Lancaster University

SCC131: Digital Systems

Topic 2: Information coding 1

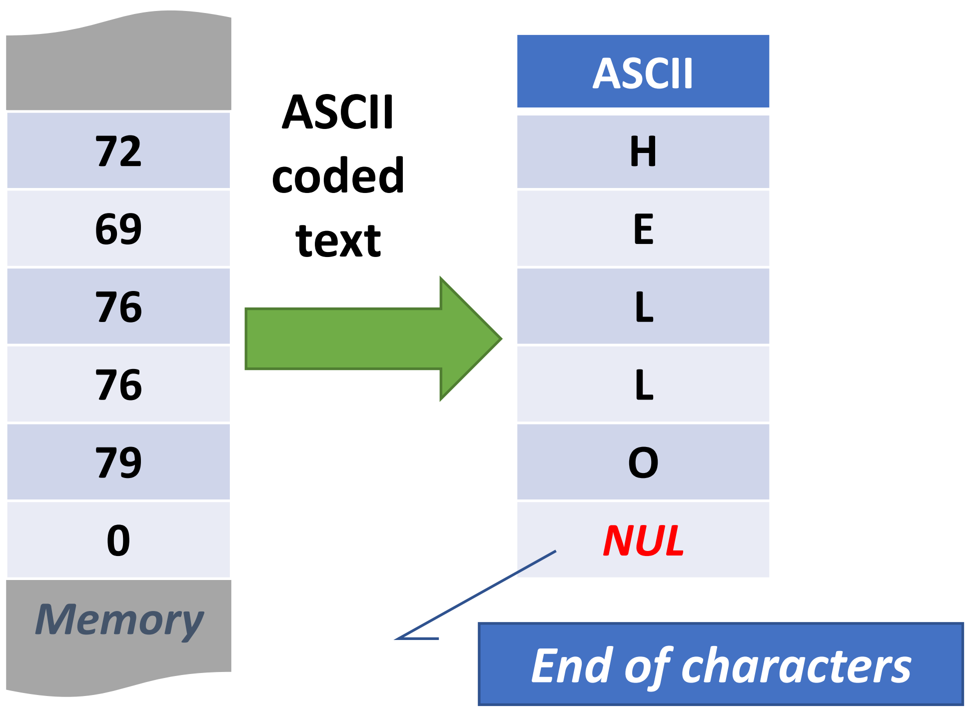
Why information coding is needed: The need for *simplicity*

- We need computer hardware to be as *simple* as possible (because of cost, performance, scalability, …)
  - So, we focus the computer hardware on handling **small, positive whole numbers**
  - Any other data type is just a **code (**or **“**representation”, or “interpretation”) that ***maps*** to positive whole numbers

For example, for a text character (such as A, B, C, D, …), we need to ***define*** what a specific number that a text should be mapped to (i.e. coding it) -> see the example in the next slides

ASCII is an example of such a code

- The American Standard Code for Information Interchange (ASCII) is a widely-used code for characters
- It defines 128 (i.e. 2^7) symbols (7-bit binary code)
  - e.g., the letters A,B,…,H,…Z are represented as 65,66,…,72,…90



From ASCII Code Chart

(e.g. 'A' is 100 0001$_{binary}$ = 65$_{decimal}$)

Lancaster University

Symbol	Decimal	Binary
A	65	01000001
B	66	01000010
C	67	01000011
D	68	01000100
E	69	01000101
F	70	01000110
G	71	01000111
H	72	01001000
I	73	01001001
J	74	01001010
K	75	01001011
L	76	01001100
M	77	01001101
N	78	01001110
O	79	01001111
P	80	01010000
Q	81	01010001
R	82	01010010
S	83	01010011
T	84	01010100
U	85	01010101
V	86	01010110
W	87	01010111
X	88	01011000
Y	89	01011001
Z	90	01011010

Symbol	Decimal	Binary
a	97	01100001
b	98	01100010
c	99	01100011
d	100	01100100
e	101	01100101
f	102	01100110
g	103	01100111
h	104	01101000
i	105	01101001
j	106	01101010
k	107	01101011
l	108	01101100
m	109	01101101
n	110	01101110
o	111	01101111
p	112	01110000
q	113	01110001
r	114	01110010
s	115	01110011
t	116	01110100
u	117	01110101
v	118	01110110
w	119	01110111
x	120	01111000
y	121	01111001
z	122	01111010

How to represent numbers?

Lancaster University

- In practice, we can have numbers such as negative numbers, fractions, floating-point numbers ...
- All need to map to **small, positive whole numbers** held in memory
- Let’s see how we do these in the following slides: ->

Decimal: a starting point

Lancaster University

- Early computers used decimal, “just like us”
- But what *is* decimal (base-10)? Essentially, it is columns of powers-of-10 (10^x)

Most Significant digit — ***Least Significant*** digit

Thousands	Hundreds	Tens	Units
10^3	10^2	10^1	10^0
1	*9*	*8*	*4*

We can view it like this...

