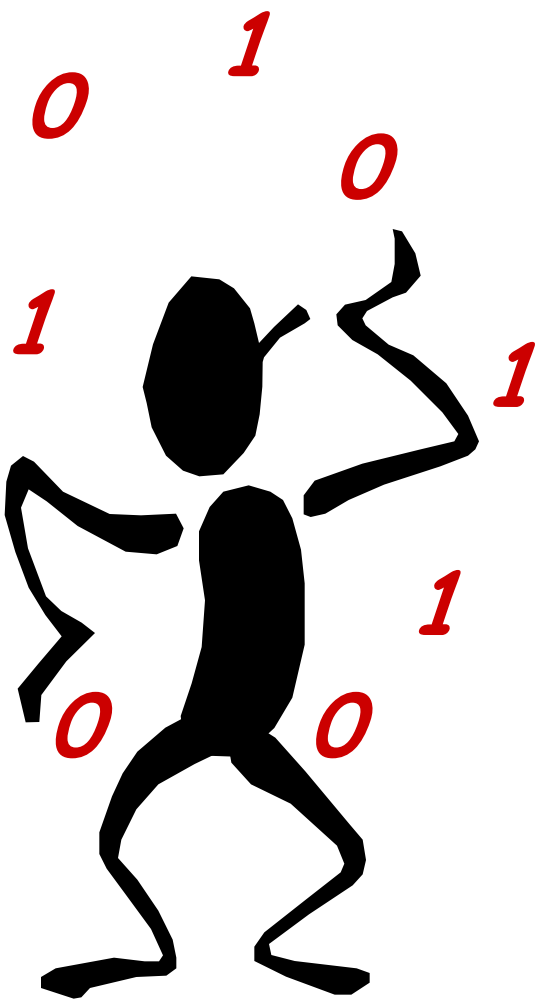
# Representing Information




## "Bit Juggling"

- Representing information using bits
- Number representations
- Some other bits

· Chapters 1 and 2.3,2.4

# Motivations


- Computers Process Information
- Information is measured in bits
- By virtue of containing only “switches” and “wires” digital computer technologies use a binary representation of bits
- How do we use/interpret bits?
- We need standards of representations for
  - Letters
  - Numbers
  - Colors/pixels
  - Music
  - Etc.

# Encoding

- **Encoding describes the process of *assigning representations to information***
- **Choosing an appropriate and efficient encoding is a real engineering challenge (and an art)**
- **Impacts design at many levels**
  - **Mechanism (devices, # of components used)**
  - **Efficiency (bits used)**
  - **Reliability (noise)**
  - **Security (encryption)**

# Fixed-Length Encodings

**If all choices are equally likely (or we have no reason to expect otherwise), then a fixed-length code is often used. Such a code should use at least enough bits to represent the information content.**

