Data Representations

- We know that a computer's CPU and main memory can only store binary bits.
  - How are individual characters and character strings stored in a computer?
  - How are integers stored in a computer?
  - How are negative numbers represented?
  - How are floating point numbers, such as 2.75, represented in a computer?

Characters

- We need to distinguish between:
  - The concept of a character.
  - How a character looks on a screen or page.
- The concept of "upper case A" can have many appearances depending on the font used to display the character.
  - A (Arial) A (Times New Roman)

    A (Comic Sans MS) A (Brush Script MT)
- The concept of "upper case A" is different from the concept of "numeral 4".

Characters (2)

- Each character (concept of a character) is represented by a sequence of ones and zeros.
  - Rather than writing the individual ones and zeros we will group the bits into groups of 8 bits (called a byte) or groups of bytes (called words).
  - Each byte will be given in hexadecimal notation.
    - 0x5E is the bit sequence 01011110
- There are several standards for assigning a code to each character.

Characters (3)

- The 2 main standards are:
  - ASCII (American Standard Code for Information Interchange)
  - Unicode
    - An extension of ASCII to allow for different “languages”.
- IBM has its own “standard” called EBCDIC.

ASCII

USASCII code chart

b_7 b_6 b_5 →					0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
b_4	b_3	b_2	b_1	Row ↓ / Column →	0	1	2	3	4	5	6	7
0	0	0	0	0	NUL	DLE	SP	0	@	P	`	p
0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0	0	1	0	2	STX	DC2	"	2	B	R	b	r
0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
0	1	0	0	4	EOT	DC4	$	4	D	T	d	t
0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
1	0	0	0	8	BS	CAN	(	8	H	X	h	x
1	0	0	1	9	HT	EM	)	9	I	Y	i	y
1	0	1	0	10	LF	SUB	*	:	J	Z	j	z
1	0	1	1	11	VT	ESC	+	;	K	[	k	{
1	1	0	0	12	FF	FS	,	<	L	\	l	|
1	1	0	1	13	CR	GS	–	=	M	]	m	}
1	1	1	0	14	SO	RS	.	>	N	^	n	~
1	1	1	1	15	SI	US	/	?	O	_	o	DEL

Extending ASCII

- While ASCII worked great for English and simple “typewriter” systems, it could not cope with accented characters very well.
  - â could be made by an “a” character followed by a backspace (BS) character and then a circumflex (“^”) character.
- Screen-based displays enabled simple graphics to be constructed from block graphics characters.

Extended ASCII Table

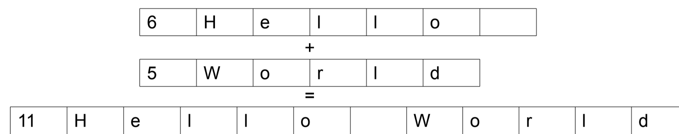
Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
128	80	Ç	160	A0	á	192	C0	└	224	E0	α
129	81	ü	161	A1	í	193	C1	┴	225	E1	ß
130	82	é	162	A2	ó	194	C2	┬	226	E2	Γ
131	83	â	163	A3	ú	195	C3	├	227	E3	π
132	84	ä	164	A4	ñ	196	C4	─	228	E4	Σ
133	85	à	165	A5	Ñ	197	C5	┼	229	E5	σ
134	86	å	166	A6	ª	198	C6	╞	230	E6	µ
135	87	ç	167	A7	º	199	C7	╟	231	E7	τ
136	88	ê	168	A8	¿	200	C8	╚	232	E8	Φ
137	89	ë	169	A9	⌐	201	C9	╔	233	E9	Θ
138	8A	è	170	AA	¬	202	CA	╩	234	EA	Ω
139	8B	ï	171	AB	½	203	CB	╦	235	EB	δ
140	8C	î	172	AC	¼	204	CC	╠	236	EC	∞
141	8D	ì	173	AD	¡	205	CD	═	237	ED	φ
142	8E	Ä	174	AE	«	206	CE	╬	238	EE	ε
143	8F	Å	175	AF	»	207	CF	╧	239	EF	∩
144	90	É	176	B0	░	208	D0	╨	240	F0	≡
145	91	æ	177	B1	▒	209	D1	╤	241	F1	±
146	92	Æ	178	B2	▓	210	D2	╥	242	F2	≥
147	93	ô	179	B3	│	211	D3	╙	243	F3	≤
148	94	ö	180	B4	┤	212	D4	╘	244	F4	⌠
149	95	ò	181	B5	╡	213	D5	╒	245	F5	⌡
150	96	û	182	B6	╢	214	D6	╓	246	F6	÷
151	97	ù	183	B7	╖	215	D7	╫	247	F7	≈
152	98	ÿ	184	B8	╕	216	D8	╪	248	F8	°
153	99	Ö	185	B9	╣	217	D9	┘	249	F9	∙
154	9A	Ü	186	BA	║	218	DA	┌	250	FA	·
155	9B	¢	187	BB	╗	219	DB	█	251	FB	√
156	9C	£	188	BC	╝	220	DC	▄	252	FC	ⁿ
157	9D	¥	189	BD	╜	221	DD	▌	253	FD	²
158	9E	₧	190	BE	╛	222	DE	▐	254	FE	■
159	9F	ƒ	191	BF	┐	223	DF	▀	255	FF	