1984

Thousands	Hundreds	Tens	Units
1×10^3	9×10^2	8×10^1	4×10^0
1×1000	9×100	8×10	4×1

$$1000 + 900 + 80 + 4 = 1984$$

Multiplication

Lancaster University

1984 x 10 = 19840

Multiply by 10 → ***Shift left*** *by one decimal place*
(feeding in a 0 at the right)...

10 thousands	Thousands	Hundreds	Tens	Units
0 x 10^4	*1* x 10^3	*9* x 10^2	*8* x 10^1	*4* x 10^0
0 x 10000	*1* x 1000	*9* x 100	*8* x 10	*4* x 1

$$\mathbf{(\underline{1}000 + \underline{9}00 + \underline{8}0 + \underline{4}) \times 10 \quad = \quad 10000 + 9000 + 800 + 40 + 0 \quad = \quad 19840}$$

Multiplying by the radix

- So…
  - To multiply by 10, we shift left by one place, and…
  - To divide by 10,   we shift right by one place
- Note the general principle here: to multiply or divide by radix^n, we shift n places:
  - For positive n (*multiplication*), we shift left by n places
  - For negative n (*division*), we shift right by n places

Thousands	Hundreds	Tens	Units
1	9	8	4

Infinite and finite precision

Lancaster University

- Given any two integers *i* and *j*, we normally (i.e., in everyday life) assume that *i + j*, *i - j*, and *i x j* will always generate a valid (integer) result

…the underlying assumption is that the range of integers is ***infinite***

- But what if we're restricted to a **finite number of digits**, as is usually the case in computer systems?

Thousands	Hundreds	Tens	Units
1	9	8	4

$x \; 10^{n} = ?$

Finite precision leads to arithmetic overflow

- If the result is too big to store, we incur an *arithmetic overflow* error

…If we only have 4 places to store the number (base-10)

Thousands	Hundreds	Tens	Units
1	9	8	4

x 10…

Thousands	Hundreds	Tens	Units
9	8	4	

1

0

= 9840?!

Negative numbers

- So far, we've considered only *positive* numbers, how about negative numbers?
- Let's now look at two simple coding approaches for *negative* numbers:

  1. Sign and magnitude
  2. Excess *n*

- (Remember again…
  - Computer design typically only allows for a fixed number of digits: **small, positive whole numbers**
  - So, we must, map our negative numbers into this space)

Approach 1: sign and magnitude

Sign is	Magnitude		
negative?	Hundreds	Tens	Units
No	9	8	4

- This "works", but has two drawbacks:
  1. We sacrifice a column for the sign indicator (called ***flag***)
  2. We get *two* representations of zero

  Both positive and negative 0.messy: when we have to test for zero, we must do it twice! Complicates hardware and/or software)

Sign is	Magnitude		
negative?	Hundreds	Tens	Units
No	0	0	0

Sign is	Magnitude		
negative?	Hundreds	Tens	Units
Yes	0	0	0

Approach 2: excess *n*

- We use what was the “sign” column to represent a so-called ***excess***
- Best introduced by example: ***code the number 150 using four decimal columns…***
  - We can do this is using “excess 5000”:
    put a 5 in the thousands, column …

Stored value *(excess 5000)*			
Thousands	**Hundreds**	**Tens**	**Units**
5	**1**	**5**	**0**

We’re initially considering a positive value number

Excess *n* in action

- We **code** (i.e. store) a number by **adding** the excess to it
  - Ensures that negative value numbers get coded as positive value
- We **decode** a representation by **subtracting** the excess from it
- Example: code values in the range -5..+4 using 'excess 5'…

To code this:	**-5**	**-4**	**-3**	**-2**	**-1**	**0**	**1**	**2**	**3**	**4**
We use this: *(value + 5)*	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

(Essentially, adding the excess has the effect of "*sliding along*")

Given this, we can build "simple" hardware to deal with our "small, positive whole numbers" ...and then use a subset of the available range for negative values.

