# Digital Representation of Information
## (signed whole numbers)

- *(from last time)* What about ***subtraction***?
  - Computer treats subtracting a number as adding a *negative* number
  - Must first know how *negative* numbers are represented
- Can you think of a straightforward way to represent negative numbers?
- *Sign-magnitude* representation

  *Why this convention?*
  - *Most significant bit* = *sign bit* (0 is positive, 1 is negative)
  - Other bits represent magnitude (same as unsigned numbers)
  - (illustrate using 4 bits)



# Digital Representation of Information
## (signed whole numbers)

- Sign-magnitude (example using 4 bits):

| decimal | binary | decimal | binary |
|---|---|---|---|
| 0 | 0000 | -0 | 1000 |
| 1 | 0001 | -1 | 1001 |
| 2 | 0010 | -2 | 1010 |
| 3 | 0011 | -3 | 1011 |
| 4 | 0100 | -4 | 1100 |
| 5 | 0101 | -5 | 1101 |
| 6 | 0110 | -6 | 1110 |
| 7 | 0111 | -7 | 1111 |

## Digital Representation of Information
### (signed whole numbers)

- Why most modern computers don't use sign-magnitude representation?
  - Arithmetic is very complicated
    - Consider the rules for adding 2 signed numbers (for example):
      *If numbers have same sign, ...*
      *If the numbers differ in sign, ...*
  - Has 2 representations for 0
- Can you think of another straightforward way to represent negative numbers?
- *Ones' complement* representation
  - Flip *zeroes to ones* and *ones to zeroes*
  - (illustrate using 4 bits)



## Digital Representation of Information
### (signed whole numbers)

- Ones' complement (example using 4 bits):

| decimal | binary | decimal | binary |
|---|---|---|---|
| 0 | 0000 | -0 | 1111 |
| 1 | 0001 | -1 | 1110 |
| 2 | 0010 | -2 | 1101 |
| 3 | 0011 | -3 | 1100 |
| 4 | 0100 | -4 | 1011 |
| 5 | 0101 | -5 | 1010 |
| 6 | 0110 | -6 | 1001 |
| 7 | 0111 | -7 | 1000 |

# Digital Representation of Information
## (signed whole numbers)

- Ones' complement (reason for the name):
  - Consider using ***8*** bits to represent 51 and –51 (or any other pairs within the range of values that can be represented):

| decimal | binary | decimal | binary |
|---|---|---|---|
| 51 | 00110011 | -51 | 11001100 |

```
      8 ones
     11111111b
     00110011b = 51
(-)  ---------
     11001100b = -51
```

ones

```
-51 = 11001100b = 11111111b - 51
```

$n$-bit ones' complement of x = $\underbrace{1......1}_{n\text{ ones}}$b - x



# Digital Representation of Information
## (signed whole numbers)

- Why most modern computers don't use ones' complement representation?
  - Solves the arithmetic problem (suffered by sign-magnitude representation)
    - To add two ones' complement numbers:
      (1) Add as if unsigned
      (2) If there's carry out the left end, add it back to the right end (***end-around carry***)
    - To subtract a ones' complement number from another:
      (1) Take the ones' complement of the number to be subtracted
      (2) Add as if unsigned
  - But still has 2 representations for 0
- How do most modern computers represent negative numbers?
- *Two's complement* representation (illustrate using 4 bits)
  - *Take ones' complement* then *add 1*
  - (illustrate)

## Digital Representation of Information
(signed whole numbers)

- Two's complement (example using 4 bits):

| decimal | binary | decimal | binary |
|---|---|---|---|
| 0 | 0000 | **(0)** | **(0000)** |
| 1 | 0001 | -1 | 1111 |
| 2 | 0010 | -2 | 1110 |
| 3 | 0011 | -3 | 1101 |
| 4 | 0100 | -4 | 1100 |
| 5 | 0101 | -5 | 1011 |
| 6 | 0110 | -6 | 1010 |
| 7 | 0111 | -7 | 1001 |
| | | **-8** | **1000** |

