Number Representations

CSE 102

Make binary more human friendly

- Hexadecimal representation – base 16
- Commonly called “hex” but don’t be confused, it is not base 6, it is base 16
- Character set 0-9, a-f (alternately A-F)
  - a=10, b=11, c=12, d=13, e=14, and f=15
- C notation is to prefix hex with symbol `0x` (e.g., `0x12`, `0xa3`)

Positional notation applies

$$xyz_{16} = x \cdot 16^2 + y \cdot 16^1 + z \cdot 16^0$$
$$= x \cdot 256 + y \cdot 16 + z$$

So $02c_{16} = 0 \cdot 256 + 2 \cdot (16) + 12 = 44_{10}$

or 0x02c, which is the shorthand I will typically use in class

Benefits of Hex

- Real beauty of hex notation is ease with which one can move back and forth between hex and binary, since $16 = 2^4$
- To transform hex number (e.g., 0x3d50) to binary we expand each hex digit to 4 bits of binary:

3	d	5	0
0011	1101	0101	0000

Binary to Hex Transformation

- To transform binary number (e.g., 1001000) to hex we group into 4-bit groups (starting from right) and rewrite each group in hex

100	1000	
4	8	= 0x48

- Or, e.g., 110101110

1	1010	1110	
1	a	e	= 0x1ae

What about fractions?

- Positional number systems work on both sides of the decimal point (radix point).
- If radix is r (n integer digits, m fractional digits):

$$val = a_{n-1} \cdot r^{n-1} + a_{n-2} \cdot r^{n-2} + \dots + a_0 \cdot r^0 + a_{-1} \cdot r^{-1} + a_{-2} \cdot r^{-2} + a_{-m} \cdot r^{-m}$$

- e.g., $wx.yz_{16} = w \cdot 16 + x + y \cdot 16^{-1} + z \cdot 16^{-2}$
  or $wx.yz_2 = w \cdot 2 + x + y \cdot 2^{-1} + z \cdot 2^{-2}$

Two kinds of numbers

- Integers – radix point is assumed to be at the far right end of the digits:
  - E.g. 01001110.
- Fixed point – radix point is at a given, fixed location:
  - E.g. 0100.1110
  - 0.1001110 is a common representation on digital signal processors

Q notation

- Qn.m means a number with n+m bits (digits), n integer and m fractional. Sign bit is often in addition to this.
- E.g., Q3.4 for 0100.1100, with value 4.75
- Qm means a number with m+1 bits, m are fractional
- E.g., Q3 notation would have 4 bits and the following values
  - $wxyz = w.xyz = w \cdot (-1) + x \cdot (1/2) + y \cdot (1/4) + z \cdot (1/8)$
  - range is now -1 to +7/8, with resolution 1/8

Floating point representation

What about the reals? Use scientific notation.

In base 10: $x \cdot 10^y$ $0.32 \times 10^{-3} = 0.00032$

In base 2: $x \cdot 2^y$ called floating point

x ↑ mantissa; y ↑ exponent

IEEE Floating Point

- Limited range of x and y (fixed # of bits) means we cannot represent every real number exactly
- IEEE std. 754 describes a standard form for floating point number representations
  - Single precision is 32 bits in size
  - Double precision is 64 bits in size

Single precision (32 bits)

31	30 … 23	22 … 0
s	exponent (e)	fraction (f)
1	8 bits	23 bits

value = $(-1)^s \times 2^{e-127} \times \underline{1}.f$

↑ hidden "1"

range = $\pm\ 2 \times 10^{\pm 38}$

31	30 … 23	22 … 0
s	exponent (e)	fraction (f)

- s = 0, e = 0, f = 0 ⇒ value = zero
- e = 255, f = 0 ⇒ value = $(-1)^s \times$ infinity
- e = 255, f ≠ 0 ⇒ value = "not a number" triggers exception
- e = 0, f ≠ 0 ⇒ denormalized
  value = $(-1)^s \times 2^{-126} \times \underline{0}.f$
  ↑ hidden "0"
- Note use of sign-magnitude for entire number, and excess notation (excess 127) for exponent

Double precision (64 bits)

```
 63 62          52  51                  0
| s | exponent (e) | fraction (f)        |
  1    11 bits          52 bits
```

value = $(-1)^s \times 2^{e-1023} \times \underline{1}.f$
↑ hidden "1"

range = $\pm\ 2 \times 10^{\pm 308}$
$e = 0, f \neq 0 \Rightarrow$ denormalized
value = $(-1)^s \times 2^{-1022} \times \underline{0}.f$

