Unit 10

Signed Representation Systems
Binary Arithmetic

BINARY REPRESENTATION SYSTEMS REVIEW

Interpreting Binary Strings

- Given a string of 1's and 0's, you need to know the representation system being used, before you can understand the value of those 1's and 0's.
- Information (value) = Bits + Context (System)

01000001 = ?

Binary Representation Systems

- Integer Systems
  - Unsigned
    - Unsigned (Normal) binary
  - Signed
    - Signed Magnitude
    - 2's complement
    - Excess-N*
    - 1's complement*

- Floating Point
  - For very large and small (fractional) numbers

- Codes
  - Text
    - ASCII / Unicode
  - Decimal Codes
    - BCD (Binary Coded Decimal) / (8421 Code)

* = Not fully covered in this class
Review of Number Systems

- Number systems consist of
  1. A base (radix) \( r \)
  2. \( r \) coefficients [0 to \( r-1 \)]

- Human System: Decimal (Base 10):
  - 0,1,2,3,4,5,6,7,8,9

- Computer System: Binary (Base 2): 0,1

- Human systems for working with computer systems (shorthand for human to read/write binary)
  - Octal (Base 8): 0,1,2,3,4,5,6,7
  - Hexadecimal (Base 16): 0-9,A,B,C,D,E,F (A thru F = 10 thru 15)

Binary Examples

\[
(1001.1)_{2} = 8 + 1 + 0.5 = 9.5_{10}
\]

\[
\frac{8}{2} \cdot \frac{4}{2} \cdot \frac{2}{2} \cdot \frac{1}{2} \cdot \frac{.5}{2} = 9.5_{10}
\]

\[
(10110001)_{2} = 128 + 32 + 16 + 1 = 177_{10}
\]

Unique Combinations

- Given \( n \) digits of base \( r \), how many unique numbers can be formed? \( r^n \)
  - What is the range? [0 to \( r^n-1 \)]

  - 2-digit, decimal numbers (\( r=10, n=2 \))
    - 0-9
    - 100 combinations: 00-99

  - 3-digit, decimal numbers (\( r=10, n=3 \))
    - 0-9
    - 1000 combinations: 000-999

  - 4-bit, binary numbers (\( r=2, n=4 \))
    - 0-1
    - 16 combinations: 0000-1111

  - 6-bit, binary numbers (\( r=2, n=6 \))
    - 0-1
    - 64 combinations: 000000-111111

Main Point: Given \( n \) digits of base \( r \), \( r^n \) unique numbers can be made with the range [0 - \( (r^n-1) \)]

Approximating Large Powers of 2

- Often need to find decimal approximation of a large powers of 2 like \( 2^{16} \), \( 2^{32} \), etc.

  - \( 2^{16} = 2^8 \cdot 2^8 \approx 64 \cdot 10^3 = 64,000 \)

- Use following approximations:
  - \( 2^{20} = 10^3 \) (1 thousand) = 1 Kilo-
  - \( 2^{20} = 10^6 \) (1 million) = 1 Mega-
  - \( 2^{30} = 10^9 \) (1 billion) = 1 Giga-
  - \( 2^{40} = 10^{12} \) (1 trillion) = 1 Tera-

- For other powers of 2, decompose into product of \( 2^{10} \) or \( 2^{20} \) or \( 2^{30} \) and a power of 2 that is less than \( 2^{10} \)
  - 16-bit half word: 64K numbers
  - 32-bit word: 4G numbers
  - 64-bit dword: 16 million trillion numbers

  - \( 2^{28} = 2^8 \cdot 2^{20} \approx 256 \cdot 10^6 = 256,000,000 \)

  - \( 2^{32} = 2^8 \cdot 2^{24} \approx 4 \cdot 10^9 = 4,000,000,000 \)
Decimal to Unsigned Binary

- To convert a decimal number, $x$, to binary:
  - Only coefficients of 1 or 0. So simply find place values that add up to the desired values, starting with larger place values and proceeding to smaller values and place a 1 in those place values and 0 in all others.

$$25_{10} = \begin{array}{cccccc}
32 & 16 & 8 & 4 & 2 & 1 \\
0 & 1 & 1 & 0 & 0 & 1 \\
\end{array}$$

For $25_{10}$, the place value 32 is too large to include so we include 16. Including 16 means we have to make 9 left over. Include 8 and 1.

