AP COMPUTER SCIENCE
TOPIC IV: DATA REPRESENTATION

PAUL L. BAILEY

1. Memory

The *memory* of a computer is where the computer stores temporary information; when you turn off the computer, the contents of the memory are gone. The memory may be viewed as a sequence of slots which may contain either a zero and or a one. The number of slots is the indicates size of that particular computer's memory.

Each slot is called a *bit*. These bits are grouped together into blocks of various sizes. Then each block is capable of holding a natural number, stored in base two. If the block contains n bits, it can contain a number between 0 and $2^n - 1$.

A *byte* is a block of eight bits. Thus a byte can contain a number between 0 and 256.

A *word* is a block of w bits, where w depends on the particular computer; we call w the *word size* of that computer. The word size is typically a multiple of eight by a power of two, but this is not necessarily the case. For example, old DEC PDP computers used 12, 18, and 36 bit words, interpreted as 4, 6, and 12 octal digits, respectively. Intel chips used in DOS and Windows computers have used 16, 32, and more recently, 64 bit words, interpreted as 4, 8, and 16 hexadecimal digits, respectively.

The computer views its memory as broken up into words; the number of bits in the memory is a multiple of the word size.

The computer accesses its memory by enumerating its words; the number (or location) of a word is called its *address*. In the simplest case, the computer stores an address as a word, so the computer's memory contains at most 2^w words, numbered from 0 to $2^w - 1$. The maximum number of words the computer can access is called the *address space* of the computer. We note that, using segmentation, Intel chips do not work this way, and their address spaces are larger then 2^w.

When the computer stores information in its memory, it stores different types of information in different ways. These types are called *data types*. Later, it interprets the information according to its data type; in particular, it needs to know the data type of the information stored in a specific area of its memory, in order to interpret it correctly.

We now discuss the format in which a computer stores various data types. Keep in mind that the memory in actually a sequence of bits, or from a slightly different point of view, the memory is a sequence of words, each of which is a sequence of bits.

Date: Monday, September 7, 2021.

2. Integers

Consider a computer with word size w; there are w bits in each word.

2.1. **Unsigned integers.** A nonnegative integer is typically called an *unsigned integer* in computer jargon. One unsigned integer is stored in a single word. The bits of the word are interpreted as representing the integer in base two. In this way, a word can store any integer between 0 and $2^w - 1$.

Example 1. The word size is $w = 8$ bits, so a word can hold an integer between 0 and 255.

A given word contains the bits 10010011. Interpret this word as an unsigned integer.

Solution. Since $(10010011)_2 = 128 + 16 + 2 + 1 = 147$, this word is interpreted as the integer 147. □

If the computer adds to unsigned integers in two words and places the result in another word, it is possible that overflow may occur; that is, the result may be too large to fit in a single word. In this case, the highest order bit is dropped.

Suppose a and b are integer stored in words of length w bits. Since a word can only contain integers between 0 and $2^w - 1$, the sum $a + b$ is actually the remainder when $a + b$ is divided by 2^w.

Example 2. The word size is $w = 8$. Let a, b, and c represent words containing bits. Suppose $a = 11001100$ and $b = 01110111$, and the computer adds these and stores the result in c. Find c. Also find the decimal representation of a, b, and c.

Solution. The binary sum of a and b is 101000011, which is 9 bits long. The highest bit is dropped, so the computer stores $c = 01000011$. In decimal, a represents $128 + 64 + 8 + 4 = 204$, and b represents $64 + 32 + 16 + 4 + 2 + 1 = 119$. Note that $204 + 119 = 323$, which is greater than 255. Now c represents $64 + 2 + 1 = 67$, which is the remainder when 323 is divided by 256. □

Warning 1. The Java programming language does not support unsigned integers.

2.2. **Signed Integers.** A *signed integer* is a word which is interpreted an integer which may be positive or negative. The high bit is reserved to the sign, where 0 means nonnegative and 1 means negative. This cuts in have the number of positive integers which may be stored.

In a word of length w, a signed integer varies between -2^{w-1} and $2^{w-1} - 1$. For example, if $w = 8$, a word is capable of storing numbers between -128 and 127.

