Fixed Point Arithmetic and the Packed Decimal Format

This set of lectures discusses the idea of **fixed point arithmetic** and its implementation by Packed Decimal Format on the IBM Mainframes.

The topics include

1. A review of IBM S/370 floating point formats, focusing on the precision with which real numbers can be stored.
2. The difference between the precision requirements of business applications and those of scientific applications.
3. An overview of the Packed Decimal format, as implemented on the IBM S/370 and predecessors.

We begin with a review of IBM notation for denoting bit numbers in a byte or word. From our viewpoint, it is a bit non–standard.

IBM S/370: Terminology and Notation

The IBM 370 is a byte–addressable machine; each byte has a unique address.

The standard storage sizes on the IBM 370 are byte, halfword, and fullword.

Byte	8 binary bits	
Halfword	16 binary bits	2 bytes
Fullword	32 binary bits	4 bytes.

In IBM terminology, the leftmost bit is bit zero, so we have the following.

Byte

0	1	2	3	4	5	6	7

Halfword

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Fullword

0 – 7	8 – 15	16 – 23	24 – 31

Comment: The IBM 370 seems to be a “big endian” machine.

S/370: Available Floating Point Formats

There are three available formats for representing floating point numbers.

Single precision	4 bytes	32 bits: 0 – 31
Double precision	8 bytes	64 bits: 0 – 63
Extended precision	16 bytes	128 bits; 0 – 127.

The standard representation of the fields is as follows.

Format	Sign bit	Exponent bits	Fraction bits
Single	0	1 – 7	8 – 31
Double	0	1 – 7	8 – 63
Extended	0	1 – 7	8 – 127

NOTE: Unlike the IEEE–754 format, greater precision is not accompanied by a greater range of exponents.

The precision of the format depends on the number of bits used for the fraction.

Single precision	24 bit fraction	1 part in 2^{24}	7 digits precision *
Double precision	56 bit fraction	1 part in 2^{56}	16 digits precision **

*2^{24} = 16,777,216 ** $2^{56} \approx (10^{0.30103})56 \approx 10^{16.86} \approx 7 \bullet 10^{16}$.

Precision Example: Slightly Exaggerated

Consider a banking problem. Banks lend each other money overnight.

At 3% annual interest, the overnight interest on $1,000,000 is $40.492.

Suppose my bank lends your bank $10,000,000 (ten million).

You owe me $404.92 in interest; $10,000,404.92 in total.

With seven significant digits, the amount might be calculated as $10,000,400.
My bank loses $4.92.

I want my books to balance to the penny. I do not like floating point arithmetic.

TRUE STORY

When DEC (the Digital Equipment Corporation) was marketing their PDP–11
to a large New York bank, it supported integer and floating point arithmetic.

At this time, the PDP–11 did not support decimal arithmetic.

The bank told DEC something like this:
"Add decimal arithmetic and we shall buy a few thousand. Without it – no sale."

What do you think that DEC did?

Precision Example: Weather Modeling

Suppose a weather model that relies on three data to describe each point.

1. The temperature in Kelvins. Possible values are 200 K to 350 K.
2. The pressure in millibars. Typical values are around 1000.
3. The percent humidity. Possible ranges are 0.00 through 1.00 (100%).

Consider the errors associated with single precision floating point arithmetic.
The values are precise to 1 part in 10^7.

The maximum temperature errors resulting at any step in the calculation would be:

$3.5 \bullet 10^{-5}$ Kelvins

$1.0 \bullet 10^{-4}$ millibars

$1.0 \bullet 10^{-5}$ percent in humidity.

As cumulative errors tend to cancel each other out, it is hard to imagine
the cumulated error in the computation of any of these values
as becoming significant to the result.

Example: A weather prediction accurate to ±1 Kelvin is considered excellent.

Encoding Decimal Digits

There are ten decimal digits. As $2^3 < 10 \leq 2^4$, we require four binary bits in order to represent each of the digits.

Here are the standard encodings. Note the resemblance to hexadecimal, except that the non–decimal hexadecimal digits are not shown.

0	0000	**5**	0101
1	0001	**6**	0110
2	0010	**7**	0111
3	0011	**8**	1000
4	0100	**9**	1001

Note that the binary codes 1010, 1011, 1100, 1101, 1110, 1111
for hexadecimal digits A B C D E F
are not used to represent decimal digits.

Packed decimal representation will have other uses for these binary codes.

Zoned Decimal Data

Remember that all textual data in computing are input and output as character codes. IBM S/370 uses 8–bit EBCDIC to represent characters.

On input, these character codes are immediately converted to Zoned Decimal format, which uses 8–bit (one byte) codes to represent each digit. With a slight exception, it is identical to EBCDIC.

Here are the EBCDIC representations of each digit, shown as 2 hex digits.

Digit	Code		**Digit**	Code	
	Hex	Binary		Hex	Binary
0	F0	1111 0000	**5**	F5	1111 0101
1	F1	1111 0001	**6**	F6	1111 0110
2	F2	1111 0010	**7**	F7	1111 0111
3	F3	1111 0011	**8**	F8	1111 1000
4	F4	1111 0100	**9**	F9	1111 1001

Packed Decimal Data

Packed decimal representation makes use of the fact that only four binary bits are required to represent any decimal digit.

Numbers can be **packed**, two digits to a byte.

How do we represent signed numbers in this representation?
The answer is that we must include a sign “half byte”.

