# Computer System Support for Scientific and Engineering Computation

Lecture 3 - May 10, 1988 (revision date May 23, 1988)

## 1 Representation of Integers

This first part of this lecture discusses the representation of integers. Specifically, the topics covered include:

1. Which Radix is Best?
2. Representation of Unsigned Integers
3. Representation of Signed Integers

Many of the topics presented are basic; however, it is important to understand that many of the issues raised in exercise 1.4 are fully understood by very few people.

The second part of this lecture discusses social and legal issues.

### 1.1 Which Radix is Best?

The interpretation of a string of digits is based on the radix, or base, of the number which that string represents. For example, "237.6" can be interpreted as follows.

| Radix | Power Sum | Decimal Value |
|---|---|---|
| 8 | $2x8^2 + 3x8 + 7x1 + 6/8$ | 159.75 |
| 10 | $2x10^2 + 3x10 + 7x1 + 6/10$ | 237.6 |
| 16 | $2x16^2 + 3x16 + 7x1 + 6/16$ | 567.375 |
| $\beta > 7$ | $2x\beta^2 + 3x\beta + 7x1 + 6/\beta$ | |

Given a radix, $\beta$, the digits $d_0, d_1, d_2, \ldots$ and the digit-set $\{0, 1, 2, \ldots, \rho\}$, where $\rho \equiv \beta - 1$, each digit $d_j$ is drawn from the digit-set so that $0 \le d_j \le \rho$.

The table below describes some familiar radices.

| Name | $\beta$ | Digit-Set |
|---|---|---|
| BINARY | 2 | 0, 1 = $\rho$ |
| TERNARY | 3 | 0,1,2 = $\rho$ |
| QUATERNARY | 4 | 0,1,2,3 = $\rho$ |
| OCTAL | 8 | 0,1,2,3,4,5,6,7 = $\rho$ |
| DECIMAL | 10 | 0,1,2,3,4,5,6,7,8,9 = $\rho$ |
| HEXADECIMAL | 16 | 0,1,2,3,4,5,6,7,8,9,0,A,B,C,D,E,$F$ = $\rho$ |

Which radix is best? From a hardware standpoint, consider a circuit with a given noise immunity, from a components point of view:

1054 2-state devices afford a total of $2^{1054} = 1.93x10^{317}$ states
665 3-state devices afford a total of $\cancel{2}3^{665} = 1.93x10^{317}$ states
527 4-state devices afford a total of $\cancel{2}^{527}_{4} = 1.93x10^{317}$ states

This suggests that a computer built from 4-state devices would require half as many components as binary, and perhaps be cheaper.

For a given speed and level of noise-immunity, one can construct a plausible model which would make it seem as though 2-state devices are less expensive, since extra energy is required to chose one state over another. In other words, for a given noise immunity in-circuit:

$$\begin{array}{rcl} \text{cost of n-state device} & = & (n-1) \text{ units} \\ \text{No. of n-state devices} & = & k/\ln(n) \text{ units} \\ \hline \text{Total cost of system} & = & k(n-1)/\ln(n) \text{ units} \end{array}$$

Thus, the best $n \geq 2$ is $n = 2$. This argument does not consider such things as redundancy for error correction, and is only suggested as plausible.

Binary, it seems, will always be with us from the mere fact that all of today's computers have bits. As for radices $\beta > 2$, we use decimal for historical reasons. We might see hexadecimal, $\beta = 2^4$ and other powers of 2 for technical reasons (e.g. bus widths, register widths, byte-addressing.) Therefore, for the purposes of this lecture, assume that $\beta$ is even, and is either 10 or a power of 2.

## 1.2 Representation of radix $\beta$ Unsigned Integers

Given a string of $p$ digits $d_j$ drawn from the digit-set $\{0, 1, 2, \ldots, \rho\}$, we can say that

$$\begin{array}{rcl} I & = & [d_{p-1}d_{p-2}\ldots d_1 d_0] \\ & := & \text{a string of digits} \\ u(I) & = & d_{p-1}\beta^{p-1} + d_{p-2}\beta^{p-2} + \ldots + d_1\beta + d_0 \\ & := & \text{value of I as an unsigned integer} \\ \text{u([00...00]} & = & 0 \leq u(I) \leq \beta^p - 1 = u([\rho\rho\ldots\rho]) \end{array}$$

You can think of $u(I)$ as $u([\ldots 000I])$, where $I$ is embedded in an infinitely long string. When you add two or more of these, a carry-out, $c_p$, may signal overflow. Overflow, then, means that the result $I$ is misleading for lack of $c_p = 1$ on the left.

As far as unsigned integers go,

$$\{p \text{ digits of radix } \beta = 2^k\} \equiv \{pk \text{ bits of radix } \beta = 2\}.$$

The reason for persisting in the notion of radix $\beta = 2^k$ is to allow us to handle multi-word integer arithmetic, exploiting an add-with-carry instruction (where each digit is a word in memory). It is important to look at the instruction set to ensure that the carry code is not changed by another operation used only for indexing.

Unfortunately, high-level languages do not usually provide access to the carry condition code without sacrificing performance. Those that lack a carry- or borrow-out bit, or the instructions add-with-carry/substract-with-borrow impede the programming of multi- word integers. In these cases, it is usually best to code an optimized assembly language routine.

### 1.2.1 Radix-dependent effects on Unsigned Integers

What is the difference between decimal arithmetic and binary arithmetic, as it pertains to unsigned integers? You cannot tell the difference between decimal arithmetic and binary arithmetic until overflow occurs. For a given wordsize, however, we can determine how much sooner overflow will occur in decimal than in binary.