**ex. Decimal digits 10 = {0,1,2,3,4,5,6,7,8,9}**

**4-bit BCD (binary code decimal)**

$$\log_2(10) = 3.322 < 4 bits$$

**ex. ~84 English characters = {A-Z (26), a-z (26), 0-9 (10), punctuation (8), math (9), financial (5)}**

**7-bit ASCII (American Standard Code for Information Interchange)**

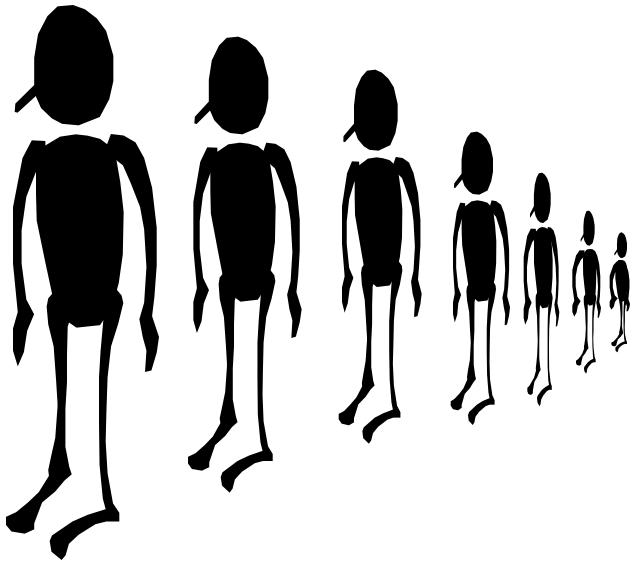
$$\log_2(84) = 6.392 < 7 bits$$

# Encoding Positive Integers

**It is straightforward to encode positive integers as a sequence of bits. Each bit is assigned a weight. Ordered from right to left, these weights are increasing powers of 2. The value of an n-bit number encoded in this fashion is given by the following formula:**

$$v = \sum_{i=0}^{n-1} 2^i b_i$$

| $2^{11}$ | $2^{10}$ | $2^9$ | $2^8$ | $2^7$ | $2^6$ | $2^5$ | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |



$2^4 = 16$

$+ 2^6 = 64$

$+ 2^7 = 128$

$+ 2^8 = 256$

$+ 2^9 = 512$

$+ 2^{10} = 1024$

$2000_{10}$

# Octal

**Often it is convenient to cluster groups of bits together for a more compact representation. The clustering of 3 bits is called Octal. Octal is not that common today.**

$$v = \sum_{i=0}^{n-1} 8^i d_i$$

| $2^{11}$ | $2^{10}$ | $2^9$ | $2^8$ | $2^7$ | $2^6$ | $2^5$ | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |

= $2000_{10}$

3 7 2 0

**03720**


**Octal - base 8**

000 - 0
001 - 1
010 - 2
011 - 3
100 - 4
101 - 5
110 - 6
111 - 7

$0*8^0 = 0$
$+ 2*8^1 = 16$
$+ 7*8^2 = 448$
$+ 3*8^3 = 1536$
$2000_{10}$

# Hex

**Clusters of 4 bits are used most frequently. This representation is called hexadecimal. The hexadecimal digits include 0-9, and A-F, and each digit position represents a power of 16.**

$$v = \sum_{i=0}^{n-1} 16^i d_i$$

| $2^{11}$ | $2^{10}$ | $2^9$ | $2^8$ | $2^7$ | $2^6$ | $2^5$ | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | = $2000_{10}$ |
| | | | 7 | | | | d | | | | 0 | |