It is interesting to examine how a computer stores negative numbers. Now we realize that subtracting a number means adding its negative. Thus the computer stores a negative number in such a way that, when the numbers are added according to the addition algorithm already outlined, the desired result is obtained.

Computers store negative in a form known as *two's complement*. Since in a word of length w, 2^w is treated as zero (as described above), then $-a$ and $2^w - a$ will be treated identically. The two's complement of a with respect to word length w is $2^w - a$, expressed in binary.

Example 3. Let $w = 8$ and $a = 00110011$. Find the two's complement of a.

Solution. We have $(00110011)_2 = 32 + 16 + 2 + 1 = 51$, and $256 - 51 = 205$. Converting 205 to binary yields $(205)_{10} = (11001101)_2$.

Note that 11001101 may be obtained from 00110011 by flipping every bit of 00110011, and adding 1. □

A word of length w equals its highest possible value $(2^w - 1)$ if and only if every bit is equal to one. If a is a word and b is obtained from a by flipping every bit (changing zero to one and changing one to zero), then the sum of a and b has every bit set to one. So, it is $2^w - 1$. Adding one to this gives 2^w, which gives zero when stored in a word of length w.

In conclusion, in order to negate a word a, the computer flips every bit of a, and adds one, using binary addition. To subtract a from b, the computer constructs negative a, and adds b to this.

To negate a signed integer:

(a) Flip every bit
(b) Add 1

If the original number was negative, this procedure will cause an overflow bit; drop it. If the original number had a 1 as its least significant bit, it may be easier to reverse the order:

(a) Subtract 1.
(b) Flip every bit.

This actually produces the same result (try it).

Example 4. A computer stores -5 in a signed byte. Find the bits which are stored.

Solution. Since $(5)_{10} = (101)_2$, 5 is stored as 00000101. Flipping every bit gives 11111010. Adding one gives 11111011. The computer stores 11111011. □

Example 5. A byte contains the bits 10011001. Interpret this as a signed integer.

Solution. Since the high bit is set, the number is negative. We negate it, to find the positive integer of which it is the negative.

Subtracting one gives 10011000. Flipping every bit gives 01100111. Converting this to decimal gives $64 + 32 + 4 + 2 + 1 = 103$. So, the original stored integer was -103. □

Example 6. Let $a = (51)_{10}$ and $b = (100)_{10}$. Show how the computer of word length $w = 8$ would compute $b - a$.

Solution. Note that, in binary, we have $a = 00110011$ and $b = 01100100$. Now let $c = 11001101$; this is the two's complement of a, and is interpreted by the computer as negative a. Now $b - a = b + c$ internally. We have

$$(01100100)_2 + (11001101)_2 = (100110001)_2;$$

however, this result is nine bits long, and cannot fit in a word. The high bit is dropped, and the result is $b - a = 00110001$. Note that $(00110001)_2 = 32 + 16 + 1 = 49$; also, $100 - 51 = 49$. □

3. Floating Point Numbers

We think of rational numbers as a quotient of integers; it is possible to write a computer program to do arithmetic on rational numbers stored as a pair of integers with no common factors; however, typical computer hardware does not do this automatically.

However, most modern computers can, in their hardware, deal with a format known as *floating point*. This is very similar to what is commonly referred to as *scientific notation*.

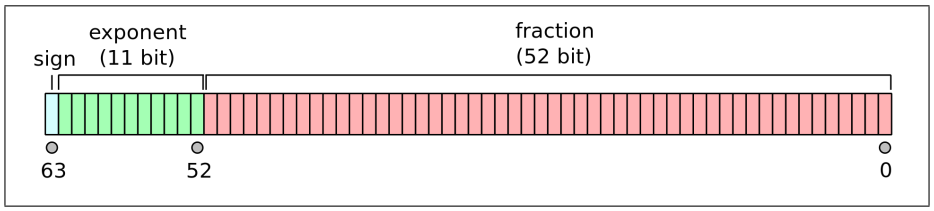
Consider a computer with word length w, or, a computer which can easily handle data of bit length w. To store a floating point number x, the w bits of the word are broken up into three parts;

