Ranges

At this point it should be clear that the binary number system can represent any whole-number (integer) value possible by the decimal number system, yet there is a limitation defined by the number of bits available for the representation—the *word size*.

For any whole number value, whether *integer/signed* (positive or negative) or *cardinal/unsigned/all-positive* (only positive), the word size limits the range of values that can be stored.

Consider the following examples (as practise, determine the values in decimal),

ex: (cardinal; 8-bit word, all-positive)

```
1111 1111₂ (largest)          _____10
.
.
.
0000 0001₂ (smallest)         _____10
0000 0000₂ (zero)
```

ex: (integer; 8-bit word, 2's comp. negation)

```
0111 1111₂ (largest positive) _____10
.
.
.
0000 0001₂ (smallest positive)_____10
0000 0000₂ (zero)
1111 1111₂ (smallest negative)_____10
.
.
.
1000 0000₂ (largest negative) _____10
```

In both cases the count of values represented (i.e., the *number* of numbers) is based strictly on the word size: 8-bits => 2^8 = 256 values (this includes zero (0)).

A nice "rule of thumb" for establish the range of a binary number, regardless of the word size ,

(all-positive): 111—111 (largest)
…
000—000 (smallest)

(2's comp.): 011—111 (largest pos)
…
000—001 (smallest pos)
000—000 (zero)
111—111 (smallest neg)
…
100—000 (largest neg)
[and the largest negative is always: `-(largest positive+1)` *]*

In programming, changing a variable from an 8-bit data type (**char** in C++, or **byte** in VB) to a 16-bit data type (**long** in C++, or **integer** in VB) allows the variable to store values of a much larger range.

> *If data types with wider word sizes represent better, larger ranges, why are so many data types used? What is the disadvantage of using data types that are too large?*

The range of possible values based on the word size is not only an issue with integer representations. Fractional representations (fixed- and floating-point; seen in the next lecture) are also constrained by the word size used to define the given data type.

Following lists are of data types in C++ and VB. Examine the word sizes for each type and ranges of possible values.

C++ (32-bit—Unix, Linux, Windows) data types, word sizes, and ranges

Type	**bits (bytes)**	**Range**
unsigned char	8 (1 byte)	0 to 255 (ASCII)
char	8 (1 byte)	-128 to 127 (ASCII)
short int	16 (2 bytes)	-32,768 to 32,767
unsigned int	32 (4 bytes)	0 to 4,294,967,295
int	32 (4 bytes)	-2,147,483,648 to 2,147,483,647
unsigned long	32 (4 bytes)	0 to 4,294,967,295
enum	32 (4 bytes)	-2,147,483,648 to 2,147,483,647
long	32 (4 bytes)	-2,147,483,648 to 2,147,483,647
float	32 (4 bytes)	$3.4*10^{-38}$ to $3.4*10^{38}$
double	64 (8 bytes)	$1.7*10^{-308}$ to $1.7*10^{308}$
long double	80 (10 bytes)	$3.4*10^{-4932}$ to $1.1*10^{4932}$

Visual Basic (32-bit—Windows) data types, word sizes, and ranges

Type	**bits (bytes)**	**Range**
Byte	8 (1 byte)	0 to 255 (ASCII)
Boolean	16 (2 byte)	True or False
Integer	16 (2 bytes)	-32,768 to 32,767
Long	32 (4 bytes)	-2,147,483,648 to 2,147,483,647
Singe Positive	32 (4 bytes)	$1.40129*10^{-45}$ to $3.402823*10^{38}$
Single Negative	32 (4 bytes)	$-3.402823*10^{38}$ to $-1.40129*10^{-45}$
Object	32 (4 bytes)	*a memory reference address to object*
Double Positive	64 (8 bytes)	$4.94065645841247*10^{-24}$ to $1.79769313486232*10^{308}$
Double Negative	64 (8 bytes)	$-1.79769313486232*10^{308}$ to $-4.94065645841247*10^{-24}$
Currency	64 (8 bytes)	-922337203685477.5808 to 922337203685466.5807
Date	64 (8 bytes)	Jan 1, 100 to Dec 31, 9999
String	*10 bytes + string length*	0 to 2 billion characters
User-Defined	??	*size is the sum of all data types encapsulated in the new type*
Variant	Date, Time, Floating Point, or String	16 to 22 bytes in length minimum; ranges in size

Numeric Cycling

As described above, in binary calculations, the word size limits the range of values by *constraining* the bit-width of every number. For example, this means that in an addition calculation, both operands (the original values) and the result must all be of the same word size. If the result of a calculation exceeds the word size, the extra significant bit (left-most) is lost.

