CSE 220:

Systems Fundamentals I

Unit 2:

Number Systems

Digital Abstraction

- Most physical variables are **continuous**
  - Temperature
  - Voltage on a wire
  - Frequency of an oscillation
  - Position of a mass
- Although voltage, charge and other electrical quantities are continuous in nature, modern computers are all digital and work with **discrete values**
- The continuous nature of electricity is "abstracted away" and we consider only "high" and "low" voltage, or the "presence" and "absence" of electric charge: i.e., 1s and 0s (**bits**: **b**inary dig**it**s)

Binary Digits

- So, all data is ultimately represented in a computer in terms of binary digits
  - Each bit is either a 0 or a 1
- Groups of bits represent larger values
  - 1101 1110 1010 1101 1011 1110 1110 1111
  - We usually write spaces between groups of four or eight bits, depending on the situation. More on this soon.

Positional Notation

- The scheme we use in modern times for representing numbers is called **positional notation**
- The position of a digit determines how much it contributes to the number's value
- With **decimal** (**base-10** or **radix-10**), the **place-values** are powers of 10:

  …, 10^3, 10^2, 10^1, 10^0, 10^{-1}, 10^{-2}, 10^{-3}, …

  …, 1000s, 100s, 10s, 1s, $^1/_{10}$s, $^1/_{100}$s, $^1/_{1000}$s, …
- 642.15 really means $(6 \times 10^2) + (4 \times 10^1) + (2 \times 10^0) + (1 \times 10^{-1}) + (5 \times 10^{-2})$

Positional Notation

- More generally, in base-10 notation, the sequence of digits $d_k d_{k-1} \ldots d_2 d_1 d_0$ stands for the **polynomial expansion** $(d_k \times 10^k) + (d_{k-1} \times 10^{k-1}) + \cdots + (d_2 \times 10^2) + (d_1 \times 10^1) + (d_0 \times 10^0)$
- We can generalize this to arbitrary bases. In radix-*k* we use *k* distinct symbols (digits), and the place-values are powers of *k.*
- Radix-2 (binary) notation example:
$$10101_2 = (1 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)$$
$$= 16 + 4 + 1$$
$$= 21_{10}$$
- When working with multiple radixes, **always** include a subscript to identify the radix!

Positional Notation

- In some circumstances it's more natural to write numbers in base 8, called **octal**, or base 16, called **hexadecimal**
- With octal there are 8 digits: 0, 1, 2, 3, 4, 5, 6, 7 with place-values that are powers of 8:

  …, 8^3, 8^2, 8^1, 8^0, 8^{-1}, 8^{-2}, 8^{-3}, …

  …, 256s, 64s, 8s, 1s, $^1/_8$s, $^1/_{64}$s, $^1/_{256}$s, …
- $376_8 = (3 \times 8^2) + (7 \times 8^1) + (6 \times 8^0)$
$$= 192 + 56 + 6$$
$$= 254_{10}$$
- With hexadecimal we should have 16 digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, where the letters A through F represent ten through fifteen, respectively

Positional Notation

- Radix-16 (**hexadecimal**) notation example:
$$3E0_{16} = (3 \times 16^2) + (14 \times 16^1) + (0 \times 16^0)$$
$$= 768 + 224 + 0$$
$$= 992_{10}$$
- Radix-*k* fractions involve negative powers of *k*
$$10.011_2 = (1 \times 2^1) + (0 \times 2^0) + (0 \times 2^{-1}) + (1 \times 2^{-2}) + (1 \times 2^{-3})$$
$$= 2 + 0.25 + 0.125$$
$$= 2.375$$
- Numbers with terminating decimal representations might not have terminating representations in other radixes.
- For example: $0.2_{10} = 0.\overline{0011}_2$

Binary, Octal and Hexadecimal

Binary	Octal	Hex	Decimal
0	0	0	0
1	1	1	1
10	2	2	2
11	3	3	3
100	4	4	4
101	5	5	5
110	6	6	6
111	7	7	7
1000	10	8	8
1001	11	9	9
1010	12	A	10
1011	13	B	11
1100	14	C	12
1101	15	D	13
1110	16	E	14
1111	17	F	15

- Note that the digits "10" represent the base of a number in its own base
- 10_2 is two
- 10_8 is eight
- 10_{16} is sixteen
- 10_{10} is ten
- Why does it work out like this?