The IBM format for packed decimal allows for an arbitrary number of digits in each number stored. The range is from 1 to 31, inclusive.

After adding the sign “half byte”, the number of hexadecimal digits used to represent any number ranges from 2 through 32.

Numbers with an even digit count are converted to proper format by the addition of a leading zero; 1234 becomes 01234.

The system must allocate an integral number of bytes to store each datum, so each number stored in Packed Decimal format must have an odd number of digits.

Packed Decimal: The Sign “Half Byte”

In the S/370 Packed Decimal format, the sign is stored “to the right” of the string of digits as the least significant hexadecimal digit.

The standard calls for use of all six non–decimal hexadecimal digits as sign digits. The standard is as follows:

Binary	Hex	Sign	Comments
1010	A	+	
1011	B	–	
1100	C	+	The standard plus sign
1101	D	–	The standard minus sign
1110	E	+	
1111	F	+	A common plus sign, resulting from a shortcut in translation from Zoned format.

Zoned Decimal Data

The **zoned decimal format** is a modification of the EBCDIC format.

The zoned decimal format seems to be a modification to facilitate processing decimal strings of variable length.

The length of zoned data may be from 1 to 16 digits, stored in 1 to 16 bytes.

We have the address of the first byte for the decimal data, but need some “tag” to denote the last (rightmost) byte.

The assembler places a “sign zone” for the rightmost byte of the zoned data.

The common standard is X’C’ for non–negative numbers, and X’D’ for negative numbers.

The format is used for constants possibly containing a decimal point, but it does not store the decimal point.

As an example, we consider the string “–123.45”.

Note that the format requires one byte per digit stored.

Creating the Zoned Representation

Here is how the assembler generates the zoned decimal format.

Consider the string “–123.45”.

The EBCDIC character representation is as follows.

Character	–	1	2	3	.	4	5
Code	6D	F1	F2	F3	4B	F4	F5

The decimal point (code 4B) is not stored.

A bit later we shall see the reason for this.

The sign character is implicitly stored in the rightmost digit.

The zoned data representation is as follows.

1	2	3	4	5
F1	F2	F3	F4	D5

The string “F1 F2 F3 F4 C5” would indicate a positive number,
with digits “12345”.

Packed Decimal Data

The packed decimal format is the one preferred by business for financial use.
Zoned decimal format is not used for arithmetic, but just for conversions.

As is suggested by the name, the packed format is more compact.

Zoned format	one digit per byte
Packed format	two digits per byte (mostly)

In the packed format, the rightmost byte stores the sign in its rightmost part,
so the rightmost byte of packed format data contains only one digit.

All other bytes in the packed format contain two digits, each with value in 0 – 9.
This implies that each packed constants always has an odd number of digits.

A leading 0 may be inserted, as needed.

The standard sign fields are:

negative	X'D'
non–negative	X'C'

The length may be from 1 to 16 bytes, or 1 to 31 decimal digits.

Examples	+7	**| 7C |**
	– 13	**| 01 | 3D |**

Example: Addition of Two Packed Decimal Values

Consider two **integer values**, stored in packed decimal format.

Note that 32–bit two's complement representation would limit the integer to a bit more than nine digits: –2, 147, 483, 648 through 2, 147, 483, 647.

Integers stored as packed decimals can have 31 digits and vary between

–9, 999, 999, 999, 999, 999, 999, 999, 999, 999, 999 and

+9, 999, 999, 999, 999, 999, 999, 999, 999, 999, 999 $(9.99 \bullet 10^{30})$

Consider the addition of the numbers –97 and 12,541.

–97 would be expanded to –097 and stored as **`097D`**.

12,541 would be stored as **`12541C`**.

The CPU does what we would do. It first pads the smaller number with 0's.

`00097D`

`12541C`

It then performs the operation, denoted as "12541 – 97", and stores the result as 12444, or **`12444C`** in packed decimal format.

Fixed Point: Where is the Decimal Point?

We shall now consider a sequence of numbers, in which each member is divided by ten to get the next one. This will be a short finite sequence.

The number 12345 is represented in packed decimal as **`12345C`**.

The number 1234.5 is represented in packed decimal as **`12345C`**.

The number 123.45 is represented in packed decimal as **`12345C`**.

The number 12.345 is represented in packed decimal as **`12345C`**.

The number 1.2345 is represented in packed decimal as **`12345C`**.

The number .12345 is represented in packed decimal as **`12345C`**.

Note that the format does not store the decimal point or any indication of the location of the decimal point. That is up to the code.

Put another way, what is the sum of **`012C`** and **`4C`**?

Is this really 12 + 4, with sum 16, represented as **`016C`**?

Is this really 12.0 + 0.4, with sum 12.4, represented as **`124C`**?

Fixed Point: Where is the Decimal Point? (Page 2)

Consider the following addition problem: 1.23 + 10.11405.

The answer is obviously 11.34405.

But the Packed Decimal format does not store the decimal point, so

1.23 is stored as **`123C`** and 10.11405 is stored as **`1011405C`**.

Using our previous logic, we line up the numbers **`0000123C`**
`1011405C`

The result is denoted **`10111528C`**, which might be 10.111528.
It cannot represent the correct answer.

This is a "feature" of **fixed point arithmetic**, in which all numbers are stored with the decimal point in the same place. In this system, the code must guarantee that 1.23 is stored as 0123000C.

The result is now **`0123000C`**
`1011405C,` stored as 1134405C.

The code must interpret and output this as the value 11.34405.