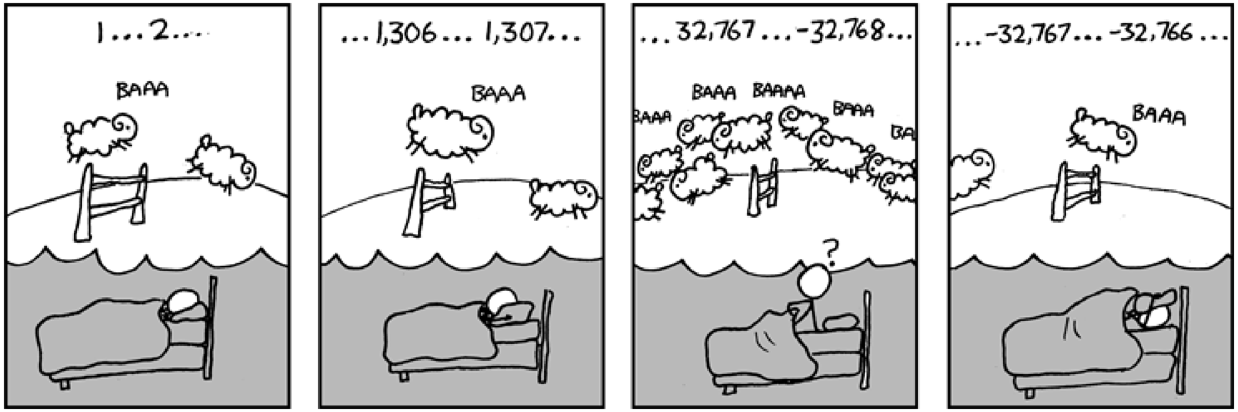
CS 261
Fall 2018

Mike Lam, Professor



https://xkcd.com/571/

Integer Encodings

Integers

- Topics
  - C integer data types
  - Unsigned encoding
  - Signed encodings
  - Conversions

Integer data types in C99

C data type	Minimum	Maximum	
[signed] char	−127	127	1 byte
unsigned char	0	255	
short	−32,767	32,767	2 bytes
unsigned short	0	65,535	
int	−32,767	32,767	2 bytes
unsigned	0	65,535	
long	−2,147,483,647	2,147,483,647	4 bytes
unsigned long	0	4,294,967,295	
int32_t	−2,147,483,648	2,147,483,647	4 bytes
uint32_t	0	4,294,967,295	
int64_t	−9,223,372,036,854,775,808	9,223,372,036,854,775,807	8 bytes
uint64_t	0	18,446,744,073,709,551,615	

Figure 2.11 Guaranteed ranges for C integral data types. The C standards require that the data types have at least these ranges of values.

Integer data types on stu

All sizes in bytes; sizes in red are larger than mandated by C99

char	1	int8_t	1
unsigned char	1	uint8_t	1
		bool	1
short	2		
unsigned short	2	int16_t	2
		uint16_t	2
int	4		
unsigned int	4	int32_t	4
		uint32_t	4
long	8		
unsigned long	8	int64_t	8
long long	8	uint64_t	8
unsigned long long	8	size_t	8

Unsigned integer encoding

- Bit i represents the value 2^i
  - Bits typically written from most to least significant (i.e., $2^3\ 2^2\ 2^1\ 2^0$)
  - This is the same encoding we saw on Tuesday!
  - No representation of negative numbers

$$1 = 1 = 0\cdot 2^3 + 0\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0 = [0001]$$

$$5 = 4 + 1 = 0\cdot 2^3 + 1\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0 = [0101]$$