Like a hexadecimal digit, a Binary Coded Decimal digit consumes 4 bits of storage; although 6 states of the 16 available are unused, 3 bits is not enough: $2^3 < 10$.

- p decimal digits consume at least 4p bits
- Decimal Overflow will occur at $10^p \doteq 2^{3.322p}$
- Binary Overflow will occur at $2^{4p}$
- $\frac{4-3.322}{4} \doteq \frac{1}{6}$
- Therefore, decimal appears to squander about 1 bit out of 6.

This is not necessarily so, however. If you represent 3 BCD digits in 10 bits, then we can reconstruct the above $p = 3n$ BCD digits from $10n$ bits in memory. This means that in decimal, overflow will occur at $10^{3n}$, or $2^{9.9658n}$. In binary, overflow will occur at $2^{10n}$.

Thus compressed decimal squanders about 1 bit out of 292 ($\frac{10-9.9658}{10}$), compared to the one bit in six wasted by BCD. Such a compressed decimal encoding was used to conserve memory in the Workslate, by Convergent Technologies.

## 1.3 Representation of Signed Integers

The representation of signed integers is the same as for unsigned integers, the difference is in the interpretation. As expected, the representation of I = $[d_{p-1}d_{p-2}\ldots d_1 d_0]$ of $p$ digits. $d_j$. The number of possible strings of I is $\beta_p$, an even number.

We would hope that:

1. The integer 0 is representable by such a string, and $+0 = -0$ should be the same string.
2. If $i$ is is an integer representable by such a string, so is $-i$.

But we are faced with a dilemma: we cannot have both of the properties above, because then the number of representable integers would have to be odd. Thus, either ±0 is ambiguous, or some integer $i$ cannot be negated. We could, however, declare either -0, or the number that overflows when negated, illegal; a scheme implemented by Apple does this.

The following discuss several different ways in which to represent a signed integer in radix $\beta$.

### 1.3.1 Signed Magnitude

The signed magnitude representation of signed integers uses one sign bit or digit and $p$ digits $d_j$, and thus negation may be accomplished by flipping one bit. Specifically,

$$\begin{aligned} I &= [\pm d_{p-1}d_{p-2}\ldots d_1 d_0] \\ &:= \text{a string of digits} \\ sm(I) &= \pm u([0I]) \\ &:= \text{sign-magnitude value of I} \\ sm([-\rho\rho\ldots\rho] &= -(\beta^p - 1) \le sm(I) \le \beta^p - 1 = sm([\rho\rho\ldots\rho]) \end{aligned}$$

Given the above, the reader should note that the wordsize required to represent such a signed integer is wider than $p$ digits, since we must account for the sign bit. Also note that $[+00\ldots00]$ and $[-00\ldots00]$ must be treated by both hardware and software as having the same value—$sm([\pm00\ldots00]) = 0$.

Overflow occurs in signed magnitude representation when a carry-out or a borrow-out interferes with the sign-digit (i.e. when $c_p \neq 0$.)

### 1.3.2 $\beta$-Complement

The $\beta$-complement signed value of $I$ is the function $u(I)$; the most common example is two's complement. We all learned this as "flip all the bits and add one". A generalized definition is:

$$v = \begin{cases} u(I) & \text{if } 0 \le d_{p-1} < \beta/2 \\ u(I) - \beta^p & \text{if } d_{p-1} \ge \beta/2 \end{cases}$$

For example, given $\beta = 10$ and $p = 3$

$$\begin{array}{llll} v[500] = -500 & v[501] = -499 & v[502] = -498 & \ldots \\ v[998] = -2 & v[999] = -1 & v[000] = 0 & \ldots \\ v[001] = 1 & v[002] = 2 & \ldots & v[449] = 499 \end{array}$$

Thus, we can conclude that for $\rho = \beta - 1$

$$v([\frac{\beta}{2}00\ldots00]) = -\frac{1}{2}\beta^p \le v(I) \le \frac{1}{2}\beta^p = v([(\frac{\beta}{2} - 1)\rho\rho\ldots\rho\rho]).$$

Note that the negation of the most negative number, $[\frac{\beta}{2}00\ldots00]$, overflows to itself (e.g. if $\beta = 2$, -[100...00] overflows to [100...00].) This can be understood by embedding $\beta$-complement strings in infinite unsigned strings according to the following "P-adic notation" :

If $0 \le d_{p-1} < \beta/2$, embed I in $[\ldots000\ldots000I]$. If $\beta/2 \le d_{p-1}$, embed I in $[\ldots\rho\rho\rho\ldots\rho\rho\rho I]$. This means that $\ldots\rho\rho\rho\ldots\rho\rho\rho00\ldots00 \equiv -100\ldots00$.

Overflow of $\beta$-complement ADD occurs when the infinite part of the result string does not contain all the same digits (all $\rho$ or all $\beta$) because the rightmost has been altered by a carry-out that did not propagate all the way out.

### 1.3.3 $\rho$-Complement

This section describes the characteristics of a number represented by $\rho$-complement (e.g. ones' complement.) This is much simpler than the previous representation. Specifically, the $\rho$-complement representation of signed integers can be expressed as follows.

$$v_\rho = \begin{cases} u(I) & \text{if } 0 \leq d_{p-1} < \beta/2 \\ u(I) - (\beta^p - 1) & \text{if } d_{p-1} \geq \beta/2 \end{cases}$$

This can be illustrated using the same example given in the previous section.

$$\begin{array}{llll} v_\rho[500] = -499 & v_\rho[501] = -498 & v_\rho[502] = -497 & \dots \\ v_\rho[998] = -1 & v_\rho[999] = -0 & v_\rho[000] = +0 & \dots \\ v_\rho[001] = 1 & v_\rho[002] = 2 & \dots & v_\rho[449] = 499 \end{array}$$

Note the existence of two zeros. Also note that when $\beta = 2$, binary ones' complement is indistinguishable functionally from binary sign-magnitude, except in the case of multi-word arithmetic, when we must deal with an "end-around carry".

