Word Size & Endianness
Word size

- Any given computer architecture has a “word size”.
- Word size determines the number of bits used to store a memory address (a pointer in C).
- Therefore you can $2^{\text{wordsize}}$ number of memory addresses.
- Until recently, most machines used 32 bits (4 bytes) as word size
  - Limits addresses to 4GB of total RAM
- These days, machines have 64-bit word size, actually only uses 48 bits of it for addresses
  - Potentially, could have $2^{48}$ addresses, that’s a lot of memory.
  - Theoretically up to 65,000 times amount of RAM of 32-bit systems. (~260TB)
Word-oriented memory organization

- Address of a word in memory is the address of the first byte in that word.

- Consecutive word addresses differ by 4 (32-bit) or 8 (64-bit).
Byte ordering in a word

- There are two different conventions of byte ordering in a word

- **Big Endian**
  - Examples: Sun, PowerPC Mac, Internet
  - Most significant byte has lowest address

- **Little Endian**
  - Examples: x86, ARM processors running Android, iOS, Windows
  - Most significant byte has highest address

- In other words, if you have a multi-byte word, what order do the bytes appear? What “end” of the word does the MSB live at?
Byte ordering in a word con’t

- Variable \( x \) has 4-byte value of \( 0x01234567 \)
- Address given by dereferencing \( x \) is \( 0x100 \)

<table>
<thead>
<tr>
<th>Big Endian</th>
<th>0x100 0x101 0x102 0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>01 23 45 67</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Little Endian</th>
<th>0x100 0x101 0x102 0x103</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>67 45 23 01</td>
</tr>
</tbody>
</table>

- We can test this programmatically. See \textit{memory/endian.c}
Byte ordering representation in C

- Casting any pointer to unsigned char* allows is to treat the memory as a byte array.

- Using printf format specifiers
  - %p - print pointer
  - %x - print value in hexadecimal

- See memory/byte_ordering.c
Floating Point
Fractional binary numbers

- How can we represent fractional binary numbers?
- One idea: use same approach as with decimal numbers, except use powers of 2 (as opposed to 10).
- So what is $1011.101_2$?
Fractional binary numbers

- How can we represent fractional binary numbers?
- One idea: use same approach as with decimal numbers, except use powers of 2 (as opposed to 10).
- So what is $1011.101_2$?

\[
(1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) + (1 \times 2^{-1}) + (0 \times 2^{-2}) + (1 \times 2^{-3}) \\
8 + 2 + 1 + \frac{1}{2} + \frac{1}{8} \\
11.625_{10}
\]
Fractional binary numbers

- How can we represent fractional binary numbers?
  - One idea: use same approach as with decimal numbers, except use powers of 2 (as opposed to 10).
  - So what is $1011.101_2$?
    \[(1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) + (1 \times 2^{-1}) + (0 \times 2^{-2}) + (1 \times 2^{-3})\]
    \[8 + 2 + 1 + \frac{1}{2} + \frac{1}{8}\]
    \[= 11.625_{10}\]
  - Going the other direction
    - $5 \frac{3}{4} \rightarrow 101.11_2$
    - $2 \frac{7}{8} \rightarrow 10.111_2$
That way of representing floating point numbers is simple, but has two significant limitations.

Only numbers that can be written as the sum of powers of 2 can be represented exactly.

- **Example**
  - $1/3$ \(0.0101010101[01]…_2\)
  - $1/5$ \(0.001100110011[0011]…_2\)
  - $1/10$ \(0.0001100110011[0011]…_2\)

Just one possible location for the binary point.

- This limits how many bits can be used for the fractional part and the whole number part.
- Range of representation much too small.
IEEE Floating Point

- **IEEE Standard 754**
  - Established in 1985 as uniform standard for floating point arithmetic
  - Before that, many idiosyncratic formats
  - Supported by all major CPUs

- **Driven by numerical concerns**
  - Nice standards for rounding, overflow, underflow
  - Numerical analysts and computer scientists collaborated to yield a performant and elegant solution.
IEEE precision options

Single precision: 32 bits

Double precision: 64 bits
Floating Point Representation

- **Numerical form**
  \((-1)^s \times M_2 \times 2^E\)
  - Sign bit \(s\) determines whether number is negative or positive
  - **Significand** \(M\) *normally* a fractional value, range \([1.0, 2.0)\)
  - **Exponent** \(E\) weights value by (possibly negative!) power of two

- **Encoding** (3 bit vectors that encode a floating point number)
  - Most significant bit is the sign bit \(s\)
  - \(\text{exp}\) field *encodes\( E\) (but is not equal to \(E\))
  - \(\text{frac}\) field *encodes\( M\) (but is not equal to \(M\))
Interpreting IEEE Values

- Three possible methods by which we evaluate a given bit vector representing a floating point type.
  - ‘Normalized’ values
  - ‘Denormalized’ values (more properly referred to as “Subnormal”)
  - ‘Special’ values

- Normalized is the most common case.

