Chapter 1: Arithmetic on digital computers

In this chapter we explain what a computer is, how it operates, and how it can be *programmed* to carry out sequences of instructions.

1. What is a computer?

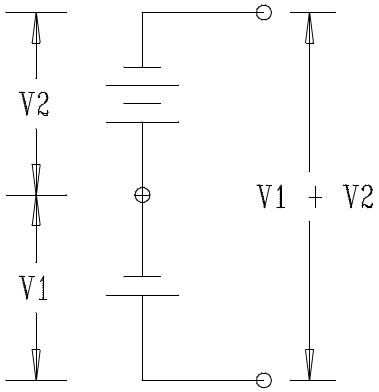
A computer is any device for manipulating numbers. Examples that come to mind are the abacus, the checkered counting table (from which the term "exchequer" is derived), the slide rule, mechanical tabulators and calculators, and more recently, analog and digital electronic computers.

Analog computers

One way to define a digital computer is by contrast with a predecessor, the analog computer. An electronic analog computer represents numbers as voltages. Then it adds two numbers by placing the corresponding voltages in series and measuring the sum, as in the figure to the right. An analog computer can represent differential equations of the form

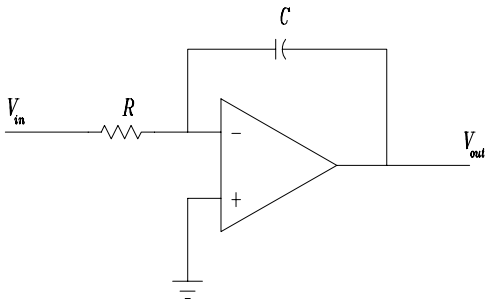
$$\frac{dx}{dt} = f(x, t)$$

using electronic circuits like that shown to the right[1], that integrate time-varying signals, and writing



$$I \overset{df}{=} \frac{dQ}{dt} = f\left(Q(t), t\right\}\Big)$$

(As a practical engineering matter, electronic circuits that *integrate* their input are more stable and less noisy than ones which *differentiate* it[2].)



In the days when an electronic digital computer was a large, slow, unreliable, expensive affair made of mechanical relays or vacuum tubes, analog computers were popu-

1. A capacitor stores charge, the integral of current with respect to time.
2. P. Horowitz and W. Hill, *The Art of Electronics*, 2nd ed. (Cambridge U. Press, New York, 1990), p. 222ff.

lar engineering tools. Their chief disadvantage lay in the realm of numerical precision: the most precisely constructed electronic circuit rarely can maintain voltages within a tolerance of ±0.1%. Digital computers, on the other hand, work with integers (whole numbers) so their precision is perfect (barring internal failure, of course).

Digital computers

So in a nutshell, a digital computer is a machine for manipulating whole numbers. It is called "digital" because in the binary system, integers are represented by strings of ones and zeros, so the digits[3] making up an integer are 1 or 0. These values may in turn be represented by a voltage level on a wire: high voltage might mean 1, and low voltage would then mean 0. So a set of N capacitors can represent an N-bit binary integer, lying in the range 0 (all capacitors discharged) to $2^N - 1$ (all capacitors charged).

Exercise

Why is $2^N - 1$ the largest integer representable by N binary digits?

End of Exercise

We can think of a capacitor bank as a kind of *memory element*. As long as the capacitors maintain their charges the memory cell will "remember" the number stored therein. A practical memory cell must include a method for charging and discharging the capacitors (storing/erasing the number) as well as for observing their settings without modifying them (fetching the number). Obviously, a usable computer memory consisting of many elements (or cells) must also assign a numerical "address"—a unique integer that distinguishes it from others in the system—to each memory cell.

By interconnecting switching circuits that can be set and observed *via* external signals, one may also construct circuits that perform the functions of two-valued logic. The most primitive of these functions is (bitwise) logical **NOT**, that reverses a bit: that is, if the bit was 1, **NOT** makes it 0, and *vice-versa*. A circuit for **NOT** is shown on p. 6 below.

Next in complexity are functions that combine two inputs (switch settings) to produce a single output. For example, the logic function **AND** produces 0 as an output if either of its inputs is 0, or if both are 0.

