several latches connected in parallel to provide memory for a word.

Why does the clocked $D$-latch work? One could trace through the circuit with all possible values of $D$, clock, and $V$, but that is confusing and unenlightening. An alternative is to translate to propositional logic and use some identities. For notational convenience, let us use $C$ for the clock signal and $V$ for the output. Let $X$

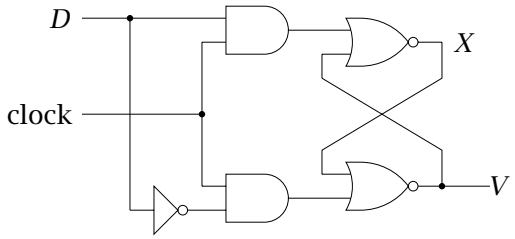


Table 8: A clocked $D$-latch. It is a one-bit memory in which $V$ retains the value that D had when the clock was last high.

---

be the output of the top nor-gate as indicated in Table 8. The outputs of the two and-gates are $D \wedge C$ and $\neg D \wedge C$, so we have

$$\begin{aligned} X &= (D \wedge C) \textbf{ nor } V \\ V &= (\neg D \wedge C) \textbf{ nor } X \end{aligned} \tag{1}$$

When $C$ is false (or 0, signifying that the clock signal is low), both conjunctions are also false. Using the identity $\bot$ **nor** $A = \neg A$, which is easy to check, the equations become

$$\begin{aligned} X &= \neg V \\ V &= \neg X \end{aligned}$$

This is the stable situation mentioned above.

When $C$ is true, the equations (1) become

$$\begin{aligned} X &= D \textbf{ nor } V \\ V &= \neg D \textbf{ nor } X \end{aligned}$$

This is an application of the identity $A \wedge \top = A$ in Table 3. Because working with **nor** is unintuitive, we use another easily-checked identity, $A$ **nor** $B = \neg A \wedge \neg B$, to rewrite the equations.

$$\begin{aligned} X &= \neg D \wedge \neg V \\ V &= D \wedge \neg X \end{aligned}$$

Eliminate $X$ by substituting the right-hand side of the first equation for $X$ in the second, and use a DeMorgan law to obtain

$$V = D \wedge \neg(\neg D \wedge \neg V) = D \wedge (D \vee V).$$

Truth tables now tell us that the only way that this can hold is if $V = D$, which is what we claimed happens when the clock signal is high. When the clock goes low again, $V$ retains its value.

This kind of memory is usually used on the processor unit itself. It is too complicated and expensive to be used for the computer's main memory. The random access memory on a computer is usually implemented with simple capacitors that store an electrical charge.

## 8 Floating-Point Numbers

Many numbers that we encounter in our computations are not integers; they have fractional parts. For completeness, we describe the representation that most computers use for numbers that are not integers.

We introduce a "binary point" that is used analogously to a decimal point. The expression $-10011.11001_2$ is the binary representation for$-19.78125_{10}$. The positions to the right of the binary point represent $1/2$, $1/4$, $1/8$, and so on, so that the fractional part .11001 corresponds to $1/2 + 1/4 + 1/32$. We can write the number as

$$-1.001111001 \times 2^4.$$

(A binary purist would have written the exponential part as $10_2{}^{100_2}$, but that seems excessively cumbersome.) These numbers are called *floating-point* because the binary point "floats" according to the exponent. Unless the number under consideration is zero, we normalize the presentation so that there is a single 1 to the left of the binary point.

### 8.1 Single and Double Precision

A number in binary scientific notation is determined by its sign, fraction, and exponent. Most computers now use IEEE Standard 754, which specifies a format for storing these three quantities. There are two variants of the standard: single precision and double precision. They correspond to `float` and `double`, respectively, in Java.

The *single precision* format uses four bytes, or 32 bits: one bit for the sign, eight bits for the exponent, and 23 bits for the fractional part. The sign bit is 1 if the number is negative and 0 otherwise. The sign bit is followed by the eight bits for the exponent, and then the fractional part is placed in the remaining bits, padded with 0's on the right if necessary.

The exponent is a signed value, but it is not represented as a two's complement integer. Instead it uses the *excess* 127 representation. The value of the exponent is the unsigned value of the eight bits minus 127. In this representation, the exponent can take on values from $-127$ through 128.

The number in the example above would have the single precision representation

1 10000011 00111100100000000000000.

The actual unsigned value of the eight exponent bits is $132_{10}$; subtracting 127 gives the actual exponent of 4. Note that only the fraction proper, and not the 1 to the left

| | Single Precision | Double Precision |
|---|---|---|
| Sign bits | 1 | 1 |
| Exponent bits | 8 | 11 |
| Fraction bits | 23 | 52 |
| Exponent system | excess 127 | excess 1023 |
| Exponent range | $-126$ to 127 | $-1022$ to 1023 |
| Largest normalized | about $2^{128}$ | about $2^{1024}$ |
| Smallest normalized | $2^{-126}$ | $2^{-1022}$ |
| Smallest denormalized | about $10^{-45}$ | about $10^{-324}$ |
| Decimal range | about $10^{-38}$ to $10^{38}$ | about $10^{-308}$ to $10^{308}$ |

Table 9: A summary of the two types of IEEE floating-point numbers. (Adapted from Tanenbaum, Structured Computer Organization, Prentice-Hall, third edition, 1990.)