Decimal to Another Base

- To convert a decimal number, $x$, to base $r$:
  - Use the place values of base $r$ (powers of $r$). Starting with largest place values, fill in coefficients that sum up to desired decimal value without going over.

$$75_{10} = \begin{array}{cccccc}
256 & 16 & 1 \\
0 & 4 & B \text{ hex} \\
\end{array}$$

Binary Representation Systems

- Integer Systems
  - Unsigned
    - Unsigned (Normal) binary
  - Signed
    - Signed Magnitude
    - 2's complement
    - 1's complement*
    - Excess-N*
- Floating Point
  - For very large and small (fractional) numbers

Codes

- Text
  - ASCII / Unicode
- Decimal Codes
  - BCD (Binary Coded Decimal) / (8421 Code)

* = Not covered in this class
Unsigned and Signed

• Normal (unsigned) binary can only represent positive numbers
  – All place values are positive
• To represent BOTH positive and negative numbers we must use the available binary codes differently, some for the positive values and others for the negative values
  – We call these *signed* representations

Signed Number Representation

• 2 Primary Systems
  – ______________________
  – ______________________ (most widely used for integer representation)

Signed numbers

• All systems used to represent negative numbers split the possible binary combinations in half (half for positive numbers / half for negative numbers)
• In both signed magnitude and 2’s complement, positive and negative numbers are separated using the MSB
  – _______ means negative
  – _______ means positive

Signed Magnitude System

• Use binary place values but now MSB represents the sign (1 if negative, 0 if positive)

<table>
<thead>
<tr>
<th>0 to 15</th>
<th>-7 to +7</th>
<th>-127 to +127</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-bit Unsigned</td>
<td>4-bit Signed Magnitude</td>
<td>8-bit Signed Magnitude</td>
</tr>
<tr>
<td>Bit 3</td>
<td>Bit 2</td>
<td>Bit 1</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>
Signed Magnitude Examples

4-bit Signed Magnitude

\[
\begin{array}{cccc}
1 & 1 & 0 & 1 \\
4 & 2 & 1 \\
\hline
+ & - & + & - \\
\end{array}
\]

\[
\frac{1}{4} \div \frac{1}{2} \div \frac{1}{8} = -5
\]

\[
\begin{array}{cccc}
0 & 0 & 1 & 1 \\
4 & 2 & 1 \\
\hline
+ & - & + & + \\
\end{array}
\]

\[
\frac{0}{4} \div \frac{0}{2} \div \frac{0}{8} = +3
\]

\[
\begin{array}{cccc}
1 & 1 & 1 & 1 \\
4 & 2 & 1 \\
\hline
+ & + & + & + \\
\end{array}
\]

\[
\frac{1}{4} \div \frac{1}{2} \div \frac{1}{8} = -7
\]

8-bit Signed Magnitude

\[
\begin{array}{cccccccc}
1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\
64 & 32 & 16 & 8 & 4 & 2 & 1 \\
\hline
+ & - & + & + & + & + & + & + \\
\end{array}
\]

\[
\frac{1}{64} \div \frac{0}{32} \div \frac{0}{16} \div \frac{0}{8} \div \frac{0}{4} \div \frac{0}{2} \div \frac{0}{1} = -19
\]

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\
64 & 32 & 16 & 8 & 4 & 2 & 1 \\
\hline
+ & - & + & + & + & + & + & + \\
\end{array}
\]

\[
\frac{0}{64} \div \frac{0}{32} \div \frac{0}{16} \div \frac{0}{8} \div \frac{0}{4} \div \frac{0}{2} \div \frac{0}{1} = +25
\]

Important: Positive numbers have the same representation in signed magnitude as in normal unsigned binary.