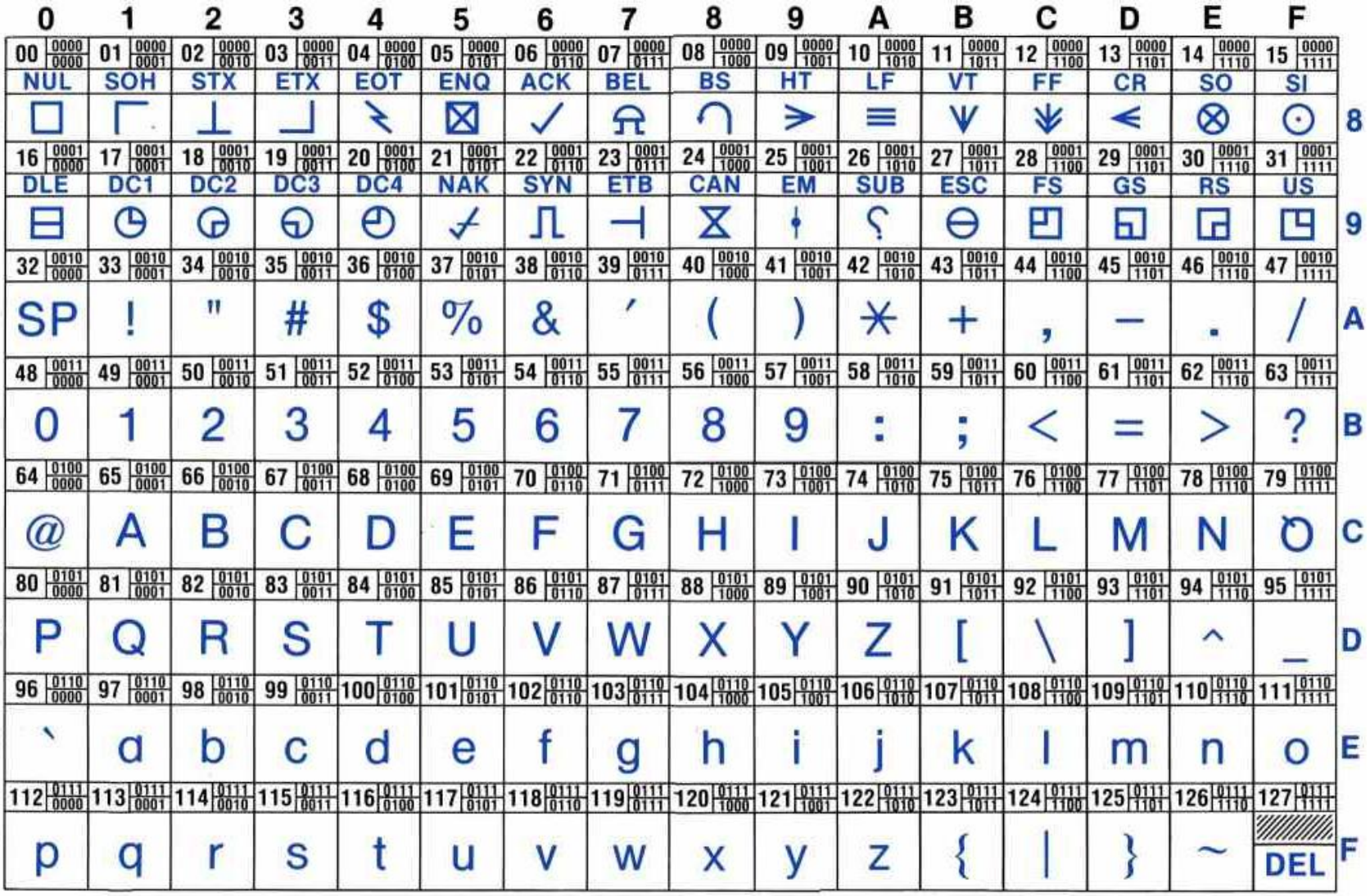
**0x7d0**

**Hexadecimal - base 16**

| | |
|---|---|
| 0000 - 0 | 1000 - 8 |
| 0001 - 1 | 1001 - 9 |
| 0010 - 2 | 1010 - a |
| 0011 - 3 | 1011 - b |
| 0100 - 4 | 1100 - c |
| 0101 - 5 | 1101 - d |
| 0110 - 6 | 1110 - e |
| 0111 - 7 | 1111 - f |

$$\begin{aligned} 0*16^0 &= 0 \\ +\ 13*16^1 &= 208 \\ +\ 7*16^2 &= 1792 \\ &\hline \\ &= 2000_{10} \end{aligned}$$

# Encoding Text in ASCII



# Unicode

- **ASCII is biased towards western languages. English in particular.**
- **There are, in fact, many more than 256 characters in common use:**

  **â, m, ö, ñ, è, ¥, 揗, 敕, 흵, カ, א, ג, ж, ŝ, ค**

- **Unicode is a worldwide standard that supports all languages, special characters, classic, and arcane**
- **Several encoding variants 16-bit (UTF-8)**

| | | | | |
|---|---|---|---|---|
| **ASCII equiv range:** | 0xxxxxxx | | | |
| **Lower 11-bits of 16-bit Unicode** | 110yyyyx | 10xxxxxx | | |
| **16-bit Unicode** | 1110zzzz | 10zyyyyx | 10xxxxxx | |
| | 11110www | 10wwzzzz | 10zyyyyx | 10xxxxxx |

# Some Bit Tricks

- **You are going to have to get accustomed to working in binary. It will be helpful throughout your career as a computer scientist.**
- **Here are some helpful guides**

1. **Memorize the first 10 powers of 2**

| | |
|---|---|
| $2^0 = 1$ | $2^5 = 32$ |
| $2^1 = 2$ | $2^6 = 64$ |
| $2^2 = 4$ | $2^7 = 128$ |
| $2^3 = 8$ | $2^8 = 256$ |
| $2^4 = 16$ | $2^9 = 512$ |