Embedding $\rho$-complement strings in infinite unsigned strings is accomplished by embedding $I$ in [...000...000$I$.000...] if $0 \leq d_{p-1} < \beta/2$. If $d_{p-1} \geq \beta/2$, then embed $I$ in [...$\rho\rho\rho$...$\rho\rho\rho I.\rho\rho\rho$...]. Thus:

$$\begin{array}{rcc} 0.\rho\rho\rho\ldots & \equiv & 1.000\ldots \\ \ldots\rho\rho\rho.000\ldots & \equiv & -1.000\ldots \\ \ldots\rho\rho\rho.\rho\rho\rho\ldots & \equiv & 0 \end{array}$$

Overflow of $\rho$-complement Add occurs when the infinite part of the result string does not have all the same digits (all $\rho$'s or all $\beta$'s) after end-around-carry. For those concerned that end-around-carry might cycle infinitely, take as an example $\beta = 2$. In order to have a carry at all, a 0 must have been generated somewhere in I. Thus, the carry would stop there.

**Theorem 1** *End-around carry cannot recirculate forever; one pass is enough.*

The other problem with using $\rho$-complement is the "minus zero nuisance"— $I + (-I) = -0$. This problem was addressed by Seymour Cray's subtractive adder, which functioned such that: $x + y = x - (-y)$. The thought behind this was that $-0$ can not be produced unless both operands are $-0$. Unfortunately, the negate operation did simple bit flipping; it was thought unreasonable to look over all the bits. Thus, negation of zero -(0) produces (-0). (The CDC 17X , 6600, 7600, and Cyber 1xx have this property.) Additionally, Cray discovered how to implement fast carry-look-ahead in ones' complement so that it was no slower than two's complement despite end-around carry (see Thornton's book on the CDC 6600).

### 1.3.4 Which Representation is Preferable?

The previous sections discussed three ways of representing signed integers:

1. Sign Magnitude

2. $\beta$-Complement

3. $\rho$-Complement.

Most current machines use two's complement. Generally, $\beta$-complement is better for multi-word arithmetic. (provided that both add-with-carry and subtract-with-borrow are present.) The reason is that it is easier to understand, it needs no end-around carry, and it can't confuse the sign-bit with carry. If add-with-carry etc, are not present in higher level languages, you can't tell the difference in complexity, although different programs may be needed.

### 1.4 Exercise

Write a program in a higher-level language to tell whether your computer uses

- Binary vs. Decimal.
- Sign-Magnitude vs. a Complement.

(Binary sign-magnitude is indistinguishable from binary ones' complement.)

## 2 Legal and social issues

One of life's big disappointments is that people who design floating point, too often leave parts of the implementation "shaky". Nonetheless, the product is alleged to have the properties which were intended, and sold as if it did. After all, there are two entirely different views on the importance of reliability in numerical software:

1. Some prefer programs that always work.
2. More prefer programs that run twice as fast, even though they may sometimes be wrong. Most people can tell whether a numerical program is fast or slow, but not whether it is right or wrong.

Those familiar with Gresham's Law:

*The bad money drives the good out of circulation.*

might see a parallel that might be called Gresham's Law of Computing:

*The fast drives out the slow even if the fast is wrong.*

Responsible engineers must be certain that their calculations are correct. This group, (e.g. Bechtel, Lockheed), must be able to refute allegations of negligence in case the product produced by the calculations is later found to be incorrect and have harmful consequences. They must be able to reproduce, exactly, the calculations in question.

The fact is that not all manufacturers produce reliable math functions. But who is responsible for what? Serious questions about liability and legal responsibility are being raised. The numerous issues which fragment the market lead us to the conclusion that we can't really assess who is responsible.

Previously, manufacturers and users of computers were regarded as roughly peers technically. Consequently they were presumed competent to assess and negotiate risk, so the usual principles of contract law prevailed.

In the future, computer users may be regarded as the consumers that most actually are, most lacking technical expertise to assess risk as peers with manufacturers. They may then come to be covered by the legal protections afforded to consumers of television sets and automobiles, namely a right to expect reasonable behavior in normal use. But who defines "reasonable" and "normal"? Governments and bodies like Underwriters' Laboratories prescribe standards for many consumer products. Would it be better for computer arithmetic to be defined voluntarily and cooperatively by its implementors and users? If not, then regulation may come from above and may not be well thought out.

Legal precedents aren't clear. Pharmaceutical companies were formerly held absolutely liable for their products. Thus Robbins was bankrupted by claims relating to Dalkon Shield even though all normal testing was performed prior to marketing that product.

However in early 1988 the California Supreme Court relieved pharmaceutical firms of such absolute liability on the grounds that otherwise no pharmaceutical company could afford to develop a new product. Manufacturers of general aviation aircraft have petitioned Congress for similar relief. Will the disclaimers attached to computer hardware and software stand up in court? Will it take an act of Congress to issue a new CPU or computer?

Proprietary rights is another issue which needs to be discussed. Some people believe that all computer software ought to be in the public domain. Software, especially floating-point math libraries, placed in the public domain benefit the computing community as a whole. An example of public domain software which has benefited many is TeX, the typesetter created by Knuth. Knuth believes that if TeXwere proprietary, it would not be as widely used and would not have benefited as many people. By making software available at no cost to the public, more people and companies are able and willing to use it. Thus, these companies need not reinvent the wheel by creating similar software, possibly not as well tested and debugged, on their own.

