Character Sets

bit patterns to represent natural numbers (unsigned numbers) and integers (signed numbers). Let's briefly review how to represent characters.

ASCII system: old-fashioned way to represent characters.1 character fits 1byte. 8 bits 2^8 = 256 char, but ASCII uses lower-order 7 bits to distinguish char,128 different char.Character set is everything you on an American standard keyboard plus some formatting characters. Germans uses umlauts can always use Unicode,which requires 2 bytes per character. Humans use tables to figure out which bit pattern corresponds to which character.

Floating-Point Numbers

Numbers represented inside computers are binary rationals. This has consequences for accuracy with which can represent arbitrary rational numbers, and, a fortiori,arbitrary real numbers.
Computers can't represent all real numbers, but can represent a finite subset of rationals. These are often reasonable approximations to real numbers. (In math, rationals are not a subset of reals). Actually, we will use finite bit patterns to exactly represent a finite subset of binary fractions, or binary rationals.In a binary rational, denominator is a power of 2. As registers have finite length,computers can only represent some binary rational num. gap betn representable binary rationals decreases if use more bits. Thus, we use a finite subset of the binary rationals to provide approx to a bounded subset of the rationals and to a bounded subset of the reals.Every binary rational has a finite decimal expansion, but converse is not true.It helps to spend a moment on fixed-point numbers before moving on to floating-point num,also helps to have a sensible presentation strategy. Our strategy is to develop a consistent, intuitive blackboard notation for floating-point num, and only then worry about how blackboard notation can be adapted to correspond to a register with a fixed number of bits.The reason for caring about fixed-point numbers is that, as we shall see,every floating-point num is a scaled version of a (1+f)-bit binary fixed-point num, where 'f' is num of bits set aside for fractional part of significand. A fixed-point num consists of an integral part and a fractional part,with 2 parts separated by a radix point. If a radix-r fixed-point number has 'm' "integer" digits and 'n' "fractional" digits, its value is: m-1

x = Sigma x_i * r^i = (x_m-1 ... x_0 <point> x_-1 ... x_-n)_r
i=-n

Digits right of radix point are given negative indices and their weights are negative powers of radix. In an (m+n)-digit radix-r fixed-point number system with 'm' whole digits, numbers from 0 to r^m - r^-n, in increments of r^-n, can be represented. We call r^-n the step size, or resolution. Obviously, we will focus on radix 2.For example, in a (2+3)-bit binary fixed-point number system,

2.375 =(1 * 2^1) + (0 * 2^0) + (0 * 2^-1) + (1 * 2^-2) + (1 * 2^-3) =(10.011).

(I won't write the radix specifier "2"; it is understood). In this number system, the values 0 = (00.000) through 2^2 - 2^-3 = 3.875 =(11.111) are representable in steps of 2^-3 = 0.125. For a fixed sum (m+n), there is a trade-off. A larger 'm' leads to an enlarged range of numbers. A larger 'n' leads to increased precision in specifying numbers, which, for us, means better approximations to real numbers. There are standard procedures to convert between decimal and binary fixed-point, which are amusing if somewhat mechanical.However, they do make bite-size homework problems, so here are two examples.

Example: Convert decimal 2.9 to (3+5)-bit binary fixed point. Integer 2 is handled separately: (010.???).

Now .9 * 2 = .8 1
.8 * 2 = .6 1
.6 * 2 = .2 1
.2 * 2 = .4 0
.4 * 2 = .8 0

.8 * 2 = .6 1

Just taking first five fractional digits gives (010.11100) = 2.875.
But we can do something with the sixth digit: since it is 1, we can add 1 to the current approximation. This gives (010.11101) = 2.90625, which is closer to 2.9 than is (010.11100). This last refinement is called rounding. We won't use rounding even if it often improves accuracy.
Example: Convert (1101.0101) to decimal.This is obviously 13 and 5/16, which is 13.3125. The binary rational is 213/16.
Although there are several ways to encode signed fixed-point numbers, we will choose just one. In fixed point notation, we will write sign explicitly. -5.75 in (3+5)-bit binary fixed point is -(101.11000).We need some terminology.When we get to floating-point num, we see they are represented as: +/- s * 2^e.
3 components of representation are: 1) sign 2) significand 's', 3) exponent 'e'. That is, plus or minus some binary rational times 2 raised to some positive or negative (integer) power.We can use "blackboard notation" as a stepping stone to representing floating-point numbers in registers. Blackboard notation is easier because

i) we use trivial representations of sign and exponent,
ii) we use examples of terminating significands, or for nonterminating ones make use of repeating binary expansions of indeterminate length.