# More Tricks with Bits

1. **Memorize the first 10 powers of 2**

2. **Memorize the prefixes for powers of 2 that are multiples of 10**

**$2^{10}$ = Kilo (1024)**
**$2^{20}$ = Mega (1024*1024)**
**$2^{30}$ = Giga (1024*1024*1024)**
**$2^{40}$ = Tera (1024*1024*1024*1024)**
**$2^{50}$ = Peta (1024*1024*1024 *1024*1024)**
**$2^{60}$ = Exa (1024*1024*1024*1024*1024*1024)**

# Even More Tricks with Bits

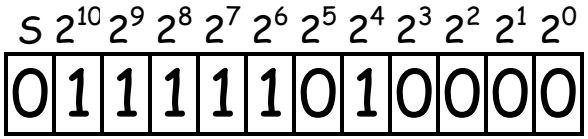
| 01 | 0000000011 | 0000001100 | 0000101000 |
|---|---|---|---|

1. When you convert a binary number to decimal, first break it down into clusters of 10 bits.
2. Then compute the value of the leftmost remaining bits (1) find the appropriate prefix (GIGA) (Often this is sufficient)
3. Compute the value of and add in each remaining 10-bit cluster

# Signed-Number Representations

- **There are also schemes for representing signed integers with bits. One obvious method is to encode the sign of the integer using one bit. Conventionally, the most significant bit is used for the sign. This encoding for signed integers is called the SIGNED MAGNITUDE representation.**

$$v = -1^{S} \sum_{i=0}^{n-2} 2^{i} b_i$$

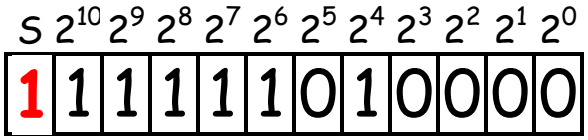



**2000**

# Signed-Number Representations

- **There are also schemes for representing signed integers with bits. One obvious method is to encode the sign of the integer using one bit. Conventionally, the most significant bit is used for the sign. This encoding for signed integers is called the SIGNED MAGNITUDE representation.**

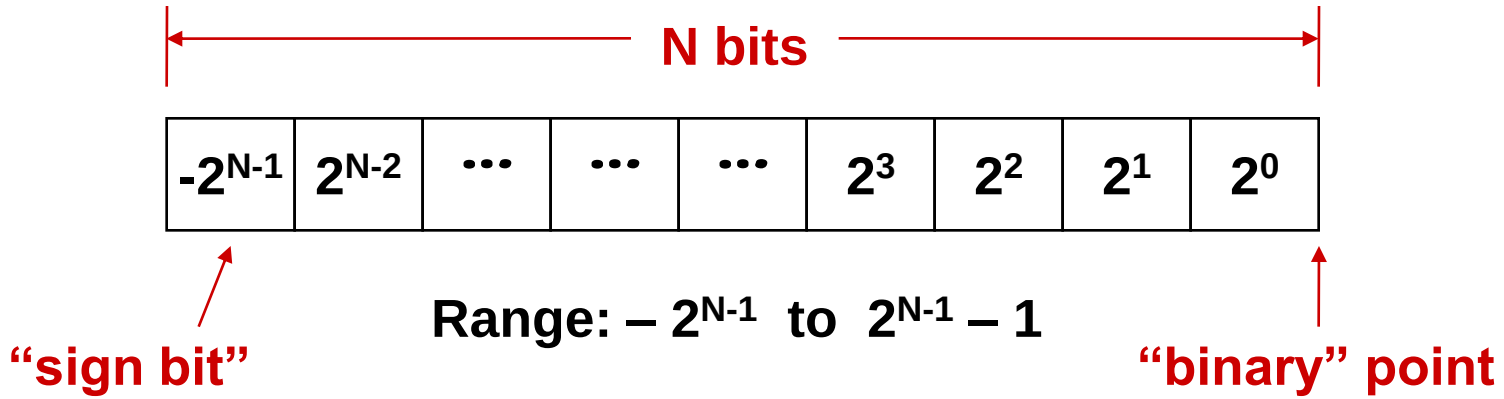
$$v = -1^{S} \sum_{i=0}^{n-2} 2^{i} b_i$$