Companies which use public domain software benefit more than just the savings in resources needed to develop it. These companies enjoy a sense of legal protection or security by using such software, instead of developing one of their own. For instance, if a customer of company A, which uses Berkeley's 4.3 BSD math library, encounters a problem arising from a flaw in a math function, company A can divert the blame to the provider of the math library. Thus company A can readily divest itself from tricky legal problems in areas in which it is very difficult to assign responsibility.

Congress has exempted certain things from proprietary claims. In a sense, what is provably optimal can't be copyrighted or patented. Thus nobody can claim proprietary rights on theorems, fundamental constants, or other facts. One can only claim proprietary rights on things which have elements of arbitrariness in them. Even though one cannot have proprietary rights to mathematical formula and constants, proprietary rights are granted to assembly language code and similar items.

One of the reasons for making software public domain is that if we make it good enough, nobody can claim proprietary rights on it. Therefore, there are incentives for us to continue improving our software to the point of perfection. This way we can develop a computing environment which will be ever more reliable.

As discussed in lecture 1, reliability in the world of computing does not mean a low incidence of error or failure. The occurrences of errors, no matter how infrequently, can cause enormous damage. All known causes of failure should be uncovered and fixed. By reliability, we mean the conformity to specifications and documentations, that is, the software or hardware should do *exactly* as documented, with all limitations clearly stated. This is really not an easy task to accomplish.

An interesting paradox is that not only is the counsel of perfection unreasonable, there exist all kinds of strange anomalies. These anomalies are "strange" because we can actually prove that they exist. As an example, even if all roundings in a computation are performed accurately, there remain strange anomalies which cannot be totally eliminated. So, our objective is to describe things in such a manner that these anomalies, as they must exist, and maybe a few other problems which we do not know how to fix, are acceptable to most people to be incorporated in their mental paradigms. Software and hardware should behave as expected taking into consideration the limitations resulting from the anomalies.

What are our obligations to correct errors? Human beings are expected to make mistakes, but we must not leave our errors uncorrected. However, corporations often feel that to admit a mistake is a major and serious blow to their reputation. As such, very often we find ourselves in a situation where we cannot convince our peers to correct known mistakes. For instance, Apple's implementation of the copysign function is quite different from the one recommended in the IEEE 754 standard: the function's arguments are in the reverse order. The problem here is not which is better, but the inconsistency. Apple preserved consistency within its library at a cost of inconsistency among differing vendors' IEEE implementations. It would have been wiser to name the Apple function "signcopy" to emphasize the difference.

In the next lecture we will discuss ways to do arithmetic exactly and the various ways to represent floating-point numbers. We will also talk about benchmarking and exception handling.

# RADIX or BASE

What does " 237.6 " mean?

| RADIX | POWER SUM | DECIMAL VALUE |
|---|---|---|
| 8 | $2\times8^2 + 3\times8 + 7\times1 + 6/8$ | 159.75 |
| 10 | $2\times10^2 + 3\times10 + 7\times1 + 6/10$ | **237.6** |
| 16 | $2\times16^2 + 3\times16 + 7\times1 + 6/16$ | 567.375 |
| $\beta$ | $2\times\beta^2 + 3\times\beta + 7\times1 + 6/\beta$<br>(Only if $\beta > 7$.) | ...... |

cf. "Mixed Radix" £ / s / d

$d < 12$ (12 pence / shilling)

$s < 20$ (20 shillings / pound)

W. Kahan
10 May 89

# RADIX = $\beta$

digits $d_0, d_1, d_2, \ldots$

digit-set $\{0, 1, 2, \ldots, \rho\}$

where $\rho \equiv \beta - 1$

Each digit $d_j$ is drawn from the digit-set, so

$$0 \leq d_j \leq \rho$$

| NAME | RADIX $\beta_{10}$ | $\rho = \beta - 1$ in digit-set |
|---|---|---|
| **BINARY** | 2 | $0,\ 1 = \rho$ |
| TERNARY | 3 | $0, 1, 2 = \rho$ (or $\bar{1}, 0, 1$) |
| QUATERNARY | 4 | $0, 1, 2, 3 = \rho$ |
| OCTAL | 8 | $0, 1, 2, 3, 4, 5, 6, 7 = \rho$ |
| **DECIMAL** | 10 | $0, 1, 2, 3, 4, 5, 6, 7, 8, 9 = \rho$ |
| HEXADECIMAL | 16 | $0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F = \rho$<br>$(F = 15_{10})$ |

# WHO USES THESE RADICES ?

**BINARY** ... *all* computers have bits in them nowadays.

**TERNARY** ... U.S.S.R. & POLAND ?

**OCTAL** ... Burroughs B65xx floating-pt.
[ Print-out of binary on PDP-11, ... ]

**DECIMAL** ... Almost ALL CALCULATORS.
... IBM 650, 1620 ; .... ( bi-quinary )
{ BCD usually } ... " Decimal Add/Subtract Adjust "
... ASCII decimal in, e.g., ELXSI 6400
... ALWAYS encoded in Binary nowadays.

**HEXADECIMAL** ... IBM 370 (& Amdahl, ...) floating-point
[ Print-out of binary ]

**$\beta = 100$** ... old VISICALC, some other SOFTWARE.

**$\beta = 256$** ... the RICE UNIV. machine (1950's)

WHICH RADIX IS BEST ?

# WHICH RADIX IS BEST?

From COMPONENTS point of view ...

| | | |
|---|---|---|
| 1054 | 2- state devices | $2^{1054} = 1.93 \times 10^{317}$ |
| 665 | 3- state devices | $3^{665} = 1.93 \times 10^{317}$ |
| 527 | 4- state devices | $4^{527} = 1.93 \times 10^{317}$ |
| ⋮ | ⋮ ⋮ | ⋮ |

with SAME RELIABILITY (noise immunity)

⇒ 2-state devices seem less expensive

( MODEL: is it plausible?