Signed Magnitude Range

- Given n bits...
  - MSB is sign
  - Other n-1 bits = normal unsigned place values
  - Range with n-1 unsigned bits = \([0 \text{ to } 2^{n-1} - 1]\)

Range with n-bits of Signed Magnitude

\([-2^{n-1} - 1 \text{ to } +2^{n-1} - 1]\)

Disadvantages of Signed Magnitude

1. Wastes a combination to represent _____

\[0000 = 1000 = ____\]

2. Addition and subtraction algorithms for signed magnitude are different than unsigned binary (we’d like them to be the same to use same HW)

\[\begin{array}{cc}
4 & 6 \\
\hline
-6 & -4 \\
\end{array}\]

Swap & make res. negative

2’s Complement System

- Normal binary place values except MSB has

\[\begin{array}{cccccc}
& & & & & \\
& & & & & \\
& & & & & \\
& & & & & \\
& & & & & \\
\end{array}\]

- MSB of 1 = _____

4-bit

- Unsigned

\[\begin{array}{cccc}
8 & 4 & 2 & 1 \\
\hline
\end{array}\]

\[\rightarrow 0 \text{ to } 15\]

4-bit

- 2’s complement

\[\begin{array}{cccc}
8 & 4 & 2 & 1 \\
\hline
\end{array}\]

\[\rightarrow -8 \text{ to } +7\]

8-bit

- 2’s complement

\[\begin{array}{cccccccc}
64 & 32 & 16 & 8 & 4 & 2 & 1 \\
\hline
\end{array}\]

\[\rightarrow -128 \text{ to } +127\]
2’s Complement Examples

4-bit 2’s complement

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2’s complement</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>-8</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

-8 + 4 + 2 + 1 = -3

Notice that +3 in 2’s comp. is the same as in the unsigned system

8-bit 2’s complement

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2’s complement</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>-128</td>
<td>64</td>
<td>32</td>
<td>16</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

-128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = -127

Important: Positive numbers have the _______ representation in 2’s complement as in normal unsigned binary

2’s Complement Range

- Given n bits...
  - Max positive value = _____________
    - Includes all n-1 positive place values
  - Max negative value = _____________
    - Includes only the negative MSB place value

Range with n-bits of 2’s complement

[-2^{n-1} to +2^{n-1}–1]

- Side note – What decimal value is 111...11?

Important: Positive numbers have the _______ representation in 2’s complement as in normal unsigned binary

Comparison of Systems

Signed Mag.

<table>
<thead>
<tr>
<th>Signed Mag.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>+1</td>
</tr>
<tr>
<td>-1</td>
</tr>
</tbody>
</table>

2’s comp.

<table>
<thead>
<tr>
<th>2’s comp.</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
</tr>
<tr>
<td>+3</td>
</tr>
</tbody>
</table>

Unsigned and Signed Variables

- In C, _______ variables use unsigned binary (normal power-of-2 place values) to represent numbers
  
  unsigned char x = 147;

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>64</td>
<td>32</td>
<td>16</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

  = +147

- In C, signed variables use the ________________ (Neg. MSB weight) to represent numbers
  
  char x = -109;

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>-128</td>
<td>64</td>
<td>32</td>
<td>16</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

  = -109
**IMPORTANT NOTE**

- All computer systems use the **2's complement system** to represent **signed integers**!
- So from now on, if we say an integer is **signed**, we are actually saying it uses the 2's complement system unless otherwise specified.
  - We will not use "signed magnitude" unless explicitly indicated.

**Zero and Sign Extension**

- Extension is the process of increasing the number of bits used to represent a number without changing its value.

  - **Unsigned = Zero Extension (Always add leading ____'s):**
    - $111011 = \underline{111011}$
    - Increase a 6-bit number to 8-bit number by ____ extending
  - **2’s complement = Sign Extension (Replicate ____ bit):**
    - pos. $011010 = ____011010$
    - neg. $110011 = ____110011$

**Zero and Sign Truncation**

- Truncation is the process of decreasing the number of bits used to represent a number without changing its value.

