Chapter 2: Computer Memory and Storage, Representing Numbers, Random Numbers

Memory and Computer Representation of Data

The computer memory contains millions of transistors which at any time can be in one of two physical states, usually labelled 0 and 1.

Each transistor carries 1 bit of information.

$$\begin{aligned} 8 \text{ bits} &= 1 \text{ byte} \\ 2^{10} \text{ bytes} &= 1 \text{ K byte} \equiv 1024 \text{ bytes} \\ 2^{20} \text{ bytes} &= 1 \text{ Mega byte} \\ 2^{30} \text{ bytes} &= 1 \text{ Giga byte} \\ 2^{40} \text{ bytes} &= 1 \text{ Tera byte} \end{aligned}$$

How many bits are needed?

(A) Characters

English:	26	lower case:	a,b,c,....
	26	upper case:	A,B,C,....
	10	numerals:	1,2,3,....
≈	15	'extras'	+−.,;:* etc.
Total ≈	77		

Given N bits we can create 2^N different combinations
Example: N=2, $\{0,0\}$, $\{0,1\}$, $\{1,0\}$, $\{1,1\}$
N=6 gives $2^6 = 64$ combinations, too low for CHARACTERS.
In fact use 1 byte = 8 bits for each character.

(B) Book

say 40 lines × 80 characters per page,
1 page = 3200 bytes ≈ 3 K bytes.
300 pages ≈ 1 M byte.

(C) Music

To detect a frequency of 20KHz you need 40,000 valves of pressure per second. If each valve is given by 16 bits (CD quality) this amounts to
80 K Bytes / second
10 M Bytes / minute
650 M Bytes / 65 minute CD...about the capacity of a single disc.

(D) Pictures

Each dot on a computer image is called a PIXEL.
A good screen might have 1080 × 780 ≈ 1 M PIXELS.
Each pixel has colour and brightness specified by (about) 3 bytes.
Therefore a single image requires 3 Mb.
(A chemical photograph contains approximately 30-40 Mb of information.)

Representation of Numbers

Integers: Two's Complement Arithmetic

Integers are usually stored using an integer number of bytes, hence one usually refers to 8-bit (see below), 16-bit, 32-bit (default value on many computers) or 64-bit integers. The number of bits controls the range of integers that can be stored, e.g., 8-bits allows $2^8 = 256$ combinations, and so allows only 256 integers to be stored.

One method for storing (8-bit) integers on the computer, that leads to a convenient binary 'arithmetic', is known as the *two's complement method*. According to this method, the left-most bit represent $-2^7 = -128$, (i.e. minus the expected value), so that, for example

$$11010101 = -1 \times 2^7 + 1 \times 2^6 + 1 \times 2^4 + 1 \times 2^2 + 1 \times 2^0 = -43$$

i.e. the left-most bit represents -2^7 then 2^6, 2^5,...., 2^0. This allows the range $[-128, 127]$ to be stored.

How are these numbers added and subtracted?

Addition

Addition proceeds just like ordinary decimal addition

Example

$$\begin{array}{rcr} & 1001\,1101 & -\,99 \\ + & \underline{0001\,0100} & +\,20 \\ & 1011\,0001 & -\,79 \end{array}$$

Subtraction

For subtraction we exploit the following theorem and then use addition.

Theorem

Suppose $f(n)$ is a function that flips all the bits in n (e.g. $f(1001\,0111) = 0110\,1000$) then

$$-n = f(n) + 1$$

Proof

$$\begin{aligned} f(n) + n = 1111\,1111 &= -1 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + ... + 1 \times 2^0 \\ &= -128 + (64 + 32 + 16 + 8 + 4 + 2 + 1) \\ &= -1 \qquad \text{as} \quad \sum_{i=0}^{m} 2^i = 2^{m+1} - 1 \end{aligned}$$

Therefore a calculation

$$a = b - c$$

can be rewritten as

$$a = b + (-c) = b + f(c) + 1$$

This uses only addition and bit flipping, both of which are fast operations. Note, however, that in two's complement arithmetic we have the unusual results

$$2 \times 64 = -128 \quad \text{and} \quad 1 + 127 = -128$$

both because the left bit = -128. Also

$$2 \times -128 = 0,$$

as multiplying by 2 shifts bits to the left, i.e.

$$2 \times 0000\,1101 = 0001\,1010$$

(compare multiplying by 10 in decimal!)

