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 = ?

  65_{10} 

  41_{BCD} 

  ‘A’_{ASCII}

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

\[
(1001.1)_2 = 8 + 1 + 0.5 = 9.5_{10}
\]

\[
(10110001)_2 = 128 + 32 + 16 + 1 = 177_{10}
\]

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 ($r=10$, $n=2$)
  3-digit, decimal numbers ($r=10$, $n=3$)
  4-bit, binary numbers ($r=2$, $n=4$)
  6-bit, binary numbers ($r=2$, $n=6$)

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.

  \[2^{16} = 2^6 \times 2^{10} \approx 64 \times 10^3 = 64,000\]

• Use following approximations:
  – $2^{10} = 10^3$ (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

  \[2^{28} = 2^8 \times 2^{20} \approx 256 \times 10^6 = 256,000,000\]

  \[2^{22} = 2^2 \times 2^{30} \approx 4 \times 10^9 = 4,000,000,000\]
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.

\[
\begin{array}{c}
25_{10} = \\
32 & 16 & 8 & 4 & 2 & 1 \\
0 & 1 & 1 & 0 & 0 & 1
\end{array}
\]

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.

\[
\begin{array}{c}
75_{10} = \\
256 & 16 & 1 \\
0 & 4 & B_{\text{hex}}
\end{array}
\]

Signed Magnitude
2’s Complement System

**SIGNED SYSTEMS**

Binary Representation Systems

- Integer Systems
  - Unsigned
  - 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)

4-bit Unsigned

<table>
<thead>
<tr>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

0 to 15

4-bit Signed Magnitude

<table>
<thead>
<tr>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

-7 to +7

8-bit Signed Magnitude

<table>
<thead>
<tr>
<th>Bit 7</th>
<th>Bit 6</th>
<th>Bit 5</th>
<th>Bit 4</th>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>+/−</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>

-127 to +127
### Signed Magnitude Examples

**4-bit Signed Magnitude**

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>= -5</th>
</tr>
</thead>
<tbody>
<tr>
<td>+/-</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>= +3</th>
</tr>
</thead>
<tbody>
<tr>
<td>+/-</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Notice that +3 in signed magnitude is the same as in the unsigned system.

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>= -7</th>
</tr>
</thead>
<tbody>
<tr>
<td>+/-</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

**8-bit Signed Magnitude**

<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>
<th>= -19</th>
</tr>
</thead>
<tbody>
<tr>
<td>+/-</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>

Range with n-1 unsigned bits = [0 to $2^{n-1} - 1$]

Range with n-bits of Signed Magnitude

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

---

### Signed Magnitude Range

- Given n bits...
  - MSB is sign
  - Other n-1 bits = normal unsigned place values
- Range with n-1 unsigned bits = [0 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)

### 2’s Complement System

- Normal binary place values except MSB has
  - MSB of 1 = _____

<table>
<thead>
<tr>
<th>4-bit Unsigned</th>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

0 to 15

<table>
<thead>
<tr>
<th>4-bit 2’s complement</th>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

-8 to +7

<table>
<thead>
<tr>
<th>8-bit 2’s complement</th>
<th>Bit 7</th>
<th>Bit 6</th>
<th>Bit 5</th>
<th>Bit 4</th>
<th>Bit 3</th>
<th>Bit 2</th>
<th>Bit 1</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></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 to +127
2’s Complement Examples

4-bit 2’s complement

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>= -5</th>
</tr>
</thead>
<tbody>
<tr>
<td>-8</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>= +3</th>
</tr>
</thead>
<tbody>
<tr>
<td>-8</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>= -1</th>
</tr>
</thead>
<tbody>
<tr>
<td>-8</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

8-bit 2’s complement

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>= -127</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>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>= +25</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>
<td></td>
</tr>
</tbody>
</table>

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?

Comparison of Systems

<table>
<thead>
<tr>
<th>Signed Mag.</th>
<th>0</th>
<th>+1</th>
<th>+2</th>
<th>+3</th>
<th>+4</th>
<th>+5</th>
<th>+6</th>
<th>+7</th>
</tr>
</thead>
<tbody>
<tr>
<td>2’s comp.</td>
<td>-7</td>
<td>-6</td>
<td>-5</td>
<td>-4</td>
<td>-3</td>
<td>-2</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>0000</td>
<td>0001</td>
<td>0010</td>
<td>0011</td>
<td>0100</td>
<td>0101</td>
<td>0110</td>
<td>0111</td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>1101</td>
<td>1110</td>
<td>1111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Unsigned and Signed Variables

• In C, _______ variables use unsigned binary (normal power-of-2 place values) to represent numbers
  ```c
  unsigned char x = 147;
  ```
  ```c
  1 0 0 0 1 0 0 1
  128 64 32 16 8 4 2 1
  ```
  = +147

• In C, signed variables use the ____________ (Neg. MSB weight) to represent numbers
  ```c
  char x = -109;
  ```
  ```c
  1 0 0 0 1 0 0 1
  -128 64 32 16 8 4 2 1
  ```
  = -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):
  
  \[
  111011 = \underbrace{00000000}_{\text{8-bit extension}}
  \]

  Increase a 6-bit number to 8-bit number by _____ extending

  2’s complement = Sign Extension (Replicate _____ bit):

  pos. \[011010 = \underbrace{00000000}_{\text{8-bit extension}}\]

  neg. \[110011 = \underbrace{00000000}_{\text{8-bit extension}}\]

