Chapter II
Binary Data Representation

The atomic unit of data in computer systems is the *bit*, which is actually an acronym that stands for BInary digiT. It can hold only 2 values or states: 0 or 1, true or false, ...therefore it can carry the smallest amount of meaningful information possible. This two state property enables us to build simple, cheap, and reliable machines that can process information using electro-mechanical, in the early days of computing, and, currently, electronic components.

Although a bit is small and simple, it can be used to represent any kind of information by simply using many of them. This is similar to the way a few symbols of the alphabet can describe virtually anything. The number of possible values that can be expressed with n bits grows very fast as n increases; each time we use one more bit, the number of represented values doubles. So we do not need too many bits to represent a useful set of values: n bits represent 2^n different values.

II.1 Representing numbers

Representing natural numbers (positive integers, commonly called unsigned integers in CS) is very easy. It is exactly the same as with numbers in the decimal system that humans use (because we have 10 fingers): we simply place the digits horizontally one after the other and the position of each digit determines its significance. The last digit (on the right hand side, called the *least-significant*[1]) counts the units, the one to its left counts the 2's, the next one the 4's and so on. Figure II.1 shows this schematically; it also shows how to convert a binary number to a base-10 number.

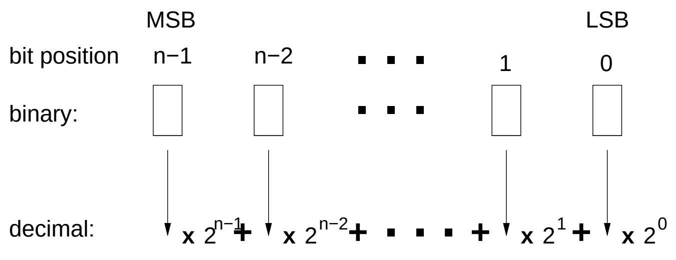

Figure II.1: Binary numbers.

[1]The leftmost bit is called the *most-significant*.

The common arithmetic operations are also exactly the same as with base-10 numbers. See the example below for addition and subtraction:

```
  01101       01101
 +01011      -01011
 ------------------
  11000       00010
```

Although we can use as many digits as we want/need when we do calculations, computers cannot handle infinitely long sequences of bits directly in hardware; it can be done in software with hardware support in the form of special instructions. Therefore, computers operate on data types which are fixed-length bit vectors. Note that this fixed-length constraint applies to all types of data, not just numbers.

The 'natural' unit of access in a computer is called a *word*. Nearly all instructions operate on words. Most computers today use 32-bit words but this is currently being changing to 64. Another commonly used data type is *byte*, which is 8 bits. Other common data types include *short*, 16 bits, and *long*, which is 64 bits.

Overflow

The fixed length of data types can lead to problems when operating on numbers, as the result of an operation may be too large to be represented by the number of bits available to a specific data type. This condition is called *overflow* and it is usually the responsibility of the program (or the operating system) to detect and deal with them. When a number overflows, it 'wraps around' and appears much smaller than what it is supposed to be. For this reason computer arithmetic using n bits is sometimes called modulo 2^n arithmetic, i.e. it appears as if each number has been divided by 2^n and what the computer stores is the remainder.

II.1.1 What about negative numbers?

Since we do not have any symbols other than 0, 1 available (i.e. no '-', '+', etc.), we have to agree to a convention for representing negative numbers using bits. One such convention is the sign-magnitude representation where the first bit (the leftmost) holds the number's sign: 1 for negative, 0 for positive. This representation complicates the design of circuits implementing basic operations, such as addition, therefore is no longer used in modern computers to represent integers.

Instead of sign-magnitude, a representation, called *2's complement*, is used. The idea of how to represent negative numbers in 2's complement, comes from the result one gets when subtracting an unsigned number from a smaller unsigned number. For example, with 4-bit data types, subtracting 0001 from 0000 produces a result of 1111[2], thus 1111 represents -1. In general for an n bit data type in 2's complement, the most significant bit has a negative weighting, while all the others have the usual positive weightings. Figure II.2 shows this schematically. Compare this figure with figure II.1 showing the unsigned integer binary representation.