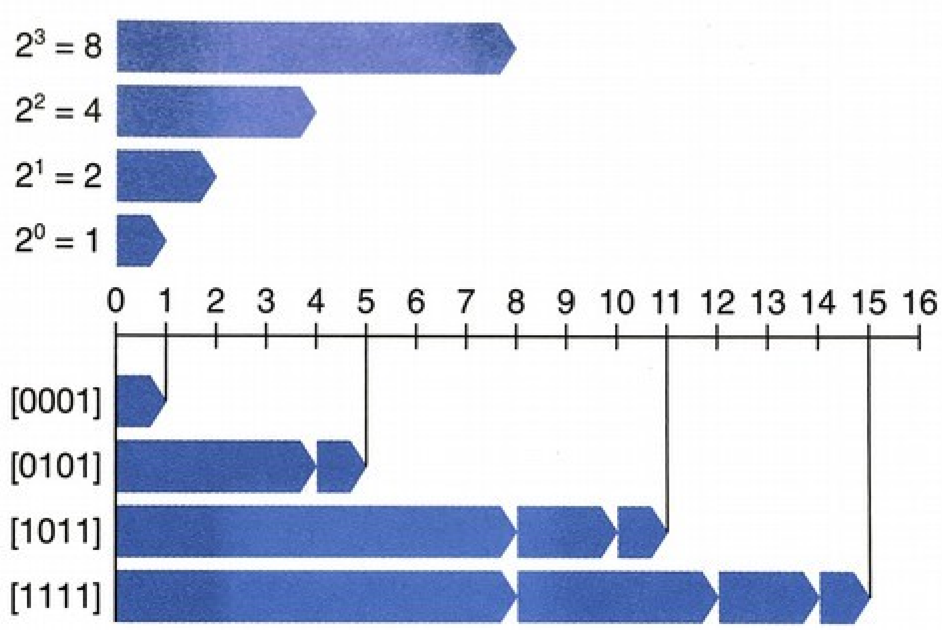
$$11 = 8 + 2 + 1 = 1\cdot 2^3 + 0\cdot 2^2 + 1\cdot 2^1 + 1\cdot 2^0 = [1011]$$

$$15 = 8 + 4 + 2 + 1 = 1\cdot 2^3 + 1\cdot 2^2 + 1\cdot 2^1 + 1\cdot 2^0 = [1111]$$

Unsigned integer encoding

- Textbook's notation
  - Each bar represents a bit
  - Add together bars to represent the contributions of each bit value to the overall value

Figure 2.12
Unsigned number examples for $w = 4$. When bit i in the binary representation has value 1, it contributes 2^i to the value.



Signed integer encodings

- Sign magnitude
  - Most natural and intuitive
- Ones’ complement
  - Helps with two’s complement conversions
- Two’s complement
  - Cleanest arithmetic but not intuitive
  - Most modern signed integer types use this!

Sign magnitude

- Sign magnitude
  - Interpret most-significant bit as a sign bit
  - Interpret remaining bits as a normal unsigned int (the magnitude)
  - Disadvantages:
    - Two zeros: -0 and +0 [1000 and 0000]
    - Less useful for arithmetic because the sign bit has no relationship with the magnitude--cannot use unsigned arithmetic logic!

```
0 011 = 3
1 011 = -3

0 111 = 7
```

```
0 111 (7)
1 011 (-3)
? 010
```

Caution: language technicalities

- Ones’ complement and two’s complement are both an **operation** and an **encoding**
  - e.g., “perform two’s complement” vs “the number is stored in two’s complement”
- The operation represents the action necessary to negate a number in that encoding.
  - e.g., performing two’s complement (ones’ complement and add one) negates a number in two’s complement encoding
- If you have a value in a particular encoding:
  - If the sign bit is not set, it’s a “regular” positive number
  - If it is set, perform the operation to recover the positive value

Ones’ complement

- Ones’ complement
  - Invert all the bits (~ operator in C) to negate
  - Still have two representations of zero (1111 and 0000)
  - Also, less useful for arithmetic than two’s complement
  - However, there is a neat trick: to perform two’s complement, just do ones’ complement then add one

Ex: 5 = 0101 → (one’s comp.) → 1010 → (add one) → 1011 = -5 (-8 + 2 + 1)

