DATA REPRESENTATION

Bringing Teachers Together Virtually
August 2022
Sandy Graham (sandy.graham@uwaterloo.ca)
Basic Principles

■ All data on a digital device is represented internally by a sequence of 0s and 1s
  - e.g. numbers, characters, images, sound, video, ...
  - Hexadecimal digits
    ■ easier for humans to read
    ■ a straightforward mapping from bit sequences (i.e. 4 bits → 1 hex digit)

■ Fixed length vs. variable length encoding

■ Compression
  - Using fewer bits to represent the same information (lossless) or a good approximation (lossy) of the information
  - e.g. zip, jpg, mp3, mp4, ...
Numbers

- Unsigned integers
  - binary equivalent
- Signed integers
  - sign bit
  - ones complement
  - twos complement
- Non integers
  - fixed point
  - floating point
A New Way to Subtract

- The traditional subtraction algorithm is hard because of borrowing
- Assume we are working with fixed width integers – e.g. 4-digit integers
  - Subtract 8413 - 5927 (answer is 2486)
  - Take each digit from the second operand, determine the 9s complement
    - find each digit’s difference from 9, no borrowing necessary
    - 5927 → 4072
  - Add 1 → 4072+ 1 = 4073
  - Add this number to the first operand → 8413 + 4073 = 12486
  - Ignore the extra digit → 2486
- Works with any base
Twos Complement

- Range of integers that are represented by n bits is $-2^{n-1} ... 2^{n-1} - 1$
- Flip the bits and add 1
  - *Ignore an extra bit*
- This action negates the integer
  - e.g. 8-bit integers if $x = 00101101$ then $-x = 11010010 + 1 = 11010011$
  - if $y = 10010001$ then $-y = 01101110 + 1 = 01101111$
- Subtraction can be thought of as negating the second operand and then adding
- Non-negative integers most significant bit is 0, negative integers most significant bit is 1
  - *Fixed width integer calculations can cause overflow*
Overflow

- Can happen when adding two positive integers or two negative integers where the sum is a value outside of the range of values that can be represented
  - *E.g. maximum signed integer value in C is 214783647 or $2^{31} - 1$*

- Recognize overflow when the most significant bit (MSB) does not make sense
  - *E.g. assume 8 bit signed integers using twos complement representation*
    - $01101001 + 01110110 = 11011111 \Rightarrow$ overflow
    - $105 + 118$ is out of range
    - Negate $11011111 \Rightarrow 00100001 = 33_{10}$
    - $105 + 118 = -33$

- Numeric values outside of the range may be managed by software
Non Integer Numbers

- **Fixed point**
  - Some bits designated for the integer part, some bits for the fraction part
  - Limited range but faster

- **Floating point**
  - Normalized representation i.e. $1.????? \times 2^??$
    - Implicit bit (all values start with 1 except for 0)
  - A sign bit, some bits designated for the fraction part, some bits for the exponent, includes a bias
  - **IEEE 754 standards**
    - Single precision (32 bits), double precision (64 bits), quadruple precision (128 bits)
    - Special values when exponent bits are all 0 or all 1
Interpreting Floating Point Numbers

- Assume a single precision number:
  - bits 0 ... 22 fraction, bits 23 ... 30 exponent, bit 31 sign, bias is 127

- Normal values: \((-1)^S \times (1+F) \times 2^{(E-bias)}\)
  - E.g. 01000011001011101010000000000000
  - S = 0, E = 10000110, F = 010111010100000000000000
  - Anything raised to the power of 0 is 1
  - \(10000110_2 = 134_{10}\), so \(E - 127\) is 7
  - Normalized value is \(1.010111010100000000000000\) \(\times 2^7\)
    which is \(10101110.101_2\) or \(174.625\)
Converting Fractions to Binary