  - **Unsigned = Zero Truncation (Remove leading 0’s):**
    - $00111011 = 111011$
    - Decrease an 8-bit number to 6-bit number by truncating 0’s. Can't remove a '1' because value is changed
  - **2’s complement = Sign Truncation (Remove _____ of sign bit):**
    - pos. $00011010 = ____$
    - neg. $1110011 = ____$

**Data Representation**

- In C/C++ variables can be of different types and sizes:
  - **Integer Types (signed and unsigned)**
    - **C Type** | **Bytes** | **Bits** | **ATmega328**
    - (unsigned) char | 1 | 8 | byte
    - (unsigned) short [int] | 2 | 16 | word
    - (unsigned) long [int] | 4 | 32 | -1
    - [unsigned] long long [int] | 8 | 64 | -1
    - int | ?2 | ?2 | ?2
  1 Can emulate but has no single-instruction support
  2 Varies by compiler/machine (avr-gcc: int = 2 bytes, g++ for x86: int = 4-bytes)
  - **Floating Point Types**
    - **C Type** | **Bytes** | **Bits** | **ATmega328**
    - float | 4 | 32 | N/A
    - double | 8 | 64 | N/A

---

1. Theorem: A number in 2's complement representation is negative if its sign bit is 1.
2. Any copies of the MSB can be removed without changing the numbers value. Be careful not to change the sign by cutting off ALL the sign bits.
ARITHMETIC

10.29

ARITHMETIC

10.30

Binary Arithmetic

• Can perform all arithmetic operations (+, -, *, /) on binary numbers
• Can use same methods as in decimal
  – Still use carries and borrows, etc.
  – Only now we carry when sum is 2 or more rather than 10 or more (decimal)
  – We borrow 2's not 10's from other columns
• Easiest method is to add bits in your head in decimal (1+1 = 2) then convert the answer to binary (2₁₀ = 10₂)

10.31

Binary Addition

• In decimal addition we ______ when the sum is 10 or more
• In binary addition we carry when the sum is __ or more
• Add bits in binary to produce a sum bit and a carry bit

0 1 1 1
+ 0 0 1 1

0 1 0 1

1 0 1 0

0 1 1 1
+ 0 0 1 1

0 1 0 1

- 0 1 0 1

10.32

Binary Addition & Subtraction

• In decimal addition we ______ when the sum is 10 or more
• In binary addition we carry when the sum is __ or more
• Add bits in binary to produce a sum bit and a carry bit
**Binary Addition**

1.  
   
   \[
   \begin{array}{c}
   \text{0110 (6)} \\
   \text{+ 0111 (7)} \\
   \text{1101 (13)}
   \end{array}
   \]

   \[
   \begin{array}{c}
   \text{0110 (6)} \\
   \text{+ 0111 (7)} \\
   \text{1101 (13)}
   \end{array}
   \]

   carry bit \quad sum bit

2.  
   
   \[
   \begin{array}{c}
   \text{10} \\
   \text{+ 0110 (6)} \\
   \text{110 (13)}
   \end{array}
   \]

   carry bit \quad sum bit

3.  
   
   \[
   \begin{array}{c}
   \text{110} \\
   \text{0110 (6)} \\
   \text{+ 0111 (7)} \\
   \text{1101 (13)}
   \end{array}
   \]

   carry bit \quad sum bit

**Hexadecimal Arithmetic**

- Same style of operations
  - Carry when sum is 16 or more, etc.

\[
\begin{array}{c}
\text{4 D}_{16} \\
\text{+ B 5}_{16}
\end{array}
\]

\[
\begin{array}{c}
\text{---} \\
\text{---}
\end{array}
\]

\[
\begin{array}{c}
\text{16} \\
\text{1}
\end{array}
\]