Text – Characters and Strings

- ASCII – American Standard Code for Information Interchange
  - 7-bit codes representing basic Latin characters and numbers [A-Z, a-z, 0-9], some common punctuation, and control characters
  - There are a number of extensions to 8 bits, but only the 7-bit codes really standard.
- Unicode – 8- or 16-bit codes extending to a much wider set of languages
  - The first 128 codes are equivalent to the 7-bit ASCII standard

C Strings

- Strings are sequences of ASCII characters, stored one byte per character (8 bits), terminated by a NULL (zero) character
- E.g., "Hello!"

01001000	'H'	0x48
01100101	'e'	0x65
01101100	'l'	0x6c
01101100	'l'	0x6c
01101111	'o'	0x6f
00100001	'!'	0x21
00000000	NULL	0x00

ASCII Facts

- Numerical digits are assigned in order of increasing value

  i.e., '0' = 0x30
  '1' = 0x31
  '2' = 0x32

  '9' = 0x39

- For single character, value conversion is simply a difference of 0x30

More ASCII Facts

- Letters are also assigned in lexicographical order:

  'A' = 0x41
  'B' = 0x42

  'Z' = 0x5a

  'a' = 0x61
  'b' = 0x62

  'z' = 0x7a

- Upper/lower case conversion is simply a difference of 0x20

Still More ASCII Facts

- First 32 characters (0-0x1f) are control codes:

0x00	^@	null (C string terminator)
0x07	^G	bell
0x0a	^J	line feed
0x0c	^L	form feed
0x0d	^M	carriage return

Line breaks are not standardized

- End of line conventions differ by operating system:
  - In MS Windows: 0x0a, 0x0d is end of line
  - In Unix/Linux: 0x0a is end of line
  - 0x0a, linefeed, is sometimes called ‘newline’
- In C, ‘\n’ is mapped to OS end of line termination convention

Java Strings

- Strings are represented via the class “String”
- String objects are immutable
- The character encoding is system specific, e.g., either UTF-8 or UTF-16 (typical).
- The length is an instance variable in the object (in most implementations)
- The characters are stored in a char[] array (again, in most implementations)

Unicode

- Standard for character representation
  - Supports wide variety of languages, symbols
- UTF-8
  - Variable length code with 8-bit code units
  - U+0000 to U+007F are the same as ASCII
- UTF-16
  - Uses 16-bit code units, also variable length
  - Latin + Greek + Cyrillic + Coptic + Armenian + Hebrew + Arabic + Syrian + Tāna + N’Ko fit in 16 bits
- UTF-32
  - Uses 32-bit code units, fixed length

Images

- Consider the following bits:

0x002400081881423c

0000 0000 0010 0100 0000 0000 0000 1000

0001 1000 1000 0001 0100 0010 0011 1100

- Make 1 dark and 0 light:

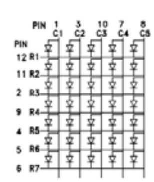
Images

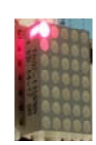
- Arrange in rows, one byte per row:

- Each bit is a “pixel” in the image

Controlling pixels

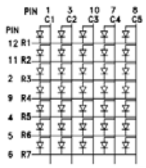
- Common approach is row, column multiplexing





- Extend with intensity control for each pixel
  - 8 bits → 0 is “off”, 255 (or 0xff) is “on”

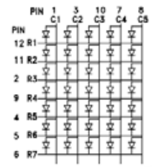
Row-based Multiplexed Control



for r = 1 to 7
  wait until next row time
  set row$_r$ LOW
  set all other rows HIGH
  for c = 1 to 5
    set column$_c$ to value for row$_r$
    (HIGH for on, LOW for off)
  end for
end for

This needs series resistors on each column

Column-based Multiplexed Control



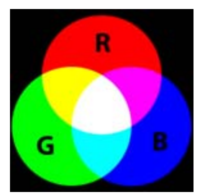
for c = 1 to 5
  wait until next column time
  set column$_c$ HIGH
  set all other columns LOW
  for r = 1 to 7
    set row$_r$ to value for column$_c$
    (LOW for on, HIGH for off)
  end for
end for

This needs series resistors on each row

Add color and more pixels

Color

- Additive color – primaries Red, Green, Blue



- Position close together and put diffuser above
  - This builds one pixel