Binary → Decimal Conversion

- Algorithm: Start with initial value of 0. Process bits from *left-to-right order*, i.e., from **most significant bit** (msb) to **least significant bit** (lsb)
  - Double the value from previous step.
  - Add the next bit value.

Binary → Decimal Example

- Convert 1011100_2 to decimal (going left to right)

  1011100_2

  1

  $1 \times 2 + 0 = 2$

  $2 \times 2 + 1 = 5$

  $5 \times 2 + 1 = 11$

  $11 \times 2 + 1 = 23$

  $23 \times 2 + 0 = 46$

  $46 \times 2 + 0 = 92$

- This algorithm will work for any radix. Just multiply by the radix after each step.

Binary → Decimal Example

- Convert 101011101_2 to decimal

Octal → Decimal Example

- Convert 417_8 to decimal

Decimal → Binary Conversion

- Algorithm: repeatedly divide the decimal representation by 2, writing the remainders in *right-to-left order*, i.e., from **least significant bit** (lsb) to **most significant bit** (msb)
- Continue dividing until the quotient is 0

Decimal → Binary Example

- Convert 123_{10} to binary
- 123 / 2 = 61 rem. 1
- 61 / 2 = 30 rem. 1
- 30 / 2 = 15 rem. 0
- 15 / 2 = 7 rem. 1
- 7 / 2 = 3 rem. 1
- 3 / 2 = 1 rem. 1
- 1 / 2 = 0 rem. 1
- Answer: 1111011_2

Decimal → Binary Example

- Convert 1528_{10} to binary

Decimal → Hexadecimal Example

- The decimal-to-hexadecimal conversion works largely in the same way, but with division by 16
- 3241 / 16 = 202 rem. 9
- 202 / 16 = 12 rem. 10
- 12 / 16 = 0 rem. 12
- Answer: $CA9_{16}$

- This algorithm will also work for any radix. Just divide by the radix after each step.

Decimal → Octal Example

- Convert 1528_{10} to octal

Decimal Fractions → Binary

- Algorithm: generate the bits in *left-to-right order*, starting from the radix point:
  - Multiply the decimal value by 2. If the product is greater than 1, the next bit is 1. Otherwise, the next bit is 0.
  - Drop the integer part to get a value less than 1.
  - Continue until 0 is reached (a terminating expansion) or a pattern of digits repeats (a non-terminating expansion)
- The resulting representation is called **fixed-point format**

Decimal Frac. → Binary Example

- Convert 0.4_{10} to binary
- $0.4 \times 2 = 0.8$ $\quad 0.8 < 1$, so write a 0 $\quad \cong 0.0$
- $0.8 \times 2 = 1.6$ $\quad 1.6 \geq 1$, so write a 1 $\quad \cong 0.01$
  - Drop the integer part
- $0.6 \times 2 = 1.2$ $\quad 1.2 \geq 1$, so write a 1 $\quad \cong 0.011$
  - Drop the integer part
- $0.2 \times 2 = 0.4$ $\quad 0.4 < 1$, so write a 0 $\quad \cong 0.0110$
- Since we arrived at a decimal fraction we have already seen, the pattern will repeat
- Final answer: $0.\overline{0110}_2$

Decimal Frac. → Binary Example

- Convert 13.85_{10} to binary

Bin ↔ Oct ↔ Hex Conversion

- Because 8 and 16 are powers of 2, converting between bases 2 and 8, and between bases 2 and 16 is very simple
- Binary → Octal
  - Working *right-to-left*, take bits in groups of 3, converting the groups into octal digits (Why 3? Because $2^3 = 8.$)
  - Example: $10110101_2 \rightarrow 10\ 110\ 101 \rightarrow 265_8$
- Binary → Hexadecimal
  - Working *right-to-left*, take bits in groups of 4 (a **nibble**), converting the groups into hexadecimal digits ($2^4 = 16$)
  - Example: $10110111101_2 \rightarrow 101\ 1011\ 1101 \rightarrow 5BD_{16}$
- Two nibbles = eight bits = one **byte**