Let's try some examples in blackboard notation. Recall that, in scientific notation, we usually write a single digit to the left of the decimal point, e.g., 3.26 * 10^5. We adopt the same convention for _binary rationals_, e.g., 1.01 * 2^-2 = 0.3125. That is, we write a positive binary rational with a single nonzero digit to the left of the radix point, and continue with the fractional part, all this times some (positive or negative) power of two.
Example: Convert (1101.1010) to blackboard floating point.
(1101.1010) = (1101.1010) * 2^0
= (1.1011010) * 2^3
= (1.1011010) [3]
Example: Convert (0.000111) to blackboard floating point.
(0.000111) = (0.000111) * 2^0
= (1.110000) * 2^-4
= (1.110000) [-4]
When move binary point left, increase exponent;when move binary point right, we decrease exponent.
This is almost as far as we can take our blackboard notation. To show actual bit patterns, we need to know how many bits are available for the significand, and how many bits are available for the exponent. Also, in standard-computer bit patterns, we will drop the (thus far explicit) "1." in that it goes without saying. This allows us to explicitly represent only part of the significand.
Consider a 16-bit register. One bit is used to represent sign, leaving us with 15 bits. After reflection, we choose to use 4 bits (one hex digit)to represent the signed exponent in two's complement semantics. That leaves only 11 bits to store fractional part of the significand.
Example: Represent 5/16 as a floating-point number in a 16-bit register. I will use the order: sign, exponent, fractional part. Now,
5/16 is 1.01 * 2^-2. The 4-bit exponent in two's complement is 1110(hex e).
So the full 16 bits are: 0 | 1110 | 01000000000, which is 7200 in hex.
To sum up,use: i) one bit for sign, ii) 4 bits to represent exponent in 2's complement,iii) 11 bits for digits to the right of the radix point in the true significand.
Example: Put (1.1010101) [-3] in a 16-bit register.
This is: 0 | 1101 | 10101010000, which is 6d50 in hex.
Example: Put (1.11) [4] in a 16-bit register.
This is: 0 | 0100 | 11000000000, which is 2600 in hex.
Example: Put 1/5 in a 16-bit register.
0.2 = (0.<0011>*) = (1.<1001>*) [-3]
This is: 0 | 1101 | 10011001100, which is 6ccc in hex.

Other people have given this matter much thought. In the current IEEE floating-point standard, a floating-point number has three components:
i) a sign +/-, ii) a significand 's', and iii) an exponent 'e'.
The exponent is a signed integer represented in biased format (a fixed bias is added to it to make it into an unsigned number). We are not responsible for exponent bias. I mention it only for completeness.
The significand is a fixed-point number in the range [1,2). Because the binary representation of the significand always starts with "1.", this fixed "1." is omitted ("hidden"), and only the fractional part of the significand is explicitly represented.
What is represented in this way?Ans: +/- s * 2^e. short (32-bit) and long (64-bit) floating-point formats. Short format ranges 1.2 * 10^-38 to 3.4 * 10^38. long format ranges from 2.2 * 10^-308 to 1.8*10^308.
Let's say a few more words about current IEEE standard. If a word has 32 bits, and there are 8 exponent bits,then significand has 23 bits (plus 1 hidden),significand range is [1,2 - 2^-23](or[1,2), if you prefer), and the exponent bias is 127. There are also bit patterns for 0, Infinity, and Not-a-Number (NaN). Again, we are not responsible for this.
Let's see how a floating-point number would be laid out in a 32-bit word. Bit 31 would be the sign bit of the binary rational. Then,bits 30 through 23 (8 bits) would store the 8-bit exponent field (including the sign of the exponent), while bits 22 through 0 (23 bits) would store a 23-bit binary rational in the range [0,1 - 2^-23]. (I oversimplify slightly; the value 0 must be handled separately).
To summarize the current IEEE standard for floating-point numbers, in the short (32-bit) format, we have the sign bit, 8 bits for the exponent, and 23 bits for the fractional part of the significand (the hardware adds the implicit hidden "1."), while in the long (64-bit format), we have the sign bit, 11 bits for the exponent, and 52 bits for the fractional part of the significand (the hardware adds the implicit hidden "1."). Most computers offer double-precision floating point. As we have just seen, we increase the exponent field from 8 bits to 11 bits,and the fraction field from 23 bits to 52 bits. Although double precision does increase the exponent range, its primary advantage is in its greater precision, which leads to greater accuracy (closer approximation of reals by binary rationals).
Note: In both 32-bit and 64-bit machines, single-precision floating-point arithmetic means use of 32-bit registers, and double-precision floating-point arithmetic means use of 64-bit registers.means use of 32-bit words.