A floating point number x is stored in three parts: the *sign* s, the *mantissa* f (also known as the *significand* or *fraction*), and the *exponent* e. The

$$x = s \cdot f \cdot 2^e \quad \text{where } s = \pm 1 \text{ and } 1 \leq f < 2.$$

Let us allocate w bits to store our floating point number; we call w the *precision* of the representation. We store x using the w bits broken into three groups. The sign requires 1 bit, the part representing the exponent requires p bits, and the part representing the mantissa requires q bits. Thus $w = 1 + p + q$. The most significant (leftmost) bit stores the sign; the next p bits store a representation of the exponent, and the least significant (rightmost) q bits store the mantissa.

A 64-bit precision number could be stored thusly:



We could store the exponent as a signed integer e, and the mantissa as an unsigned integer v interpreted as $f = \dfrac{v}{2^{q-1}}$. In practice, it is possible to play a couple tricks to improve the amount of information which is stored in a fixed number of bits, and to allow us to store things like $\pm\infty$ and `NaN` ("not a number").

The first trick is to store the exponent as an unsigned integer u, with a fixed (predetermined) *exponent bias offset* o subtracted from it. Thus $e = u - o$. For example, an 8-bit exponent would store numbers from 0 to 255; an bias of $o = 127$ would be subtracted from it, for a range of -126 to 127. For an x bit exponent, the bias would typically be $o = 2^{x-1} - 1$.

The second trick is to avoid storing the leading 1 in the mantissa. To understand this, note that in decimal scientific notation, we would write 4.21×10^3 and not 0.421×10^4 or 42.1×10^2; we always write exactly one nonzero digit. In binary, however, the only nonzero digit is 1, so we can simply assume the 1 exists. So if the mantissa contains the bits `01011`, the actual value we interpret is $(101011)_2 = (43)_{10}$. Thus we store the mantissa as an unsigned integer v, where $f = 1 + \dfrac{v}{2^q}$.

Lastly, the sign $s = \pm 1$ is stored as one bit, say b, where $b = 0$ or $b = 1$, and $s = (-1)^b$.

We summarize the variables we have used to describe floating point representations.

- x is the number to be stored.
- $s = \pm 1$ is the sign, f is the fractional mantissa, e is the exponent, and $x = s \cdot f \cdot 2^e$.
- w is the precision, 1 is the number of sign bits, p is the number of exponent bits, and q is the number of mantissa bits.
- b is the sign bit, u is the unsigned integer give by the p exponent bits, v is the unsigned integer given by the q mantissa bits, and o is the bias.
- Thus $s = (-1)^b$, $e = u - o$ and $f = 1 + \dfrac{v}{2^q}$.

The Institute for Electrical and Electronics Engineers (IEEE) floating point standard IEEE 754 specifies the preferred values for p, q, and o, as indicated in the following chart. Quarter precision is not actually in the standard.

The following table lists the bit lengths and the biases for the various precisions for a number x.

Type	Sign	Exponent p	Mantissa q	Total bits w	Bias o
Quarter Precision*	1	3	4	8	3
Half Precision	1	5	10	16	15
Single Precision	1	8	23	32	1023
Double Precision	1	11	52	64	16383

Example 7. A byte holds `01011101`. Interpret this as a number according to the scheme outlined above.

Solution. A floating point number is stored in $w = 8$ bits. The sign requires 1 bit, the exponent uses $p = 3$ bits, and the mantissa uses $q = 4$ bits. The exponent bias is $o = 3$.

We break the byte into three parts:

$$\texttt{01010101} \quad \Rightarrow \quad \texttt{0 101 1101}.$$

Each part is interpreted as an unsigned integer.

The first part is the sign bit `0`, which means $b = 0$, so $s = 1$ (the number is positive).

The second part is the exponent, and its three bits `101` represent the unbiased exponent $u = 5$, from which we subtract the bias $o = 3$ to get $e = 2$.