Unicode

- Even with the extended ASCII table, the table is very oriented towards Latin based alphabets.
- As computers became more prevalent across the world it became necessary to accommodate other alphabets.
  - Cyrillic
  - Kanji and Hiragana
  - Arabic
- Unicode uses 16-bit codes.
  - 256 tables of 256 characters each.
  - ASCII is bottom half of the first table.

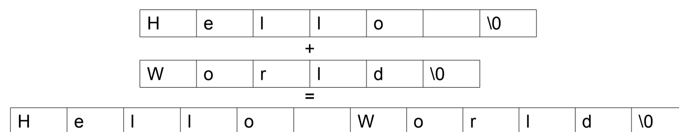
Strings

- Strings are sequences of characters.
- Fixed length strings
  - Older programming languages, e.g. Pascal, used fixed length strings.
  - The first byte gave the length of the string (number of characters in the string) and the remaining bytes were the characters in the string.



Strings (2)

- Variable length string
  - These are much more flexible than fixed length strings.
  - Variable length strings are terminated by a NULL character (0x00 or \0).
  - Concatenating strings (joining strings together is a little more complicated.



Integers

- A single byte can represent 256 different values, 0 up to 255.
- For larger numbers we need to use multiple bytes.
  - A 2 byte word can represent 2^{16}=65,536 values, 0 up to 65535.
  - A 4 byte word can represent 2^{32}=4,294,967,296 values.
  - No matter how many bytes you add there is always a limit.

2's Complement Numbers

- Lets us limit ourselves to bytes for the moment.
- There are 2 ways to implement negative numbers.
  - Use a sign bit.
    - If the most significant bit is 1 then the number is negative, and if it is 0 then it is positive.
  - Though simple it has a few problems.
    - There are two ways to represent zero.
      - 0x00 = 00000000 = +0
      - 0x80 = 10000000 = -0

2's Complement Numbers (2)

- Arithmetic manipulation is messy
  - Try adding 1 to -1 and you get:
  - 0x81 + 0x01 = 0x82 = -2 ??????
  - You would need different hardware for adding two positive numbers and a positive number and a negative number.

- 2's complement
  - To convert a positive number into a negative number subtract it from 0x00.

