Representing Characters and Numbers in a Computer – Part 1

Introduction to Computer Science

Module Code: 4CC509

Overview

- Last Thursday we looked at number systems and looked at binary, octal and hexadecimal
- In this lecture, we will start to look at how characters and numbers are stored in a computer using binary.
- We will continue this in the next lecture as well.

Storing Characters

- A binary number is used to represent each character
- The first major portable standard was ASCII which uses 7 bits to represent each character
- The full set of characters can be seen at http://www.asciitable.com/
- Later evolved into 8 bit version called Latin-1 Extended ASCII character set. Because it allows for 256 characters, it included accented characters and other special symbols (for example, the £ sign is not in the original ASCII character set).

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

EBCDIC

- Another character encoding that is still used today on IBM mainframes is EBCDIC.
- http://www.astrodigital.org/digital/ebcdic.html
- Not so widely adopted since the letters are not contiguous – makes it awkward to write sorting programs, etc
- However, it is worthwhile knowing it exists in case you ever need to convert mainframe data or handle the translation of data that is being transferred from a PC to a mainframe (which does happen!)

Unicode

- 8 bits is not enough to handle the characters for some languages. For example, Japanese requires many more than 8 bits to represent the characters
- This meant that different encoding schemes were introduced that used 16 or more bits e.g. Shift-JIS, EUC, …
- The goal of Unicode was to have one character set that could represent every character in every language

Unicode

- Unicode uses 16 bits for each character
- The first 256 characters map on to the extended ASCII character set
- For lists of Unicode code charts, look at:
  - http://www.unicode.org/charts/
- For a full index of character names in alphabetic order, look at:
  - http://www.unicode.org/charts/charindex.html

Storing Numbers

- We have seen that a computer stores numbers in binary.
- However, if we look at a particular 4 bytes in memory, what number does the value stored in that 4 bytes actually represent?
- The answer is that it depends on the *type* of number stored at that location

Example

- A particular 4 bytes of memory contain the following values:

1000000 1	1011000 0	0000000 0	0000000 0

- If we treat the contents of these 4 bytes (i.e. 32 bits) as one number, what number is it?

Unsigned Integers

- Usually stored in 8, 16, 32 or 64 bits.
- A 8 bit location can hold values in the range 0 to 255 (2^8 -1)
- A 16 bit location can hold values in the range 0 to 65,535 (2^{16} – 1)
- A 32 bit location can hold values in the range 0 to 4,294,967,295 (2^{32} -1)
- A 64 bit location can hold values in the range 0 to 18,446,744,073,709,551,615 (2^{64} -1)

Our Example

- If we consider our 4 bytes of memory as an unsigned integer:

10000001	10110000	00000000	00000000

- will represent the unsigned integer 2175795200 ($2^{31} + 2^{24} + 2^{23} + 2^{21} + 2^{20}$)

Introduction to Computer Science 4CC509

Signed Integers

- If we want to represent signed integers, we need to have a way of representing the sign of the number, i.e. is it positive or negative.
- We could simply allocate the first bit in the location to represent the sign. For example, in an 8 bit location we could use the first bit to indicate whether the number is positive or negative and the other 7 bits to represent the number part

Signed Integers

- For example, in one byte (8 bits) -7 could be represented as follows:
  10000111
- +15 would be represented as 00001111
- This byte could now store numbers in the range -127 to +127

What’s the Problem with this?

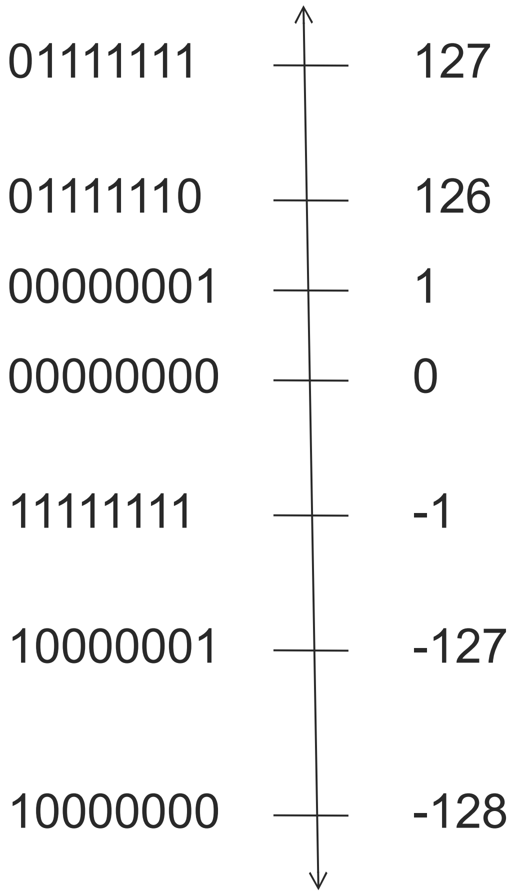
- What does 00000000 represent?
- What does 10000000 represent?
- You end up with two values of 0 which causes real problems.
- Not suited to arithmetic

