

(extract)



z/OS

Assembler

Michel Castelein - Arcis Services

11 June 2010

arcis@advalvas.be

<http://www.arcis-services.net/>

(extract)**EBCDIC and zoned decimal**

Using standard EBCDIC, every displayable character is represented by *one* byte¹.

There are a number of different versions of EBCDIC, customized for different countries². For instance, *codepage* 1148 (Cp1148) is “ECECP Multilingual”.

Codepage 1148 - Belgium, Switzerland - ECECP

	-0	-1	-2	-3	-4	-5	-6	-7	-8	-9	-A	-B	-C	-D	-E	-F
0-		0001	0002	0003	009C	0005	0086	0007	0097	008D	008E	000B	000C	000D	000E	000F
1-	0010	0011	0012	0013	009D	0085	0016	0087	0018	0019	0092	008F	001C	001D	001E	001F
2-	0080	0081	0082	0083	0084	000A	0017	001B	0088	0089	008A	008B	008C	0005	0006	0007
3-	0090	0091	0016	0093	0094	0095	0096	0004	0098	0099	009A	009B	0014	0015	009E	001A
4-	0020	00A0	00E2	00E4	00E9	00E1	00E3	00E5	00E7	00F1	005B	002E	003C	0028	002B	0021
5-	&	é	ê	ë	è	í	î	ï	ì	ß]	\$	*)	;	^
6-	-	/	Â	Ä	À	Á	Ã	Å	Ç	Ñ	!	,	%	_	>	?
7-	ø	É	Ê	Ë	È	Í	Î	Ï	Ì	`	:	#	@	'	=	"
8-	Ø	a	b	c	d	e	f	g	h	i	«	»	ð	ý	þ	±
9-	°	j	k	l	m	n	o	p	q	r	ª	º	æ	,	Æ	€
A-	µ	~	s	t	u	v	w	x	y	z	;	¿	Ð	Ý	þ	®
B-	¢	£	¥	·	©	§	¶	¼	½	¾	¬		-	¨	'	×
C-	{	A	B	C	D	E	F	G	H	I	-	ô	ö	ò	ó	õ
D-	}	J	K	L	M	N	O	P	Q	R	¹	û	ü	ù	ú	ÿ
E-	\	÷	S	T	U	V	W	X	Y	Z	²	Ô	Ö	Ò	Ó	Õ
F-	0	1	2	3	4	5	6	7	8	9	³	Û	Ü	Ù	Ú	

e.g. character 'A' = hexadecimal value 'C1' (1 byte)
character string 'BraVO' = X'C29981E5D6' (5 bytes)
C'C1'

¹ High Level Assembler also supports *double-byte character set* (DBCS) and **UNICODE** characters.

² See <http://www.ibm.com/servers/eserver/iseries/software/globalization/codepages.html>

(extract)

C' '(space)	= X'40'
C'1234'	= X'F1F2F3F4'
C'-0.25'	= X'60F04BF2F5'
C'08/07/2005'	= X'F0F861F0F761F2F0F0F5'

- The first four bits of the byte is called the *zone* and represent the category of the character.
- The last four bits of the byte is called the *number* and identify the specific character.

Notice the character representation of *decimal digits*: the *zone* is always X'F', i.e. C'0' is X'F0', C'1' is X'F1', C'2' is X'F2', ..., C'9' is X'F9'.

- However, do not confuse EBCDIC representation with *zoned decimal* representation.

*Zoned decimal data:*

- A zoned decimal representation stores a decimal digit (d) in the right nibble of each byte.
- For all but the rightmost byte, the left nibble is the numeric zone nibble (i.e. X'F').
- The sign of a zoned decimal number (s) is represented in the left nibble of the rightmost byte.

i.e.

Fd	Fd	...	Fd	sd
----	----	-----	----	----

For zoned decimal data (as well as for packed decimal data¹), the *sign nibble* (i.e. s) must have a hexadecimal value in the range of A to F where:

- A, C, E, or F is used to indicate a positive value. → CAFE (+)
- B or D is used to indicate a negative value.