For given noise-immunity in-circuit,

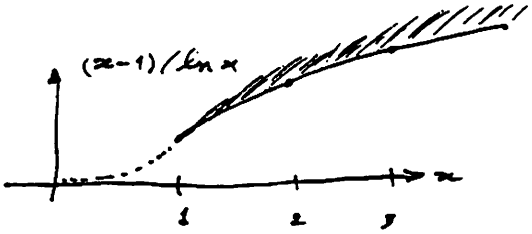
What about Redundancy used for Error-Correction?

Cost of n-state device = (n-1) units ... at least

No. of n-state devices = K/ln(n) units

Total cost of system = K.(n-1)/ln(n) ... at least

Best $n \geq 2$ is $n=2$



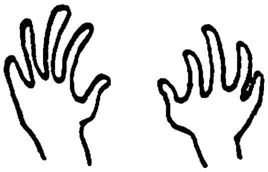
)

$\overset{\circ}{\circ\circ}\!\!\Rightarrow$ 2-state components are most economical for a given level of reliability.

BINARY WILL ALWAYS BE WITH US!

# REASONS FOR RADICES $\beta > 2$

1. DECIMAL ($\beta = 10$) for HISTORICAL REASONS.



2. HEXADECIMAL ($\beta = 2^4$) and other powers of 2

because of
- BUS WIDTHS
- REGISTER WIDTHS
- BYTE-ADDRESSING (CHARACTERS)
- IBM 360 & 370 [A MISTAKE!]

$\overset{\circ}{{}_{\circ\circ}}\Longrightarrow$ Assume $\beta$ is EVEN

either 10 or a power of 2

RULE OUT $\beta = 3$ and other peculiar radices for now.

# Radix $\beta$ representation of

## UNSIGNED INTEGERS

with $p$ digits $d_j$

drawn from $\{0,1,2,\dots,\rho\}$. $(\rho=\beta-1)$

$$I := [\,d_{p-1}\ d_{p-2}\ \cdots\ d_1\ d_0 .\,]$$ a string of digits.

$u(I) :=$ value of $I$ as an UNSIGNED INTEGER

$$= d_{p-1}\beta^{p-1} + d_{p-2}\beta^{p-2} + \cdots + d_1\beta + d_0\cdot 1 .$$

$$u([00\dots00]) = 0 \le u(I) \le \beta^{p}-1 = u([\rho\rho\cdots\rho\rho]) .$$

$$u(I) = u([\dots\ 000\ \underbrace{d_{p-1}\ d_{p-2}\ \cdots\ d_1\ d_0}_{\text{$I$ embedded in an infinitely long string.}}\,])$$

$$\begin{array}{rl}
 & [\,a_{p-1}\ \ a_{p-2}\ \cdots\ a_1\ \ a_0\,] \\
+ & [\,b_{p-1}\ \ b_{p-2}\ \cdots\ b_1\ \ b_0\,] \\
\hline
c_p & [\,d_{p-1}\ \ d_{p-2}\ \cdots\ d_1\ \ d_0\,]
\end{array}$$

$\uparrow$

carry-out

$c_p = 0$ or $1$ ... OVERFLOW $\equiv (c_p = 1)$

means "result" $I$ is misleading for lack of $c_p = 1$ on left.

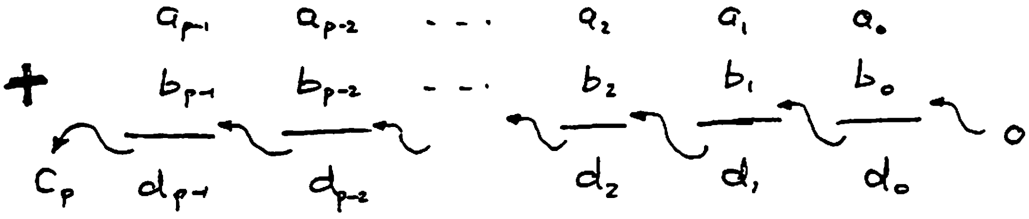
As far as <u>UNSIGNED INTEGERS</u> go,

$$\{ p \text{ digits of radix } \beta = 2^k \} \equiv \{ pk \text{ bits of radix } \beta = 2 \}.$$

Reason for persisting in notion of radix $\beta = 2^k$ is for

MULTI-WORD INTEGER ARITHMETIC;

cf. ADD-WITH-CARRY instruction:



$a_{p-1}$ $a_{p-2}$ . . . $a_2$ $a_1$ $a_0$
+ $b_{p-1}$ $b_{p-2}$ . . . $b_2$ $b_1$ $b_0$
0
$C_p$ $d_{p-1}$ $d_{p-2}$ $d_2$ $d_1$ $d_0$

... each digit is a WORD in memory.

(... similarly, SUBTRACT-WITH-BORROW instruction.)

HIGHER-LEVEL LANGUAGES that lack
- CARRY/BORROW-out bit (side-effect/status)
- ADD-WITH-CARRY / SUBTRACT-WITH-BORROW

impede programming of multi-word integers.

As far as UNSIGNED INTEGERS go,

$\{$ DECIMAL arithmetic $\} \equiv \{$ BINARY arithmetic $\}$

UNTIL OVERFLOW occurs.

For a given wordsize, how much sooner must OVERFLOW occur with DECIMAL than with BINARY ?