The third part is the mantissa, and its four bits `1101` imply that $v = 13$, so should be viewed as

$$f = 1 + \frac{13}{2^4} = (1.1101)_2 = 2 + \frac{1}{2} + \frac{1}{4} + \frac{1}{16} = \frac{45}{16}.$$

Thus the number stored is

$$s \cdot f \cdot 2^e = 1 \times \frac{45}{16} \times 2^2 = \frac{45}{4} = (11.25)_{10}.$$

□

The observant reader will have realized at this point that our model has some duplication and some ambiguities. Firstly, we note that because of the "hidden bit", the mantissa cannot be zero, so we need to designate how zero is stored.

Also, $1^e = 1$ for all e, so there exist some duplicate representations unless we state otherwise; in particular, if $v = 0$, any value for u will produce ± 1.

Moreover, $0^e = 0$ (unless $e = 0$) and $f^0 = 1$ (unless $f = 0$), so we need to at least consider those potential ambiguities.

These difficulties are managed by treating $u = 0$ (all exponent bits off) and $u = 2^p - 1$ (all exponent bits on) as special cases. That is, for these two values of u, the meaning is not $x = s \times f \times 2^e$, but rather, is specified separately as follows.

- **Zero:** Zero is stored with all bits off. So $b = 0$, $u = 0$, and $v = 0$, which implies $s = 1$, $f = 1$, and $e = 0$. Note that in this case $x \neq 1$.
- **Negative Zero:** If $b = 1$, $u = 0$, and $v = 0$, we interpret the value as "negative zero". This is normally treated as zero, although whereas dividing a positive number by zero gives infinity, dividing a positive number by negative zero yields negative infinity.
- **Subnormalized Numbers:** If $u = 0$ and $v \neq 0$, the number is interpreted as a subnormal number. That is, in this case, the hidden bit is not assumed to be 1 but rather zero. This allows for storing numbers less than the smallest normalized number.
- **Infinities:** Plus or minus infinity is stored with $v = 0$ and $u = 2^p - 1$ (all ones). Then the high bit determines the sign of infinity.
- **NaN's:** If $u = 2^p - 1$ (all bits on) and $u \neq 0$, we have "Not a Number".

4. Text

Computers store text as a sequence of characters. Characters are represented in the computer as numbers. There are various standard way of representing characters as numbers; the standards which is most commonly used today are ASCII and its extension, Unicode.

4.1. **ASCII.** The acronym ASCII stands for American Standard Code for Information Interchange. This standard uses seven bits; it is a mapping of numbers between 0 and 127 to how they should be interpreted by an output device.

The characters are grouped into four types:

- Alphabetic: upper and lower case letters
- Numeric: the digits 0 through 9
- Punctuation: other printable characters
- Control: nonprinting characters, used to control devices

The following table lists the standard; control characters are in angle brackets. We use `<sp>` to indicate a space, which is considered a printable character.

0	<NUL>	16	<SO>	32	<sp>	48	0	64	@	80	P	96	‘	112	p
1	<SOH>	17	<XON>	33	!	49	1	65	A	81	Q	97	a	113	q
2	<STX>	18	<DC2>	34	"	50	2	66	B	82	R	98	b	114	r
3	<ETX>	19	<XOF>	35	#	51	3	67	C	83	S	99	c	115	s
4	<EOT>	20	<DC4>	36	$	52	4	68	D	84	T	100	d	116	t
5	<ENQ>	21	<NAK>	37	%	53	5	69	E	85	U	101	e	117	u
6	<ACK>	22	<SYN>	38	&	54	6	70	F	86	V	102	f	118	v
7	<BEL>	23	<ETB>	39	’	55	7	71	G	87	W	103	g	119	w
8	<BS>	24	<CAN>	40	(	56	8	72	H	88	X	104	h	120	x
9	<TAB>	25	<EM>	41	)	57	9	73	I	89	Y	105	i	121	y
10	<LF>	26	<SUB>	42	*	58	:	74	J	90	Z	106	j	122	z
11	<VT>	27	<ESC>	43	+	59	;	75	K	91	[	107	k	123	{
12	<FF>	28	<FS>	44	,	60	<	76	L	92	\	108	l	124	|
13	<CR>	29	<GS>	45	-	61	=	77	M	93	]	109	m	125	}
14	<SO>	30	<RS>	46	.	62	>	78	N	94	^	110	n	126	~
15	<SI>	31	<US>	47	/	63	?	79	O	95	_	111	o	127	<RUB>