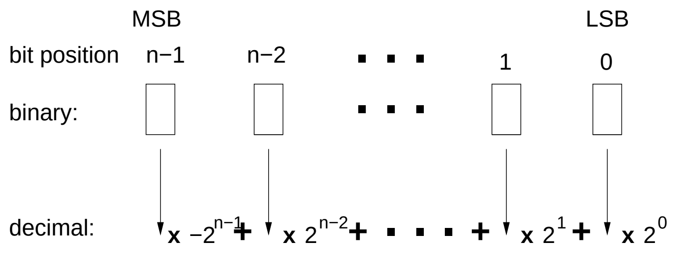

Figure II.2: Signed (2's complement) binary numbers.

Note that all positive numbers start with 0 and all negative start with 1, exactly as with the sign-magnitude representation. This is very convenient as we can rapidly determine whether a number is positive or negative. Another important point is that the lowest negative number 10...00 (-2^{n-1} in base 10) would not fit in n bits if it was converted to positive. Although this asymmetry can be confusing at times, circuits implementing 2's complement arithmetic are the same as those for unsigned numbers, which makes 2's complement a very good representation for signed numbers.

To negate a number x, regardless if it is negative or positive, we simply invert (toggle 0 to 1 and vice versa) all its bits and add 1 (at the least significant bit - LSB). A faster, but trickier, method is to start from the LSB and scan the number toward the left end. While we encounter zeros, we do nothing; the first one seen, is also left untouched, but from then on all bits are inverted.

With 2's complement numbers represented differently than unsigned numbers, the conditions and the behaviour of overflow change too. A positive overflow, i.e. a positive number becoming too large, will produce a negative number, since it will start with a 1. Likewise a negative overflow produces a positive number. This explains what happens in some programming languages when you use an integer to keep a running sum for quite a long time and eventually (and surprisingly) you get a negative result. Note that Java throws an exception when overflow happens.

Sign-extension

In many situations a 2's complement number needs to be converted to a larger data type, for example a signed byte converted into a 16-bit short. This is done by taking the most-significant bit of the byte (the origin) and replicating it

[2]There would also be a borrow carried forward "for ever".

to fill the unused bits of the 16-bit short (the target data type). This operation is called *sign extension*. Sign-extension is based on the fact that as adding 0's to the left of a positive number does not change its value, adding 1's to the left of a negative number, does not change its value too.

Shifting

A useful operation with binary numbers is *shifting*, i.e. moving the bits of a data type to the left or to the right. When shifting left, 0's fill up the empty bit places. Note that this operation is, in effect, a multiplication; when a number is shifted left by n bits, it is multiplied by 2^n.

When shifting right a 2's complement number, it makes sense to fill in the empty spaces with copies of the MSB, for the same reason as with sign extension. This operation is effectively a division by a power of 2. Because shifts are also useful for processing data types other 2's complement numbers, most processors have another version of shift-right where the empty spaces are filled in with 0's. To differentiate the two, the former is usually called *arithmetic* shift right, while the latter is called *logical* shift right.

II.1.2 Hexadecimal representation

Because binary numbers can be very long and therefore rather tedious for humans to use, the *hexadecimal* notation is very commonly used in computer programming. Hex numbers (short for hexadecimal) have 16 as their base and use the usual 0-9 digits and also A-F[3] to represent 10-15. For example 0xaf (0x is a common prefix used in computing to signify that a hex number follows) is $10 \times 16 + 15 = 175$ (remember 0xa = 10, 0xf = 15). Since the base of hex numbers is a power of 2, for every 4 binary digit combination there is a corresponding hex digit. So a binary number can be converted to hex by directly substituting every 4 bits with 1 hex digit and vice versa.