If we think of 1 as **TRUE** and 0 as **FALSE** we see this makes sense: *a* **AND** *b* will be **TRUE** only if both propositions *a* and *b* are **TRUE**.

Similarly we can define **OR** (**TRUE** if either input is, **FALSE** if both are **FALSE**) and **XOR** ("exclusive or", **TRUE** only if one input is **TRUE** and the other **FALSE**). As we shall see, it is possible to perform

---

3. ***bi***nary digi***t***s are called "***bits***"

arithmetic—addition and multiplication (which is merely repeated addition)—by compounding the primitive logical operations.

Finally, one may combine logical operations to produce complex circuits that interpret numbers as instructions. That is, different input numbers produce different sets of actions. Such a device is called a *central processing unit*, or CPU.

A modern digital computer consists of at least the following components:

- CPU
- Storage elements that hold instructions
- Storage elements that hold data[4]
- Interaction with the outside world:
  - keyboard for input ("standard input device");
  - printer or screen (CRT) for output ("standard output device").

The input device allows us to place instructions in the instruction store and data in the data store (*i.e.*, "program the computer"). The CPU reads instructions sequentially from the instruction store, and performs operations on the data. Finally, the results of its cogitations are returned *via* the output device.

2. Representing numbers

The numbers most of us meet in daily life are expressed in two common notations: the archaic Roman system (found on clocks and movie copyright notices) and the more recent decimal or Arabic positional notation. Other notational systems, such as the Mayan, Egyptian or Babylonian, have been devised other times and places. This section explains the binary number system, the two-valued logic that can be based on it, and how arithmetic may be translated into logical operations.

The binary number system

As anyone knows who has tried it, addition and subtraction are difficult using Roman numerals. Multiplication and long division are well-nigh impossible. It is therefore hardly surprising that little progress in computation—especially in applying mathematics to engineering and finance—took place before 1000 AD when Arabic positional notation[5] reached the West.

Arabic notation represents an integer as a sum of multiples of integer powers of 10 (why it is called the *decimal* system), in fact, as a polynomial (with $x = 10$) whose coefficients are the digits; thus, *e.g.*,

4. These two sets of storage can be the same ("Von Neumann architecture") or distinct ("Harvard architecture"). Both systems are used in practice.
5. Which some say was invented in India.

$$\begin{aligned} 732 &= 7 \times 10^2 \\ &+ 3 \times 10^1 \\ &+ 2 \times 10^0 \end{aligned} \tag{1}$$

The *positional* convention 732 is a shorthand expression of Eq. 1 omitting explicit powers of ten and explicit addition signs.

In elementary school we are taught Arabic notation and its corresponding arithmetic rules by rote. So ingrained are these rules we seldom think of them in algebraic terms—for example, adding two numbers is precisely equivalent to adding two polynomials.

Once we realize what is going on, however, it becomes clear there is nothing sacred about the number 10 (the "x" in the above polynomial of degree 2). In fact, the x in the polynomial could have been any positive integer $N > 1$, and the coefficients (digits) would then be integers in the range $[0,...N-1]$. The integer N is called the *base* of the numbering system.

Several decades of computer programming have boiled down the possible number bases to four that are most commonly employed: decimal (because that is what human beings are taught in school), binary (base 2), octal (base 8) and hexadecimal (base 16).

In base–2 arithmetic the only possible digits are 1 and 0. From our previous remarks about a digital computer being made of switches we now begin to see why the binary system might be useful in computer design.

However, what is the hexadecimal system for? Briefly, $16 \equiv 2^4$. This fact makes it easy to convert between hexadecimal and binary number representations. Hexadecimal, as we shall see, is $4\times$ more compact than binary, as well as being easier for humans to read and write, so for this, as well as additional reasons cited below, hexadecimal has become popular and common.

As we have already remarked, in binary notation the only possible digits are 1 and 0. So typical binary numbers might be 110, 1010, 1101, *etc.* How do we convert these to/from decimal notation? We use the polynomial form, thus writing

$$\begin{aligned} 110 &= 1 \times 2^2 \\ &+ 1 \times 2^1 \\ &+ 0 \times 2^0 \end{aligned} \tag{2}$$

Equation 2 is easily seen to add up to $0 + 2 + 4 \equiv 6$ in decimal notation. Similarly,

$$\begin{aligned} 1010 &= 0 \times 2^0 \\ &+ 1 \times 2^1 \\ &+ 0 \times 2^2 \\ &+ 1 \times 2^3 \end{aligned} \tag{3}$$

$$\equiv 0 + 2 + 0 + 8 = 10$$

Question:
What decimal number does the binary number 1101 represent?