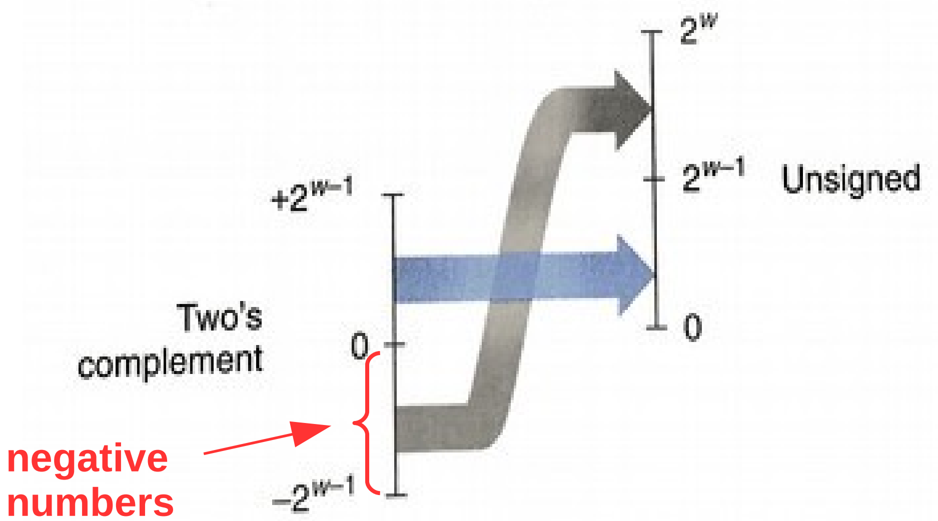
Aside: Why does this work? The sum of a number x and its ones’ complement is all ones (or 2^N-1 where N is the number of bits), so its ones' complement can be expressed as 2^N-1 - x. Because taking the two’s complement of x is equivalent to subtracting x from 2^N, if we add one to the ones' complement the results are equal:

$(2^N-1 - x) + 1 = 2^N - x$

Two's complement encoding

- Two's complement
  - Take ones' complement then add one to negate
    - Equivalently: subtract number from 2^N where N is the number of bits
  - Implication: half of all values as negative
    - One more negative number than positive numbers
  - Positive numbers "wrap around" to negative ones halfway through

2's Comp.		**Unsigned**
-1	1111	15
	…	
-7	1001	9
-8	1000	8
7	0111	7
	…	
1	0001	1
0	0000	0

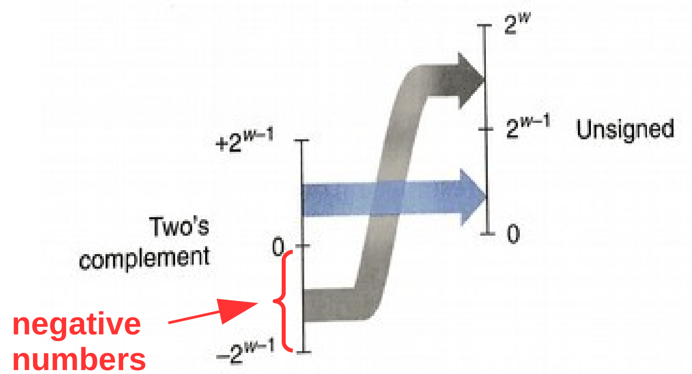


Comparison

- We'll see one more signed integer encoding next week: "offset binary" / "biased" / "excess"
  - For now, here's a comparison (for 1-byte integers):

Binary	Unsigned	Sign Mag	Ones' C	Two's C	Offset-127
1111 1111	255	-127	-0	-1	128
1111 1110	254	-126	-1	-2	127
…	…	…	…	…	…
1000 0001	129	-1	-126	-127	2
1000 0000	128	-0	-127	-128	1
0111 1111	127	127	127	127	0
0111 1110	126	126	126	126	-1
…	…	…	…	…	…
0000 0001	1	1	1	1	-126
0000 0000	0	0	0	0	-127

Comparison



Binary	Unsigned	Sign Mag	Ones' C	Two's C	Offset-127
1111 1111	255	-127	-0	-1	128
1111 1110	254	-126	-1	-2	127
…	…	…	…	…	…
1000 0001	129	-1	-126	-127	2
1000 0000	128	-0	-127	-128	1
0111 1111	127	127	127	127	0
0111 1110	126	126	126	126	-1
…	…	…	…	…	…
0000 0001	1	1	1	1	-126
0000 0000	0	0	0	0	-127

Two’s complement encoding

- Alternate interpretation: value of most significant bit is negated
  - i.e., start at most negative number and build back up towards zero

Figure 2.12
Unsigned number examples for $w = 4$. When bit i in the binary representation has value 1, it contributes 2^i to the value.