**-2000**

- The Good: Easy to negate, find absolute value
- The Bad:
  - Add/subtract is complicated; depends on the signs
  - Two different ways of representing a 0
  - It is not used that frequently in practice

# 2's Complement Integers




**The 2's complement representation for signed integers is the most commonly used signed-integer representation. It is a simple modification of unsigned integers where the most significant bit is considered negative.**

$$v = -2^{n-1} b_{n-1} + \sum_{i=0}^{n-2} 2^i b_i$$

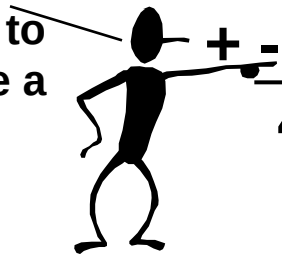
**8-bit 2's complement example:**

$$11010110 = -2^7 + 2^6 + 2^4 + 2^2 + 2^1$$
$$= -128 + 64 + 16 + 4 + 2 = -42$$

# Why 2's Complement?

**If we use a two's complement representation for signed integers, the same binary addition mod $2^n$ procedure will work for adding positive and negative numbers (don't need separate subtraction rules). The same procedure will also handle unsigned numbers!**

**When using signed magnitude representations, adding a negative value really means to subtract a positive value. However, in 2's complement, adding is adding regardless of sign. In fact, you NEVER need to subtract when you use a 2's complement representation.**



**Example:**

$$\begin{array}{rrcl} & 55_{10} & = & 00110111_2 \\ + & 10_{10} & = & 00001010_2 \\ \hline & 65_{10} & = & 01000001_2 \end{array}$$

$$\begin{array}{rrcl} & 55_{10} & = & 00110111_2 \\ + & -10_{10} & = & 11110110_2 \\ \hline & 45_{10} & = & 100101101_2 \end{array}$$

# 2's Complement Tricks

- **Negation – changing the sign of a number**
  - **First complement every bit (i.e. 1 $\rightarrow$ 0, 0 $\rightarrow$ 1)**
  - **Add 1**

  **Example: 20 = 00010100, -20 = 11101011 + 1 = 11101100**

- **Sign-Extension – aligning different sized 2's complement integers**
  - **Simply copy the sign bit into higher positions**

# CLASS EXERCISE

## 10's-complement Arithmetic
### (You'll never need to borrow again)

**Step 1) Write down two 3-digit numbers that you want to subtract**

**Step 2) Form the 9's-complement of each digit in the second number (the subtrahend)**

**Step 3) Add 1 to it (the subtrahend)**

**Step 4) Add this number to the first**

**Step 5) If your result was less than 1000,**
**form the 9's complement again and add 1**
**and remember your result is negative**
**else**
**subtract 1000**

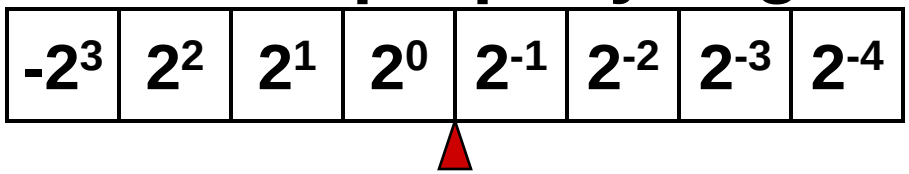
**Helpful Table of the 9's complement for each digit**

| |
|---|
| $0 \rightarrow 9$ |
| $1 \rightarrow 8$ |
| $2 \rightarrow 7$ |
| $3 \rightarrow 6$ |
| $4 \rightarrow 5$ |
| $5 \rightarrow 4$ |
| $6 \rightarrow 3$ |
| $7 \rightarrow 2$ |
| $8 \rightarrow 1$ |
| $9 \rightarrow 0$ |