- Note that the ***leftmost bit*** *still serves well as the* ***sign bit***



## Digital Representation of Information
(signed whole numbers)

*note "–" sign*

- Polynomial expansion for two's complement number:
  - $-d_n2^n + d_{n-1}2^{n-1} + d_{n-2}2^{n-2} + \ldots + d_22^2 + d_12^1 + d_0$
- Can still use *Horner's scheme* to evaluate:
  - $-d_n2^n + d_{n-1}2^{n-1} + d_{n-2}2^{n-2} + \ldots + d_22^2 + d_12^1 + d_0 =$
    $((\ldots((-d_nr + d_{n-1})2 + d_{n-2})2 + \ldots + d_2)2 + d_1)2 + d_0$
- Example ($n$ is 6):
  - 1011010b =
    $-1\times2^6 + 0\times2^5 + 1\times2^4 + 1\times2^3 + 0\times2^2 + 1\times2^1 + 0\times2^0 =$
    $(((((-1\times2 + 0)2 + 1)2 + 1)2 + 0)2 + 1)2 + 0 = -38$

# Digital Representation of Information
## (signed whole numbers)

- Two's complement (reason for the name):
  - Commonly mentioned relationship with ones' complement:
    **(n-bit two's complement of x) = (n-bit ones' complement of x) + 1**
  - From previous discussion (of Ones' complement):
    **(n-bit ones' complement of x) = (n ones)b – x**
  - Substituting:
    **(n-bit two's complement of x) = (n ones)b – x + 1**
  - Rearranging:
    **(n-bit two's complement of x) = (n ones)b + 1 – x**
  - Or:
    **(n-bit two's complement of x) = $2^n$ – x**

two



# Digital Representation of Information
## (signed whole numbers)

- Simple algorithm for obtaining two's complement of a binary number in 1 step:
  - Traverse the bits from the *rightmost bit* to the *leftmost bit* and do the following during the traversal:
    - ☞ Keep *all digits until and including the first 1* unchanged
    - ☞ Flip all further digits (if any) to the *left* of the first 1
  - Example:

    *traverse*

    *flip these* *first 1*

    *given number:* **11010 100**b

    *Two's complement of given number:* **00101 100**b
  - For further examples, look under the earlier slide that gives *Two's complement example using 4 bits*

# Digital Representation of Information
## (signed whole numbers)

- Why simple algorithm works:
  - Not-so-simple algorithm (by definition):
    - Flip all bits
    - Add 1
  - Simple algorithm:
    - Traverse bits from right to left
    - Locate "1st occurrence of 1" & leave it (& any 0's to its right) alone
    - Flip all bits left of "1st occurrence of 1"
  - Simple algorithm ≡ Not-so-simple algorithm:
    - Given: `<some pattern>1<n 0s>`
    - Complement: `<complement of some pattern>0<n 1s>`
    - Complement + 1: `<complement of some pattern>1<n 0s>`

compare

- 2 "don't apply" situations:
  - `<some 0s>` → only 1 representation for zero
  - `1<some 0s>` → can't represent complement of most negative number



# Digital Representation of Information
## (signed whole numbers)

- Distinguishing between two's complement *representation* and two's complement *operation*:
  - The *two's complement representation* of *–x* is the result of applying *two's complement operation* to the representation of ***x***
  - Examples:
    - **00110011b** is the 8-bit two's complement *representation* of **51**
    - **11001101b** is the 8-bit two's complement *representation* of **-51**
    - The two's complement *operation* applied to **00110011b** (= **51**) is **11001101b** (= **-51**)
    - The two's complement *operation* applied to **11001101b** (= **-51**) is **00110011b** (= **51**)

## Digital Representation of Information
### (signed whole numbers)

- Adding two's complement numbers:
  - Simply add the numbers as if they were unsigned numbers
  - If there's any carry out the left end, throw it away
  - Example cases (using 8 bits):

```
   11  1    (carries)
    01001010b =  74
    11001101b = -51
(+)----------   ---
   100010111b =  23
   ↑
   throw away
```

```
     11 11 (carries)
    10110110b = -74
    00110011b =  51
(+)----------   ---
    11101001b = -23
```

```
   111111  (carries)
    10110110b =  -74
    11001101b =  -51
(+)----------   ----
   110000011b = -125
   ↑
   throw away
```