This situation leads to an interesting phenomenon in binary representation called a numeric cycling, such that it seems as though the number series never ends (although it should because of word size). but repeats over and over again when repeated *addition* or *subtraction* operations are performed.

> ***Note:*** *A numeric "cycle" exists only with 1's complement and 2's complement. Sign magnitude does not describe a sequential transition from negative to positive, or positive to negative.*

Consider the numeric series of a 4-bit word, all-positive,

1111_2	15_{10}	largest positive
1110	14	
1101	13	
1100	12	
1011	11	
1010	10	
1001	9	
1000	8	
0111	7	
0110	6	
0101	5	
0100	4	
0011	3	
0010	2	
0001	1	smallest positive
0000	0	zero

And the same 4-bit word size but under 2's complement,

0111_2	7_{10}	largest positive
0110	6	
0101	5	
0100	4	
0011	3	
0010	2	
0001	1	smallest positive
0000	0	zero
1111	-1	smallest negative
1110	-2	
1101	-3	
1100	-4	
1011	-5	
1010	-6	
1001	-7	
1000	-8	largest negative

The *cycle* in both cases is apparent when attempting to "go beyond" either the *top* or *bottom* of each range. This attempt, and failure, is called an "overflow error" (see below).

Using the 4-bit words above, consider calculations in decimal and similar calculations in binary,

ex: (1) with 4-bit word, 2's complement

```
   7            0111
+ 1₁₀        + 0001₂
----           ----
   8₁₀          1000₂  = -8₁₀   so 7+1=-8, a cycle from top to bottom
```

ex: (2) with 4-bit word, 2's complement

```
  -8            1000
+-1₁₀        + 1111₂
----           ----
 -9₁₀          10111₂  = 0111₂ = 7₁₀   so -8-1=7, cycle from bottom to top
```

In the next examples, although "all-positive" can not store negative numbers, the CPU performs 2's comp if a negative value is used or subtraction is attempted; but clearly, with all-positive the final result can never be negative.

ex: (3) with 4-bit word, all-positive

```
   15            1111
+   1₁₀        + 0001₂
 ----           ----
   16₁₀         10000₂  = 0₁₀   so 15+1=0, a cycle from top to bottom
```

ex: (4) with 4-bit word, 2's complement

```
   0            0000
+-1₁₀        + 1111₂
----           ----
 -1₁₀          1111₂  = 1111₂ = 15₁₀   so 0-1=15, a cycle from bottom to top
```

Notice in examples (2) and (3), the "extra bit" must be discarded because of the word size limitation (overflow).

Overflow

"Overflow errors" are the result of a calculation producing a result that exceeds the word size and can not be stored (represented) correctly. Once an overflow error occurs there is no way it can be corrected; further, there is no reason to attempt a correction—the result exceeds the word size, and the word size can not be changed in mid-calculation.

A little forethought on the part of the programmer/designer can reduce the possibility of errors by selecting data types that correctly accommodate the values to be stored. But this is not complete prevention, so algorithms must be included to detect overflow errors.

Most (if not all) programming languages include such algorithms in programs when they are *compiled* from high-level code (C++, Java, Pascal, VB, COBOL, etc.) to machine-language (language of the CPU). Modern CPUs also have routines available within the "math co-processor" chip for overflow detection, but the currently running program must explicitly call these routines during a calculation.

Consider how an overflow error can be detected at the completion of a calculation?
(Hint: for "all-positive," consider the magnitude of the value. For "2's comp," consider the sign of the result.)