# IBM 370 Basic Data Types

This lecture discusses the basic data types used on the IBM 370,

1. Two's–complement binary numbers
2. EBCDIC (**E**xtended **B**inary **C**oded **D**ecimal **I**nterchange **C**ode)
3. Zoned Decimal Data
4. Packed Decimal Data
5. Floating Point Numbers

**Assumption:** The student is expected to be familiar with

a) Two's–complement integer arithmetic (from CPSC 2105)

b) The ASCII character set (also CPSC 2105)

c) The IEEE–754 floating point standard (also CPSC 2105)
We shall not spend much time on floating point.

This lecture will make frequent comparisons to the CPSC 2105 material.

# Terminology and Notation

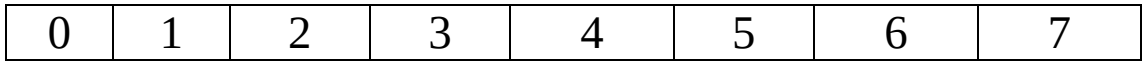
The IBM 370 is a byte–addressable machine; each byte has a unique address.

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

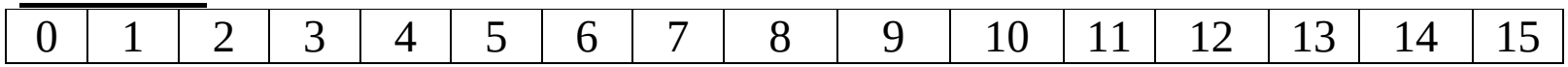
| | | |
|---|---|---|
| Byte | 8 binary bits | |
| Halfword | 16 binary bits | 2 bytes |
| Fullword | 32 binary bits | 4 bytes. |

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

**Byte**




**Halfword**




**Fullword**


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

Reference: Textbook, Chapter 5 (page 103) and Chapter 9 (pages 208 & 209).

# Parity Bits

This is a detail with almost no implications for assembly language programming.

Each 8–bit byte is stored in nine bits of primary memory.
The bit layout in the memory is as follows.

| P | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|

Consider the following two characters, stored as EBCDIC codes.

“0” X’F0’ 1111 0000 “1” X’F1’ 1111 0001

The IBM 370 standard is to use **odd parity**, meaning that the
nine–bit storage location contains an odd number of one bits.

Here is the storage for the two sample characters.

“0”

| P | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |

“1”

| P | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 |

Reference: Textbook, Chapter 1 (page 4)

# Binary Arithmetic

The IBM 370 uses standard two's–complement integer arithmetic.

Binary integers come in three sizes: byte, halfword, and fullword.

| | | | |
|---|---|---|---|
| B | 8–bit signed integers | $-2^7$ to $2^7 - 1$ | –128 to +127. |
| H | 16–bit signed integers | $-2^{15}$ to $2^{15} - 1$ | –32768 to 32767 |
| F | 32–bit signed integers | $-2^{31}$ to $2^{31} - 1$ | |

**NOTE:** Figure 9–1, on page 212 of the textbook, is in error.

Again, IBM terminology is nonstandard. Here is how the textbook describes the general purpose registers, which are used to store binary data.

> "Each register contains 32 data bits, numbered from left to right as 0 through 31. … For binary values, bit 0 (to the left) is the sign bit and bits 1 – 31 are data."

The IBM documentation will tend to discuss 32–bit signed integers as "31 bit data". This makes sense; it is just not the present–day standard.

Reference: Textbook, Chapter 6 (page 107), Chapter 9 (pages 208 – 214)

# Character Data

The IBM series computers store character data in EBCDIC form. As the name suggests, this is an extension of an earlier format, **BCD** (**B**inary **C**oded **D**ecimal).

EBCDIC is an 8–bit code; each character is represented by eight bits.
These codes are conventionally represented as two hexadecimal digits.

This table shows the hexadecimal form of the EBCDIC for a number of characters.
The ASCII codes are also given, just for comparison.

| Character | EBCDIC | ASCII |
|---|---|---|
| blank | 40 | 20 |
| 0 – 9 | F0 – F9 | 30 – 39 |
| A – I | C1 – C9 | 41 – 49 |
| J – R | D1 – D9 | 4A – 52 |
| S – Z | E2 – E9 | 53 – 5A |
| a – i | 81 – 89 | 61 – 69 |
| j – r | 91 – 99 | 6A – 72 |
| s – z | A2 – A9 | 73 – 7A |

NOTE: For these EBCDIC characters, the second digit is always a decimal digit.
This shows the origin as punch card codes.

# Bytes and Zones

Character data is stored one character per byte.

The need for efficient processing of character data may be the origin of the popularity of byte addressability in computers.