Bin ↔ Oct ↔ Hex Conversion

- Octal → Binary
  - Working *left-to-right*, replace each octal digit with its 3-bit binary equivalent
  - Example: $250_8 \rightarrow 010\ 101\ 000 \rightarrow 10101000_2$
- Hexadecimal → Binary
  - Working *left-to-right*, replace each hexadecimal digit with its 4-bit binary equivalent
  - Example: $3FE5_{16} \rightarrow 0011\ 1111\ 1110\ 0101 \rightarrow$ 11111111100101_2
- Conversion between octal and hexadecimal is not so straightforward. It's easiest to convert the given representation into binary and then into the desired base.

Bin ↔ Oct ↔ Hex Conversion

- Convert $7BA3.BC4_{16}$ to octal

- Convert 2713_8 to hexadecimal

Base 2^M ↔ Base 2^N

- General algorithm for converting from base 2^M to 2^N
  - Write out the binary equivalent of the base 2^M number and, going right-to-left, form groups of *N* bits
- Convert 2713_8 to base 4
- Because $4 = 2^2$, we will take bits in pairs (right-to-left)

- General algorithm for converting from base 2^M to 2^N
  - Write out the binary equivalent of the base 2^M number and, going right-to-left, form groups of *N* bits

General Base Conversions

- What if we want to convert between two bases other than the ones we've studied?
- Generally it's easiest to convert the given representation into decimal, and then from decimal into the desired base
- Example: convert 215_7 to base 5
- $215_7 = (2 \times 7^2) + (1 \times 7^1) + (5 \times 7^0)$
  $= 98 + 7 + 5$
  $= 110_{10}$
- 110 / 5 = 22 rem. 0
- 22 / 5 = 4 rem. 2
- 4 / 5 = 0 rem. 4
- Answer: $215_7 = 420_5$

Integer Encodings: Unsigned

- Now we'll start to see how integers are encoded in hardware
- In **unsigned integer** encodings, the numbers 0, 1, 2, …, $2^N - 1$ are typically encoded as follows:

0	→	000 … 00
1	→	000 … 01
2	→	000 … 10
3	→	000 … 11
…		
$2^N - 1$	→	111 … 11

Integer Encodings: Unsigned

- Binary arithmetic with unsigned integers (particularly, addition), is done in the usual way
- With decimal addition we can have *carried* values:

Carry values: 11

$$\begin{array}{lr} & 74_{10} \\ & \underline{+89_{10}} \\ \textit{Sum}: & 163_{10} \end{array}$$

- The same thing can happen with binary addition

Carry values: 1 1 1

$$\begin{array}{lr} & 0\;1\;0\;0\;1\;0\;1\;0_2 \\ & \underline{+\;0\;1\;0\;1\;1\;0\;0\;1_2} \\ \textit{Sum}: & 1\;0\;1\;0\;0\;0\;1\;1_2 \end{array}$$

Binary Addition Example

- Add the following two 8-bit binary numbers:

$$\begin{array}{r} 1\;1\;0\;1\;0\;1\;1\;1_2 \\ \underline{+\;0\;1\;0\;1\;0\;0\;0\;1_2} \end{array}$$

- We had an **overflow**.
- In **fixed-precision** integer arithmetic, it is possible for an arithmetic operation, such as addition, to result in an overflow. The leftmost carry value is dropped, resulting in an incorrect sum.
- The programmer must be aware of the range of representable values to avoid unintentional overflow

Integer Encodings: Signed

- There are several different signed integer encodings which permit the representation of both positive and negative numbers.
- The hardware designer chooses between the encodings to make the arithmetic hardware simpler or more efficient for certain operations.
- Sometimes the programmer needs to be aware of what encoding is being used:
  - To format numbers for input or output.
  - To understand the range of representable values and the conditions under which overflow can occur.