4.2. **Printable characters.** Alphabetic, numeric, and punctuation characters are called *printable characters*. We list these.

```
Alphabetic characters:      ABCDEFGHIJKLMNOPQRSTUVWXYZ
                            abcdefghijklmnopqrstuvwxyz

Numeric characters:         0123456789

Punctuation:                <sp> !"#$%&’()*+,-./:;<=>?@[\]^_‘{|}~
```

4.3. **Control characters.** Control characters are used to control the behavior of peripheral devices, or may be sent by peripheral devices to the computer as signals.

Certain control characters are used very frequently, and in a more or less standard way. We list some of these.

```
  0 <NUL> Null              Used as a string terminator in the
                            C programming language
  3 <ETX> End Transmission  Breaks the program execution
  7 <BEL> Bell              Sounds a bell or beep
  8 <BS>  Backspace         Delete previous character
  9 <TAB> Tab               Separates columns;
                            sometimes used as a field separator
 10 <LF>  Line Feed         Move the cursor down one line;
                            used as a line separator in UNIX
 13 <CR>  Carriage Return   Put the cursor at the beginning
                            of the current line;
                            output of the "Enter" key
 17 <XOF> X-off             Resume transmission
 19 <XON> X-on              Pause transmission
 27 <ESC> Escape            Exits some programs
127 <RUB> Rubout            Delete current character;
                            sometimes delete previous character
```

Pressing the control key together with another key toggles bit 6; since $2^6 = 64$, this has the effect of subtracting 64 from the ASCII value of a letter. Thus CTRL-C often stops the program execution, CTRL-S pauses transmission, and CTRL-Q resumes transmission. Also, CTRL-G is a beep.

4.4. **Strings.** A *string* is a finite sequence of characters; usually the word string refers to a sequence of characters in memory, without any carriage returns. It is one word, or name, or description. In most programming languages, strings are surrounded by quotation marks when they are referred to.

To reference a string, the computer needs to know where the string starts (its beginning position) and where it stops (its ending position). There are two standard ways of doing this, which we refer to as the Pascal and C conventions.

- Pascal: the first byte holds the length of the string, so only strings containing up to 255 characters can be stored;
- C: the first byte holds the first character; the last character in the string is followed by a zero byte, call the string terminator.

For example, consider the string `"This is a test"`. The length of this string is 14 (we include the spaces but not the quotation marks as characters which we count). Pascal and C each allocate 15 bytes to store this string.

Pascal stores the number 14 followed by the numbers which are the ASCII values of the characters:

```
  14  84  104 105 115 32  105 115 32  97  32  116 101 115 116
[len] T   h   i   s  <sp> i   s  <sp> a  <sp> t   e   s   t
```

C stores the ASCII values of the characters, followed by a byte containing zero.

```
  84  104 105 115 32  105 115 32  97  32  116 101 115 116 0
  T   h   i   s  <sp> i   s  <sp> a  <sp> t   e   s   t   [end]
```

4.5. **Text files.** Text files contain sequences of bytes containing numbers, interpreted as characters via ASCII. Lines in a text file as separated according to two different standards:

- UNIX: lines are separated by a single `<LF>` (ASCII 10).
- DOS: lines are separated by a `<CR><LF>` (ASCII 13 followed by ASCII 10).

This difference is nightmarish for programmers, and is another example of the horrible mistakes made by Microsoft (UNIX preceded DOS by more than a decade).

5. Images

We describe a simplified version of how a computer may store an image.

The atomic unit of storage is a *pixel*. This word may refer to a physical attribute of the computer monitor, or to a logical attribute of the storage protocol. It is the latter meaning which we use.

