Memory Model Tutorial I : big and little endian

1. Bits, Cells, Bytes, Words, Big Endian and Little Endian

All computers divide memory into cells that have consecutive addresses. The most common cell size is 8 bits. Such a 8-bit cell is called a byte. The reason for using 8 bits in a byte is that ASCII characters are 7 bits long, so one ASCII character plus a parity bit fits into a byte. Bytes are generally grouped into 4-byte (32-bit) or 8-byte (64-bit) words.

In the following we will consider an architecture with an 8-bit cell (a byte) and a 32-bit word (4 bytes). Generally, the memory is addressed by byte. The n-th word has address n in word, 4*n in byte and 32*n in bit, cf. Fig. 1.

…	0x01	0x00	0x01	0x15	0x00	0x0B	0x0C	0x10	0x00	0x00	0x0A	0x12	**data**
89	88	80	72	64	56	48	40	32	24	16	8	0	**bit**
12	11	10	9	8	7	6	5	4	3	2	1	0	**byte**
3				2				1				0	**word**

Fig. 1. Linear memory addresses (in decimal) with random data (in hexadecimal).

Because of the word structure (4 bytes), it is natural to represent memory as a two dimensional table with a word for each row, cf. Fig. 2.

				Address (in byte)
0x00	0x00	0x0A	0x12	**0**
3	**2**	**1**	**0**	
0x00	0x0B	0x0C	0x10	**4**
7	**6**	**5**	**4**	
0x01	0x00	0x01	0x15	**8**
11	**10**	**9**	**8**	

Fig. 2. 4-byte memory addresses with random data.

The bytes in a word can be numbered from right-to-left or left-to-right. The latter system, where the numbering begins at the “big” end (i.e. high order bit) is called big endian (e.g. Motorola processors), in contrast to little endian (e.g. Intel processors), cf. Fig. 3.

Address (in byte)	**Big endian**				**Little endian**				**Address (in byte)**
0	0	1	2	3	3	2	1	0	**0**
4	4	5	6	7	7	6	5	4	**4**
8	8	9	10	11	11	10	9	8	**8**
12	12	13	14	15	15	14	13	12	**12**
	(a)				**(b)**				

Fig. 3. (a) Big endian memory (b) Little endian memory.
The values of the byte cells correspond to their address.

It is important to understand that big and little endian refers to the ordering of *bytes*. But the *bit* ordering "inside" individual bytes and words is always right to left: In both, big endian and little endian systems, a 32-bit integer with numerical value of 6 is represented by the bits 110 in the rightmost (low order) 3 bits of a word and zeros in the leftmost 29 bits, i.e. in the big endian system, the 110 bits are in byte 3 (or 7, or 11, etc.), whereas in the little endian system they are in byte 0 (or 4, or 8, etc.). In both cases the word containing the integer has address 0 (or 4, or 8, etc.)

2a. An Example for some data

In Fig. 4 we represent the following data, in both big and little endian systems :

1. The integer 1713 at address 0 (hexadecimal: 0x6B1).
2. The string “my cat” at address 4 (ASCII: m=0x6D, y=0x79, space=0x20, c=0x63, a=0x61, t=0x74).
3. The character 'c' at address 12 (ASCII: c=0x63).
4. The character 'd' at address 16 (ASCII: d=0x64).

Address	Big endian				Little endian				Address
0x0	0x00	0x00	0x06	0xB1	0x00	0x00	0x06	0xB1	**0x0**
0x4	0x6D	0x79	0x20	0x63	0x63	0x20	0x79	0x6D	**0x4**
0x8	0x61	0x74	***	***	***	***	0x74	0x61	**0x8**
0xC	0x63	***	***	***	***	***	***	0x63	**0xC**
0x10	0x64	***	***	***	***	***	***	0x64	**0x10**
	(a)				**(b)**				

Fig. 4. Big and little endian memory.
With hexadecimal memory cells and address values.
*** stands for garbage data.

2b. An Example for C statements

We will now represent the state of the memory after the execution of a C statement. The memory regions where the data will be stored have an abstract name (i.e. the value of the variable *k* will be stored at the address *&k* which is chosen at compile time).

Similar to the previous example, Fig. 5 represents the state of the memory after the execution of the following C statements, in both big and little endian systems :

1. int k = 1713;
2. char s[8] = “my cat”;
3. char c1 = 'c' ;
4. char c2 = 'd' ;

Remark: Note that the string “my cat” ends with NULL (= '\0' = 0x00). Therefore there is an extra character at line s+4 of Fig. 5.
Remark 2: s is declared as being an array of characters, therefore s already is an address (the address where the array starts).

Address	Big endian				Little endian				Address
&k	0	0	1713		0	0	1713		**&k**
s	m	y		c	c		y	m	**s**
s + 4	a	t	\0	***	***	\0	t	a	**s + 4**
&c1	c	***	***	***	***	***	***	c	**&c1**
&c2	d	***	***	***	***	***	***	d	**&c2**
	(a)				**(b)**				

Fig. 5. Big and little endian memory.
With symbolic memory cell and address values.
*** stands for garbage data.

2c. Simplified Memory Representation

In order to simplify the state representation of the memory we will make another abstraction step. This simplified representation will be independent from the endian system (big or little) and from the word dimension (usually 4-bytes). Fig. 6 shows a simplified version of the example discussed previously.

Variable		Address	
Name	**Value**	**Name**	**Possible value**
k	1713	&k	0x00
	my cat	s	0x04
c1	c	&c1	0x0C
c2	d	&c2	0x10

Fig. 6. Simplified Memory of Fig. 5
With symbolic names and values for the variables and addresses and with symbolic names and possible hexadecimal values for the addresses (the value of the addresses are determined at run time).

Furthermore we will hide all addresses, showing only the variable names and values, cf. Fig. 7. Pointers will be represented by arrows.

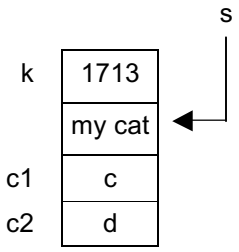

Fig. 7. Simplified Memory of Fig. 6
With symbolic names and values for the variables.

Sometimes it is interesting to represent an array in a more precise way, cf. Fig.8.

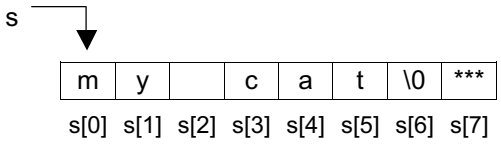

Fig. 8. Memory of an array.
Example with symbolic indexes for the C statement :
char s[8] = “my cat”;

References

[1] Andrew S. Tanenbaum, *Structured Computer Organization*, 5th Edition, Prentice Hall, 2006.

[2] Wikipedia: http://en.wikipedia.org/wiki/Endianness : more about big and little endian

Authors: Béat Hirsbrunner and Fulvio Frapolli, University of Fribourg, Switzerland
Version 0.2, 25 February 2008 (BH+FF); Version: 0.3, 15 July 2010, rev. Sept. 2013 (BH)