The *preferred* sign codes¹ are C (Credit) for a positive value, and D (Debit) for a negative value.

e.g. printable characters	EBCDIC	zoned decimal
123	X'F1F2F3'	X'F1F2 <u>C</u> 3'
123	X'F1F2F3'	X'F1F2 <u>F</u> 3'
+456	X' <u>4</u> EF4F5F6'	X'F4F5 <u>C</u> 6'
-890	X' <u>6</u> 0F8F9F0'	X'F8F9 <u>D</u> 0'

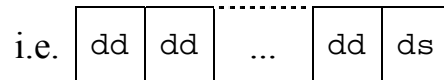
- Only *unsigned* decimal integers *can* have identical EBCDIC and zoned decimal values.

¹ See further.

(extract)

Packed decimal

In the packed decimal format, each byte contains two decimal digits (*dd*), except for the rightmost byte, which contains a decimal digit (*d*) in the left half and the sign code (*s*) in the right half.



- F, C (credit), and D (debit) are the sign codes *produced* by the CPU.
- However, any nibble A, B, C, D, E, or F will be recognized as valid sign code.
 - C, A, F, or E is used for plus (+).
 - B or D is used for minus (-).



The conversion of zoned decimal into packed decimal is done by means of the *pack* (**PACK**) instruction: this machine instruction swaps the nibbles of the rightmost byte, and removes the zones of the preceding bytes (if any).

e.g. X'F3' 1-byte zoned decimal

↓ PACK ↓

X'3F' 1-byte packed decimal

X'F9F8F7C6' 4-byte zoned decimal

↓ PACK ↓

X'09876C' 3-byte packed decimal

- Because data consists of whole bytes, packing an *even* number of bytes yields one leading 0 as *padding* nibble.

The conversion of packed decimal into zoned decimal is done by means of the *unpack* (**UNPK**) instruction: this machine instruction swaps the nibbles of the rightmost byte, and expands the nibbles of the preceding bytes (if any) into full bytes by prefixing every nibble with X'F'.

e.g. X'123F' 2-byte packed decimal

↓ UNPK ↓

X'F1F2F3' 3-byte zoned decimal

X'7D' 1-byte packed decimal

↓ UNPK ↓

X'D7' 1-byte zoned decimal

- In COBOL, packed decimal is also known as *internal decimal*.
- In COBOL, zoned decimal is also known as *external decimal*.
- Another synonym for zoned decimal is *unpacked decimal*.

(extract)

Fixed-point

Fixed-point numbers are *signed* binary integers.

- The length of a fixed-point is 2 (halfword) or 4 bytes (fullword).
- The leftmost bit is used as *sign bit*.
 - In z/OS, bits, bytes, and words are numbered from *left to right*, beginning with *zero*.



Therefore, the sign bit is referred to as *bit 0*.

- The remaining bits are a positional representation based on two, i.e. a positional representation with radix two.

When the sign bit is *zero*, the fixed-point represents a *positive* value.

$$\begin{aligned}
 \text{e.g. } 26 &= 16 + 8 + 2 = 2^{**4} + 2^{**3} + 2^{**1} \\
 &= 11010_2 \\
 &= \text{X}'001\text{A}' \quad (\text{2 bytes, i.e. a halfword}) \\
 &= \text{X}'0000001\text{A}' \quad (\text{4 bytes, i.e. a fullword}) \\
 &= (1 * 16^{**1}) + (10 * 16^{**0}) = 16 + 10 = 26
 \end{aligned}$$

When the sign bit is *one*, the fixed-point represents a *negative* value, whereas one uses the *two's complement notation*.