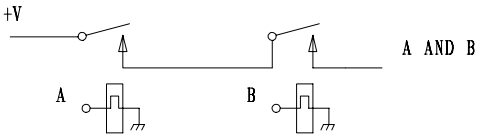
The rules for adding and subtracting binary numbers are precisely the same as those we learned for decimal arithmetic: for example,

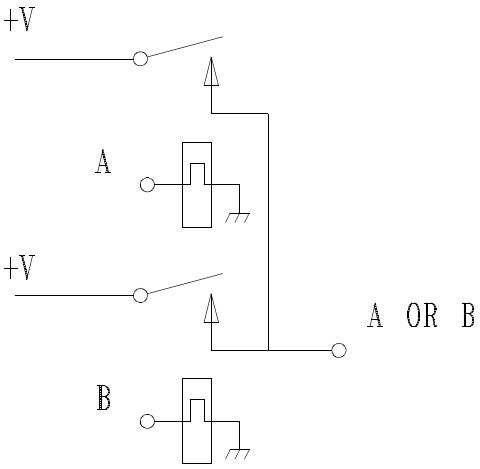
$$\begin{array}{r} 1011 \\ +\,101 \\ \hline 10000 \end{array};$$

we see this is right because 11 + 5 = 16.

3. Arithmetic and logic

George Boole, a British mathematician of the 19th Century, developed a system of 2-valued logical operations, now called Boolean algebra in his honor. This system can be applied to switching circuits to produce devices that perform logical operations. For example, we can think of two relay switches in series as **AND** (the output is "high", *i.e.* **TRUE**, only when both switches are closed, meaning both inputs are **TRUE**). Similarly, two relay switches in parallel produce an **OR** (the output is **TRUE** when either input—or both—is **TRUE**).





Logical operators can be summarized by their *truth tables*. If we represent **TRUE** by 1 and **FALSE** by 0, then we may write the truth table for **AND** below:

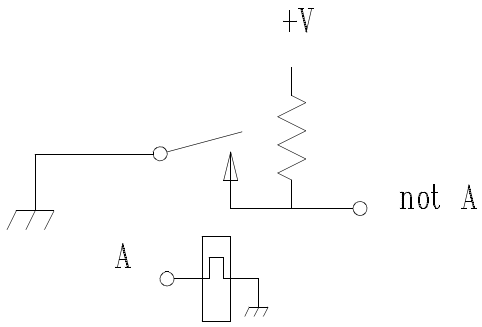
Truth Table for AND

	A 1	0
B		
1	1	0
0	0	0

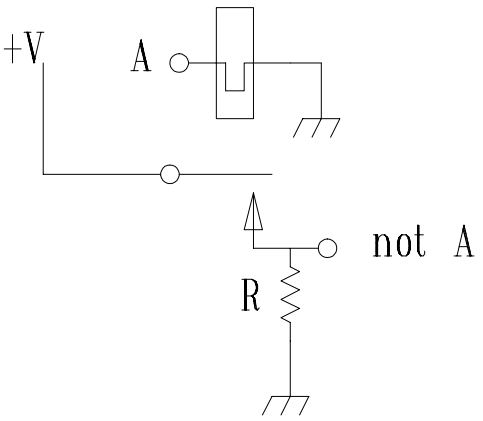
Similarly **OR** has the truth table

Truth Table for OR

B \ A	1	0
1	1	1
0	1	0

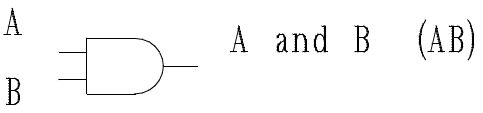


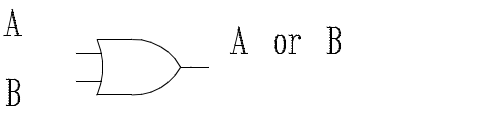
Finally, NOT could be accomplished by one normally open relay and a pull up resistor, or by a normally closed relay and a pull-down resistor, as shown in the two figures to the right.