Since bits can be used to represent any type of information, a hex number could also represent data types other than numbers. Remember that hexadecimal numbers are just a convenience, data are represented in binary in computers!

II.2 Floating point numbers

In addition to integers, computers are also used to perform calculations with real numbers. The representation of these numbers in computers is typically done using the *floating point* data type. Real numbers can be represented in binary using the form $(-1)^s \times f \times 2^e$. Therefore a floating point type needs to hold s (the sign), f and e. Note that floating point numbers use sign-magnitude for the fraction part, f.

Under IEEE 754, the most prevalent floating point format, f is *normalised*

[3]Lowercase a-f may also be used.

before storage, which means shifting the binary point so that there is one non-zero digit before the binary point[4]. For example, the value $\frac{3}{4}$ is represented as 1.1×2^{-1}. The value 1.1 is known as the *mantissa* and the -1 as the *exponent*.

Because the non-zero digit before the binary point must always be a 1, it is not necessary to store it. The 32-bit format uses one bit to store the sign of the number, 8 bits to store the exponent, and 23 bits to store the digits of the mantissa after the binary point. In contrast, the 64-bit format uses 11 bits for the exponent, and 52 bits for the mantissa, giving a greater range of numbers and more accuracy.

The details of arithmetic operations on floating point numbers are beyond the scope of this course. These operations can be performed in multiple steps using integer arithmetic, shifting, etc. To speed them up, modern processors contain a specialised floating point arithmetic unit and special instructions. Most processors also provide a separate bank of registers for floating point numbers and there are instructions to transfer values between these and the general purpose registers. The MIPS provides add, subtract, multiply, divide, negate, absolute value and comparison instructions on 32-bit and on 64-bit floating point numbers, as well as instructions to convert numbers between the 32 and 64-bit formats and between floating point and integer representations.

II.3 Characters and Strings

In addition to number crunching, computers are used to process text. Therefore characters (and punctuation marks) have to be encoded in a binary format. The most common representation is ASCII which stores one character into a byte. Java, being a modern language, uses Unicode (16-bits) for representing characters which means that the alphabets of most languages of the world can be encoded.

Words and phrases are created by combining characters into *strings*. The tricky issue here is how to tell where the string ends. One convention (adopted by the C language) is to use a special character to indicate the end of the string. Another, adopted by Java, is to use an accompanying variable, which contains the length of the string, 'packaged' — in an object — together with the string itself.

Warning: Don't confuse (or mix without thinking) characters with numbers! `0` (the number) and '`0`' (the character) are quite different when stored, say, in a byte; the former is stored as `0x00` while the latter is stored as `0x30`. While it makes sense to do arithmetic with the numbers, doing standard binary arithmetic with ASCII represented numbers will not produce the correct results. Try it out, if you don't believe! Note that all data types can be represented in hex and

[4]In this case the FP number is $(-1)^s \times (1 + f) \times 2^e$.

hex numbers use some letters as digits, so be careful not to confuse characters with hex digits.

II.4 Binary representation of non-data

As explained earlier, not only the data but instructions are also stored in the computer memory. Therefore, instructions are also represented in a binary form and need to be *encoded* somehow. We will see the encodings for some MIPS instructions shortly.

The important point to remember is that the contents of a memory location could be anything: an instruction, a 2's complement integer, four characters, ... There is nothing in the memory location itself that describes what is stored there. It is the responsibility of the program to use these contents as they should be used. There is some support from the processor itself though, usually in coordination with the operating system, which ensures that a program cannot modify a part of the memory that it is not supposed to.

A program written in a high-level language 'keeps' data in variables. In the actual computer system most of the data are stored in the memory. Therefore a mechanism for mapping variables to memory locations is needed. The details of this mechanism are revealed in more advanced courses (compilers). What one needs to know, for now, is that we need a kind of 'meta-data', which say where the real data are in the memory. Because these meta-data point to the data, they are called pointers and since memory addresses are unsigned numbers, pointers are numbers themselves and can be 'processed' as such.