1 decimal digit consumes at least 4 bits :

e.g. BCD = Binary Coded Decimal

vs. HEXADECIMAL : $\overbrace{0, 1, 2, 3, 4, 5, 6, 7, 8, 9,}\ \underbrace{A, B, C, D, E, F}_{\text{discards}}$
($\beta = 2^4$)

$\therefore$ p decimal digits consume at least 4p bits ?

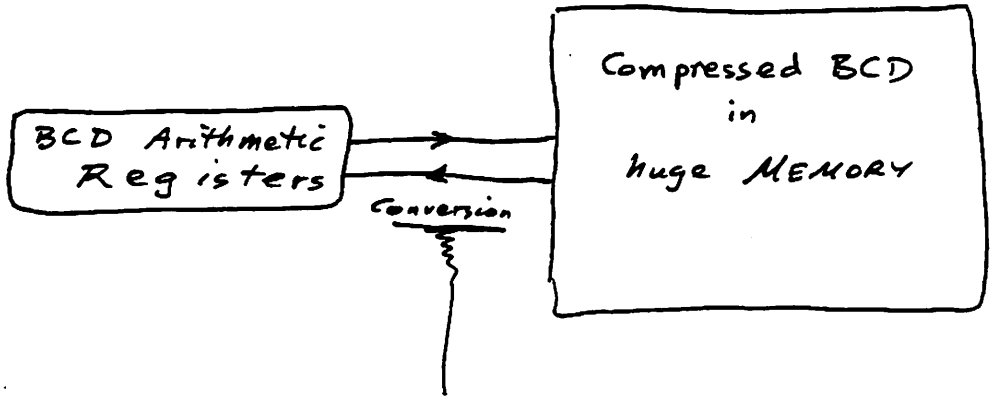
$\text{Overflow}_{\text{DECIMAL}}$ @ $10^p$ vs. $\text{Overflow}_{\text{BINARY}}$ @ $2^{4p}$

i.e. $10^p \doteq 2^{3.322p}$ vs $2^{4p}$

$\therefore$ DECIMAL appears to squander about 1 bit in 6 $\left\{ \dfrac{4-3.322}{4} \doteq \dfrac{1}{6} \right\}$

NOT NECESSARILY SO !

# COMPRESSION of 3 BCD digits → 10 bits.



CONVERSION of every 3 BCD digits ⟷ 10 bits.

| 3 BCD digits | 10 bits |
|---|---|
| 0, 1, 2, ··· , 998, 999 | 0, 1, 2, · , 998, 999, $\underbrace{1000, 1001, \ldots, 1024}_{\text{discards}}$ |

$p = 3n$ DECIMAL DIGITS consumes $10n$ bits in memory.

Overflow @ $10^{3n}$ vs Overflow @ $2^{10n}$
DECIMAL BINARY

i.e. $10^{3n} \doteqdot 2^{9.9658n}$ vs $2^{10n}$

∴ COMPRESSED DECIMAL appears to SQUANDER about

1 bit in 292 $\left\{ \frac{10 - 9.9658}{10} \doteqdot \frac{1}{292} \right\}$

Such a scheme has been used (for floating-point) to conserve memory in the WORKSLATE, by CONVERGENT TECHNOLOGIES INC.

∴ MEMORY ECONOMY IS NOT NECESSARILY A REASON TO PREFER BINARY OVER DECIMAL.

# RADIX $\beta$ representation of SIGNED INTEGERS

with $p$ digits $d_j$ .

$I = [\,d_{p-1}\ d_{p-2} \cdots d_1\ d_0\,]$ is a string of digits.

How many such strings $I$ ? ... $\beta^p$ ,

an EVEN NUMBER.

DESIDERATA

0) The integer $0$ is representable by such a string,

and $+0 = -0$ should be the same string.

1) If $i$ is an integer representable by such a string,

so is $-i$ .

DILEMMA : We cannot have both 0) and 1) ,

because then the number of representable integers would have to be ODD.

∴ ⟹ EITHER ±0 IS AMBIGUOUS,

OR SOME INTEGER $i$ CANNOT BE NEGATED.

# SIGN - MAGNITUDE

Radix $\beta$ representation of

## SIGNED INTEGERS

with $p$ digits $d_i$

and one sign-digit (bit).

$\underline{I} := [\pm\ d_{p-1}\ d_{p-2} \cdots d_1\ d_0]$ ... a string of digits.

$sm(I)$ := sign-magnitude value of $I$

$= \pm\, u([0I])$ resp. ($u(I)$ = unsigned value of $I$)

[Note: word-size is WIDER than $p$ digits.]

$$sm([-\rho\rho \cdots \rho\rho]) = -(\beta^p - 1) \le sm(I) \le \beta^p - 1 = sm([+\rho\rho \cdots \rho\rho])$$

Note: $[+00 \cdots 00]$ and $[-00 \cdots 00]$ must be treated by hardware & software as having the same value, namely $sm([\pm 00 \cdots 00]) = 0$.

OVERFLOW occurs when a carry-out or a borrow-out contends with the sign-digit, i.e. when $c_p \neq 0$.

$$\begin{array}{r} [\pm a_{p-1}\ a_{p-2} \cdots a_1\ a_0] \\ +\ [\pm b_{p-1}\ b_{p-2} \cdots b_1\ b_0] \\ \hline [\underset{\overset{\uparrow\uparrow}{?}}{\pm [c_p}\ d_{p-1}\ d_{p-2} \cdots d_1\ d_0] \end{array}$$

# $\beta$- complement

## Radix $\beta$ representation of

## SIGNED INTEGERS

with $p$ digits $d_j$.