An image is stored as a grid, with x pixels across and y pixels down. Each pixel is a position in this grid, and represents a specific color.

The *resolution* of the image refers to $x \times y$.

One pixel represents a specific color. This is stored as a number, so the entire image requires xy numbers.

The *depth* of the color refers to the number of bits used to represent a color. In *indexed* mode, the number specifying the color is used as an index to a *palette*. In *direct* mode, the bits of the number are broken up into three blocks, which specify the intensity of the primary colors red, green, and blue. The combined intensities of these colors is often referred to as *RGB*.

We describe the evolution of IBM PC compatible graphics adaptors.

- **CGA** resolution 320×200, choice of two palettes with 4 colors each, and a choice of low or high intensity for each color.
- **EGA** resolution 640×350, with a sixteen color palette. The sixteen colors of the palette are selected from a choice of 64 colors ($64 = 2^6$, with 2 bits for each primary color intensity)
- **VGA** resolution 640×400, with a sixty four color palette, selected from a possible 262144 colors ($262144 = 2^{18}$, with 6 bits per primary color)
- **SVGA** Super VGA allows for variations. It is now common to use 8 bits per primary color, for 256 possible intensities per color, giving $2^{24} = 16777216$ possible colors.

6. Sound

Sound is the variation in air pressure. Thus, sound can be modeled as a single real valued function of time. Such a function is represented on a computer breaking up time into small time units of equal size, and recording the air pressure at that moment. This is then stored as a sequence of numbers, each representing the air pressure.

7. Appendix

We list some of the history of word sizes. Note that IBM microcomputers and compatibles used Intel chips, whereas Macintosh used Motorola chips.

- DEC PDP-8 used a twelve bit word ($2^{12} = 4096$). Introduced in 1965, this was the first successful minicomputer, and was built with integrated circuits.
- Intel 4004 was a four bit microprocessor ($2^4 = 16$). Introduced in 1971, this was the first single chip microprocessor.
- Intel 8008 was an eight bit microprocessor ($2^8 = 256$). Introduced in 1974.
- Intel 8088 was a sixteen bit microprocessor ($2^{16} = 65536$) with an eight bit bus. Introduced in 1979 and used in the first IBM PC.
- Intel 8086 was a sixteen bit microprocessor with a sixteen bit bus. Introduced in 1978.
- Intel 80286 was sixteen bits. Introduced in 1982 and used the IBM AT.
- Intel 80386 was 32 bits ($2^{32} = 4294967296$). Introduced in 1985 and used in the high end IBM PS/2. DOS still used 16-bit compatibility mode. Later, Microsoft developed the Windows 95 operating system, which used the 32-bit instruction set.
- Intel 80486 was 32 bits. Introduced in 1989.
- Intel Pentium was 32 bits with a 64 bit bus. Introduced in 1993.
- Intel Pentium II was 32 bits. Introduced in 1997.
- Intel Pentium III was 32 bits. Introduced in 1999.
- Intel Pentium 4 was 32 bits. Introduced in 2000.
- Intel Itanium was 64 bits ($2^{64} \approx 1.84 \times 10^{19}$). Introduced in 2001 with a completely new (not backwards compatible) instruction set.

8. Problems

Problem 1. An eight bit computer adds the unsigned integers 200 and 100, and stores the result in a byte. Find the bits that are stored in the byte, and find the value of the byte as an unsigned integer.

Problem 2. An eight bit computer stores -100 as a signed integer in a byte. Find the bits that are stored in the byte.

Problem 3. A byte holds the bits `11001100`.

(a) Interpret this byte as an unsigned integer.
(b) Interpret this byte as a signed integer.
(c) Interpret this byte as a floating point number as in Example **??**.

Problem 4. A program stores the string "`Sept 11, 2001`" in memory using the Pascal convention. Determine the sequence of numbers (expressed in decimal) which is stored.

Problem 5. A program finds the sequence

```
79 83 65 77 65 32 57 47 49 49 0
```

in memory, and interprets it as a string stored with the C convention. Find the string.

Department of Mathematics, Paragon Science Academy
Email address: paul.bailey@basised.com