Sign/Magnitude Numbers

- In N-bit **sign/magnitude encoding**, the most significant bit (leftmost bit) is used as a **sign bit** (0 = “positive”, 1 = “negative”), and the remaining $N-1$ bits represent the **magnitude** (absolute value) of the number, as in the unsigned scheme.
- Range: $\left[-2^{N-1}+1, 2^{N-1}-1\right]$
- Example (8-bit precision):
  - $+75 \Rightarrow 01001011$
  - $-15 \Rightarrow 10001111$
- Problems with sign/magnitude encoding:
  - Two encodings for zero: $+0$ and -0
  - Subtraction is somewhat complicated

One’s Complement Encoding

- The N-bit **one’s complement** encoding represents integers in the range $\left[-\left(2^{N-1}-1\right), 2^{N-1}-1\right]$ as follows:

0	$\rightarrow$	$000\ldots00$	-0	$\rightarrow$	$111\ldots11$
1	$\rightarrow$	$000\ldots01$	-1	$\rightarrow$	$111\ldots10$
2	$\rightarrow$	$000\ldots10$	-2	$\rightarrow$	$111\ldots01$
3	$\rightarrow$	$000\ldots11$	-3	$\rightarrow$	$111\ldots00$
…			…		
$2^{N-1}-1$	$\rightarrow$	$011\ldots11$	$-\left(2^{N-1}-1\right)$	$\rightarrow$	$100\ldots00$

- To obtain the negative of a value, complement (“flip”) all the bits:
  - change all 0s to 1s and all 1s to 0s

One’s Complement Encoding

- Addition may require an “end around carry”:

```
   0 0 0 0 0 1 0 1       5
 + 1 1 1 1 1 1 0 0      -3
 -------------------------
 1 0 0 0 0 0 0 0 1
 +               1
 -------------------------
   0 0 0 0 0 0 1 0       2
```

One’s Complement Example

- Perform the computation $-28 - 37$ in 8-bit one's complement and convert the result to decimal.

One’s Complement Issue

- The presence of two representations for zero adds complexity to a circuit that implements addition
- So we can use a different encoding called **two’s complement** that avoids this problem

Two’s Complement Encoding

- The N-bit **two’s complement** encoding represents integers in the range $\left[-2^{N-1}, 2^{N-1}-1\right]$ as follows:

0	$\rightarrow$	$000 \ldots 00$			
1	$\rightarrow$	$000 \ldots 01$	-1	$\rightarrow$	$111 \ldots 11$
2	$\rightarrow$	$000 \ldots 10$	-2	$\rightarrow$	$111 \ldots 10$
3	$\rightarrow$	$000 \ldots 11$	-3	$\rightarrow$	$111 \ldots 01$
…			…		
$2^{N-1}-1$	$\rightarrow$	$011 \ldots 11$	-2^{N-1}	$\rightarrow$	$100 \ldots 00$

Two’s Complement Encoding

- To negate a number in two’s complement encoding, use one of the following (equivalent) rules:

1. Complement all the bits and increment the result.
   Example: negate 3
   $00000011 \rightarrow 11111100 \rightarrow 11111101$
2. Complement all the bits to the left of the rightmost 1.
   Example: negate 9
   $00001001 \rightarrow 11110111$

- The value 2^{N-1} has no representation in N-bit two’s complement notation, but -2^{N-1} does.

Two’s Complement

- What is the 8-bit two’s complement representation of −99?

- What is the decimal equivalent of the two’s complement number 11001010?

Two’s Complement of Zero

- What happens if we take the two’s complement of the number 0 written as an 8-bit number?

The Two’s Complement Wheel

- Unlike mathematical integers, which inhabit a number line, in computer arithmetic the numbers lie on a circle.
- An overflow occurs when crossing the boundary between the greatest representable positive number and the least representable negative number.