Reals and Round-Off Error

Reals are stored as floating point numbers

$$\pm 1.\underbrace{fffffffffff}_{\text{mantissa}} \times 2^{\overbrace{eeeeeeee}^{\text{exponent}}}$$

In the case of 32-bit (standard) floating point numbers, the mantissa is usually 23 bits long, and the exponent is an 8-bit (two's complement) integer in the range $[-128, 127]$.

Representing the reals in this way has several consequences:

Rational Fractions

In base 2, just like base 10, many rational fractions have a repeating pattern after the decimal point.

Example

Store 1/7 (in base 10) as a decimal in binary.

$$\begin{aligned} \tfrac{1}{7} &= 2^{-3}(1 + \tfrac{1}{7}) \\ &= 2^{-3} + 2^{-6}(1 + \tfrac{1}{7}) \qquad \text{! substituting for 1/7 from above} \\ &= (0.\overline{001})_2 \qquad \text{! in binary} \end{aligned}$$

Example

Store 1/10 (in base 10) as a decimal in binary.

$$\begin{aligned} \tfrac{1}{10} &= 2^{-4}(\tfrac{16}{10}) \\ &= 2^{-4}(1 + \tfrac{6}{10}) \\ &= 2^{-4} + 2^{-5} + 2^{-4}.\tfrac{1}{10} \\ &= 2^{-4} + 2^{-5} + 2^{-8} + 2^{-9} + 2^{-8}.\tfrac{1}{10} \\ &= (0.0\overline{0011})_2 \end{aligned}$$

In binary, then, both 1/7 and 1/10 are repeating fractions! Since the reals are stored with a finite mantissa (typically 23 bits) even 1/10 will only be approximately stored.

To improve precision we can use reals with more bits, e.g. Salford allows 64-bit reals, which can be called with the declaration

```
integer, parameter::long=selected_real_kind(p=12)
real(kind=long)::x                ! makes x a 64 bit real
```

Ordinary reals have 23 digits past the decimal point in binary $\approx$ 7 digits past the decimal point in base 10. p=12 in the expression above asks for AT LEAST 12 digits in base 10 (in fact you get 16 for 64 bit reals).

Examples of Round-off Error

Consider the sum 1+x for some small number x.
For 32-bit reals, x can be as small as 10^{-38} ($1.0000000 \times 2^{11111111} = 2^{-128} = 10^{-38}$).
BUT 1+x cannot have an exponent of -128. Because $2^0 = 1$, it must have exponent = 0. Therefore

$$1 + x = 1.f_1 f_2 f_3 ....f_{22} f_{23} \times 2^0$$

The smallest possible value of x is therefore $2^{-23} \approx 10^{-7}$ (giving $f_1 = f_2 = ... = f_{22} = 0$, $f_{23} = 1$). Then

$$1 + x = 1.00000000000000000000001 \times 2^0$$

If $|x| < 2^{-23}$ then 1+x=1.

Summing Convergent Series

Suppose we want to calculate the summation

$$\sum_{n=1}^{5000} \frac{1}{n^2}$$

The final 1500 terms in this series are all small $< 10^{-7}$ so if we add the summation FORWARDS (starting) from the first term, they will not contribute to the final answer as there is a 1+x =1 round-off error for every term. However, taken together, the final 1500 terms add to give $\approx 1.36 \times 10^{-4}$, a significant error!

Solution One (not entirely foolproof) way to get around this is to add the terms in the summation BACKWARDS, i.e. from the smallest term first. This will minimise the accumulated round-off error.

Quadratic Equations

Consider a quadratic equation

$$ax^2 + bx + c = 0.$$

and assume that $b > 0$ for what follows (although the argument is easily modified). The roots of this equation are

$$x_1 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}, \quad \text{and} \quad x_2 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

Supposing we have $b^2 \gg 4ac$. Then $\sqrt{b^2 - 4ac} \approx b(1 - 2ac/b^2)$ and

$$x_1 \approx -\frac{b}{a}, \quad \text{and} \quad x_2 = -\frac{c}{b}$$

Note that $|x_1| \gg |x_2|$ since $x_1/x_2 = b^2/ac \gg 1$.

A problem with round-off error may arise if $4ac < 10^{-7}b^2$. Then the computer will calculate $b^2 - 4ac = b^2$ and will then calculate $x_2 = 0$!!!!