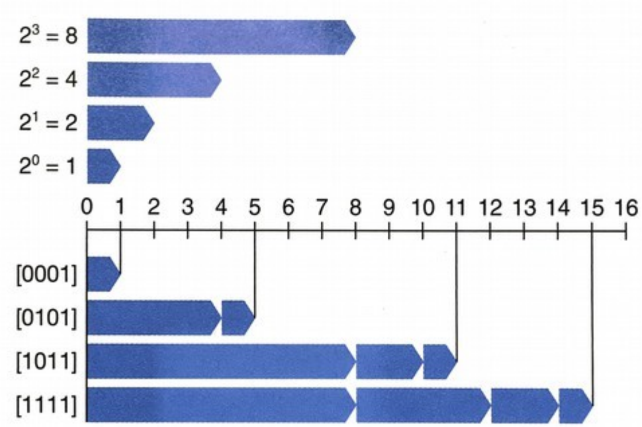
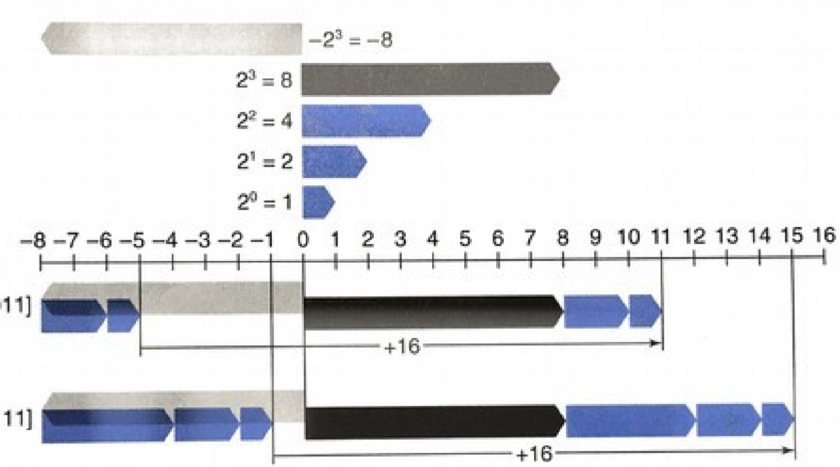


Figure 2.16
Comparing unsigned and two’s-complement representations for $w = 4$. The weight of the most significant bit is -8 for two’s complement and $+8$ for unsigned, yielding a net difference of 16.



Two's complement encoding

- Two's complement advantage: uses unsigned arithmetic logic
  - (ignore carries out of the sign bit for now)

  - Ex: 5 – 3 = 5 + (-3) = 0101 + 1101 = 0010 (2)
  - Ex: 1 – 3 = 1 + (-3) = 0001 + 1101 = 1110 (-2)
  - Ex: -2 – 3 = (-2) + (-3) = 1110 + 1101 = 1011 (-5)

```
0011 = 3          0111 (7)
1100            + 1101 (-3)
1101 = -3       ----------
                  0100 (4)

0111 = 7
```

Integer representations

- Information = Bits + Context
  - What does “1011” mean? It depends!

Unsigned:	11
Sign magnitude:	-3
Ones' complement:	-4
Two's complement:	-5

Conversions

- Smaller unsigned → larger unsigned
  - Safe; zero-extend to preserve value

`0101 (5)  → 0000 0101 (5)`

- Smaller two's comp. → larger two's comp.
  - Safe; sign-extend to preserve value

`1101 (-3) → 1111 1101 (-3)`

- Larger → smaller (unsigned or two's comp.)
  - Overflow if new type isn't large enough to fit (truncate)

`0000 0101 (5)  → 0101 (5)`
`0011 0101 (53) → 0101 (5)`

- Unsigned → two's comp.
  - Overflow if first bit is non-zero (otherwise, no change)

`0101 (5)  → 0101 (5)`
`1101 (13) → 1101 (-2)`

- Two's comp. → unsigned
  - Overflow if value is negative (otherwise, no change)

`0101 (5)  → 0101 (5)`
`1101 (-2) → 1101 (13)`