---

of the binary point, appears. That allows us to squeeze one extra bit of precision into the 32-bit single precision number.

The *double precision* format is similar, except that it uses eight bytes. The sign still takes only one bit. The exponent occupies 11 bits using excess 1023 representation, and the remaining 52 bits are devoted to the fraction. With a larger possible exponent and more bits in the fraction, double precision numbers can have larger values and more "significant figures" than single precision ones. Table 9 compares the two formats.

Notice in Table 9 that the exponent range for single precision is from $-126$ to 127, not $-127$ to 128 as might be expected. The two extreme exponents are reserved for special purposes. Table 10 shows the five forms of floating-point numbers: normalized, denormalized, zero, infinity, and not a number.

The *normalized* form is the one that we have been describing so far. Let $B$ be the "excess" amount in the representation of the exponent; in the case of single precision, $B = 127$. The floating-point number


is in the normalized form if the exponent does not consist of all O's or all 1's. It has the value

$$\pm 1.\text{fraction} \times 2^{E-B},$$

where $E$ is the unsigned value of the exponent bits. As always, the value is negative if the sign bit is 1.

| | Sign | Exponent | Fraction |
|---|---|---|---|
| Normalized | ± | $0 < \exp < \max$ | any |
| Zero | ± | 0 | zero |
| Denormalized | ± | 0 | non-zero |
| Infinity | ± | max | zero |
| Not a number | ± | max | non-zero |

Table 10: The five forms for floating-point numbers. The values in the exponent column are the unsigned values, ranging from 0 to a maximum of $11\ldots1_2$. (Also adapted from Tanenbaum.)

---

The *zero* form is the representation for the number zero. All bits, except perhaps the sign bit, are zero. There are two representations of zero, one "positive" and the other "negative." A computer must be designed to recognize that they are the same.

The *denormalized* form provides a way to represent much smaller positive numbers than would otherwise be possible. The sign bit and the exponent, which consists of all 0's, are interpreted as in the normalized case. The difference is that, with denormalized numbers, the bit to the left of the binary point is taken to be 0 instead of 1. The smallest positive value that can be represented in single precision has the denormalized representation

0 00000000 00000000000000000000001,

which translates to

$+0.00000000000000000000001 \times 2^{0-127}$.

There are 23 fraction bits, so the number is $2^{-150}$, quite a bit less than the smallest normalized positive number of $2^{-126}$.

The two remaining forms, *infinity* and *not a number*, arise as the result of overflow or other errors. The IEEE standards specify the results of operating on these as well as the more conventional forms.

## 8.2 Numerical Computations

Arithmetic on floating-point numbers is analogous to arithmetic on numbers expressed in decimal scientific notation. For addition, one must shift the binary point of one term so that the exponents are the same before adding. This requires the disassembly of the sign, exponent, and fraction part before the operation and subsequent reassembly. Usually, the calculation is carried out by specialized hardware.

Other operations on real numbers, like the calculation of square roots and trigonometric functions, are done using approximation techniques.

The finite range and precision of floating-point numbers can cause errors that would not arise if one were working with perfectly exact mathematical objects. An entire branch of mathematics, numerical analysis, is devoted to studying computation with these approximations that we call floating-point numbers. In this section, we are content to indicate a few of the most common considerations.

Although we sometimes talk about floating-point "reals," many of the usual mathematical laws for real numbers fail when applied to floating-point numbers. For example, the associative law $(x + y) + z = x + (y + z)$ does not hold for floating-point; can you find a counterexample?

The order of operations makes a difference. The expression

$$-1 + \underbrace{2^{-26} + 2^{-26} + \ldots + 2^{-26}}_{2^{26}\text{ terms}}$$

is zero mathematically, but the result will be $-1$ if it is computed from left to right with single precision numbers. Individually, the terms on the right are too small to change the $-1$. It does not help much to start at the right, either; the result is $-0.75$ in that case.

Even in the best of cases, floating-point results will be approximations. This is because of small errors that creep in when input values are converted from decimal to binary and round-off errors that can occur with every calculation. A result of this observation is good advice: *Never test floating-point numbers for equality!* Two numbers may be mathematically equal, but as results of floating-point computations, the bit patterns will be slightly different. Instead of asking whether $x$ and $y$ are equal, always ask whether $|x - y|$ is less than some (small) positive tolerance.

## 9 Other Kinds of Data

So far, we have seen numbers: signed and unsigned integers and floating point numbers. The common non-numeric data types are characters, strings, boolean values, and pointers.