**Taking the Negative**

- Given a number in signed magnitude or 2’s complement how do we find its negative (i.e. \(-1 \times X\))
  - Signed Magnitude: _______________
    - 0110 = +6 => _______________
  - 2’s complement: “__________________________”
    - 0110 = +6 => __________________
  - Operation defined as:
    1. Flip/invert/not ___________ (1’s complement)
    2. Add ___ and drop any _______
       (i.e. finish with the same # of bits as we start with)
Taking the 2’s Complement

- Invert (flip) each bit (take the 1’s complement)
  - 1’s become 0’s
  - 0’s become 1’s
- Add 1 (drop final carry-out, if any)

Important: Taking the 2’s complement is equivalent to taking the negative (negating)

2’s Complement System Facts

- Normal binary place values but MSB has negative weight
- MSB determines sign of the number
  - 0 = positive / 1 = negative
- Special Numbers
  - 0 = All 0’s (00...00)
  - -1 = All 1’s (11...11)
  - Max Positive = 0 followed by all 1’s (011...11)
  - Max Negative = 1 followed by all 0’s (100...00)
- To take the negative of a number (e.g. -7 => +7 or +2 => -2), requires taking the complement
  - 2’s complement of a # is found by flipping bits and adding 1

ADDITION AND SUBTRACTION
2’s Complement Addition/Subtraction

- **Addition**
  - Sign of the numbers do not matter
  - ________________
  - ________________
- **Subtraction**
  - Any subtraction (A-B) can be converted to addition (_________) by taking the __________ of B
  - (A-B) becomes (______________________)
  - Drop any carry-out
- The _____ of the result is produced by performing the above process and need not be considered separately

2’s Complement Addition

- No matter the sign of the operands just add as normal
- Drop any extra carry out

```
0011  1101
+ 0010  + 0010
```

```
0011  1101
+ 1110  + 1110
```

Unsigned and Signed Addition

- Addition process is the same for both unsigned and signed numbers
  - Add columns right to left
- Examples:

<table>
<thead>
<tr>
<th>If unsigned</th>
<th>If signed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>1001</td>
</tr>
<tr>
<td>+ 0011</td>
<td>+ 0011</td>
</tr>
</tbody>
</table>

2’s Complement Subtraction

- Take the 2’s complement of the subtrahend and add to the original minuend
- Drop any extra carry out

```
0011 (+3)   1101 (-3)
- 0010 (+2)  - 1110 (-2)
```
Unsigned and Signed Subtraction

• Subtraction process is the same for both unsigned and signed numbers
  – Convert $A - B$ to $A + \text{Comp. of } B$
  – Drop any final carry out
• Examples:
  
<table>
<thead>
<tr>
<th>If unsigned</th>
<th>If signed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1100 (12)</td>
<td>(-4)</td>
</tr>
<tr>
<td>- 0010 (2)</td>
<td>(2)</td>
</tr>
</tbody>
</table>

Important Note

• Almost all computers use 2's complement because...
• The same addition and subtraction __________ can be used on unsigned and 2's complement (signed) numbers
• Thus we only need one ______________ to perform operations on both unsigned and signed numbers

Overflow

• Overflow occurs when the result of an arithmetic operation is ___________
  ________________
• Conditions and tests to determine overflow depend on __________ of numbers (signed or unsigned) in the operation
Unsigned Overflow

Overflow occurs when you cross this discontinuity.

10 + 7 = 17

With 4-bit unsigned numbers we can only represent 0 – 15. Thus, we say overflow has occurred.

2’s Complement Overflow

Overflow occurs when you cross this discontinuity.

5 + 7 = +12

-6 + -4 = -10

With 4-bit 2’s complement numbers we can only represent -8 to +7. Thus, we say overflow has occurred.

