Character Sets

We discussed how to use bit patterns to represent natural numbers ("unsigned numbers") and integers that might be negative ("signed numbers"). Let's briefly review how to represent characters.

The ASCII system is a slightly old-fashioned way to represent characters.

Each character fits into a byte. Normally, 8 bits would give us 2^8 = 256 characters, but ASCII only uses the lower-order 7 bits to distinguish characters. ASCII is thus able to represent 128 different characters.

The character set is everything you can type on an American standard keyboard plus some formatting characters. Germans who can't live without their umlauts can always use Unicode, which requires two bytes per character. Humans typically use tables when they try to figure out which bit pattern corresponds to which character.

Floating-Point Numbers

All numbers represented inside computers are _binary rationals_ (definition follows). This, and the finite size of computers, has consequences for the accuracy with which we can represent arbitrary rational numbers, and, a fortiori, arbitrary real numbers.

Computers can't represent all binary rationals, because of finite register size, but they can represent a bit-limited subset of the binary rationals. These are often reasonable approximations to rationals and real numbers.

Of course, bit-limited bit patterns can _exactly_ represent a finite subset of the binary rationals. Definition: A binary rational is a rational whose denominator is a power of 2.

Schemes that use bit-limited binary rationals to approximate fractional numbers introduce a trade-off between _precision_ and _range_. We get precision when the binary rationals are close together; we get range when the difference between the smallest and largest binary rational is large.

A number 'n' has a finite binary expansion if 'n' is a binary rational.

A reduced binary rational fraction is a number of the form m/2^n, where m < 2^n and the radix 2 is not a factor of 'm'. The binary expansion of such a number has exactly 'n' binary places to the right of the binary point. This identity is actually true for arbitrary radix.

It helps to spend a moment on fixed-point numbers before moving on to floating-point numbers. It also helps to have a sensible presentation strategy. Our strategy is to develop a consistent, intuitive _blackboard notation_ for floating-point numbers, 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 every floating-point number is a scaled version of a (1+f)-bit binary fixed-point number, where 'f' is the number of bits set aside for the fractional part of the significand.

A fixed-point number consists of an _integral part_ and a _fractional part_, with the two parts separated by a _radix point_. If a radix-r fixed-point number has 'm' "integer" digits and 'n' "fractional" digits, its value is:

$$x = \sum_{i=-n}^{m-1} x_i * r^i = (x_{m-1} \ldots x_0 \langle point \rangle x_{-1} \ldots x_{-n})_r$$

The digits to the right of the radix point are given negative indices and their weights are negative powers of the 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_. We will focus on radix 2.

In this case, any two adjacent (m+n)-bit fixed-point binary rationals have a fixed distance between them, viz., 2^-n.

Example:

In a (2+3)-bit binary fixed-point number system:
decimal 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 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 the first five fractional digits gives (010.11100) = 2.875.

But we _could_ 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 just write the sign explicitly.

Thus, -5.75 in (3+5)-bit binary fixed point is just -(101.11000).

We need some terminology. When we get to floating-point numbers, we will see that they are logically represented as: +/- s * 2^e. The three components of the representation are: 1) the _sign_, 2) the _significand_ 's', and 3) the _exponent_ 'e'. That is, plus or minus some binary rational times 2 raised to some positive or negative (integer) power.

This is not as foreign as it looks. In the decimal world, we often write, say:

-3.45 * 10^7, where 3.45 is a (decimal) rational in the range [1,10).

Note: The phrase "decimal rational" is ambiguous: it might be a rational number represented in decimal, or it might be an analogue of "binary rational".

We can use "blackboard notation" as a steppingstone to representing floating-point numbers in registers. Blackboard notation is easier because

i) we use trivial representations of sign and exponent, and
ii) we use examples with terminating significands, and---for nonterminating ones---make

use of repeating binary expansions of infinite 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 normalized blackboard floating point.

(1101.1010) = (1101.1010) * 2^0 <identity>

= (1.1011010) * 2^3 <normalize>

= (1.1011010) [3] <just being lazy>

Example:

Convert (0.000111) to normalized blackboard floating point.

(0.000111) = (0.000111) * 2^0 <identity>

= (1.110000) * 2^-4 <normalize>

= (1.110000) [-4] <just being lazy>

When we move the binary point left, we increase the exponent; when we move the binary point right, we decrease the 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 (a prefix of) the fractional part of the significand.

Note: A simpler word than "significand" is "coefficient". I will use "coefficient".

Consider a 16-bit register. One bit is used to represent the 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 the _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, we use:

i) one bit for the sign,
ii) 4 bits to represent the exponent in two's complement,
iii) 11 bits for the bits to the _right_ of the binary point in the full significand. (Again, these "right-of-point" bits are called the _fractional part_ of the significand).

Example:

Put (1.1010101) [-3] into a 16-bit register.

This is: 0 | 1101 | 10101010000, which is 6d50 in hex.

Example:

Put (1.11) [4] into a 16-bit register.

This is: 0 | 0100 | 11000000000, which is 2600 in hex.

Example:

Put 1/5 into a 16-bit register. Here, the true value of the fractional part of the binary-rational coefficient is an infinite binary expansion.

0.2 = (0.<0011>*) = (1.<1001>*) [-3] This is still blackboard notation.

This becomes: 0 | 1101 | 10011001100, which is 6ccc in hex.

But any notion of fixed-point has vanished from the last example. So we have another possible notation. Pretending we are mathematicians, we could write:

0.2 = 0.(0011)*

This is just the infinite binary expansion of 0.2 without normalization, and without scale factor.

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'
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 (a prefix of) the fractional part of the significand is explicitly represented.