Signed Integers

- As a solution to this, we shift the range of numbers represented by a certain number of bits so that half of the integers are negative numbers.
- For example, if we have 8 bits, instead of them representing the range 0 to 255, they are used to represent the integers in the range -128 to +127.

Introduction to Computer Science 4CC509

Storing Signed Numbers



Storing Negative Numbers

- This representation of negative numbers is called the *Two's Complement*
- When storing signed integers, an integer that begins with a 1 is a negative number.
- To determine the representation of a negative number, take the positive number, flip the bits so that a 1 becomes a 0 and vice versa and then add 1 to the result (discarding any overflow from the left)
- Exactly the same algorithm can be used to convert a negative number to a positive number.

Two's Complement Example

Original number = 1	00000001
Flip all of the bits	11111110
Add 1 to the flipped value	1
Answer = -1	11111111

Add together

Two's Complement Example

Original number = 0	00000000
Flip all of the bits	11111111
Add 1 to the flipped value	1
Answer = 0	00000000

Add together

Introduction to Computer Science 4CC509

Two’s Complement Example

Original number = -5	11111011
Flip all of the bits	00000100
Add 1 to the flipped value	1
Answer = 5	00000101

Add together

Two’s Complement

- When converting numbers, make sure you round up the number to the correct number of bits by putting 0s at the beginning before attempting the conversion.
- For example, if you are storing numbers as 16 bit values, you need to make sure all of your numbers are 16 bits.

Two’s Complement Example using 16-bit Values

Original number = 62	0000000000111110
Flip all of the bits	1111111111000001
Add 1 to the flipped value	1
Answer = -62	1111111111000010

Add together

Our Example

- If we consider our 4 bytes of memory as a signed integer:

10000001	10110000	00000000	00000000

- This represents the signed integer -2119172096

Storing Decimal Numbers

- When storing real numbers, we need to represent the parts of the number that are before and after the decimal point

Decimal Numbers

10^2	10^1	10^0		10^{-1}	10^{-2}	10^{-3}
100	10	1		$^1/_{10}$	$^1/_{100}$	$^1/_{1000}$
4	2	6	.	5	2	7

Binary Decimal Numbers

2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}
8	4	2	1	$^1/_2$	$^1/_4$	$^1/_8$
1	0	0	1 .	1	0	1

$$8 + 0 + 0 + 1 + .5 + 0 + .125$$

$$= 9.625$$

Introduction to Computer Science 4CC509

Storing Decimal Numbers

- Fixed Point
  - A set number of bits are used to store each of the parts of the number before and after the decimal point – the number is stored as two integer values.
- Floating Point
  - The number is stored as a value (the mantissa) raised to the power of an exponent

Introduction to Computer Science 4CC509

Fixed-Point Numbers

- No standard for storing fixed-point numbers
- Usually represented as a structure that contains one integer (two’s complement) to represent the part of the number before the decimal point and another integer to represent the part of the number after the decimal point.

Fixed-Point Numbers

- For example, we might use a 32-bit value to store a fixed point number using 16 bits to represent the part of the number before the decimal point and the other 16-bits to represent the part of the number after the decimal point.

Our Example

- If we consider our 4 bytes of memory as a fixed-point decimal number:

10000001	10110000	00000000	00000000

Before decimal point | After decimal point

- We just treat each 16-bit part as an integer and place a decimal point between them
- This represents the decimal number -32336.0

Floating Point

- We will leave that until next time.