$I$ := $[\, d_{p-1}\ d_{p-2}\ \dots\ d_1\ d_0 \,]$ ... a string of digits

$v_\beta$ := $\beta$-complement signed value of $I$

$$= \begin{cases} u(I) & \text{if } 0 \le d_{p-1} < \beta/2 \\ u(I) - \beta^p & \text{if } d_{p-1} \ge \beta/2 \end{cases}$$

($u(I)$ is unsigned value of $I$)

e.g. Take $p = 3$ digits decimal $(\beta = 10)$ :

$v_\beta[500] = -500$ $v_\beta[501] = -499$ $v_\beta[502] = -498$ ...

$v_\beta[498] = -2$ $v_\beta[999] = -1$ $v_\beta[000] = 0$ ...

$v_\beta[001] = 1$ ... $v_\beta[499] = 499$ .

$$v_\beta\left(\left[\tfrac{\beta}{2}\,00\dots00\right]\right) = -\tfrac{1}{2}\beta^p \le v_\beta(I) \le \tfrac{1}{2}\beta^p - 1 = v_\beta\left(\left[\tfrac{\beta}{2}\text{-}1\ \varrho\varrho\varrho\cdots\varrho\varrho\right]\right)$$

where $\varrho = \beta - 1$ .

**Note:** Negation of $\left[\tfrac{\beta}{2}\ 00\dots00\right]$ OVERFLOWS to itself.

e.g. $\beta = 2$ $-[100\dots00]$ overflows to $[100\dots00]$ .

# EMBEDDING $\beta$-complement strings in infinite unsigned strings:

If $0 \le d_{p-1} < \beta/2$ embed $[d_{p-1}\, d_{p-2} \cdots d_1\, d_0]$ in $[-000\ldots 00 d_{p-1} d_{p-2} \cdots d_1 d_0]$

If $\beta/2 \le d_{p-1}$ in $[\underbrace{-\rho\rho\rho\ldots\rho\rho}_{\substack{\text{INFINITE STRING}\\ \text{of replicated digits}}} d_{p-1}\, d_{p-2} \cdots d_1\, d_0]$ .

$$
\begin{array}{lcl}
\text{What does} & \cdots\rho\rho\rho\cdots\rho\rho\ 00\ldots00 & \text{mean ?} \\
\text{Add} & \underline{\cdots 000\ldots 01\ 00\ldots00} & ; \\
\text{get} & \ldots 000\ldots 00\ \underbrace{00\ldots00}_{p \text{ digits}} & = \text{zero !!}
\end{array}
$$

$$\therefore \quad \cdots\rho\rho\rho\cdots\rho\rho\ 00\ldots00 \ \equiv\ -100\ldots00$$

Cf. Euler's "formula" $\cdots + \beta^m + \beta^{m-1} + \beta^{m-2} + \cdots + \beta^2 + \beta + 1 = \dfrac{1}{1-\beta}$

which would be valid if $|\beta| < 1$

$$\therefore \quad \cdots + \rho\beta^{m+p} + \rho\beta^{m+p-1} + \rho\beta^{m+p-2} + \cdots + \rho\beta^{p+2} + \rho\beta^{p+1} + \rho\beta^{p} = \frac{\rho\beta^p}{1-\beta}$$

$$= -\beta^p$$

18th century reasoning?

$$
\begin{array}{rl}
 & \overbrace{\cdots\cdots}^{\text{Infinite repetition}}\ a_{p-1}\ a_{p-2}\ \cdots\ a_1\ a_0 \\
+ & \underline{\cdots\cdots\ b_{p-1}\ b_{p-2}\ \cdots\ b_1\ b_0} \\
 & \boxed{\ldots\ldots?}\ d_{p-1}\ d_{p-2}\ \cdots\ d_1\ d_0
\end{array}
$$

In BINARY, repeat SIGN BIT infinitely

OVERFLOW of $\beta$-complement ADD occurs when Infinite part of result-string has not all digits the same (all $\rho$'s or all 0's).

NOT the same as carry-out!

$\boxed{\rho = \beta - 1}$

$\rho$-complement

Radix $\beta$ representation of

SIGNED INTEGERS

with $p$ digits $d_j$ .

$I :=$ $[d_{p-1}\ d_{p-2} \cdots d_1\ d_0]$ ... a string of digits

$v_\rho$ := $\rho$-complement signed value of $I$

$$= \begin{cases} u(I) & \text{if } 0 \le d_{p-1} < \beta/2 \\ u(I) - (\beta^p - 1) & \text{if } d_{p-1} \ge \beta/2 \end{cases}$$

( $u(I)$ is unsigned value of $I$ )

e.g. Take $p = 3$ digits decimal ( $\beta = 10$ ):

$v_\rho[500] = -499$ $\quad$ $v_\rho[501] = -498$ $\quad$ $v_\rho[502] = -497$ ...

$v_\rho[997] = -2$ $\quad$ $v_\rho[998] = -1$

$$\underset{\underline{-0}}{v_\rho[999]} = \underset{\underline{+0}}{v_\rho[000]} = 0$$

$v_\rho[001] = 1$ $\quad$ $v_\rho[002] = 2$ ... $v_\rho[499] = 499$.

TWO ZEROS !

When $\beta = 2$ ... BINARY 1's complement is indistinguishable functionally from BINARY sign-magnitude except for MULTI-WORD arithmetic.