In the IBM view, the 8–bit byte is divided into
a 4–bit zone, and
a 4–bit numeric field.

This division reflects the origin of EBCDIC code as punched–card codes.

This division is shown in the following table.

| Portion | Zone | | | | Numeric | | | |
|---|---|---|---|---|---|---|---|---|
| Bit | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

Note the following important zones

| | |
|---|---|
| A – I | C |
| J – R | D |
| S – Z | E |
| 0 – 9 | F |

References: Textbook, Chapter 7 (page 137) and Chapter 8 (page 163)

# 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.

Zoned Decimal format is not much used.

# 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.

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.

# Packed Decimal Data

Arithmetic in the IBM 370 is performed on data in the packed decimal format.

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 \| |

# Why The Sign on the Right?

The first thing to note is that business data tend to be variable length. A list of debits might include the strings 3.12, 1003.47, 67.18, etc.

Consider the problem of reading a list of signed integers. We specify that each integer will have between 1 and 7 digits and be found in columns 1 – 8 of the punch card.

Here is a typical list.

```
  123
 -765
17765
 -96
```

A list such as this is easily processed by a program written in a modern high–level programming language. It is a bit of a trick to implement in assembly language.

The program must scan left to right until it finds the first non–blank character. If the character is a "`-`", the number is negative. Otherwise it is not.

While one can write a program to interpret this list correctly, it is not trivial to do so. The earlier programmers sought a different solution.

# Why The Sign on the Right? (Part 2)

All options for handling the input require the sign to be in a fixed location.

Here is one option.  Its limitations are obvious.

```
      123
-     765
    17765
-     96
```

The option selected roughly corresponds to placing the sign after the number.
The list to process would resemble the following.

```
     123
     765-
   17765
     96-
```

The digits are found in columns 1 – 7, and the sign in column 8.

Given this, the placement of the sign indicator to the right of each of the
Zoned Decimal and Packed Decimal formats was an easy choice.

# Floating Point Data

Floating point is the format of preference for scientific computations.

Floating point is not commonly used for financial applications, due to round–off problems. More on this later.

All floating point formats are of the form (S, E, F) representing $(-1)^S \bullet B^E \bullet F$

- S — the sign bit, 1 for negative and 0 for non–negative.
- B — the base of the number system; one of 2, 10, or 16.
- E — the exponent.
- F — the fraction.

The IEEE–754 standard calls for a binary base.

The IBM 370 format uses base 16.

Each of the formats represents the numbers in normalized form.

For IBM 370 format, this implies that $0.0625 < F \leq 1.0$. Note $(1/16) = 0.0625$.

# Floating Point: Storing the Exponent

The exponent is stored in excess–64 format as a 7–bit unsigned number.

This allows for both positive and negative exponents.

A 7–bit unsigned binary number can store values in the range [0, 127] inclusive.

The range of exponents is given by $0 \le (E + 64) \le 127$, or

$-64 \le E \le 63$.

The leftmost byte of the format stores both the sign and exponent.

| Bits | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| Field | Sign | Exponent in Excess–64 format | | | | | | |

**Examples**

Negative number, Exponent = –8 E + 64 = 56 = 48 + 8 = X’38’ = B’011 1000’.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| Sign | 3 | | | 8 | | | |
| 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 |

The value stored in the leftmost byte is 1011 1000 or B8.

# Converting Decimal to Hexadecimal

The first step in producing the IBM 370 floating point representation of a real number is to convert that number into hexadecimal format.

The process for conversion has two steps,
one each for the integer and fractional part.

**Example:** Represent 123.90625 to hexadecimal.

Conversion of the integer part is achieved by repeated division with remainders.

| | | | | |
|---|---|---|---|---|
| 123 / 16 | = 7 | with remainder | 11 | X'B' |
| 7 / 16 | = 0 | with remainder | 7 | X'7'. |

Read bottom to top as X'7B'. Indeed 123 = 7•16 + 11 = 112 + 11.

Conversion of the fractional part is achieved by repeated multiplication.

0.90625 • 16 = 14.5 Remove the 14 (hexadecimal E)

0.5 • 16 = 8.0 Remove the 8.

The answer is read top to bottom as E8.

The answer is that 123.90625 in decimal is represented by X'7B.E8'.

# Converting Decimal to IBM 370 Floating Point Format

The decimal number is 123.90625.

Its hexadecimal representation is 7B.E8.

Normalize this by moving the decimal point two places to the left.

The number is now $16^2 \bullet 0.7BE8$.

The sign is 0, as the number is not negative.

The exponent is 2, E + 64 = 66 = X’42’.  The leftmost byte is X’42’.

The fraction is 7BE8.

The left part of the floating point data is 427BE8.

In single precision, this would be represented in four bytes as 42 78 E8 00.

# 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?