Properties of excess *n* (1)

- Compared to 'sign and magnitude'…
  - We have regained a storage slot (the *thousands* column, in our initial example)
  - We no longer have two representations of 0

Coded value of 150 in *excess 5000*			
Thousands	**Hundreds**	**Tens**	**Units**
5	**1**	**5**	**0**

Properties of excess *n* (2)

- What range can we represent with excess *n*?
- Overall range of 0-9999: half for negative value ; half for positive value
  - -5000, …, -1, 0, …, 4999 mapped into 0, …, 4999, 5000, …, 9999

Coded value of 150 in *excess 5000*			
Thousands	Hundreds	Tens	Units
5	**1**	**5**	**0**

Choosing the value of *n*

- Why did we choose excess 5000 in our examples…??
  - *n* was **half the number of values available**: i.e. 10000/2
- Can also be seen as a function of the number of *columns*
  - Our example used 4 columns (base-10), so used 'excess 5000' ($n=5000= 10^4/2$)

Num of values available		Num of columns	Use excess…
10^1	10 (i.e. 0..9)	1	$(10^1/2) = 5$
10^2	100 (i.e. 0..99)	2	$(10^2/2) = 50$
10^3	1000	3	$(10^3/2) = 500$
10^4	10000	**4**	$(10^4/2) = 5000$

(In binary (i.e. base-2), if we use 4 columns, $n= 2^4/2 = 8$ => 'excess 8' for 4-bit binary)

Generalising to *fixed point* (i.e. fractions)

- We simply reserve some columns for the fractional part

Thousands	Hundreds	Tens	Units	Tenths	Hundredths	Thousandths
$n \times 10^3$	$n \times 10^2$	$n \times 10^1$	$n \times 10^0$	$n \times 10^{-1}$	$n \times 10^{-2}$	$n \times 10^{-3}$
n x 1000	n x 100	n x 10	n x 1	n x .1	n x .01	n x .001

e.g., **1798.059** = 1798 + 59/1000
= **1798** + **0**/10 + **5**/100 + **9**/1000

Thousands	Hundreds	Tens	Units	Tenths	Hundredths	Thousandths
1 x 1000	***7*** x 100	***9*** x 10	***8*** x 1	***0*** x .1	***5*** x .01	***9*** x .001

But there's a problem with fixed point

- We quickly run out of columns!
- Let's say we use 8 columns for numbers (base-10), and reserve half of these columns for fractional parts:
  - The usable range (i.e. integer part) is ~ ± 5000 (with excess n, n=5000= $10^4/2$)
- Then, how do we express planetary distance, or similarly very large numbers??

Solution: trade *precision* for *range*

- As we know, in everyday life, people often use approximate or order-of-magnitude values
  - "The mass of the Sun is ~ 2 x 10^{33}g"*...to the gram ????*
  - For very small numbers, we generally talk of numbers "to *n* decimal places"
- So, **except finance, etc.,** we **can** often use approximate values, and trade accuracy for range

  It leads to the idea of ***Floating Point*** representation…

(First: a brief recap on *exponents)*

Lancaster University

- Recall normally how we've been referring to numbers:

Number	1 x	Exponent
0. 0001	10^{-4}	-4
0. 0010	10^{-3}	-3
0. 0100	10^{-2}	-2
0. 1000	10^{-1}	-1
1. 0000	10^{0}	0
10. 0000	10^{1}	1
100. 0000	10^{2}	2
1000. 0000	10^{3}	3
10000. 0000	10^{4}	4

mantissa *exponent*

- So we can, e.g., represent 150.2 as **1.502×10^{2}**
  … or represent 0.015 as **1.5×10^{-2}**

Background source on Moodle: Pitfalls of Floating Point;
What Every Computer Scientist Should Know About Floating Point

Floating point

- The basic idea is to represent the *mantissa* and the *exponent* as separate fixed-point numbers

Mantissa (0 ≤ mantissa < 1)			Exponent		Value
Sign	Tenths	Hundredths	Sign	Value	
+	.0	0	+	0	0
+	.4	0	+	0	.4
+	.5	0	+	1	*.5 x 10^{1} (= 5)*
-	.2	1	+	1	-2.1
+	.1	0	+	9	100,000,000
+	.1	0	-	1	*.1 x 10^{-1} (= .01)*

mantissa

exponent

n.b. need separate sign representations for exponent **and** for mantissa

Precision in floating point

- As noted, we increased range by *sacrificing precision*

Mantissa (0 ≤ mantissa < 1)			Exponent		Value
Sign	Tenths	Hundredths	Sign	Value	
+	.2	0	+	9	200,000,000
+	.7	2	-	4	0.000,072
-	.1	1	-	4	-0.000,011

$$+0.72 \times 10^{-4}$$

Only **2 digits of precision**, and “only” **1 digit of range**

N.B., we *normalise* by putting the first non-zero mantissa digit in the leftmost column (so **0.1×10^1** rather than 0.01×10^2)

Summary

- Computers only fundamentally deal with **small, positive whole numbers** – everything else is a matter of coding
- When fixed numbers of columns are used to represent numbers, we need to be aware of the possibility of *overflow*
- ‘Sign and magnitude’ and ‘excess n’ are two different coding schemes for representing negative numbers
- *Floating point* coding removes the range limitations of fixed point, but at the expense of precision