## Digital Representation of Information
### (signed whole numbers)

- More about two's complement representation:
  - Two's complement arithmetic is even easier than 1's complement
    - ☞ Though computing 2's complements is slightly harder
  - There's no negative zero problem:
    **-00000000b = 11111111b + 1**
    **= 00000000b** (when carry out the left end is discarded)
  - Unfortunately, a new problem is introduced → there's a negative number for which there's no corresponding positive number:
    - ☞ In 8-bit Two's complement: **10000000b = -128**
    - ☞ Taking Two's complement: **01111111b + 1 = 10000000b** (!)
  - There's no way to avoid this anomaly (*do you see why?*)
  - Having a negative number without a corresponding positive doesn't commonly present problems in two's complement...
    - ☞ Because the anomaly occurs at the fringes
    - ☞ But often that number must be tested for and treated specially

# Digital Representation of Information
## (signed whole numbers)

- Two's complement in hex:
  - We can of course do it this way:
    - Convert hex number into binary
    - Take two's complement in binary
    - Convert two's complement in binary into hex
  - But it's also easily done directly in hex:
    - Taking ones' complement is equivalent to subtracting the number from all **1**s in binary
    - In hex it is subtracting the number from all **F**s
      <u>TIP:</u> It's probably easiest to mentally convert each digit to decimal, do the subtraction, then convert back to hex
    - Once the ones' complement is obtained, add 1 to get the two's complement
      <u>CAVEAT:</u> Be sure to do the addition in *hex*
      **1 + 9 = A** (*not* **10**)
      **1 + F = 0** carry **1**

*Example:*
Find two's complement of **3DA6h**

```
      FFFFh
      3DA6h
(-)   -----
      C259h   (ones' complement)
         1h
(+)   -----
      C25Ah   (two's complement)
```



# Digital Representation of Information
## (signed whole numbers)

- Which two's complement hex numbers are negative?
  - (*Can you come out with the rules?*)
  - If the first hex digit is in the range **8-F**, ...
    - The number is *negative*
  - If the first hex digit is in the range **0-7**, ...
    - The number is *positive*
  - (*Can you explain why?*)

# Digital Representation of Information
## (signed whole numbers)

- Changing lengths of two's complement representations
  - *Why do we need to know how to do that?*
  - In two's complement arithmetic, ...
    - All operands must be of the *same* length
    - *First bit* of an operand is always interpreted as the *sign bit*
  - And we need...
    - More bits to represent large numbers than small numbers
  - There are thus occasions when we have to...
    - Increase or decrease the number of bits in our representation
  - So we need to know how to do it



# Digital Representation of Information
## (signed whole numbers)

- Shortening two's complement numbers (in binary):
  - Can remove as many bits as we like from the left end as long as
    - They are all the *same as the sign bit*
    - The *original sign bit must be retained* in the shortened representation

| | |
|---|---|
| 11111111 11011000<br>↓<br>11011000 | ✓ |
| 11110111 11011000<br>↓<br>11011000 | ✗ |
| 11111111 01011000<br>↓<br>01011000 | ✗ |

# Digital Representation of Information
## (signed whole numbers)

- Lengthening two's complement numbers (in binary):
  - (*What do you think?*)
  - Can always be done
    - Simply copy the sign bit into as many positions to the left as necessary
    - Process is called *sign extension*

```
         11011000
            ↓
11111111 11011000
-----------------
         01011000
            ↓
00000000 01011000
```

✓ ✓



# Digital Representation of Information
## (signed whole numbers)

- Shortening two's complement numbers in hex:
  - (*Can you arrive at some guidelines?*)
  - Only *series* of **0**'s or *series* of **F**'s can be removed (*why?*)
  - **0**'s can be removed from the *left end* as long as...
    - The *first remaining digit* is in the range **0-7** (*why?*)
  - **F**'s can be removed from the *left end* as long as...
    - The *first remaining digit* is in the range **8-F** (*why?*)
- Lengthening two's complement numbers in hex:
  - (*Can you arrive at some guidelines?*)
  - Only *series* of **0**'s or *series* of **F**'s can be added (*why?*)
  - If the *leftmost digit* is in the range **0-7**...
    - As many **0**'s as needed can be added to the *left* end (*why?*)
  - If the *leftmost digit* is in the range **8-F**...
    - As many **F**'s as needed can be added to the *left* end (*why?*)