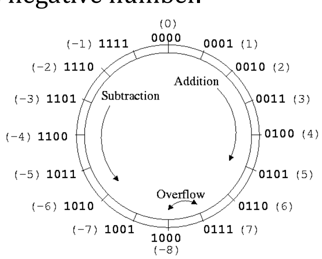
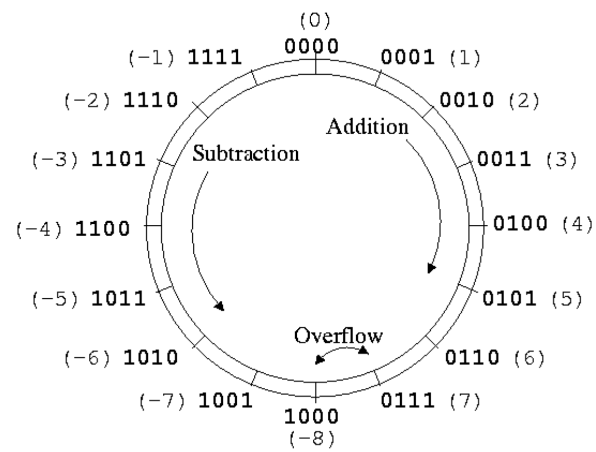


Image: users.dickinson.edu/~braught/courses/cs251f02/classes/notes07.html

The Two’s Complement Wheel



Two's Complement Encoding

- The leftmost bit is the sign bit, which tells whether the number is positive (0) or negative (1).
- The rightmost bit tells whether the number is even (0) or odd (1).
- Multiplication by 2 is accomplished by a **left shift** (introduce 0 at right), dropping the msb:

$-3 \rightarrow -6$

$11111101 \rightarrow 11111010$

- Division by 2 is accomplished by a **right arithmetic shift** (replicate sign bit at left), dropping the lsb:

$-6 \rightarrow -3$

$11111010 \rightarrow 11111101$

Overflow in Two's Complement

- In signed arithmetic using two's complement encoding, there is a possibility of overflow, when the correct result is "too large" or "too small" to be represented.
- Overflow cannot occur when adding numbers of opposite sign (or when subtracting numbers of the same sign).
- Overflow can occur when adding numbers of the same sign (or subtracting numbers of opposite sign).

```
  0 1 1 1 1 1 1 0      126
+ 0 0 0 0 0 0 1 1    +   3
-----------------------------
  1 0 0 0 0 0 0 1     -127
```

- The overflow can be detected by noticing that the sum has opposite sign from the addends.

Sign Extension

- When a two's complement number is extended to more bits, the sign bit must be copied into the most significant bit positions
  - This is called **sign-extension**
- Examples: rewrite each of the following numbers (3 and − 3) in 8-bit two's complement
  - 3 is 0011 → 00000011
  - −3 is 1101 → 11111101

Excess-*k* Encoding

- **Excess-*k* encoding** is another signed integer encoding that is important due to its use in representing real numbers
- In excess-k, also called **bias-*k***, integer i is represented by the unsigned encoding of $i + k$.
  - For example, in excess-127:
  - 3 is represented as $3 + 127 = 130 \rightarrow 10000010$
  - -5 is represented as $-5 + 127 = 122 \rightarrow 01111010$
- We note that both positive and negative numbers are represented as unsigned values. In excess-127:
  - −127 is 00000000 and +128 is 11111111
- The advantage of excess-*k* notation is that ordering of positive and negative numbers is preserved, which makes comparing two values (e.g., <, >) very straightforward

Comparison of Integer Encodings

N decimal	N binary	−N sign/mag	−N 1's comp.	−N 2's comp.	−N bias-127
1	00000001	10000001	11111110	11111111	01111110
2	00000010	10000010	11111101	11111110	01111101
3	00000011	10000011	11111100	11111101	01111100
4	00000100	10000100	11111011	11111100	01111011
5	00000101	10000101	11111010	11111011	01111010
10	00001010	10001010	11110101	11110110	01110101
50	00110010	10110010	11001101	11001110	01001101
90	01011010	11011010	10100101	10100101	00100101
100	01100100	11100100	10011011	10011100	00011011
127	01111111	11111111	10000000	10000001	00000000
128	10000000	N/A	N/A	10000000	N/A

What About Real Numbers?