Solution A robust quadratic solver proceeds as follows. First calculate

$$x_1 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

as normal. Then note that

$$x_1 x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a} \times \frac{-b + \sqrt{b^2 - 4ac}}{2a} = \frac{4ac}{4a^2} = \frac{c}{a}$$

Now recover the 'problem' root x_2 from

$$x_2 = \frac{c}{ax_1}.$$

Clearly if $x_1 \approx -b/a$ then $x_2 \approx -c/b$ as it should! We have avoided round-off error.

Random Numbers

A computer is an entirely *deterministic* device, i.e. it does not have access to any genuinely random process. 'Random' numbers must therefore be generated from a deterministic sequence - ideally one which 'appears' to be random to the casual observer (although of course is not really). 'Random' numbers generated in this fashion are adequate for most pratical purposes.

The Linear Congruence Algorithm

One popular and relatively simple algorithm is as follows:

1. Choose 3 numbers a, c and m where $a < c < m$.
2. Choose a number $I_0 < m$
3. Generate the sequence I_0, I_1, I_2,.... by

   $$I_{j+1} = (aI_j + c) \bmod m$$

   Example: $m = 7$, $a = 2$, $c = 3$

   $$\text{then} \quad I_0 = 5,\ I_1 = (2 \times 5 + 3) \bmod 7 = 6,\ I_2 = 1,\ I_3 = 5,\ I_4 = 6,\ \text{etc.}$$

4. This generates a sequence of integers between 0 and $m - 1$. Dividing by m gives a sequence of reals between 0 and $1 - 1/m$.
5. Some choices of a, c, m are better than others

   GOOD CHOICE : $m = 233280, a = 9301, c = 49297$ (Numerical Recipes)

   BAD CHOICE : $m = 8, a = 2, c = 4$ $I_0 = 2,\ I_1 = 0,\ I_2 = 4,\ I_3 = 4,\ I_4 = 4,$ etc.

The Instrinsic Subroutine

In FORTRAN random numbers can be called using the intrinsic subroutine

```
Call random_number(x)
```

This uses an algorithm chosen by the compiler company. Like the linear congruence algorithm, it will be *deterministic*, i.e. every time it runs it will give the same random numbers.

To change from run to run, can use

```
Call random_seed()
```

or alternatively add

```
print *,'enter number < 1000
read *,nran
do i=1,nran
   call random_number(x)
end do                    !returns last value of x
```

Generating Other Random Variables using the Intrinsic Subroutine

We can use the random variable X from the intrinsic subroutine to generate random variables with the probability distribution of our choice.

Recall: X is uniformly distributed on $[0, 1]$.

Continuous Random Variables

Suppose we want to generate a random variable Y with probability density $P(Y)$. Recall that for a realisation Y_i of Y

$$P(Y) = \lim_{\delta Y \to 0} \frac{\text{Prob}(Y \leq Y_i < Y + \delta Y)}{\delta Y}, \qquad \int_{Y_{\text{min}}}^{Y_{\text{max}}} P(Y)\, dY = 1.$$

To obtain Y in terms of X,

$$\text{Set } P(Y)\, dY = P(X)\, dX = dX$$

Then rearrange and solve

$$\frac{dX}{dY} = P(Y) \quad \text{with} \quad x(Y_{min}) = 0,$$

and invert to get $Y(X)$. Then the random variable $Y = Y(X)$ will have the correct distribution.

Example

Find Y(X) so that

$$P(Y)\, dY = \exp\{-Y\}\, dY, \qquad 0 < Y < \infty$$

Following the above procedure

$$\frac{dX}{dY} = \exp\{-Y\}, \qquad X(0) = 0.$$

Solving get

$$X = 1 - \exp\{-Y\},$$

and rearranging

$$Y(X) = \log \frac{1}{1 - X}$$

gives the correct distribution.

Discrete Random Variables

We can also generate discrete random variables using the intrinsic subroutine

This can be done using integer variables. Suppose we want a discrete random variable Z to take integer values i1 to i2 inclusive with equal probability. Then the following FORTRAN lines can be used

```
Integer::Z,i1,i2
```

```
call random_number(x)
Z=(i2-i1+1)*x+i1
```

Example: Coin Toss

Want a discrete random variable a that takes the values 0 or 1 with equal probability. Then i1=0, i2-i1+1=2, so

```
Integer::a
call random_number(x)
a=2*x
```