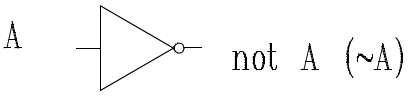
We now explain the logical notation we shall use here: if A is one input and B another, then the result of "ANDing" them together is written AB. That is, AB is a shorthand for "A AND B". Again, the shorthand for "A OR B" is $A + B$. Finally, if we apply the NOT operator to an input, as in NOT A, we write it as $\overline{A}$. These may be tabulated as:

Operation	Notation
A AND B	AB
A OR B	$A + B$
NOT A	$\overline{A}$





The three basic logical operators are sometimes represented as circuit blocks, shown to the right.



The exclusive-OR operator, XOR, has the truth table shown below, and the symbol ⊕.

Truth Table for XOR

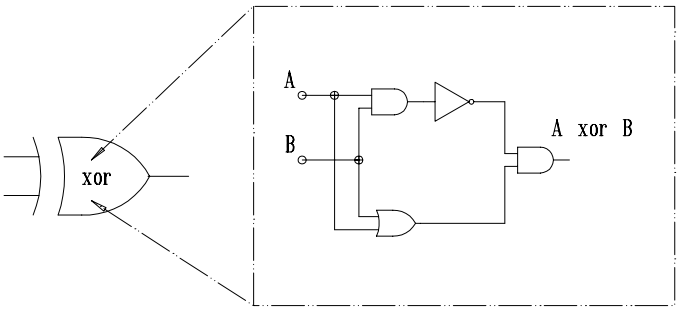
B \ A	1	0
1	0	1
0	1	0

XOR can be constructed from **AND**, **OR** and **NOT**: we see that the inverse of the truth table for **AND**, *i.e.* that of $\overline{AB}$, is:

Truth Table for $\overline{AB}$

B \ A	1	0
1	0	1
0	1	1

If we AND this output with the output from $A + B$, we achieve the truth table of XOR. That is, we can write $A \oplus B = (\overline{AB})(A + B)$. In terms of circuit elements we might express this result as the figure to the right.

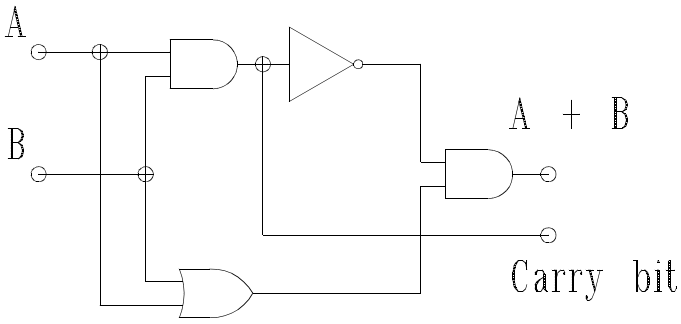


Binary arithmetic

We next need to see how binary arithmetic can be performed using logical elements.

Addition

Consider addition: to add two binary numbers we add their corresponding digits. Clearly, the digit that results from adding a pair of digits has the same truth table as **XOR**. That is, we want two 0's or two 1's on input to yield an output digit 0, whereas 1 and 0 or 0 and 1 should produce 1.



However, this is not the whole story. If we add 1 and 1 we get two, *i.e.* 10 in binary notation. So we must arrange to carry the 1 into the next column, as we do when adding by hand. If we **AND** the two inputs at the same time we **XOR** them, we obtain a carry digit (bit) only when both inputs are 1, which is what we want. Note we do not need an extra **AND** gate for this, the signal we need can be found after the first **AND**, as shown to the right.

There are several ways we can arrange to perform addition with carrying, on numbers larger than 1 (*i.e.* represented by a row of bits). To understand a full addition circuit, we imagine numbers are stored by an array of switches called a "register". Usually registers in modern computers are 4 ("nybble"), 8 ("byte"), 16 ("word"), 32 ("dword") or 64 ("qword") bits wide, although in some special-purpose machines they may be 20 or 21 bits wide. Register widths that are integer powers of 2 are —for obvious reasons—especially convenient.