- Denormalized is for representing 0 and provide a property known as “gradual underflow” where values are spaced evenly near 0.

- Special, is well, special.

- Contents of exp determine ‘type’ i.e. how to encode & interpret.
Floating point encoding number line
Normalized values

- Precondition: \( exp \neq 000\ldots0 \) and \( exp \neq 111\ldots1 \)

- For some bit pattern: \( \text{value}_{10} = (-1)^s \cdot M \cdot 2^E \)

- \( s = \text{sign bit} \)

- \( E = [exp] - \text{bias} \)
  - \( \text{bias} = 2^{k-1} - 1 \)
  - \( k = \text{number of bits in } exp \)

- \( M = 1.\text{frac} \)
  - By assuming the leading bit is 1, we get an extra bit for “free”
  - Smallest value when all bits are zero: 000\ldots0, \( M = 1.0 \)
  - Largest value when all bits are one: 111\ldots1, \( M = 2.0 - \varepsilon \)
Normalized encoding example

- float f = 15213.0;
  \[15213_{10} = 111011011011012\]
  \[= 1.11011011011012 \times 2^{13}\]

\[
\text{value}_{10} = (-1)^s \times M_2 \times 2^E
\]
Normalized encoding example, con’t

- float f = 15213.0;
  \[15213_{10} = 111011011011011_2\]
  \[= 1.1101101101101_2 \times 2^{13}\]

- Significand (aka Mantissa)
  \[M = 1.1101101101101_2 \times 2^{13}\]
  \[frac = 110110110110100000000000_2\]

\[\text{value}_{10} = (-1)^s \times M_2 \times 2^E\]
Normalized encoding example, con’t

- `float f = 15213.0;`
  
  \[ 15213_{10} = 11101101101101_2 \]
  
  \[ = 1.1101101101101_2 \times 2^{13} \]

- **Significand (aka Mantissa)**
  
  \[ M = 1.1101101101101_2 \times 2^{13} \]
  
  \[ frac = 110110110110100000000000000_2 \]

- **Exponent**
  
  \[ E = 13 \]
  
  \[ bias = (2^8 - 1) = 127_{10} \]
  
  \[ exp = E + bias = 140_{10} = 10001100_2 \]

\[ s \quad exp \quad frac \]

\[
\begin{array}{c|c|c}
0 & 10001100 & 110110110110100000000000000
\end{array}
\]
Range of Expression

- For normalized 32-bit single precision…
  - The value of $\exp$ 1…254
  - The value of $E$ -126…127
  - Fairly large numbers; $\leq 2^{127}$
  - Fairly small numbers; $\geq 2^{-126}$

- For normalized 64-bit double precision, obviously this range is greater.

- Note that there is always a leading 1 in the value of mantissa $M$ for ‘normalized values’, so we cannot represent numbers that are very small.

- Next, we will observe what happens when $\exp$ is either 00…0 or 11…1
Denormalized values

- Precondition: $exp = 000\ldots0$

- For some bit pattern: $value_{10} = (-1)^s \cdot M_2 \cdot 2^E$

- $M = 0.\frac{\text{[frac]}}{}$
  - No implicit 1 prefix.
  - Allows for representation of numbers closer to 0

- $E = 1 - \text{bias}$
  - $\text{bias} = 2^{k-1} - 1$
  - $k =$ number of bits in $exp$
  - Differs from ‘normalized’, as $exp$ is 0, so use 1

- If $exp = 000\ldots0$, $\frac{\text{frac}}{} = 000\ldots0$ represents 0.0

- If $exp = 000\ldots0$, $\frac{\text{frac}}{} \neq 000\ldots0$ represent numbers evenly spaced near 0
Special values