EMBEDDING ρ-complement strings in infinite unsigned strings;

$\left\{\begin{matrix}\text{binary}\\ \text{decimal}\end{matrix}\right\}$ point

If $0 \le d_{p-1} < \beta/2$ embed $[d_{p-1}\, d_{p-2} \cdots d_1\, d_0]$ in

$$[\cdots 000 \cdots 00\, d_{p-1}\, d_{p-2} \cdots d_1\, d_0 \bullet 000 \cdots]$$

If $d_{p-1} \ge \beta/2$ embed $[d_{p-1}\, d_{p-2} \cdots d_1\, d_0]$ in

$$[\underbrace{\cdots \rho\rho\rho \cdots \rho\rho}_{\text{Infinite string of replicated digits}}\, d_{p-1}\, d_{p-2} \cdots d_1\, d_0 \bullet \underbrace{\rho\rho\rho \cdots}_{\text{Infinite string of replicated digits}}]$$

What does $0 \bullet \rho\rho\rho \cdots\cdots$ mean? $1.000\cdots$ .

What does $\cdots \rho\rho\rho\rho \bullet 000 \cdots$ mean? $-1.000 \cdots$ .

What does $\cdots \rho\rho\rho\rho \bullet \rho\rho\rho \cdots$ mean? $0$ .

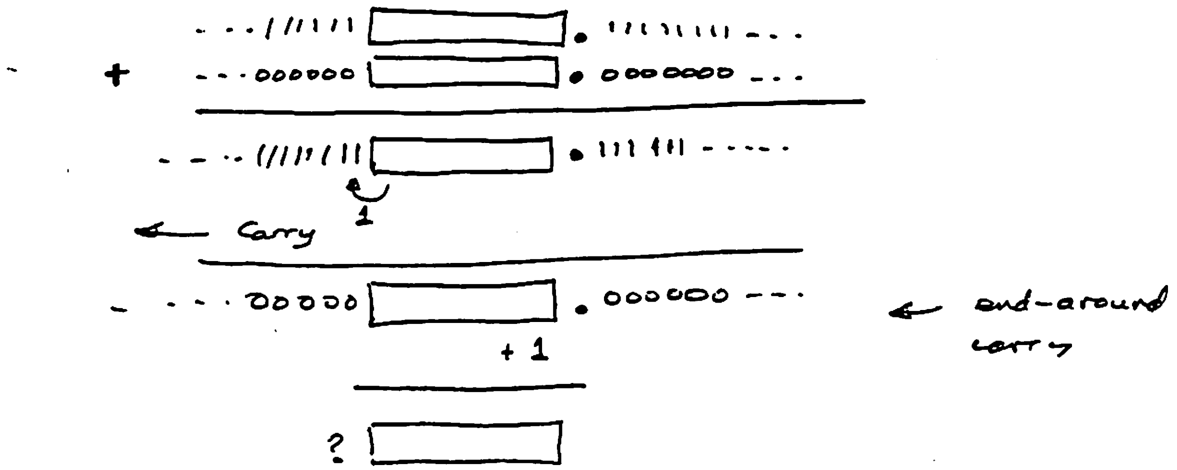
$$\begin{array}{rl} & \overbrace{\cdots\cdots}^{\text{Infinite Repetition}}\; a_{p-1}\; a_{p-2} \cdots a_1\; a_0 \bullet \overbrace{\cdots\cdots}^{\text{Infinite Repetition}} \\ + & \cdots\cdots\; b_{p-1}\; b_{p-2} \cdots b_1\; b_0 \bullet \cdots\cdots \\ \hline & \boxed{\cdots ?}\; d_{p-1}\; d_{p-2} \cdots d_1\; d_0 \bullet \boxed{\cdots\cdots} \end{array}$$

In BINARY, repeat SIGN BIT infinitely.

OVERFLOW of ρ-complement ADD occurs when Infinite part of result string has <u>not</u> all digits the same (all ρ's or all 0's) after END-AROUND CARRY.

# END-AROUND CARRY ?

Take $\beta = 2$ : $\rho = 1$ ( 1's complement )



Theorem : End-around carry cannot re-circulate forever; one pass is enough.

## MINUS ZERO NUISANCE :

$$I + (-I) = -0$$

in $\rho$-complement arithmetic.

$$\text{e.g.}\quad \begin{array}{rcl} (+1) & = & \boxed{0\ 0\ 1} \\ +\ (-1) & = + & \boxed{1\ 1\ 0} \\ \hline (-0\ ) & = & \boxed{1\ 1\ 1} \end{array}$$

in 1's complement

( UNIVAC 11xx )

# Seymour Cray's contributions to 1's complement arithmetic

1's complement subtractive adder

$$x + y \equiv x - (-y)$$

$\Rightarrow$ can't produce $-0$ unless both operands are $-0$ 's.

( But, alas $-(0) \rightarrow (-0)$ . )

CDC 6600
7600
Cyber 1x

( Also, Cray found how to inplement fast carry-look-ahead in 1's complement so that it was no slower than 2's complement despite end-around carry : see Thornton's book on CDC 6600. )

Which of SIGN-MAGNITUDE
β-COMPLEMENT
ρ-COMPLEMENT

is preferable ?

β-complement is better for

MULTI-WORD INTEGER ARITHMETIC

provided $\left\{ \begin{array}{l} \text{ADD-WITH-CARRY} \\ \text{SUBTRACT-WITH-BORROW} \end{array} \right\}$ are present

because it is EASIER TO UNDERSTAND,
NEEDS NO END-AROUND CARRY,
CAN'T CONFUSE SIGN-BIT with CARRY.

Without ADD-WITH-CARRY etc.,
i.e. in HIGHER-LEVEL LANGUAGES,
you can't tell the difference in complexity,
though different programs may be needed!

CAN YOU WRITE A PROGRAM
in, say, PASCAL or FORTRAN or C
to tell whether your computer uses

- BINARY vs DECIMAL ?
- SIGN-MAGNITUDE vs a COMPLEMENT ?

<u>Answer</u>: Yes,
except binary sign-magnitude is
INDISTINGUISHABLE
from binary 1's complement.