Characters are represented as integers, with a mapping between the letters and the numbers that represent them. One common encoding is ASCII, which represents characters with a single byte. In that system, the byte 0011 0100 is the character `4` (not to be confused with the *integer* 4), and 0110 1101 is the lower-case letter `m`. The specifics of the correspondence are usually not important.

In recent years, people have recognized the limitations of the small size of the ASCII character set. An alternative is the Unicode encoding, which uses sixteen bits and has a much richer collection of letters. Java uses the Unicode system, and its primitive data type `char` is a sixteen-bit unsigned integer.

The boolean values, false and true, can be represented by single bits, but on most computers it is inconvenient to work with a data object smaller than a byte. Most frequently, boolean values are one word, the same size as integers. Usually the word with all 0's corresponds to false and the word whose unsigned value is 1 corresponds to true. Other words may or may not correspond to boolean values. (In C, there is no specific boolean type. That language uses the type `int`, interpreting zero as false and any non-zero value as true. C++ recognizes the same convention, but recently the `bool` type was added to the language, and programmers are encouraged to use it.)

As we shall see in another part of the course, the memory in a computer is simply an array of bytes. Pointers are usually (but not always) interpreted as indices into that array. That is, a location in memory by counting the number of bytes from the beginning, and a pointer is an unsigned integer which signifies a location. It is a matter of some debate whether or not a programmer should make extensive use of this representation.

Arrays are formed by placing the elements next to one another in memory. Strings can be viewed as arrays of characters, and in many programming languages they are exactly that. In other languages, strings are have a more sophisticated representation.

## 10 Transistors

Although most computer scientists can function quite well without knowing exactly how gates work, it is interesting to go one level deeper into the physical computer. Most gates are constructed from semiconductor devices called *transistors*. A transistor is a kind of switch, shown as the circular object in the diagram of Table 11. The signal on one wire, called the *base*, controls the current flowing between two other wires, the *collector* and the *emitter*.

In digital circuits, there are only two relevant voltages—low and high. They are the ones at the points marked ground and $V_{\text{ref}}$, respectively, in Table 11. In other applications, like stereo sound equipment, transistors are used as amplifiers, and the variations in the signal at the base are reproduced with greater magnitude at the collector.

We take the voltage at the point marked "ground" to be zero, and supply a higher voltage at $V_{\text{ref}}$. When the voltage at the base is near zero, there is no connection

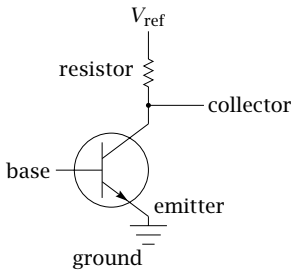


Table 11: A diagram of a typical transistor. The circular object with three leads is the transistor. The emitter is connected to ground, a zero voltage, and the collector is connected to a higher reference voltage.

---

between the collector and the emitter, and the voltage at the collector is close to the reference voltage. Conversely, when the voltage at the base is high, then current can flow between the collector and the emitter. The collector is effectively attached to the ground, and the voltage level is zero there.

The two voltage levels can be interpreted as bits, with the ground state being 0 and a high voltage being 1. When the base of a transistor is 0, the collector is 1, and *vice versa.* Under this interpretation, the transistor illustrated in Table 11 is a not-gate whose input is the base, and output is the collector. With two transistors, we can construct nor- and nand-gates, as shown in Table 12.

Using nor, not, and nand and the identities of Section 2, one can construct all the gates that we have discussed. With a large enough supply of transistors, one can construct a computer. Ten years ago, a processor chip had about twenty million transistors—give or take a factor of two. Today, a typical number is half a billion. One of Intel's high end processors has two billion.

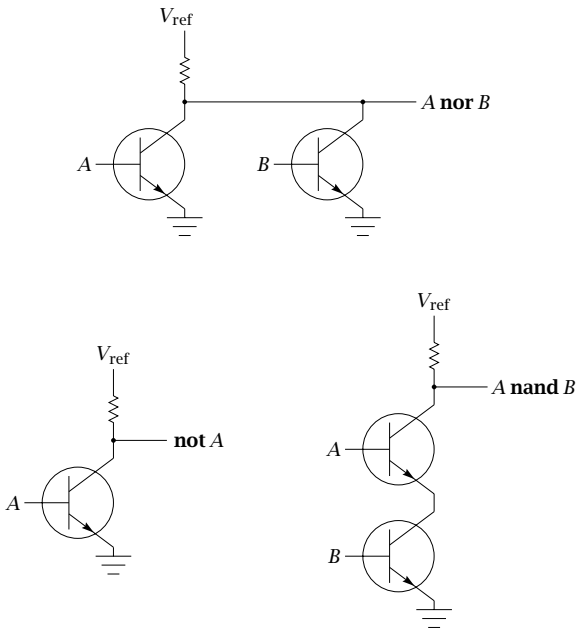


Table 12: Three common gates, nor, not, and nand, constructed from transistors.