$$\begin{aligned}
 \text{e.g. } -7 &= \text{X}'?????' \\
 &1. \text{ (halfword) binary representation of the positive value:} \\
 &\quad 0000 \ 0000 \ 0000 \ 0111_2 \\
 &2. \text{ reverse each bit:} \\
 &\quad 1111 \ 1111 \ 1111 \ 1000_2 \\
 &3. \text{ add one to get the two's complement:} \\
 &\quad 1111 \ 1111 \ 1111 \ 1001_2 \\
 -7 &= \text{X}'FFF9'
 \end{aligned}$$

Two's complement notation actually means that the leftmost bit represents a *negative* quantity.

$$\begin{aligned}
 \text{e.g. } \text{x}'FFF9' &= 1111 \ 1111 \ 1111 \ 1001_2 \\
 &= (-2^{**15}) + 2^{**14} + 2^{**13} + 2^{**12} + \dots + 2^{**0} \\
 &= -32768 + 16384 + 8192 + 4096 + \dots + 1 \\
 &= -7
 \end{aligned}$$

(extract)

Floating-point

MVS, OS/390, and z/OS use *hexadecimal* floating-point (**HFP**) formats.

- A synonym is *base 16 S/370-S/390* floating-point formats¹.
- The length of an HFP number is either 4 bytes (*single* precision), 8 bytes (*double* precision), or 16 bytes (*extended* precision).

Support for IEEE *binary* floating-point (**BFP**) data formats and machine instructions was introduced with OS/390 V2R6 and High Level Assembler (**HLASM**) Release 3.

This support provides improved performance for Java applications using floating-point operations, and improved portability for C/C++ applications that use floating-point.

- IEEE binary floating-point is also known as *base 2 IEEE-754* floating-point².
- The length of a BFP number is either 32 bits (single-precision, a.k.a. *short* floating-point), 64 bits (double-precision, a.k.a. *long* floating-point), or 128 bits (quadruple-precision, a.k.a. *extended* floating-point)³.

The value of an HFP number is $\textit{fraction} \times 16^{\textit{exponent}}$

- The leftmost bit (i.e. bit 0) indicates the *sign* of the fraction: zero for a positive, one for a negative number.
- All other bits of the leftmost byte (i.e. bits 1-7) specify the *characteristic*, i.e. an excess-64 notation of the *exponent*.
- The remaining bits (i.e. bits 8-31 for single, bits 8-63 for double, or bits 8-63 & 72-127 for extended precision⁴) specify the *fraction's* absolute value. This value is always < 1 .

The value of a BFP number is $\textit{fraction} \times 2^{\textit{exponent}}$

- The leftmost bit (i.e. bit 0) indicates the *sign* of the fraction: zero for a positive, one for a negative number.
- The *characteristics* is specified as follows⁵:
 - For a 32-bit (4-byte) BFP, the following 8 bits (i.e. bits 1-8) represent the *exponent*, using an excess-127 notation.
 - For a 64-bit (8-byte) BFP, the following 11 bits (i.e. bits 1-11) represent the *exponent*, using an excess-1023 notation.
 - For a 128-bit (16-byte) BFP, the following 15 bits (i.e. bits 1-15) represent the *exponent*, using an excess-16383 notation.

¹ The *base* is also referred to as *radix*.


² Cf. the IEEE 754-1985 standard for *binary* floating-point arithmetic.

³ On Intel processors (e.g. in Windows), one uses 80 bits to represent an IEEE extended floating-point number!

⁴ In a 16-byte HFP, the first byte of the second word is *ignored*.

⁵ If the characteristics consists of *all* ones, the BFP number represents an infinity or a NaN (“*Not-a-Number*”).