What is being represented in this way? Answer: +/- s * 2^e.

There are short (32-bit) and long (64-bit) floating-point formats. The short format ranges from 1.2 * 10^-38 to 3.4 * 10^38. The long format ranges from 2.2 * 10^-308 to 1.8 * 10^308.

Exercise:

Calculate the four bounds for strictly positive single- and double-precision floating-point binary rationals. Single: The smallest strictly positive binary rational is 2^-126, which is (approximately) 1.2 * 10^-38; the largest binary rational is (approximately) 2^128, which is (approximately) 3.4 * 10^38. Double: The smallest strictly positive binary rational is 2^-1022, which is (approximately) 2.2 * 10^-308; the largest binary rational is (approximately) 2^1024, which is (approximately)

1.8 * 10^308. Zero lies outside either range.

Note: 1.2 is more precisely 1.17549435... . The others are similar.

The distance between two adjacent (m+n)-bit floating-point binary rationals varies considerably. Consider single-precision binary rationals, which we may characterize as (8+23)-bit floating-point binary rationals. When the scale factor is near 1, the gap is near 2^-23. Consider the smallest strictly positive floating-point binary rational, which is 2^-126. The distance to the next largest binary rational is 2^-23 * 2^-126 = 2^-149. Consider the largest floating-point binary rational, which is roughly 2^128. The distance to the next smallest binary rational is 2^-23 * 2^127 = 2^104. The gap between adjacent floating-point binary rationals is a function of their position on the real line.

Let's say a few more words about the current IEEE standard. If a word has 32 bits, and there are 8 exponent bits, then the significand has 23 bits (plus 1 hidden), the 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.

I should add that there is no free lunch, and that the two most negative exponents in the exponent range are _removed_ to give us bit patterns that are used in special representations of special values.

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.

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 the two registers identified in the in the 'rs' and 'rt' fields and store the 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 format for 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, the 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.

<u>Example:</u>

'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_, the offset is the number of _bytes_ forward (positive) or backward (negative) relative to the base address. In contrast, for _branch instructions_, the 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 must multiply by 4 to get the number of bytes. 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

```
   [s.d]   [r2]    [f6]    [24]

   6 bits  5 bits  5 bits  16 bits
```

We add the immediate byte-offset -24 to 'r2' to determine a memory address. Then, we load the 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 the immediate byte-offset 24 to 'r2' to determine a memory address.
Then, we store the 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 the shifted, sign-extended 16-bit signed integer 'loop' to the memory address of the branch instruction.

We have 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 available in 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 two tricks, we expand the partial jump-target address 'done' into a full 32-bit address, and transfer control there.

Addressing modes

_____

Addressing mode is the method by which the 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 <subi r1,r1,8 in RISC-V>

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 reasonably unlikely here. 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.

In truth, error detection is done at a higher level in the hardware/software stack. Still, it is prudent to exercise great caution in programming in this manner.

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 the 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 the same as base addressing, except that the "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 _subsequently_ 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.

Special Values in IEEE Floating Point (we are not responsible for this);

_____

I have told you that the exponent field is in two's complement. Suppose we have a 4-bit exponent field. Then:

bit pattern	true exponent
0000	0
0001	1
...	...
0111	7
1000	- 8
1001	- 7
1010	- 6
...	...
1111	- 1

I don't teach the five special values (+/- 0, +/- infinity, and NaN), but I agree that they are necessary in real computers. To represent these values, we need special codes. But this means will we have to sacrifice certain bit patterns that we could otherwise have used.

4-bit two's complement runs from -8 to 7. That's 16 possible exponent values.

The IEEE standard steals two exponent values---the two smallest values, leaving you with only 14. Here are the fourteen that remain:

bit pattern	true exponent
0000	0
0001	1
...	...
0111	7
**********	
1000	- 8 <stolen to participate in special codes>
1001	- 7 <stolen to participate in special codes>
**********	
1010	- 6
...	...
1111	-1

We are left with an exponent range of -6 to 7, as I show in my lectures. Why might a student care? If someone asks you, what is the minimum normalized IEEE floating-point number in 16 bits, with a sign bit and 4 exponent bits, you need to know whether that minimum magnitude is:

1.<0:11> * 2^-6 or 1.<0:11> * 2^-8

I generally ask for maximum magnitudes, so this problem never arises, but if I were to ask for minimum magnitudes, I would tell you whether to use -6 or -8.

Stealing is real; bias is just a convenience. Amateurs usually mix stealing and bias so awkwardly---incompetently, I would say---that the student loses the big picture.

Why is bias convenient? Well, the normal range is -8 to 7. Take the largest number (i.e., 7), and add it to each of the two stolen exponents. (We also add 7 to the exponent bit patterns that have not been stolen; this does _not_ change their true values, only their representations).

```
 1000          - 8    <stolen to participate in special codes>
+0111
 ____
 1111                 <no longer an exponent; now a special code part>


 1001          - 7    <stolen to participate in special codes>
+0111
 ____
1|0000                <no longer an exponent; now a special code part>
```

That is why 32-bit normalized IEEE floating-point numbers have magnitudes that run from 2^-126 to (approximately) 2^128 (in decimal, 1.2 * 10^38 to 3.4 * 10^38---roughly).

In contrast, 64-bit normalized IEEE floating-point numbers have magnitudes that run from 2^-1022 to (approximately) 2^1024 (in decimal, 2.2 * 10^-308 to 1.8 * 10^308---roughly). See Lecture 3.

However, you are _still_ not responsible for special values, special encodings, exponent stealing, or exponent bias. This appendix is for information only. Disclosure: I may talk about exponent stealing.