Instruction Formats

Although instruction formats have consequences in terms of the ease with which certain operations can be carried out, and whose simplicity and uniformity is absolutely critical to the speed with which a sequence of machine instructions can be pipelined efficiently, they are not in themselves very interesting. Having briefly reviewed the encoding structures for different types of numbers, let us now consider instruction encoding.
In one computer, a typical machine instruction is 'add r1,r2,r3', which causes the values in 'r2' and 'r3' to be added, and the sum put into 'r1'.
A machine instruction for an arithmetic/logic operation specifies an opcode, one or more source operands, and, usually, one destination register. The opcode is a binary code (bit pattern) that specifies an operation. The operands of an arithmetic or logical instruction can come from a variety of sources. The method used to specify where the operands are to be found, and where the result must go, is called the addressing mode, or addressing scheme. For now, we assume that all operands are in registers, and discuss other addressing modes gradually.
In the computer mentioned, there are three instruction formats.
1)Register or R-type instructions operate on 2 registers identified in 'rs' and 'rt' fields, and store result in register 'rd'.
R: opcode rs rt rd <other stuff>
6 bits 5 bits 5 bits 5 bits 11 bits = 32 bits
2) Immediate or I-type instructions come in two flavors. The general formatfor an I-type instruction is:

I: opcode rs rt immediate (immediate =operand or offset)
6 bits 5 bits 5 bits 16 bits = 32 bits
i) In true immediate instructions,16-bit immediate field in bits 0 - 15 holds a 16-bit signed integer that plays the same role as 'rt' in the R-type instructions; in other words, the specified operation is performed on 'rs' and the immediate operand, and the result is written into 'rt', which is now a destination register.
Example: 'daddiu r1,r1,#-8' lays out as
[daddiu] [r1] [r1] [-8]
6 bits 5 bits 5 bits 16 bits
We add the 16-bit immediate (here, -8) to 'r1' to compute a number (often a memory address). Then, we write this number into 'r1'.
ii) In load, store, and branch instructions, the 16-bit field is interpreted as an offset, or relative address, that is to be added to the base value in register 'rs' (resp., the program counter) to obtain a memory address for reading or writing memory (resp., transfer of control).
For data accesses,offset is num of bytes forward(positive) or backward(negative) relative to base address.
For branch instructions,offset is in words,given that instructions always occupy complete 32-bit memory words. To interpret the 16-bit signed integer as a word-address offset, we multiply by 4. Since offsets can be positive or negative, this allows for branching to other instructions within +/- 2^15 (32,768) instructions of the current instruction.
We describe two data-transfer instructions, and a branch instruction, separately.
D: opcode rs rt immediate -- data transfer 6 bits 5 bits 5 bits 16 bits
Example: 'l.d f6,-24(r2)' lays out as
[l.d] [r2] [f6] [-24]
6 bits 5 bits 5 bits 16 bits
We add immediate byte-offset -24 to 'r2' to determine a memory address.Then, we load double-precision floating point number (64 bits) from that memory location and put it into floating-point register 'f6'.
Example: 's.d f6,24(r2)' lays out as
[s.d] [r2] [f6] [24]
6 bits 5 bits 5 bits 16 bits
We add immediate byte-offset 24 to 'r2' to determine a memory address.Then, we store double-precision floating point number (64 bits) in floating-point register 'f6' into that memory location.
B: opcode rs rt immediate – conditional branch 6 bits 5 bits 5 bits 16 bits
Example: 'bne r1,r2,loop' lays out as
[bne] [r1] [r2] [loop]
6 bits 5 bits 5 bits 16 bits