Imagine adding 4-bit numbers. We could employ a single ½ adder circuit if we treated each pair of input bits sequentially—say from right to left. Or we could do all 4 bit pairs at once using 4 distinct

½ adders. This example is especially noteworthy: here we first encounter the engineering tradeoff between space and time, that governs the relation between program size and speed. We see that if chip "real estate" is limited (so we only have room for one ½ adder circuit) we must expect our 4-bit adder to go four times slower than if we had room for four components.

Suppose we can do 4 bits at once. Call the two input registers A and B, and now the operations AND and XOR will be applied bitwise. We imagine a result register, C, that holds $A \oplus B$ and another, D, that holds AB shifted 1 bit to the left, with 0 as the right-most bit. Shifting one bit to the left is just multiplication by 2, so we write

$$A\ plus\ B = C\ plus D \equiv (A \oplus B)\ plus\ (2 \times AB) \qquad (4)$$

Equation 4 has the unusual property of defining the operation *plus* in terms of itself! However, despite appearances this definition is not circular. It simply means "apply the same operation to C and D, repeating until the end point becomes obvious."

So, clearly, we have

$$C\ plus\ D = (C \oplus D)\ plus\ (2 \times CD) = \Big((A \oplus B) \oplus (2 \times AB)\Big)\ plus\ 4 \times (A \oplus B)(AB)$$

We see that since multiplying any number in a 4-bit register by 16 is the equivalent of zeroing it, and that since the factor $2^n \times$ keeps doubling, at some point we are adding something *plus* zero, so we can stop. We can therefore think of an adder as a cyclic process (in what follows, $\leftarrow$ means "is replaced by"):

Begin
$A \leftarrow A \oplus B$
$B \leftarrow 2(AB)$
Repeat until $B = 0$

This basic idea, with refinements, can be applied to construct 8-, 16-, 32- or 64-bit adders.

Let us work through two examples: we suppose we have a 4-bit register to hold the result of XOR, and another to hold the carry bits 2(AB). First we add 7 and 4:

binary	**decimal**	**operation**
0111	7	
0100	4	
0011		**XOR**
1000		carry = $2AB$
1011		**XOR** again
0000		carry = $2AB = 0$

Then we add 7 and 10:

binary	decimal	operation
0111	7	
1010	10	
1101		XOR
0100		carry = $2AB$
1001		XOR again
1000		carry = $2AB = 0$
0001		XOR yet again
0000		carry = $2AB = 0$

We keep adding the "XORs" register to the "carries" register until the carry bits have fallen off the left end, *i.e.* the "carries" register contains only zero bits. Note the second result is 0001 because $7+10 = 17 = 1 \pmod{16}$.

Subtraction

Having found out how to add numbers, we next ask how to subtract them. It is of course possible to combine logic elements to make a subtractor (with borrowing) and this was done in the early digital computers. To represent numbers with both algebraic signs, an extra bit, called the *sign* bit, was reserved for each number, 0 representing a positive- and 1 a negative number.

However, more recent computers use a cleverer method—2's-complement arithmetic. To subtract one number from another, *i.e.* to evaluate $A - B$, we may rewrite it as $A + (-B)$; but how does this help? The answer lies in a special representation for negative integers. Consider a 4-bit register for simplicity: in binary notation the bit patterns 0000…1111 represent (decimal) integers from 0 to 15: that is 16 possible combinations. But there are also 16 integers in the range -8…+7. So if we wish we can map the latter range (which includes both negative and positive integers) onto the 16 bit patterns provided by a 4-bit wide register. Not all mappings are equally convenient, however. One is especially useful, as we are about to see.

The 2's-complement mapping takes the first 2^{n-1} integers (in a 2^n-wide register) as the positive integers from 0 to $2^{n-1} - 1$. In our example 4-bit register, this means 0…7. The negative integers are represented by

$$-A \equiv \overline{A} \; plus \; 1$$

The number –1 is therefore represented by 1111, –2 by 1110, –3 by 1101, –4: 1100, –5: 1011, –6: 1010, –7: 1001, –8: 1000. We see that in a sense the range –8…7 has been wrapped around so that the gap (between positive and negative integers) occurs at 7, *i.e.* ordinary 8 becomes –8, ordinary 9 becomes –7, …

What is the use of this 2's-complement representation? We can see instantly that first of all, this representation obeys the first law of negation:

$$-(-A) \equiv A \; .$$

But even more important, as one can easily see by trying it with explicit integers, one gets exactly the same result by $A + (\overline{B} + 1)$ as by the more direct $A - B$ (evaluated with the usual subtraction-with-borrowing rules).