(extract)

- The remaining bits define the *fraction's* absolute value.
 - For a 32-bit BFP, the final 23 bits (i.e. bits 9-31) represent a value < 1 .
 - For a 64-bit BFP, the final 52 bits (i.e. bits 12-63) represent a value < 1 .
 - For a 128-bit BFP, the final 112 bits (i.e. bits 16-127) represent a value < 1 .
 - *But one must add one to those values to get the BFP's fraction¹.* 

e.g. 4-byte HFP X'C1280000' = ???

byte 0 = X'C1' = 11000001₂ ⇒ bit 0 = 1₂

⇒ the fraction is negative (-)

bits 1-7 = 1000001₂ = X'41' = 65

⇒ the exponent is 65 - 64 (*bias quantity*) = 1

bits 8-31 = X'280000'

⇒ the fraction's absolute value is $2 \times 16^{-1} + 8 \times 16^{-2}$

$$= 2 \times \frac{1}{16^1} + 8 \times \frac{1}{16^2} = \frac{2}{16} + \frac{8}{256} = 0.15625$$

the value of this HFP is $-0.15625 \times 16^1 = -2.5$

e.g. 8-byte HFP X'436D100000000000' = ???

byte 0 = X'43' = 01000011₂ ⇒ bit 0 = 0₂

⇒ the fraction is positive (+)

bits 1-7 = 1000011₂ = X'43' = 67

⇒ the exponent is 67 - 64 (*bias quantity*) = 3

bits 8-63 = X'6D10000000000000'

⇒ the fraction's absolute value is $6 \times 16^{-1} + 13 \times 16^{-2} + 1 \times 16^{-3}$

$$= 0.42602539$$

the value of this HFP is $+0.42602539 \times 16^3 = +1745$

e.g. 16-byte HFP X'4134000000000000033000000000000000' = ???

byte 0 = X'41' = 01000001₂ ⇒ bit 0 = 0₂

⇒ the fraction is positive (+)

bits 1-7 = 1000001₂ = X'41' = 65

⇒ the exponent is 65 - 64 (*bias quantity*) = 1

bits 8-63 and 72-127 = X'3400000000000000000000000000000000'

⇒ the fraction's absolute value is $3 \times 16^{-1} + 4 \times 16^{-2}$

$$= 0.203125$$

the value of this HFP is $+0.203125 \times 16^1 = +3.25$

¹ This rule *does not* apply to the 80-bit IEEE binary floating-point used on Intel machines. The final 64 bits of such a floating-point represent a value that may exceed 1 because the weight of bit 16 is not 2^{-1} but 2^0 !

(extract)

Support for IEEE *decimal* floating-point (**DFP**) data formats and machine instructions was introduced with z/OS V1R10 and HLASM Release 6.

- System **Z9** is the first IBM machine to support the DFP machine instructions¹.
- IEEE decimal floating-point² was previously known as *base 10 IEEE-754r* (i.e. “revision”) floating-point.

Each DFP number represents either a NaN (“*Not-a-Number*”), an infinity, or a finite number.

- NaN is a *special* value returned as the result of certain “invalid” (BFP³ or) DFP computations, such as $0/0$, $\infty \times 0$, or $\sqrt{-1}$.
- (BFP⁴ and) DFP numbers can represent $\pm\infty$.
- The DFP finite numbers are defined by a *sign*, an *exponent* (which is a power of ten), and a decimal integer *coefficient*.

The value of a DFP finite number is given by $(-1)^{\text{sign}} \times \text{coefficient} \times 10^{\text{exponent}}$

Working directly with decimal (base 10) fractions avoids the rounding errors that otherwise typically can occur when converting decimal fractions (common in human-entered data) to binary (base 2) or hexadecimal (base 16) fractions.

- For example, the decimal value 0.10, can be represented exactly in DFP as $+1 \times 10^{-1}$, but in the single-precision BFP format, it must be approximated, giving in fact a decimal value of 0.100000001490116119384765625.

This explains why DFP is ideal for business-math applications that cannot tolerate approximations⁵.

The length of a DFP number is either 4 bytes (*short* format), 8 bytes (*long* format), or 16 bytes (*extended* format).

- Unlike HFP and BFP operands, DFP operands are stored in a specially encoded, storage-efficient format.