- What happens when you multiply 0.1 * 0.9 in Python?
- Convert a fraction to binary
  - *E.g. 0.375*
    
    \[
    \begin{align*}
    0.375 \times 2 &= 0.75 \\
    0.75 \times 2 &= 1.5 \\
    0.5 \times 2 &= 1.0 \\
    \rightarrow 0.375_{10} &= 0.011_2
    \end{align*}
    \]
  - *E.g. 0.1*
    
    \[
    \begin{align*}
    0.1 \times 2 &= 0.2 \\
    0.2 \times 2 &= 0.4 \\
    0.4 \times 2 &= 0.8 \\
    0.8 \times 2 &= 1.6 \\
    0.6 \times 2 &= 1.2 \\
    \ldots \text{repeating bits} \\
    \rightarrow 0.1_{10} &= 0.000110011\ldots_2
    \end{align*}
    \]
- Bits are truncated in this case which leads to an imprecise representation
- Rational numbers may be represented precisely in software
Character Representation

- **ASCII codes** ([http://www.ascciitable.com](http://www.ascciitable.com))
  - 7 bits representing $2^7 = 128$ standard codes (teletype machines)
  - 8 bits (1 byte) provides 128 more codes
    - extended ASCII is not standard
  - **Fixed length encoding scheme – table lookup**

- **Unicode** ([https://www.unicode.org/charts](https://www.unicode.org/charts))
  - Potentially millions of code points (currently approximately 1 million)
  - $2^{16} = 65536$ (not enough) and $2^{32} = 4294967296$ (wasted space)
  - **UTF-32 fixed length encoding scheme**
  - **UTF-16 and UTF-8 variable length encoding schemes**
Unicode Encoding

- **Code point:** $U+\text{xxxxxx}$
  - *E.g.* $U+1F94C$ →

- **UTF-16**
  - *Most code points are represented with 16 bit (2 byte) codes*
  - *Some code points are represented with 32 bit (4 byte) codes*
  - *First 2 bytes determine the code length*

- **UTF-8 (most commonly used)**
  - $U+0000 \ldots U+007F$ (1 byte), $U+0800 \ldots U+07FF$ (2 bytes), $U+0800 \ldots U+FFFF$ (3 bytes), $U+10000 \ldots U+10FFFF$ (4 bytes)
  - *First bits indicate the code length: 0..., 110..., 1110..., 11110...*
  - *E.g.* $U+1F94C \rightarrow 4$ *byte code*

- Fill in the missing bits: \[11110\ldots10\ldots10\ldots10\ldots\]
- \[0x1F94C = 000011111100101001100 \text{(21 bit code)}\]
- **UTF-8 encoding**: \[111110000100111111010010110001100\]
Program Instructions

- Basic compilation
  - Source Code $\rightarrow$ compiled $\rightarrow$ Assembly Code $\rightarrow$ assembled $\rightarrow$ Machine Code

- Machine code instructions are bit sequences

- Assembly is mostly straightforward (except for labels)

- Examples:
  - `addi $t, $s, i` 0010 00ss ssst tttt iiii iiii iiii iiii
  - `add $d, $s, $t` 0000 00ss ssst tttt dddd d000 0010 0000
  - `jr $s` 0000 00ss sss0 0000 0000 0000 0000 1000
  - `addi $1, $0, -5` 0010 0000 0000 0001 1111 1111 1111 1011
  - `add $2, $1, $1` 0000 0000 0010 0001 0001 0000 0000 0010 0000
  - `jr $31` 0000 0011 1110 0000 0000 0000 0000 1000
Other Kinds of Data

- Anything you can assign a number to can be represented by a bit sequence
- Sound: frequency, amplitude, wave forms, etc.
- Colour: RGB (red, green blue) values, HSV (hue, saturation, lightness) models, etc.
- Images: coloured pixels, row/column positions on a grid
- Video: frames of images over time
- Examples:
  - Mark Guzdial ([https://www.youtube.com/watch?v=mGc6clf_Wt4](https://www.youtube.com/watch?v=mGc6clf_Wt4))