**What did you get? Why weren't you taught to subtract this way?**

# Fixed-Point Numbers

**By moving the implicit location of the "binary" point, we can represent signed fractions too. This has no effect on how operations are performed, assuming that the operands are properly aligned.**




$1101.0110 = -2^3 + 2^2 + 2^0 + 2^{-2} + 2^{-3}$

$= -8 + 4 + 1 + 0.25 + 0.125$

$= -2.625$

**OR**

$1101.0110 = -42 * 2^{-4} = -42/16 = -2.625$

# Repeated Binary Fractions

**Not all fractions can be represented exactly using a finite representation. You've seen this before in decimal notation where the fraction 1/3 (among others) requires an infinite number of digits to represent (0.3333…).**

**In Binary, a great many fractions that you've grown attached to require an infinite number of bits to represent exactly.**

**EX:** $1 / 10 = 0.1_{10} = .0001100110011\ldots_{2}$

$1 / 5 = 0.2_{10} = .001100110011\ldots_{2} = 0.333\ldots_{16}$

# Bias Notation

- **There is yet one more way to represent signed integers, which is surprisingly simple. It involves subtracting a fixed constant from a given unsigned number. This representation is called "Bias Notation".**

$$v = \sum_{i=0}^{n-1} 2^i b_i - Bias$$

| $2^7$ | $2^6$ | $2^5$ | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 |

**EX: (Bias = 127)**

$$\begin{aligned} 6 * 1 &= 6 \\ 13 * 16 &= 208 \\ -\ 127 & \\ \hline 87 & \end{aligned}$$

**Why? Monotonicity**

# Floating Point Numbers

**Another way to represent numbers is to use a notation similar to Scientific Notation. This format can be used to represent numbers with fractions ($3.90 \times 10^{-4}$), very small numbers ($1.60 \times 10^{-19}$), and large numbers ($6.02 \times 10^{23}$). This notation uses two fields to represent each number. The first part represents a normalized fraction (called the significand), and the second part represents the exponent (i.e. the position of the "floating" binary point).**

$$Normalized \quad Fraction \times 2^{Exponent}$$


**"dynamic range" "bits of accuracy"**

# IEEE 754 Format

**Single precision format**

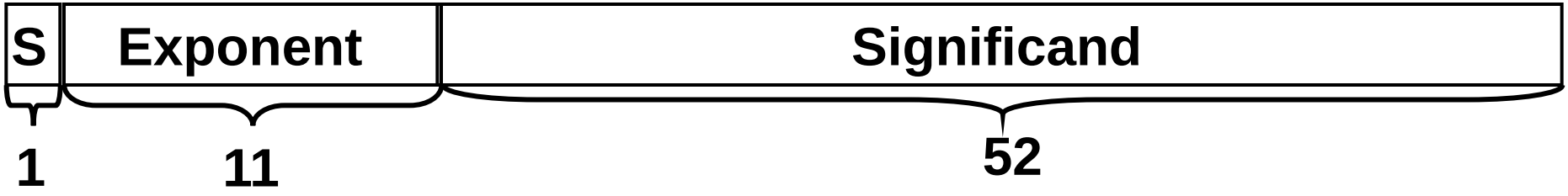
**This is effectively a signed magnitude fixed-point number with a "hidden" 1.**

**The 1 is hidden because it provides no information after the number is "normalized"**


**The exponent is represented in bias 127 notation. Why?**

$$v = -1^{S} \times 1.Significand \times 2^{Exponent-127}$$

**Double precision format**




$$v = -1^{S} \times 1.Significand \times 2^{Exponent-1023}$$

# Summary

1) **Selecting the encoding of information has important implications on how this information can be processed, and how much space it requires.**
2) **Computer arithmetic is constrained by finite representations, this has advantages (it allows for complement arithmetic) and disadvantages (it allows for overflows, numbers too big or small to be represented).**
3) **Bit patterns can be interpreted in an endless number of ways, however important standards do exist**
   - **Two's complement**
   - **IEEE 754 floating point**