IEEE specifies two possible encodings for the DFP *coefficient*⁶: (1) *Binary Integer Decimal (BID)*, designed by Intel, and (2) *Densely Packed Decimal (DPD)*, designed by IBM.

- In a BID-based DFP, the entire *coefficient* is encoded as an unsigned binary integer.
- In a DPD-based DFP, the first (leftmost) decimal digit of the *coefficient* is encoded as an unsigned binary integer, but the remaining decimal digits

¹ See the *POP*.

² Cf. the IEEE 754-2008 standard for (*binary and decimal*) floating-point arithmetic.

³ With BFP, a NaN entity is represented by a characteristics of all ones and a nonzero fraction.

⁴ With BFP, infinity is represented by a characteristics of all ones and a zero fraction.

⁵ With DFP, results are as people expect them, identical to what would be obtained using pencil and paper.

⁶ The *coefficient* is also referred to as *significand*.

(extract)

of the *coefficient* are divided into groups of three decimal digits, and each group (“delet”) is *compressed* into 10 bits.

Use the following table to decode a DPD¹-encoded “delet”, with 10 bits $b_{(0)}$ to $b_{(9)}$ (where an “x” denotes “don’t care”), into 3 decimal digits $d_{(1)}$ to $d_{(3)}$:

$b_{(6)}$	$b_{(7)}$	$b_{(8)}$	$b_{(3)}$	$b_{(4)}$	$d_{(1)}$	$d_{(2)}$	$d_{(3)}$
0	x	x	x	x	$4b_{(0)} + 2b_{(1)} + b_{(2)}$	$4b_{(3)} + 2b_{(4)} + b_{(5)}$	$4b_{(7)} + 2b_{(8)} + b_{(9)}$
1	0	0	x	x	$4b_{(0)} + 2b_{(1)} + b_{(2)}$	$4b_{(3)} + 2b_{(4)} + b_{(5)}$	$8 + b_{(9)}$
1	0	1	x	x	$4b_{(0)} + 2b_{(1)} + b_{(2)}$	$8 + b_{(5)}$	$4b_{(3)} + 2b_{(4)} + b_{(9)}$
1	1	0	x	x	$8 + b_{(2)}$	$4b_{(3)} + 2b_{(4)} + b_{(5)}$	$4b_{(0)} + 2b_{(1)} + b_{(9)}$
1	1	1	0	0	$8 + b_{(2)}$	$8 + b_{(5)}$	$4b_{(0)} + 2b_{(1)} + b_{(9)}$
1	1	1	0	1	$8 + b_{(2)}$	$4b_{(0)} + 2b_{(1)} + b_{(5)}$	$8 + b_{(9)}$
1	1	1	1	0	$4b_{(0)} + 2b_{(1)} + b_{(2)}$	$8 + b_{(5)}$	$8 + b_{(9)}$
1	1	1	1	1	$8 + b_{(2)}$	$8 + b_{(5)}$	$8 + b_{(9)}$

Prior to encoding, the *exponent* is converted to an unsigned binary value called the *biased exponent* by adding a *bias value* which is 101 for 4-byte DFP numbers, 398 for 8-byte DFP numbers, and 6176 for 16-byte DFP numbers.

- The use of the bias allows both negative and positive exponents.

Each DFP number consists of four fields:

1. The *sign bit* (**S**) is zero for plus and one for minus.
The S bit is in bit 0.
2. The *combination field* (**CF**) is a 5-bit field which occupies bits 1-5.
 - If the CF field is $\overline{1111}1_2$, the DFP number represents a NaN.
 - If the CF field is $\overline{1111}0_2$, the DFP number represents an infinity.
 - If any of the first four bits of the CF field (i.e. bits 1-4) is zero, the DFP number represents a finite number, and the CF field contains the first (leftmost) two bits of the *biased exponent* and specifies the first (leftmost) decimal digit (**LMD**) of the *coefficient*²:
 - If bits 1-2 are 00_2 , 01_2 , or 10_2 (i.e. *not* 11_2), they represent the first (leftmost) two bits of the *biased exponent*; otherwise (i.e. if bits 1-2 are 11_2), the first two bits of the *biased exponent* are in bits 3-4.
 - If bits 1-2 are 00_2 , 01_2 , or 10_2 (i.e. *not* 11_2), the LMD is an unsigned binary integer represented by the 4-bit string consisting of 0_2 followed by bits 3-5; otherwise (i.e. if bits 1-2 are 11_2), the LMD is an unsigned