- We did some base conversions involving real numbers but haven't seen yet how they can be represented in a computer
- A big disadvantage of the so-called **fixed-point format** or **fixed-precision encoding** of such numbers is that they have a very limited "dynamic range"
  - This forma can't represent very large numbers (e.g., 2^{70}) or very small numbers (e.g., 2^{-17})
- These numbers are called "fixed point" because the decimal point (or **binary point**) is fixed, which limits accuracy
- On the other hand, they give *exact* answers (i.e., no rounding errors) as long as there is no overflow

Floating-Point Format

- Because fractions can have non-terminating representations and/or might require many digits to be represented exactly, usually real numbers can only be approximately represented in a computer
  - We have to tolerate a certain amount of **representational error**
- The industry standard way used to approximate real numbers is called **floating-point format**
  - **IEEE 754 floating-point standard**
- In this scheme the binary point is allowed to "float" (i.e., be repositioned) in order to give as accurate an approximation as possible

IEEE 754 Floating-Point Standard

- The IEEE 754 standard species floating-point representations of numbers and also arithmetic operations on these representations
- IEEE 754 is essentially a form of **scientific notation**, but written in binary: $\pm 2^{exponent} \times fraction$
- This format can be encoded using three fields: a **sign bit** (s), an **exponent** (e) and a **fraction** (f), sometimes called the **mantissa**
- IEEE 754 **single-precision format** requires 32 bits and provides about 7 decimal digits of accuracy
- IEEE 754 **double-precision format** requires 64 bits and provides about 15 decimal digits of accuracy

IEEE 754 Floating-Point Standard

- 32-bit version: **single precision**

1 bit	8 bits	23 bits
sign	exponent	fraction (mantissa)

- 64-bit version: **double precision**

1 bit	11 bits	52 bits
sign	exponent	fraction (mantissa)

- Sign bit: 0 (positive) or 1 (negative)
- Exponent: stored in excess-127 for the 32-bit version
  - Excess-1023 for the 64-bit version
- Fraction: contains the digits to the right of the binary point
  - **Normalized**: the digit to the left of the point is always 1, and is not represented, giving us one bit of precision "for free"

IEEE 754 to Decimal

- Decimal value of a IEEE 754 floating point encoding is given by the formula: $(-1)^s \times 2^{e-bi} \times (1+f)$
  where:
  - s is the sign bit (0/1).
  - e is the decimal value of the exponent field
  - bias is 127 for single-precision, 1023 for double-precision.
  - f is the decimal value of the fraction field (regarded as a binary fraction)

IEEE 754 to Decimal Example

- What decimal value has the following IEEE 754 encoding?
  10111110011000000000000000000000
  1 01111100 11000000000000000000000

Decimal to IEEE 754 Example

- Encode 13.4_{10} in 32-bit IEEE 754 floating-point format

IEEE 754 Special Values

- The smallest 000…0 and largest 111…1 exponents are reserved for the encoding of special values:
  - Zero (two encodings):
    - $s = 0$ or $1, e = 000 \dots 0, f = 000 \dots 0$
  - Infinity:
    - $+\infty: s = 0, e = 111 \dots 1, f = 000 \dots 0$
    - $-\infty: s = 1, e = 111 \dots 1, f = 000 \dots 0$
  - NaN (not a number):
    - $s = 0$ or $1, e = 111 \dots 1, f =$ non-zero
    - Can result from division by zero, $\sqrt{-1}$, $\log(-5)$, etc.

IEEE 754 Format Summary

Property	Single-Precision	Double-Precision
Bits in Sign	1	1
Bits in Exponent	8	11
Bits in Fraction	23	52
Total Bits	32	64
Exponent Encoding	excess-127	excess-1023
Exponent Range	-126 to 127	-1022 to 1023
Decimal Range	$\cong 10^{-38}$ to 10^{38}	$\cong 10^{-308}$ to 10^{308}

Floating-Point Limitations

- There are $\sim 2^{32}$ different values that can be represented in single-precision floating point.
- This is the same as the number of values that can be represented using 32-bit integer encodings.
- Many values (even integers) do not have floating-point representations:
  - Examples: 33554431_{10} and 0.33554431_{10}
- Try assigning these to a `float` variable in Java and then printing them out.
- Caution: Results of floating point calculations are not exact.
- Never use floating point when exact results are essential or in equality checks, such as in conditional statements