Overflow in Addition

- Overflow occurs when the result of the addition cannot be represented with the given number of bits.
- Tests for overflow:
  - Unsigned: ________________
  - Signed: ________________

Overflow in Subtraction

- Overflow occurs when the result of the subtraction cannot be represented with the given number of bits.
- Tests for overflow:
  - Unsigned: if Cout = 0
  - Signed: if addition is p + p = n or n + n = p

If unsigned If signed

0111 0110
+ 0100 + 0101
0001 1011
FLOATING POINT

Floating Point

- Used to represent very small numbers (fractions) and very large numbers
  - Avogadro’s Number: +6.0247 * 10^{23}
  - Planck’s Constant: +6.6254 * 10^{-27}
  - Note: 32 or 64-bit integers can’t represent this range
- Floating Point representation is used in HLL’s like C by declaring variables as `float` or `double`

Fixed Point

- Unsigned and 2’s complement fall under a category of representations called “Fixed Point”
- The radix point is assumed to be in a fixed location for all numbers
  [Note: we could represent fractions by implicitly assuming the binary point is at the ________. Variables just store bits...you can assume the binary point is anywhere you like]
  - Integers: 10011101. (binary point to right of LSB)
    - For 32-bits, unsigned range is 0 to ~4 billion
  - Fractions: .10011101 (binary point to left of MSB)
    - Range [0 to 1)
- Main point: By __________ the radix point, we limit the range of numbers that can be represented
  - Floating point allows the radix point to be in a different location for each value

Floating Point Representation

- Similar to scientific notation used with decimal numbers
  - ±D.DDD * 10^{±exp}
- Floating Point representation uses the following form
  - _________________
  - 3 Fields: __________, __________, ____________ (also called _________________)
  - Bit storage
  - Fixed point Rep.
Normalized FP Numbers

- Decimal Example
  - +0.754*10^{15} is not correct scientific notation
  - Must have exactly _____ significant digit before decimal point: _______________
- In binary the only _______________ is ’1’
- Thus normalized FP format is: _______________
- FP numbers will always be normalized before being __________ in memory or a reg.
  - The 1. is actually ______________ but assumed since we always will store normalized numbers
  - If HW calculates a result of 0.001101*2^5 it must normalize to 1.101000*2^2 before storing

IEEE Floating Point Formats

- Single Precision (32-bit format)
  - 1 Sign bit (0=pos/1=neg)
  - ___ Exponent bits
    - ________ representation
    - More on next slides
  - ___ fraction (significand or mantissa) bits
  - Equiv. Decimal Range:
    - 7 digits \( \times 10^{\pm38} \)

- Double Precision (64-bit format)
  - 1 Sign bit (0=pos/1=neg)
  - ___ Exponent bits
    - ________ representation
    - More on next slides
  - ___ fraction (significand or mantissa) bits
  - Equiv. Decimal Range:
    - 16 digits \( \times 10^{\pm308} \)

Floating Point vs. Fixed Point

- Single Precision (32-bits) Equivalent Decimal Range:
  - 7 significant decimal digits \( \times 10^{\pm38} \)
  - Compare that to 32-bit signed integer where we can represent \( \pm2 \) billion. How does a 32-bit float allow us to represent such a greater range?
  - FP allows for range but sacrifices precision (can’t represent all numbers in its range)
- Double Precision (64-bits) Equivalent Decimal Range:
  - 16 significant decimal digits \( \times 10^{\pm308} \)

Exponent Representation

- Exponent needs its own sign (+/-)
- Rather than using 2’s comp. system we use Excess-N representation
  - Single-Precision uses Excess-127
  - Double-Precision uses Excess-1023
  - This representation allows FP numbers to be easily compared
- Let \( E' = \) stored exponent code and \( E = \) true exponent value
- For single-precision: \( E' = E + 127 \)
  - \( 2^1 \Rightarrow E = 1, E' = 128_{10} = 10000000_2 \)
- For double-precision: \( E' = E + 1023 \)
  - \( 2^2 \Rightarrow E = -2, E' = 1021_{10} = 011111111101_2 \)

Comparison of 2’s comp. & Excess-N
Q: Why don’t we use Excess-N more to represent negative \( E' \)’s
Single-Precision Examples

1. $2^7 = 128$, $2^3 = 8$

\[
\begin{array}{cccccc}
   & 1 & 1000 & 0010 & 110 & 0110 & 0000 & 0000 & 0000 & 0000 \\
\end{array}
\]

2. $+0.6875 = +0.1011$