Example

Evaluate 7 – 3 in binary notation directly and by adding using the 2's-complement method, assuming 4-bit registers.

Solution

The binary representation for 7 is 0111, and that for 3 is 0011. Thus we have

$$\begin{array}{r} 0111 \\ -0011 \\ \hline 0100 \end{array}$$

using the usual rules. Now we repeat using the 2's-complement method. Writing

$$-3 \rightarrow \overline{0011} + 1 = 1101$$

we have

$$\begin{array}{r} 0111 \\ +1101. \\ \hline 0100 \end{array}$$

Note that in performing the addition we had an extra carried bit that "fell off" the left end of the result register. This is typical of 2's-complement subtraction.

End of Example

A computer with one addition circuit is much easier to build than one that needs a subtractor as well. The bitwise **NOT** operation (that could be done as **XOR** with 1111, and which is usually included as a machine logical instruction anyway) is the only extra cost in the 2's-complement design.

Multiplication

Since multiplication is merely repeated addition, once we have built an adder it is straightforward to construct a multiplier. So virtually every microprocessor possesses the ability to multiply integers. On the other hand, division is a more complex process, more difficult to design hardware for. Early computers often synthesized division as a sequence of multiplications and subtractions. (In the next chapter we shall see how this is possible.) Division was much slower than multiplication that good programming technique required avoiding division as much as possible. This is still true for certain special-purpose microprocessors that lack a division instruction. Most recent general-purpose computers implement a hardware division instruction comparable in speed to multiplication.

With the preceding discussion and examples we have come to the end of our exposition of how simple switching circuits can perform logical and arithmetical operations. Since this is not a text on computer design or circuit theory, we must leave this subject here.

4. Floating point arithmetic

So far we have discussed how digital computers represent integers and compute with them. However, for scientific and engineering applications integers are too limiting. For example, on a machine with 32-bit cells the range of integers that we can represent is

$$-2147483648 \le n \le 2147483647 ,$$

i.e. about $\pm 2 \times 10^9$. For many calculations this is inadequate by many orders of magnitude. For this reason numerical analysts have developed *floating point* number representations.

Most computer languages in common use permit number entry (and display) in a nearly universal format closely related to scientific powers-of-10 notation: 0.936E7 means 0.936×10^7 and so forth. We now inquire how such numbers can be represented internally, that is, in the form of bit patterns in memory cells. Although many such representations have been invented, most languages today use the near-universal IEEE standard[6] for single- (32-bit) or double (64-bit) precision numbers.

Number representation

In general, a floating point number (excluding the sign) of s significant digits is represented as

$$0.d_1 d_2 \ldots d_{s-1} \times \beta^n \equiv \left(d_1 \times \beta^{-1} + d_2 \times \beta^{-2} + \ldots + d_{s-1} \times \beta^{-s+1} \right) \times \beta^n .$$

Here β is the base, or *radix*, and the factor $0.d_1 d_2 \ldots d_s$ —shown in "normalized" form—is called the *significand*. The integer n is the *exponent*. The symbols d_k are the *digits*; they run from 0 to β-1. In normalized form, the digit following the period (binary point in base-two notation) must be non-zero.

Since we are using the binary system, $\beta=2$, the digits are 0 or 1. This means the digit just after the binary point must be 1, hence there is no need to actually record it. By omitting it, we can get one extra digit of precision in the significand. The 32-bit IEEE representation is exhibited in the table below

bit #:	31	30	...	22	...	0
contents:	S	exponent		significand (24 bits)		

and the corresponding 64-bit representation is

bit #:	63	62	...	51	...	0
contents:	S	exponent		significand (53 bits)		

6. ANSI/IEEE 754-1985

We see that the 32-bit representation has a sign bit (0 for positive, 1 for negative) 8 bits that hold an exponent, leaving 23 bits for the significand. Thus the 32-bit representation actually has 24 significant bits, equivalent to about 7 decimal digits. The 8-bit exponent holds integers in the range from 0 to 255. However it is customary to offset the exponent by subtracting 127 from it. The exponents thus run from -127 to +127, equivalent to a dynamic range of $10^{\pm 38}$.