We compare register 'r1' and register 'r2'. If they are not equal, we add the word-offset derived from the immediate 'loop' to the current value of PC as the new value of PC.That is, we add shifted, sign-extended 16-bit signed integer 'loop' to memory address of the branch instruction(actually, instruction just below).
Wehave already seen how far we can go from the current instruction.
3) Jump or J-type instructions cause unconditional transfer of control to the instruction at the specified address. Since only 26 bits are availablein the address field of a J-type instruction, two conventions are used. First, as the 26-bit field is assumed to specify a word address (as opposed to a byte address), two zero bits are appended to the right.
We now have 28 bits. The four missing bits are stolen from PC, as will be described shortly.
J: opcode partial jump-target address – unconditional branch 6 bits 26 bits = 32 bits
Example: 'j done' lays out as
[j] [done]
6 bits 26 bits
Using our 2 tricks,we expand partial jump-target address 'done'into a full 32-bit address, and transfer control there.

Addressing modes

Addressing mode:method by which location of an operand is specified within an instruction. Our computer uses five addressing modes, which are described as follows:

1. Immediate addressing: The operand is given in the instruction itself. A simple example is 'daddi'. A second example is shown in its full glory: daddui r1,r1,#-8

Because of the 'u', this is a "natural-number" add, which makes sense, for example, when computing memory addresses. After all, addresses are themselves natural numbers, and overflow is unlikely here (that would take a programmer who was literally asleep). Our odd-looking instruction takes nonnegative 'r1', adds the immediate -8 to it, but does not bother to check for overflow. Of course, if 'r1 = 7', this is a problem.But suppose we are not computing memory addresses, and write: daddi r1,r1,#-8

Again, the second operand is given in the instruction itself. The first operand is 'r1'. The 16-bits holding -8 are the second operand(or actually the lower half of it). However, with this opcode, we must most definitely check for overflow. After all, 'r1' might be a large, negative number.

2.Register addressing: The operand is taken from, or the result placed into, a specified register. Here is an example with two source registers and one destination register:mul.d f4,f2,f6 The contents of registers 'f2' and 'f6' are read. A double precision floating-point multiply takes place, and result is placed into register 'f4'.

3.Base addressing: The operand is in memory and its location is computed by adding a byte-address offset (16-bit signed integer) to the contents of a specified base register.Examples:

l.d f6,-24(r2) ; f6 is a destination register

s.d f6,24(r2) ; f6 is an operand register

4. PC-relative addressing: This is same as base addressing,except "base" register is always PC, and a hardware trick is used to extend the signed-integer offset to 18 bits. Namely, we multiply by 4 to obtain a word-address offset, which is then sign extended to 32 bits.This addressing mode is used in conditional branches. **Example: beq r1,r2,found**

Again, the 16-bit signed number is multiplied by 4 (making it a word-address offset) and the result is added to PC. This allows branching to other instructions within +/- 2^15 words of the current instruction.

5. Absolute addressing: The addressing mode for unconditional branches is different because we don't really have a "base" register.

Example:j done

Here, we need fancier hardware tricks. 'done' is a 26-bit natural number.Multiplying by 4 gives us a 28-bit natural number. Now, if we pad the front of 'done' with the four leading bits of PC, we obtain a genuine 32-bit (word) address. Thus, we can "goto" instructions much further away than those within +/- 2^15 words of the current instruction.