0000 0010 = 0x02	2
0000 0001 = 0x01	1
0000 0000 = 0x00	0
1111 1111 = 0xFF	-1
1111 1110 = 0xFE	-2
1111 1101 = 0xFD	-3

2's Complement Numbers (3)

- Now adding positive and negative numbers does not cause any complications.
  - -1 + 2 = 0xFF + 0x02 = 0x01 = 1
  - And we did not have to even involve a carry bit!
- Multiplication by 2 still works as a shift left operation.
  - 4 x 2 = 0x04 << 1 = 0000 0100 << 1 = 0000 1000 = 0x08 = 8
  - -2 x 2 = 0xFE << 1 = 1111 1110 << 1 = 1111 1100 = 0xFC = -4

2's Complement Numbers (4)

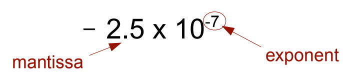
- Division by 2 is still a shift right but with a slight twist.
  - 4 / 2 = 0x02 >> 1 = 0000 0100 >> 1 = 0000 0010 = 0x02 = 2
  - -4 / 2 = 0xFC >> 1 = 1111 1100 >> 1 = 1111 1110 = 0xFE = -2
- If dividing a positive number by 2 we need to shift in a 0 to the most significant bit.
- If dividing a negative number by 2 we need to shift in a 1 to the most significant bit.
- Most CPUs have shift instructions that shift the carry bit into the register being shifted. The carry bit can be set depending on whether the register contains a negative 2's complement number (as determined by a bit in the status register.

Floating Point Numbers

- If we only have binary numbers, how can we represent numbers such as:
  - 0.00000025
  - 1,000,000,000,000,000
  - -127.5
  - -0.075
- First let us think back to how we humans have learnt to deal with very big and very small numbers.

Scientific Notation

- Using scientific notation we can reduce very large numbers and very small numbers into manageable entities.
  - $0.00000025 = 2.5 \times 10^{-7}$
  - $1{,}000{,}000{,}000{,}000{,}000 = 1.0 \times 10^{15}$
  - $-127.5 = -1.275 \times 10^{2}$
  - $-0.075 = -7.5 \times 10^{-2}$
- Each part of the scientific notation number has a specific name.
  - 2.5×10^{-7}



Scientific Notation in Binary

- The standard for scientific notation in binary is the IEEE 754 standard.
- First what does the binary number 0.1001 represent?
  - $(0 \times 2^{0}) + (1 \times 2^{-1}) + (0 \times 2^{-2}) + (0 \times 2^{-3}) + (1 \times 2^{-4})$
  - $0 + (1 \times 0.5) + (0 \times 0.25) + (0 \times 0.125)$
    $+ (1 \times 0.0625)$
  - 0.5625

Scientific Notation in Binary (2)

- Now let us try 1234.5625
- Convert the part to the left of the decimal point into binary and the part to the right of the decimal point into binary.
  - 1234.5625 = 10011010010.1001
- Now normalise the binary number
  - $1.00110100101001 \times 2^{1010}$

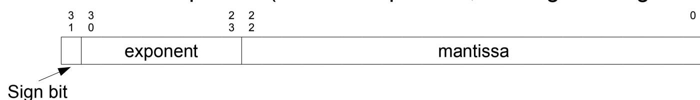
mantissa

exponent

- A nice feature of normalised binary numbers is that the digit to the left of the binary point is always a 1!

Scientific Notation in Binary (3)

- IEEE 754 covers 32-bit, 64-bit and 128-bit floating point numbers.
- A 32-bit IEEE 754 number is calculated and stored in memory as follows.
  - Convert integer part into binary.
  - Convert fractional part into binary
  - Normalise, drop leading 1 and add sufficient 0's to the start to get a 23-bit mantissa
  - Add 127 to the number of places shifted to get an 8-bit exponent (shift left is positive, shift right is negative).



Scientific Notation in Binary (4)

- So 1234.5625 as a 32-bit IEEE 754 number is:
  - 1234.5625 = 10011010010.1001
  - 1.00110100101001 x 2^{1010}
  - Sign bit = 0
  - Mantissa = 00000000000110100101001
  - Exponent = 10 + 127 = 137 = 0x89 = 10001001

31	30							23	22																						0
0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	0	0	1	0	1	0	0	1