The 64-bit floating point representation reserves 11 bits for the exponent and 52 for the significand, giving about 16 significant (decimal) figures of precision, and a dynamic range of $2^{\pm 1023} = 10^{\pm 308}$.

Exercise
The numerical co-processor units in common use today maintain 80-bit internal registers for their operands. This provides 15 bits for the exponent, and 64 for the significand (of which 3 comprise the "guard", "round" and "sticky" bits used within the fpu for error correction). That is, assuming 61 bits of precision, how many (decimal) significant figures can be represented, and what is the dynamic range?
End of Exercise

In the Intel family of processors and their clone/competitors it is unwise in general to work with 80-bit numbers since the idiosyncrasies of memory organization impose significant time penalties for moving such numbers to/from the main memory. The 32- and 64-bit floats, on the other hand, move in 1 machine cycle (1 "clock") or less[7].

Arithmetic

When we multiply two floating point numbers, we simply multiply their significands, possibly (re)normalize the result (depending on whether the leading bit is 1 or 0) and add the exponents. Similarly, to divide a float by another we perform an appropriate integer division, renormalize the result if necessary, and subtract the exponents.

Addition and subtraction are considerably more difficult, however. We have to denormalize the addend (or subtrahend) that is smaller in absolute value (that is, we shift its binary point leftward), so that the exponents of the two numbers are the same. Then we add the significands, renormalize and adjust the exponent. There is usually less work in addition than multiplication so multiplication (or division) has traditionally been much slower than addition or subtraction. However, recent advances in technology have led to hardware that executes floating point multiplication about as fast as floating point addition. (Division, however, is many times slower, hence should be avoided.)

Roundoff error

7. Less is possible because the Pentium chips incorporate several processors that operate—sometimes—in parallel, thereby achieving "superscalar" performance.

Since digital arithmetic is performed with registers and memory cells of finite length, bits inevitably "fall off" the ends of the result register. Thus a sequence of arithmetic operations can lose precision unless intermediate results are stored in double-length registers. The problem is exacerbated in floating point arithmetic. When we multiply two numbers, an "exact" significand will require twice as many bits as either of the multiplicands. But they only the same number of bits is available to store the result, so information is inevitably lost.

Two conventions are employed for rounding: chopping, in which all bits from bit s (inclusive) are dropped; and rounding, in which one adds 1 to bit s and then chops[8].

Let us represent floating point arithmetic operations in large bold text, and "exact" ones in ordinary text. Then

$$a \,\mathbf{op}\, b = (a \,\mathrm{op}\, b)\,(1+\delta)\,,$$

meaning that the result of the floating point operation is the true result times some factor not quite unity, representing roundoff error. Generally δ is of order of the numerical precision for the operations of multiplication and division. However, suppose we subtract two numbers (of the same sign) that are close in magnitude. The result will be far less precise than either of the subtrahends. For example, subtract[9] 1.0010000_d from 1.0100000_d . Each is known to 8 significant figures, but the result, The result is 0.0090000_d , which is known only to 5 significant figures. In other words, one operation has lost 3 significant figures of precision!

What can we do to save ourselves from this disaster? The answer in general is, "Not much." That is, when we perform certain types of calculation using floating point arithmetic, wherein many subtractions of numbers that are close in size may be expected, then we must expect drastic losses of precision—perhaps to the point of vitiating the calculation. Fortunately when such awful things happen, there is usually a reason (no, I do not mean you have offended the gods). The reason can almost always be stated in mathematical terms—a matrix is ill-conditioned, an equation has many closely spaced roots, and so forth. But such conditions usually arise from some aspect of the physical problem the equations are intended to represent, and can often be stated in physical terminology. When that is the case it is usually possible to see how the problem must be restated to avoid catastrophic roundoff. Only rarely does it pay in such instances to try to solve the original problem by sheer brute force, *i.e.* by extending the precision to so many significant figures that an answer of acceptable precision can still be obtained.

8. In decimal rounding one adds 5 to the s'th digit and then chops.
9. The subscript "d" means we are using decimal notation.