- **Precondition**: \( \text{exp} = 111 \ldots 1 \)
  - Represents positive or negative infinity, a result of overflow

- **If \( \text{exp} = 111 \ldots 1 \) and \( \frac{\text{frac}}{0} \neq \text{NaN} \)**
  - A case when no numeric value can be determined
  - Examples: \( \sqrt{-1} \) (Not-a-Number (NaN))

- **Examples:**
  - \( -1.0/0.0 = +\infty \)
  - \( 1.0/0.0 = -\infty \)

- **If \( \text{exp} = 111 \ldots 1 \) and \( \frac{\text{frac}}{0} = \text{NaN} \)**
  - Not-a-Number (NaN)
  - A case when no numeric value can be determined
  - Examples: \( \infty \times 0 \)
Tiny Floating Point
Tiny Floating Point

- **6-bit Floating Point Representation**
  - The sign bit $s$ is in the most significant bit
  - The next three bits are the $exp$, with a $bias$ of 3
    - Note that the $bias$ is the same for all 6-bit precision numbers!
  - The last two bits are the $frac$

- **IEEE Format**
  - normalized, denormalized and special values
Tiny Normalized Example 1

\[ \text{value}_{10} = (-1)^s \times M_2 \times 2^E \]
\[ E = \text{exp} - \text{bias} \]

- **000100_2** (smallest positive value)
  - \( s = (-1)^0 = 1 \)
  - \( M = 1.00_2 \)
  - \( \text{bias} = 2^{3-1} - 1 = 3_{10} \)
  - \( E = 001_2 - 3_{10} = 1_{10} - 3_{10} = -2_{10} \)
  - \( 1 \times 1.00_2 \times 2^{-2} = .01_2 = 0.25_{10} \)
Tiny Normalized Example 2

- $011011_2$ (largest positive value)
  - $s = (-1)^0 = 1$
  - $M = 1.11_2$
  - bias = $2^{3-1} - 1 = 3_{10}$
  - $E = 110_2 - 3_{10} = 6_{10} - 3_{10} = 3_{10}$
  - $1 \times 1.11_2 \times 2^3 = 1110_2 = 14.0_{10}$
Tiny Denormalized Example 1

\[
\text{value}_{10} = (-1)^s \times M_2 \times 2^E
E = 1 - \text{bias}
\]

- **100011_2** (smallest negative value)
  - \( s = (-1)^1 = -1 \)
  - \( M = 0.11_2 \)
  - \( \text{bias} = 2^{3-1} - 1 = 3_{10} \)
  - \( E = 1_{10} - 3_{10} = -2_{10} \)
  - \(-1 \times 0.11_2 \times 2^{-2} = -0.0011_2 = -0.1875_{10} \)
Tiny Denormalized Example 2

\[ \text{value}_{10} = (-1)^s \cdot M_2 \cdot 2^E \]
\[ E = 1 - \text{bias} \]

- **000001_2** (smallest positive less than 1)
  - \( s = (-1)^0 = 1 \)
  - \( M = 0.01_2 \)
  - \( \text{bias} = 2^{3-1} - 1 = 3_{10} \)
  - \( E = 1_{10} - 3_{10} = -2_{10} \)
  - \( 1 \cdot 0.01_2 \cdot 2^{-2} = 0.0001_2 = 0.0625_{10} \)
Tiny special values

Result of overflow or infeasibility

- $exp = 111, frac = 00$
  - 011100, 111100
- Positive or negative infinity
- $exp = 111, frac \neq 00$
  - 011101, 011110, 011111, 111101, 111110, 111111
- Not-a-Number (NaN)
Exercises
Exercise 1

\[
\text{value}_{10} = (-1)^s \cdot M_2 \cdot 2^E
\]

\[E = ? - \text{bias}\]

- \[100111_2\]
  - \[s = (-1)^s = ?\]
  - \[M = ?_2\]
  - \[\text{bias} = ?_{10}\]
  - \[E = ?_{10}\]
  - \[\text{value}_{10} = ? = -0.4375_{10}\]
Exercise 2

value\textsubscript{10} = (-1)^s \times M_2 \times 2^E

E = ? - bias

- 100001_2
  - s = (-1)^s = ?
  - M = ?_2
  - bias = ?_{10}
  - E = ?_{10}
  - value\textsubscript{10} = ? = -0.0625_{10}