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):
  
  \[
  \underbrace{111111}_{\text{8-bit truncation}}
  \]

  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. \[00011010 = \underbrace{00000000}_{\text{8-bit truncation}}\]

  neg. \[1110011 = \underbrace{00000000}_{\text{8-bit truncation}}\]

Data Representation

• In C/C++ variables can be of different types and sizes
  – Integer Types (signed and unsigned)

<table>
<thead>
<tr>
<th>C Type</th>
<th>Bytes</th>
<th>Bits</th>
<th>ATmega328</th>
</tr>
</thead>
<tbody>
<tr>
<td>[unsigned] char</td>
<td>1</td>
<td>8</td>
<td>byte</td>
</tr>
<tr>
<td>[unsigned] short [int]</td>
<td>2</td>
<td>16</td>
<td>word</td>
</tr>
<tr>
<td>[unsigned] long [int]</td>
<td>4</td>
<td>32</td>
<td>-1</td>
</tr>
<tr>
<td>[unsigned] long long [int]</td>
<td>8</td>
<td>64</td>
<td>-1</td>
</tr>
<tr>
<td>int</td>
<td>?²</td>
<td>?²</td>
<td>?²</td>
</tr>
</tbody>
</table>

  ²Can emulate but has no single-instruction support
  ³Varies by compiler/machine (avr-gcc: int = 2 bytes, g++ for x86: int = 4-bytes)

  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.

  – Floating Point Types

<table>
<thead>
<tr>
<th>C Type</th>
<th>Bytes</th>
<th>Bits</th>
<th>ATmega328</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>4</td>
<td>32</td>
<td>N/A²</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>64</td>
<td>N/A²</td>
</tr>
</tbody>
</table>
**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 (2_{10} = 10_2)

**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

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

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

**Binary Addition & Subtraction**

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

\[
\begin{array}{cccc}
0 & 1 & 1 & 1 \\
- & 0 & 1 & 0 \\
\hline
0 & 0 & 1 & 1 \\
\end{array}
\]
Binary Addition

Hexadecimal Arithmetic

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

0110 (6) + 0111 (7) = 1101 (13)

4 \text{D}_{16} + \text{B} 5_{16} = ___ 16 1

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)

Original number = +19

Resulting number = -19

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

Take the 2’s complement yields the negative of a number

2’s comp. of 0 is __

Original number = 0

Resulting number = ____

Take the 2’s complement

Original # = -8

Back to original = ____

Negative of -8 is __

(i.e. no positive equivalent, but this is not a huge problem)

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

\[
\begin{align*}
1001 & \quad x = -7 \\
0110 & \quad \text{Bit flip (1’s comp.)} \\
+ & \quad \text{Add 1} \\
0111 & \quad -x = -(\text{-7}) = +7
\end{align*}
\]

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

\[
\begin{array}{c}
0011 \\
+ 0010 \\
\end{array}
\begin{array}{c}
1101 \\
+ 0010 \\
\end{array}
\]

\[
\begin{array}{c}
0011 \\
+ 1110 \\
\end{array}
\begin{array}{c}
1101 \\
+ 1110 \\
\end{array}
\]

Unsigned and Signed Addition

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

  \[
  \begin{array}{c}
  0011 \\
  + 0010 \\
  \end{array}
  \begin{array}{c}
  1101 (+3) \\
  + 0010 (+2) \\
  \end{array}
  \begin{array}{c}
  - 0010 (+2) \\
  - 1110 (-2) \\
  \end{array}
  \]

2’s Complement Subtraction

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

\[
\begin{array}{c}
0011 (+3) \\
- 0010 (+2) \\
\end{array}
\begin{array}{c}
1101 (-3) \\
- 1110 (-2) \\
\end{array}
\]
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</td>
<td>(12) (−4)</td>
</tr>
<tr>
<td>− 0010</td>
<td>(2) (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
--- | --- | --- | ---
0110 | 0111 | 0101 | 0111
0100 | 1000 | 1011 |
FLOATING POINT

Floating Point

- Used to represent very small numbers (fractions) and very large numbers
  - Avogadro’s Number: $+6.0247 \times 10^{23}$
  - Planck’s Constant: $+6.6254 \times 10^{-27}$
  - 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
  - $\pm D.DDD \times 10^{\pm exp}$
- Floating Point representation uses the following form
  - _______________
  - 3 Fields: ____ _________ __________
    (also called _____________________)
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 x 10^{±38}

- 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 x 10^{±308}

Floating Point vs. Fixed Point

- Single Precision (32-bits) Equivalent Decimal Range:
  - 7 significant decimal digits * 10^{±38}
  - Compare that to 32-bit signed integer where we can represent ±2 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 * 10^{±308}

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 $E' = \text{stored exponent code}$ and $E = \text{true exponent value}$
- For single-precision: $E' = E + 127$
  - $2^1 \Rightarrow E = 1, E' = 128_{10} = 10000000_2$
- For double-precision: $E' = E + 1023$
  - $2^2 \Rightarrow E = -2, E' = 1021_{10} = 0111111101_2$

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. \(2^7 = 128\) \(2^1 = 2\)

|   | 1 | 1000 0010 | 110 0110 0000 0000 0000 0000 |

2. \(+0.6875 = +0.1011\)