# Digital Representation of Information
## (signed whole numbers)

- Range of signed two's complement numbers:
  - (*Can you write the range for **n** bits?*)
  - ($-2^{n-1}$) to ($2^{n-1} - 1$)
  - Examples:

| Storage Type | Range (low–high) | Powers of 2 |
|---|---|---|
| Signed byte | −128 to +127 | $-2^7$ to $(2^7 - 1)$ |
| Signed word | −32,768 to +32,767 | $-2^{15}$ to $(2^{15} - 1)$ |
| Signed doubleword | −2,147,483,648 to 2,147,483,647 | $-2^{31}$ to $(2^{31} - 1)$ |
| Signed quadword | −9,223,372,036,854,775,808 to +9,223,372,036,854,775,807 | $-2^{63}$ to $(2^{63} - 1)$ |



# Digital Representation of Information
## (signed whole numbers)

- Overflow involving two's complement numbers:
  - (*What does overflow mean?*)
  - Condition that occurs when the result of an arithmetic operation is too big to be represented with the number of bits we are using

```
     01010111 =   87
     00101010 =   42
(+) ---------------
     10000001 = -127
```

```
      10101001 = -87
      11010110 = -42
(+) ---------------
     101111111 = 127
```

  - Results are incorrect because it's not possible to represent the true answers (129, –129) as 8-bit two's complement numbers

## Digital Representation of Information
### (signed whole numbers)

- Detecting overflow (for two's complement numbers):
  - (*Can you think of two common-sense rules?*)
  - ***Rule 1:***
    - Adding 2 numbers with *different* signs
    - Overflow will *never* occur (*can you see why?*)
  - ***Rule 2:***
    - Adding 2 numbers with the *same* sign
    - Overflow occurs if the *sign of the result is different*

***Quick quiz***: What indicates *overflow in unsigned addition*?



## Digital Representation of Information
### (signed whole numbers)

- Overflow detection examples (two's complement) :

```
     01010111
     00101010
(+)  --------
     10000001
```

*(overflow)*

```
     11001011
     11010110
(+)  --------
     10100001
```

*(no overflow)*

```
     10101001
     11010110
(+)  --------
    101111111
```

*(overflow)*

```
     11001011
     01010110
(+)  --------
    100100001
```

*(no overflow)*

## Digital Representation of Information
### ("unsigned rendition of" signed whole numbers)

- (*biased* / *excess* representation)
- Basic idea is to…
  - … ***adjust/shift*** each number ***upward*** by…
  - … ***fixed*** amount so that…
  - … ***smallest*** (***most negative***) number becomes ***0***
    (*fixed amount* applied is called *bias* or *excess*)
  - … and represent adjusted numbers as ***unsigned***
- What bias to cover ***same range of numbers covered by two's complement*** with ***same number of bits***?
  - With *n* bits:
    - Range using *two's complement*: **[$-2^{n-1}$, $2^{n-1} - 1$]**
    - To cover same range *biased*: use excess of **$2^{n-1}$**
  - *E.g.*: $n = 4$, excess = $2^3 = 8$
    - Two's complement range: [-8, 7]
    - "rendered range": [0, 15]
    - Note: negative if MSB = 0, positive if MSB = 1

***excess-8*** representation of [-8, 7]

| decimal | excess-8 |
|---|---|
| -8 | 0000 |
| -7 | 0001 |
| -6 | 0010 |
| -5 | 0011 |
| -4 | 0100 |
| -3 | 0101 |
| -2 | 0110 |
| -1 | 0111 |
| 0 | 1000 |
| 1 | 1001 |
| 2 | 1010 |
| 3 | 1011 |
| 4 | 1100 |
| 5 | 1101 |
| 6 | 1110 |
| 7 | 1111 |



## Digital Representation of Information
### ("unsigned rendition of" signed whole numbers)

- (*biased* / *excess* representation continued)
- Working with numbers represented as such:
  - Involves *bias addition* during *encoding*
  - Involves *bias subtraction* during *decoding*
- Unnecessarily complicated for regular use
- Selected for use in ***IEEE-754***
  - When representing *exponent* (of floating-point number)
  - (more on this later)