¹ DPD is a *compressed form of binary-coded decimal (BCD)*, derived from Chen-Ho encoding and invented by Mike Cowlishaw.

² DFP finite numbers are not normalized, i.e. leading and trailing zeros may exist in the coefficient.

(extract)

binary integer represented by the 4-bit string consisting of 100_2 followed by bit 5.

3. The *biased exponent continuation field (BECF)* has a width of 6, 8, or 12 bits.

For DFP finite numbers, the BECF field contains the remaining bits of the *biased exponent*.

In a 4-byte DFP number, the BECF field occupies bits 6-11. In an 8-byte DFP number, it occupies bits 6-13. In a 16-byte DFP number, it occupies bits 6-17.

4. The *coefficient continuation field (CCF)* has a width of 20, 50, or 110 bits.

For DFP finite numbers, the CCF field specifies the remaining decimal digits of the *coefficient*.

In a 4-byte DFP number, the CCF field occupies bits 12-31. In an 8-byte DFP number, it occupies bits 14-63. In a 16-byte DFP number, it occupies bits 18-127.

e.g. 4-byte DPD-based DFP $X'77FDF2C3' = ???$

byte 0 = $X'77' = 01110111_2 \Rightarrow$ bit 0 = $0_2 \Rightarrow$ the DFP number is positive (+)

byte 0 = $X'77' = 01110111_2 \Rightarrow$ bits 1-5 (i.e. the **CF** field) = 11101_2

- bits 1-4 are *not* $1111_2 \Rightarrow$ this DFP number *does* represent a finite number
- bits 1-2 are $11_2 \Rightarrow$ the two most significant bits (MSBs) of the *biased exponent* reside in bits 3-4, i.e. their value is 10_2
- bits 1-2 are $11_2 \Rightarrow$ the leftmost decimal digit (LMD) of the *coefficient* is defined by the 4-bit unsigned binary integer consisting of 100_2 followed by bit 5

bit 5 is $1_2 \Rightarrow$ the value of the **LMD** is $1001_2 = 9$

bytes 0-1 = $X'77FD' = 01110111111111101_2$

\Rightarrow bits 6-11 (i.e. the **BECF** field) = 111111_2

- the *biased exponent* is defined by an unsigned binary integer consisting of the **MSBs** (i.e. 10_2) followed by the **BECF** field
- \Rightarrow the value of the *biased exponent* is $10111111_2 = 191$
- the *bias* for 4-byte DFP numbers is 101
- \Rightarrow the value of the (unbiased) *exponent* is $191 - 101 = 90$

bytes 1-3 = $X'FDF2C3' = 1111110111111001011000011_2$

\Rightarrow the two 10-bit declets in bits 12-31 (i.e. the **CCF** field) represent the following two groups of 3 decimal digits¹:

- for the 1st declet 1101111100_2 , the “ $b_{(6)}-b_{(7)}-b_{(8)}-b_{(3)}-b_{(4)}$ ” combination is $11011_2 \Rightarrow$ the 1st decimal digit represented by this declet = $8 + b_{(2)} = 8 + 0 = 8$; the 2nd decimal digit represented by this declet = $4b_{(3)} + 2b_{(4)} + b_{(5)} = 4 \times 1 + 2 \times 1 + 1 = 7$; the 3rd decimal digit represented by this declet = $4b_{(0)} + 2b_{(1)}$

¹ You must use the DPD-decoding table.

